HANDBOOK OF FLOATING-POINT ARITHMETIC

JEAN-MICHEL MULLER

NICOLAS BRISEBARRE

FLORENT DE DINECHIN

CLAUDE-PIERRE JEANNEROD

VINCENT LEFÈVRE

GUILLAUME MELQUIOND

NATHALIE REVOL

DAMIEN STEHLÉ

SERGE TORRES

BIRKHÄUSER

Jean-Michel Muller
Nicolas Brisebarre
Florent de Dinechin
Claude-Pierre Jeannerod
Vincent Lefèvre
Guillaume Melquiond
Nathalie Revol
Damien Stehlé
Serge Torres

Handbook of Floating-Point Arithmetic

Jean-Michel Muller
CNRS, Laboratoire LIP
École Normale
    Supérieure de Lyon
46, allée d'Italie
69364 Lyon Cedex 07
France
jean-michel.muller@
    ens-lyon.fr

Nicolas Brisebarre
CNRS, Laboratoire LIP
École Normale
    Supérieure de Lyon
46, allée d'Italie
69364 Lyon Cedex 07
France
nicolas.brisebarre@
    ens-lyon.fr

Florent de Dinechin
ENSL, Laboratoire LIP
École Normale
    Supérieure de Lyon
46, allée d'Italie
69364 Lyon Cedex 07
France
florent.de.dinechin@
    ens-lyon.fr

Claude-Pierre Jeannerod
INRIA, Laboratoire LIP
École Normale
    Supérieure de Lyon
46, allée d'Italie
69364 Lyon Cedex 07
France
claude-pierre.jeannerod@
    ens-lyon.fr

Vincent Lefèvre
INRIA, Laboratoire LIP
École Normale
    Supérieure de Lyon
46, allée d'Italie
69364 Lyon Cedex 07
France
vincent@vinc17.net

Guillaume Melquiond
INRIA Saclay – Île-de-
    France
Parc Orsay Université
4, rue Jacques Monod
91893 Orsay Cedex
France
guillaume.melquiond@
    inria.fr

Nathalie Revol
INRIA, Laboratoire LIP
École Normale
    Supérieure de Lyon
46, allée d'Italie
69364 Lyon Cedex 07
France
nathalie.revol@ens-lyon.fr

Damien Stehlé
CNRS, Macquarie University,
    and University of Sydney
School of Mathematics
    and Statistics
University of Sydney
Sydney NSW 2006
Australia
damien.stehle@gmail.com

Serge Torres
ENSL, Laboratoire LIP
École Normale
    Supérieure de Lyon
46, allée d'Italie
69364 Lyon Cedex 07
France
serge.torres@ens-lyon.fr

Contents

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 an 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 5.33Hz Z3 computer.

A vast quantity of very diverse arithmetics was implemented between the 1960s and the early 1980s. The radix (radices 2, 4, 16, and 10 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, and its follower, the IEEE 854-1987 "radix-independent standard." 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. It has been revised during recent years, and its new version, the IEEE 754-2008 standard, was released in August 2008.

By carefully specifying the behavior of the arithmetic operators, the 754-1985 standard allowed researchers to design extremely smart 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 hardly 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 and the recent revision of the IEEE 754 standard led us to the decision to compile into a book selected parts of the vast knowledge on floating-point arithmetic. This book is designed for programmers of numerical applications, compiler designers, programmers of floating-point algorithms, designers of arithmetic operators, and more generally the 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 by an actual program the described techniques, 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 IEEE 754 and 854 standards and the new revised standard. The second part shows how the features of the standard can be used to develop smart 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). The fourth part is devoted to the implementation of elementary functions. The fifth part presents some extensions: certification of floating-point arithmetic and extension of the precision. The last part is devoted to perspectives and the Appendix.

Acknowledgements

Jean-Michel Muller, Nicolas Brisebarre	Lyon, France
Florent de Dinechin, Claude-Pierre Jeannerod	July 2009
Vincent Lefèvre, Guillaume Melquiond	
Nathalie Revol, Damien Stehlé	
Serge Torres	

List of Figures

List of Tables

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 [220, 400] and semi-logarithmic [294] number systems, continued fractions [228, 424], rational numbers [227] and possibly infinite strings of rational numbers [275], level-index number systems [71, 318], fixed-slash and floating-slash number systems [273], and 2-adic numbers [425].

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 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

$$\pm m_0.m_1 m_2 \cdots m_{p-1} \times \beta^e,$$

where e, called the *exponent*, is an integer, and $m_0.m_1 m_2 \cdots m_{p-1}$, 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* [222], 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 [434], and a very interesting analysis of the Babylonian mathematics related to YBC 7289 was done by Fowler and Robson [138].

The arithmetic of the slide rule, invented around 1630 by William Oughtred [433], 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 [341]. And yet, the first real, modern implementation of floating-point arithmetic was in Konrad Zuse's Z3 computer, built in 1941 [66]. It used a radix-2 floating-point number system, with 14-bit significands, 7-bit exponents and 1-bit sign. 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 [347]. Photographs of Konrad Zuse and the Z3 can be viewed at `http://www.computerhistory.org/projects/zuse_z23/` and `http://www.konrad-zuse.de/`.

Readers interested in the history of computing devices should have a look at the excellent book by Aspray et al. [15].

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 [63, 85, 90, 91, 129, 414, 413, 428, 429]. 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 [222]. Another one [177] 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 [44, 76, 232] and Chapter 2) have shown that radix 2 with the *implicit leading bit convention* (see Chapter 2) 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 [10]. This standard specifies various formats, the behavior of the basic operations and conversions, and exceptional 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. 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.

---

[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 3.3.2 and Chapter 7.

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

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 [11], 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 [187].

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 worse 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 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 4.

An example of a huge calculation that requires much care was carried out by Laskar's team at the Paris observatory [243]. 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 [19], 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 [290, 122]. 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 [208] 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 [7, 306].

  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 [7].

  Also, in January 2004, a bridge between Germany and Switzerland did not fit at the border because the two countries use a different definition of the sea level.[5]

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 [289] and analyzed by Kahan [208, 291]. Let (u_n) be the sequence defined as

$$\begin{cases} u_0 &= 2 \\ u_1 &= -4 \\ u_n &= 111 - \dfrac{1130}{u_{n-1}} + \dfrac{3000}{u_{n-1}u_{n-2}}. \end{cases} \quad (1.1)$$

---

[5]See `http://www.spiegel.de/panorama/0,1518,281837,00.html`.

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.

```c
#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

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.

these computed terms is very tiny, but nonzero. This suffices to make the computed sequence "converge" to 100.

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 = 1.718281828459045235360287 47135;
      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 [352], and analyzed by various authors [93, 268],

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

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.

```c
#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.

Chapter 2

Definitions and Basic Notions

A s said in the Introduction, roughly speaking, a radix-β floating-point number x is a number of the form

$$m \cdot \beta^e,$$

where β is the *radix* of the floating-point system, m such that $|m| < \beta$ is the *significand* of x, and e is its *exponent*. And yet, portability, accuracy, and the ability to prove interesting and useful properties as well as to design smart algorithms require more rigorous definitions, and much care in the specifications. This is the first purpose of this chapter. The second one is to deal with basic problems: rounding, exceptions, properties of real arithmetic that become wrong in floating-point arithmetic, best choices for the radix, and radix conversions.

2.1 Floating-Point Numbers

Let us now give a more formal definition of the floating-point numbers. Although we try to be somewhat general, the definition is largely inspired from the various IEEE standards for floating-point arithmetic (see Chapter 3).

Main definitions

A floating-point format is (partially)[1] characterized by four integers:

- a *radix* (or *base*) $\beta \geq 2$;

- a *precision* $p \geq 2$ (roughly speaking, p is the number of "significant digits" of the representation);

---

[1] A full definition of a floating-point format also specifies the binary encoding of the significands and exponents. It also deals with special values: infinities, results of invalid operations, etc.

- two *extremal exponents* $e_{\min}$ and $e_{\max}$ such that $e_{\min} < e_{\max}$. In all practical cases, $e_{\min} < 0 < e_{\max}$.

A finite floating-point number in such a format is a number for which there exists at least one representation (M, e) such that

$$x = M \cdot \beta^{e-p+1}, \tag{2.1}$$

where

- M is an integer of absolute value less than or equal to $\beta^p - 1$. It is called the *integral significand* of the representation of x;

- e is an integer such that $e_{\min} \leq e \leq e_{\max}$, called the *exponent* of the representation of x.

The representation (M, e) of a floating-point number is not necessarily unique. For instance, with $\beta = 10$ and $p = 3$, the number 17 can be represented either by 17×10^0 or by 170×10^{-1}, since both 17 and 170 are less than $\beta^p = 1000$. The set of these equivalent representations is called a *cohort*.

The number

$$\beta^{e-p+1}$$

from Equation (2.1) is called the *quantum* of the representation of x. We will call the *quantum exponent* the number

$$q = e - p + 1.$$

The notion of quantum is closely related to the notion of *ulp* (*unit in the last place*); see Section 2.6.1.

Another way to express the same floating-point number x is by using the triplet (s, m, e), so that

$$x = (-1)^s \cdot m \cdot \beta^e,$$

where

- e is the same as before;

- $m = |M| \cdot \beta^{1-p}$ is called the *normal significand* (or, more simply, the *significand* of the representation). It has one digit before the radix point, and at most $p - 1$ digits after (notice that $0 \leq m < \beta$); and

- $s \in \{0, 1\}$ is the sign of x.

The significand is also frequently (and slightly improperly) called the *mantissa* in the literature.[2] According to Goldberg [148], the term "significand" was coined by Forsythe and Moler in 1967 [136].

_____

[2]The *mantissa* of a non-negative number is the fractional part of its logarithm.

Consider the following "toy system." We assume radix $\beta = 2$, precision $p = 4$, $e_{\min} = -7$, and $e_{\max} = +8$. The number[3] $416_{10} = 110100000_2$ is a floating-point number. It has one representation only, with integral significand 13_{10} and exponent 8_{10}, since

$$416 = 13 \cdot 2^{8-4+1}.$$

The quantum of this representation is $2^5 = 32$. Note that a representation such as $26 \cdot 2^{7-4+1}$ is excluded because $26 > 2^p - 1 = 15$. In the same toy system, the number $4.25_{10} = 17 \cdot 2^{-2}$ is not a floating-point number, as it cannot be exactly expressed as $M \cdot \beta^{e-p+1}$ with $|M| \leq 2^4 - 1$.

When x is a nonzero arbitrary real number (i.e., x is not necessarily representable in a given floating-point format), we will denote *infinitely precise significand* of x (in radix β) the number

$$\frac{x}{\beta^{\lfloor \log_\beta |x| \rfloor}},$$

where $\beta^{\lfloor \log_\beta |x| \rfloor}$ is the largest integer power of β smaller than $|x|$.

Normalized representations, normal and subnormal numbers

As explained above, some floating-point numbers may have several representations (M, e). Nevertheless, it is frequently desirable to require unique representations. In order to have a unique representation, one may want to *normalize* the finite nonzero floating-point numbers by choosing the representation for which the exponent is minimum (yet larger than or equal to $e_{\min}$). The obtained representation will be called a *normalized representation*. Requiring normalized representations allows for easier expression of error bounds, it somewhat simplifies the implementation, and it allows for a one-bit saving in radix 2.[4] Two cases may occur.

- In general, such a representation satisfies $1 \leq |m| < \beta$, or, equivalently, $\beta^{p-1} \leq |M| < \beta^p$. In such a case, one says that the corresponding value x is a *normal* number. When x is a normal floating-point number, its infinitely precise significand is equal to its significand.

- Otherwise, one necessarily has $e = e_{\min}$, and the corresponding value x is said to be a *subnormal* number (the term *denormal number* is often used too). In that case, $|m| < 1$ or, equivalently, $|M| \leq \beta^{p-1} - 1$. The special case of zero will be dealt with later.

---

[3] We will frequently write "xxx $\cdots$ xx$_b$" to designate the number whose radix-b representation is xxx $\cdots$ xx. To avoid complicated notation, we will tend to omit b each time its value is obvious from the context.

[4] We will see in Chapter 3 that the new IEEE 754-2008 standard requires normalized representations in radix 2, but not in radix 10.

With such a normalization, as the representation of a finite nonzero number x is unique, the values M, q, m, and e only depend on the value of x. We therefore call e the exponent of x, q its quantum exponent ($q = e - p + 1$), M its integer significand, and m its significand.

For instance, in radix-2 normalized floating-point arithmetic, with[5] $p = 24, e_{\min} = -126$, and $e_{\max} = 127$, the floating-point number f that is nearest to $1/3$ is a normal number. Its exponent is -2, and its integral significand is 11184811. Therefore,

$$f = 11184811 \times 2^{-2-24+1} = 0.33333334326744079589843750_{10}$$

and the quantum of f is $2^{-2-24+1} \approx 2.98 \times 10^{-8}$, while its quantum exponent is -25.

In the same floating-point system, the number 3×2^{-128} is a subnormal number. Its exponent is $e_{\min} = -126$, its quantum is 2^{-149}, and its integral significand is

$$3 \times 2^{149-128} = 6291456.$$

In radix 2, the first digit of the significand of a normal number is a 1, and the first digit of the significand of a subnormal number is a 0. If we have a special encoding (in practice, in the exponent) that tells us if a number is normal or subnormal, there is no need to store the first bit of its significand. This *leading bit convention*, or *implicit bit convention*, or *hidden bit convention* is frequently used.

Hence, in radix 2, the significand of a normal number always has the form

$$1.m_1 m_2 m_3 \cdots m_{p-1},$$

whereas the significand of a subnormal number always has the form

$$0.m_1 m_2 m_3 \cdots m_{p-1}.$$

In both cases, the digit sequence $.m_1 m_2 m_3 \cdots m_{p-1}$ is called the *trailing significand* of the number. It is also sometimes called the *fraction*.

Some "extremal" floating-point numbers are important and will be used throughout this book:

- the smallest positive normal number is $\beta^{e_{\min}}$;

- the largest finite floating-point number is

$$\Omega = (\beta - \beta^{1-p}) \cdot \beta^{e_{\max}};$$

---

[5]It is the *single-precision* format of IEEE 754-1985, and the *binary32* format of IEEE 754-2008. See Chapter 3 for details.

- and the smallest positive subnormal number is

$$\alpha = \beta^{e_{\min} - p + 1}.$$

For instance, still using the format ($\beta = 2$, $p = 24$, $e_{\min} = -126$, and $e_{\max} = +127$) of the example given above, the smallest positive normal number is

$$2^{-126} \approx 1.175 \times 10^{-38};$$

the smallest positive subnormal number is

$$\alpha = 2^{-149} \approx 1.401 \times 10^{-45};$$

and the largest finite floating-point number is

$$\Omega = \left(2 - 2^{-23}\right) \cdot 2^{127} \approx 3.403 \times 10^{+38}.$$

A note on subnormal numbers

Subnormal numbers have probably been the most controversial part of IEEE 754-1985 [148, 207]. As stated by Schwarz et al. [371], they are the most difficult type of numbers to implement in floating-point units. As a consequence, they are sometimes implemented in software rather than in hardware, which may result in huge execution times when such numbers appear in a calculation.

One can of course define floating-point systems without subnormal numbers. And yet, the availability of these numbers allows for what Kahan calls *gradual underflow*: the loss of precision is slow instead of being abrupt. For instance [205, 176], the availability of subnormal numbers implies the following interesting property: if $a \neq b$, then the computed value of $b - a$ is necessarily nonzero.[6] This is illustrated by Figure 2.1. Gradual underflow is also sometimes called *graceful underflow*. In 1984, Demmel [110] analyzed various numerical programs, including Gaussian elimination, polynomial evaluation, and eigenvalue calculation, and concluded that the availability of gradual underflow significantly eases the writing of stable numerical software.

Many properties presented in Chapter 4, such as Sterbenz's lemma (Lemma 2, page 122), are true only if subnormal numbers are available (otherwise, we must add the condition *if no underflow occurs then...* in these properties). Reference [370] presents techniques for implementing subnormal numbers in hardware at a reasonable cost.

---

[6]Note that this property is also true if a and b are subnormal numbers.

Figure 2.1: The positive floating-point numbers in the toy system $\beta = 2$ and $p = 3$. Above: normal floating-point numbers only. In that set, $b - a$ cannot be represented, so that the computation of $b - a$ in round-to-nearest mode (see Section 2.2) will return 0. Below: the subnormal numbers are included in the set of floating-point numbers.

A note on underflow

The word "underflow" can be ambiguous. For many naive users, it may mean that the exact result of an arithmetic operation has an absolute value below the smallest nonzero *representable* number (that is, when subnormal numbers are available, $\alpha = \beta^{e_{\min}-p+1}$). *This is not the definition chosen in the context of floating-point arithmetic*. As a matter of fact, there are two slightly different definitions. Unfortunately, the IEEE 754-1985 and 754-2008 standards did not make a choice between them (see Chapter 3). As a consequence, for the very same sequence of calculations, the underflow exception may be signaled on one "conforming" platform, and not signaled on another one.

Definition 1 (Underflow before rounding)**.** *In radix-β arithmetic, an arithmetic operation or function underflows if the exact result of that operation or function is of absolute value strictly less than $\beta^{e_{\min}}$.*

Definition 2 (Underflow after rounding)**.** *In radix-β, precision-p arithmetic, an arithmetic operation or function underflows if the result we would compute with an* unbounded exponent range *and precision p would be nonzero and of absolute value strictly less than $\beta^{e_{\min}}$.*

Figure 2.2 illustrates these two definitions by pointing out (assuming round-to-nearest—see Section 2.2—and radix 2) the tiny domain in which

they lead to a different conclusion. For instance in radix-2, precision-p arithmetic, as soon as $2e_{\min} < -p - 2$ (which holds in all practical cases), $\sin(2^{e_{\min}})$ should underflow according to Definition 1, and should not underflow according to Definition 2.

If we had unbounded exponents, these would be FP numbers

Floating-point numbers

$2^{e_{\min}}$

with unbounded exponents, numbers in that domain would not round to $2^{e_{\min}}$

with unbounded exponents, numbers in that domain would round to $2^{e_{\min}}$

Figure 2.2: In this radix-2 example, if the exact result is in the grey area, then there is an underflow before rounding (Definition 1), and no underflow after rounding (Definition 2).

One should not worry too much about this ambiguity, as the two definitions disagree extremely rarely. What really matters is that when the returned result is a subnormal number, an underflow is signaled. This is useful since it warns the user that the arithmetic operation that returned that result might be less accurate (in terms of relative error) than usual. Indeed, we will see throughout this book that many algorithms and properties hold "provided that no underflow occurs."

Special floating-point data

Some data cannot be expressed as normal or subnormal numbers. An obvious example is the number zero, which will require a special encoding. There are also other examples that are not fully "numeric."

- It is in general highly desirable to have a closed system so that any machine operation can be well specified (without generating any trap[7]), even if it is an invalid operation over the real numbers

---

[7]A trap is a transfer of control to a special handler routine.

(e.g., $\sqrt{-5}$ or $0/0$). A special datum, called NaN (*Not a Number*) in IEEE 754-1985 and its successors, can be introduced for this purpose. Any invalid operation will return a NaN.[8]

- Moreover, due to the limited exponent range, one needs to introduce more special values. To cope with values whose magnitude is larger than the maximum representable one, either an unsigned infinity (∞) or two signed infinities ($+\infty$ and $-\infty$) can be added to the floating–point system. If signed infinities are chosen, one may want signed zeros (denoted $+0$ and -0) too, for symmetry. The IEEE standards for floating-point arithmetic have signed zeros and infinities. As noticed by Kahan [204], signed zeros also help greatly in dealing with branch cuts of complex elementary functions. However, unless at least a third, unsigned zero is introduced, this choice also yields an asymmetry for the exact zero and the result of $(+0) + (-0)$. As an example, the system of the Texas Instruments pocket calculators has 3 zeros and 3 infinities: positive, negative, and with indeterminate sign.[9]

Konrad Zuse's Z3 computer, built in 1941, already had mechanisms for dealing with floating-point exceptions [346]. The current choices will be detailed in Chapter 3.

2.2 Rounding

2.2.1 Rounding modes

In general, the result of an operation (or function) on floating-point numbers is not exactly representable in the floating-point system being used, so it has to be *rounded*. In the first floating-point systems, the way results were rounded was not always fully specified. One of the most interesting ideas brought out by IEEE 754-1985 is the concept of *rounding mode*: how a numerical value is rounded to a finite (or, possibly, infinite) floating-point number is specified by a *rounding mode* (or *rounding direction attribute*), that defines a rounding function $\circ$. For example, when computing $a + b$, where a and b are floating-point numbers, the returned result is $\circ(a + b)$. One can define many possible rounding modes. For instance, the four rounding modes that appear in the IEEE 754-2008 standard are:

- round toward $-\infty$: $\mathrm{RD}(x)$ is the largest floating-point number (possibly $-\infty$) less than or equal to x;

- round toward $+\infty$: $\mathrm{RU}(x)$ is the smallest floating-point number (possibly $+\infty$) greater than or equal to x;

---

[8]The IEEE 754-1985 standard actually defines two kinds of NaN: *quiet* and *signaling* NaNs. See Section 3.1.6 page 69 for an explanation.

[9]http://tigcc.ticalc.org/doc/timath.html

- round toward zero: $RZ(x)$ is the closest floating-point number to x that is no greater in magnitude than x (it is equal to $RD(x)$ if $x \geq 0$, and to $RU(x)$ if $x \leq 0$);

- round to nearest: $RN(x)$ is the floating-point number that is the closest to x. A tie-breaking rule must be chosen when x falls exactly halfway between two consecutive floating-point numbers. A frequently chosen tie-breaking rule is called *round to nearest even*: x is rounded to the only one of these two consecutive floating-point numbers whose integral significand is even. This is the default mode in the IEEE 754-2008 Standard (see Chapter 3). The IEEE 754-2008 Standard also defines another tie-breaking rule: called round *ties to away* (see Section 3.4.6). In general, properties one might expect from a tie-breaking rule are sign symmetry[10] $RN(-x) = -RN(x)$, lack of statistical bias, ease of implementation, and reproducibility.

Figure 2.3 illustrates these four rounding modes.

Figure 2.3: The four rounding modes. Here we assume that x and y are positive numbers.

When the exact result of a function is rounded according to a given rounding mode (as if the result were computed with infinite precision and unlimited range, then rounded), one says that the function is *correctly rounded*. The term *exactly rounded* is sometimes used [148].

In radix 2 and precision p, how a positive real value x, whose infinitely precise significand is $1.m_1 m_2 m_3 \ldots$, is rounded can be expressed as a function of the bit *round* $= m_p$ (*round bit*) and the bit *sticky* $= m_{p+1} \vee m_{p+2} \vee \ldots$ (*sticky bit*), as summarized in Table 2.1 (see Chapter 8).

In the following, we will call a *rounding breakpoint* a value where the rounding function changes. In round-to-nearest mode, the rounding breakpoints are the exact middles of consecutive floating-point numbers. In the

---

[10]This is the only property required by the LIA-2 Standard, see Section 3.7.1, page 109.

round / sticky	RD	RU	RN
0 / 0	$-$	$-$	$-$
0 / 1	$-$	$+$	$-$
1 / 0	$-$	$+$	$- \,/\, +$
1 / 1	$-$	$+$	$+$

Table 2.1: Rounding a radix-2 infinitely precise significand, depending on the "round" and "sticky" bits. Let $\circ \in \{\mathrm{RN}, \mathrm{RD}, \mathrm{RU}\}$ be the rounding mode we wish to implement. A "$-$" in the table means that the significand of $\circ(x)$ is $1.m_1 m_2 m_3 \ldots m_{p-1}$, i.e., the truncated exact significand. A "$+$" in the table means that one needs to add 2^{-p+1} to the truncated significand (possibly leading to an exponent change if all the m_i's up to m_{p-1} are 1). The "$- \,/\, +$" corresponds to the halfway cases for the round-to-nearest (RN) mode (the rounded result depends on the chosen convention).

other rounding modes, called *directed rounding modes*, they are the floating-point numbers themselves.

Returning a correctly rounded result is fairly easily done for the arithmetic functions (addition/subtraction, multiplication, division) and the square root, as Chapters 8, 9 and 10 will show. This is why the IEEE 754-1985 standard for floating-point arithmetic requires that these functions be correctly rounded (see Chapter 3). And yet, it may be extremely difficult for some functions.[11] In such a case, if for any exact result y, one always returns either $\mathrm{RD}(y)$ or $\mathrm{RU}(y)$, one says that the returned value is a *faithful* result and that the arithmetic is faithful. Beware: sometimes, this is called a "faithful rounding" in the literature, but this is not a *rounding mode* as defined above, since the obtained result is not a fully specified function of the input value.

2.2.2 Useful properties

As shown later (especially in Chapters 4, 5, and 6), correct rounding is useful to design and prove algorithms and to find tight and certified error bounds.

An important and helpful property is that for any of the four rounding modes presented above, the rounding function $\circ$ is a *nondecreasing function*; i.e., if $x \le y$, then $\circ(x) \le \circ(y)$. Moreover, if y is a floating-point number, then $\circ(y) = y$, which means that when the exact result of a correctly rounded function is a floating-point number, we get that result exactly.

Also, if the rounding mode is *symmetric* (i.e., it is RZ or RN with a symmetrical choice in case of a tie), then a correctly rounded implementation

---

[11]If the exact value of the function is very close to a rounding breakpoint, the function must be approximated with great accuracy to make it possible to decide which value must be returned. This problem, called the *Table Maker's Dilemma*, is addressed in Chapter 12.

preserves the symmetries of a function. With the other rounding modes, properties such as

$$\mathrm{RU}(a+b) = -\mathrm{RD}(-a-b)$$

or

$$\mathrm{RD}(a \times b) = -\mathrm{RU}((-a) \times b)$$

can sometimes be used for saving a change of rounding mode if it is a complicated or costly operation.

Finally, we note that in the case of tiny or huge values, the rounding modes of the IEEE standards behave as shown by the following property. This property is useful in particular when optimizing the implementation of correctly rounded arithmetic operators (see Chapters 8, 9, and 10).

Property 1. *With* $\alpha = \beta^{e_{\min}-p+1}$ *(smallest positive subnormal number) and* $\Omega = (\beta - \beta^{1-p}) \cdot \beta^{e_{\max}}$ *(largest finite floating-point number), one has the following values when rounding the real* x:

- $\mathrm{RN}(x) = \begin{cases} +0 & \text{if } 0 < x \leq \alpha/2, \\ +\infty & \text{if } x \geq (\beta - \beta^{1-p}/2) \cdot \beta^{e_{\max}}; \end{cases}$

- $\mathrm{RD}(x) = \begin{cases} +0 & \text{if } 0 < x < \alpha, \\ +\infty & \text{if } x \geq \beta^{e_{\max}+1}; \end{cases}$

- $\mathrm{RU}(x) = \begin{cases} \alpha & \text{if } 0 < x \leq \alpha, \\ +\infty & \text{if } x > \Omega. \end{cases}$

2.2.3   Relative error due to rounding

In the following, we call the *normal range* the set of the real numbers of absolute value between $\beta^{e_{\min}}$ and $\Omega = (\beta - \beta^{1-p}) \cdot \beta^{e_{\max}}$, and the *subnormal range* the set of the numbers less than $\beta^{e_{\min}}$. When approximating a nonzero real number x by $\circ(x)$ (where $\circ$ is the active rounding mode), a relative error

$$\epsilon(x) = \left| \frac{x - \circ(x)}{x} \right|$$

happens. That relative error is plotted in Figure 2.4 in a simple case. When $\circ(x) = x = 0$, we consider that the relative error is 0.

If x is in the normal range, the relative error $\epsilon(x)$ is less than or equal to

$$\frac{1}{2}\beta^{1-p}$$

in round-to-nearest mode, and less than

$$\beta^{1-p}$$

Figure 2.4: Relative error $|x - \mathrm{RN}(x)|/|x|$ that occurs when representing a real number x in the normal range by its nearest floating-point approximation $\mathrm{RN}(x)$, in the toy floating-point format $\beta = 2, p = 3$.

in the "directed" rounding modes. If x is in the subnormal range (thus, assuming subnormal numbers are available), the relative error can become very large (it can be close to 1). In that case, we have a bound on the *absolute* error due to rounding:

$$|x - \mathrm{RN}(x)| \leq \frac{1}{2}\beta^{e_{\min}-p+1}$$

in round-to nearest mode, and

$$|x - \circ(x)| < \beta^{e_{\min}-p+1}$$

if $\circ$ is one of the directed rounding modes. More generally, by combining these relative and absolute error bounds, we find that if z is the result of the correctly rounded operation $a\top b$ (that is, if $z = \circ(a\top b)$), and if no overflow occurs, then

$$z = (a\top b)(1 + \epsilon) + \epsilon',$$

with

- $|\epsilon| \leq \frac{1}{2}\beta^{1-p}$ and $|\epsilon'| \leq \frac{1}{2}\beta^{e_{\min}-p+1}$ in round-to-nearest mode, and

- $|\epsilon| < \beta^{1-p}$ and $|\epsilon'| < \beta^{e_{\min}-p+1}$ in directed rounding modes.

Moreover, ϵ and ϵ' cannot both be nonzero [205]. One should notice that

- if z is in the normal range (i.e., if no underflow occurred) then $\epsilon' = 0$;

- if z is in the subnormal range, then $\epsilon = 0$. Moreover, in that case, if the arithmetic operation being performed is addition or subtraction ($\top$ is $+$ or $-$), then we will see (Chapter 4, Theorem 3, page 124) that the result is *exact*, so that $z = a \top b$ (i.e., $\epsilon' = 0$).

The bound on ϵ (namely $\frac{1}{2}\beta^{1-p}$ of β^{1-p}, depending on the rounding mode) is frequently called the *unit roundoff* (see Definition 6). The bounds given here on the errors due to rounding will be used in particular in Chapter 6.

2.3 Exceptions

In IEEE 754-1985 arithmetic (but also in other standards), an *exception* can be signaled along with the result of an operation. This can take the form of a status flag (which must be "sticky," so that the user does not need to check it immediately, but after some sequence of operations, for instance at the end of a function) and/or some trap mechanism.

Invalid: This exception is signaled when an input is invalid for the function. The result is a NaN (when supported). Examples: $(+\infty) - (+\infty)$, $0/0$, $\sqrt{-1}$.

DivideByZero, a.k.a. infinitary (in the LIA-2 standard): This exception is signaled when an exact infinite result is defined for a function on finite inputs, e.g., at a pole. Examples: $1/0$, $\log(+0)$.

Overflow: This exception is signaled when the rounded result with an unbounded exponent range would have an exponent larger than $e_{\max}$.

Underflow: This exception is signaled when a tiny (less than $\beta^{e_{\min}}$) nonzero result is detected. This can be according to Definition 1 (i.e., before rounding), or according to Definition 2 (i.e., after rounding).

Underflow handling can be different whether the exact result is exactly representable or not, which makes sense: the major interest in signaling the underflow exception is to warn the user that the obtained result might not be very accurate (in terms of relative error). Of course, this is not the case when the obtained result is exact. This is why, in the IEEE 754-2008 standard, if the result of an operation is exact, then the underflow flag is *not* raised (see Chapter 3).

Inexact: This exception is signaled when the exact result y is not exactly representable ($\circ(y) \neq y$, y not being a NaN).

Unlike the IEEE standards, the LIA-2 standard [191] does not regard an inexact value as an exception. It also defines an additional exception: *absolute_precision_underflow*, which is used when the angle argument of a trigonometric function is larger than some threshold (which can be changed by the

implementation). The reason for that choice is that even if a large input argument to a sine, cosine, or tangent is accurate to $1/2 \, \mathrm{ulp}$ (see Section 2.6.1 for the definition of ulp), the result could be very inaccurate or even meaningless.[12] Moreover, the underflow exception is not signaled in some particular cases where such an exception would not really be useful, e.g., for the sine of a subnormal number.

In general, the values of $e_{\min}$ and $e_{\max}$ are chosen to be almost symmetrical: $e_{\min} \approx -e_{\max}$. One of the reasons for that is that we expect an acceptable behavior of the reciprocal function $1/x$. The minimum positive normal number is $\beta^{e_{\min}}$. Its reciprocal is $\beta^{-e_{\min}}$, which is below the maximum normal number threshold as long as $-e_{\min} \leq e_{\max}$, i.e., $e_{\min} \geq -e_{\max}$. So, if one chooses $e_{\min} = -e_{\max}$ or $e_{\min} = 1 - e_{\max}$ (which is the choice in IEEE 754-2008, probably for parity reasons: the number of different exponents that can be represented in a given binary integer format is an even number, whereas if $e_{\min}$ were exactly equal to $-e_{\max}$, we would have an odd number of exponents), one has the following properties.

- If x is a normal number, $1/x$ never produces an overflow.

- If x is a finite floating-point number, $1/x$ can underflow, but the rounded result is not zero (for common values of p), as soon as subnormal numbers are available.

As explained, by Hauser [176], among others, it is often easier and cheaper to deal with an exception after the fact than to prevent it from occurring in the first place. When exception handling is not available, avoiding exceptional cases in programs requires artful numerical tricks that make programs slower and much less clear. Hauser gives the example of the calculation of the norm

$$\mathcal{N} = \sqrt{\sum_{i=1}^{N} x_i^2},$$

where the x_i's are floating-point numbers. Consider computing $\mathcal{N}$ using a straightforward algorithm (Algorithm 2.1).

Algorithm 2.1 Straightforward calculation of $\sqrt{\sum_{i=1}^{N} x_i^2}$ [176].

$S \leftarrow 0.0$
for $i = 1$ to N **do**
    $S \leftarrow \mathrm{RN}(S + \mathrm{RN}(x_i \times x_i))$
end for
return $\mathrm{RN}(\sqrt{S})$

---

[12]This is a debatable choice, since in some cases, the input argument might well be exact.

Even when the exact value of $\mathcal{N}$ lies in the normal range of the floating-point format being used, Algorithm 2.1 may fail, due to underflow or overflow when evaluating the square of one or several of the x_i's, or when evaluating their sum. There are many solutions for avoiding that. We may for instance emulate an extended range arithmetic or scale the operands (i.e., first examine the input operands, then multiply them by an adequate factor K, so that Algorithm 2.1 can be used with the scaled operands[13] without underflow or overflow, and finally divide the obtained result by K).

These solutions would lead to programs that would be reliable, yet the extended range arithmetic as well as the scaling would significantly slow down the calculations. This is unfortunate since, in the vast majority of practical cases, Algorithm 2.1 would have behaved in a satisfactory way. A possibly better solution, when exception handling is available, is to first use Algorithm 2.1, and then to resort to an extended range arithmetic or a scaling technique only when overflows or underflows occurred during that preliminary computation.

2.4 Lost or Preserved Properties of the Arithmetic on the Real Numbers

The arithmetic on real numbers has several well-known properties. Among them:

- addition and multiplication are commutative operations: $a + b = b + a$ and $a \times b = b \times a$ for all a and b;

- addition and multiplication are associative operations: $a + (b + c) = (a + b) + c$ and $a \times (b \times c) = (a \times b) \times c$ for all a, b, and c;

- distributivity applies: $a \times (b + c) = a \times b + a \times c$.

When the arithmetic operations are correctly rounded, in any of the four rounding modes presented in Section 2.2, floating-point addition and multiplication remain commutative:[14] if $\circ$ is the active rounding mode then, of course, $\circ(a + b) = \circ(b + a)$ and $\circ(a \times b) = \circ(b \times a)$ for all floating-point numbers a and b. However, associativity and distributivity are lost. More precisely, concerning associativity, the following can occur.

---

[13]Notice that Hammarling's algorithm for routine xNRM2 in LAPACK goes only once through the x_i's but nevertheless avoids overflow. This is done by computing the scaling factor K on the fly while computing the norm.

[14]However, in a programming language, swapping the terms may yield a different result in practice. This can be noticed in the expression a * b + c * d when a Fused multiply-add is used; see, e.g., Section 7.2.3.

- In some extreme cases, $\circ(a + \circ(b + c))$ can be *drastically* different from $\circ(\circ(a+b)+c)$. A simple example, in radix-β, precision-p arithmetic with round-to-nearest mode is $a = \beta^{p+1}$, $b = -\beta^{p+1}$, and $c = 1$, since

$$\mathrm{RN}(a + \mathrm{RN}(b + c)) = \mathrm{RN}(\beta^{p+1} - \beta^{p+1}) = 0,$$

  whereas

$$\mathrm{RN}(\mathrm{RN}(a + b) + c) = \mathrm{RN}(0 + 1) = 1.$$

  Many studies have been devoted to the finding of good ways of reordering the operands when one wants to evaluate the sum of several floating-point numbers. See Chapter 6 for more details.

- If no overflow or underflow occurs, $P_1 = \circ(\circ(a \times b) \times c)$ can be *slightly* different from $P_2 = \circ(a \times \circ(b \times c))$. More precisely, in radix-β, precision-p arithmetic with round-to-nearest mode:

$$\mathrm{RN}(a \times b) = a \times b \times (1 + \epsilon_1),$$

  with $|\epsilon_1| \leq \frac{1}{2}\beta^{1-p}$, so that

$$P_1 = \mathrm{RN}(\mathrm{RN}(a \times b) \times c) = a \times b \times c \times (1 + \epsilon_1)(1 + \epsilon_2),$$

  with $|\epsilon_2| \leq \frac{1}{2}\beta^{1-p}$. Similarly,

$$P_2 = \mathrm{RN}(a \times \mathrm{RN}(b \times c)) = a \times b \times c \times (1 + \epsilon_3)(1 + \epsilon_4),$$

  with $|\epsilon_3|, |\epsilon_4| \leq \frac{1}{2}\beta^{1-p}$. Therefore,

$$\left(\frac{1 - \frac{1}{2}\beta^{1-p}}{1 + \frac{1}{2}\beta^{1-p}} \right)^2 \leq \frac{P_1}{P_2} \leq \left(\frac{1 + \frac{1}{2}\beta^{1-p}}{1 - \frac{1}{2}\beta^{1-p}} \right)^2,$$

  which gives

$$\frac{P_1}{P_2} = 1 + \epsilon,$$

  with

$$|\epsilon| \leq 2\,\beta^{1-p} + 2\left(\beta^{1-p}\right)^2 + 3/2\left(\beta^{1-p}\right)^3 + \cdots. \tag{2.2}$$

  One should notice that this bound is rather tight. For instance, in the binary32 arithmetic of the IEEE 754-2008 standard ($\beta = 2$, $p = 24$, $e_{\min} = -126$, $e_{\max} = 127$; see Section 3.4, page 79), the bound on ϵ given by (2.2) is $4.00000048 \times 2^{-24}$, whereas, if $a = 8622645$, $b = 16404663$, and $c = 8647279$, then

$$\frac{P1}{P2} = 3.86\cdots \times 2^{-24}.$$

- In case of overflow or underflow, $\circ(a \times \circ(b \times c))$ can be *drastically* different from $\circ(\circ(a \times b) \times c)$. For instance, in binary64 arithmetic ($\beta = 2$, $p = 53$, $e_{\min} = -1022$, $e_{\max} = 1023$; see Section 3.4 page 79), with $a = b = 2^{513}$ and $c = 2^{-1022}$, $\mathrm{RN}(\mathrm{RN}(a \times b) \times c)$ will be $+\infty$, whereas $\mathrm{RN}(a \times \mathrm{RN}(b \times c))$ will be 16.

2.5 Note on the Choice of the Radix

2.5.1 Representation errors

As stated in Chapter 1, various different radices were chosen in the early days of electronic computing, and several studies [56, 44, 76, 232] have been devoted to the best radix choice, in terms of maximal or average representation error. These studies have shown that radix 2 with the implicit leading bit convention gives better worst-case or average accuracy than all other radices.

Cody [76] studied static and dynamic characteristics of various floating-point formats. Let us present his explanations in what is called the *static* case. Assume that floating-point numbers are represented in radix β, where β is a power of 2, and that their significands and exponents are stored on w_s and w_e bits, respectively. As explained in Section 2.2.3, when a nonzero number x in the normal range is represented by the nearest floating-point number $\text{RN}(x)$, a relative representation error

$$\left| \frac{x - \text{RN}(x)}{x} \right|$$

is committed. We want to evaluate the maximum and average values of this relative error, for all x between the smallest positive normal floating-point number $\beta^{e_{\min}}$ and the largest one $\Omega = \beta^{e_{\max}} \cdot (\beta - \beta^{1-p})$.

First, notice that for any given integer k, if $\beta^k x$ remains between $\beta^{e_{\min}}$ and Ω, $(\beta^k x - \text{RN}(\beta^k x))/(\beta^k x)$ is equal to $(x - \text{RN}(x))/x$, so that it suffices to compute the maximum and average values for x between two consecutive powers of β, say, $1/\beta$ and 1.

Second, for evaluating average values, one must choose a probability distribution for the significands of floating-point numbers. For various reasons [161, 222], the most sensible choice is the *logarithmic distribution*, also called *Benford's law* [24]:

$$P(s) = \frac{1}{s \ln \beta}.$$

We easily get the following results.

- If $\beta > 2$, or if $\beta = 2$ and we do not use the hidden bit convention (that is, the first "1" of the significand is actually stored), then the maximum relative representation error is

$$\text{MRRE}(w_s, \beta) = 2^{-w_s - 1} \beta.$$

- If $\beta = 2$ and we use the hidden bit convention, we have

$$\text{MRRE}(w_s, 2) = 2^{-w_s - 1}.$$

- If $\beta > 2$, or if $\beta = 2$ and we do not use the hidden bit convention, then the average relative representation error is

$$\mathrm{ARRE}(w_s, \beta) \approx \int_{1/\beta}^{1} \left(\frac{1}{s \ln \beta} \right) \frac{2^{-w_s} ds}{4s} = \frac{\beta - 1}{4 \ln \beta} 2^{-w_s}.$$

- If $\beta = 2$ and we use the hidden bit convention, that value is halved and we get

$$\mathrm{ARRE}(w_s, 2) \approx \frac{1}{8 \ln 2} 2^{-w_s}.$$

These values seem much in favor of small radices, and yet, we must take into account the following. To achieve the same dynamic range (i.e., to have a similar order of magnitude of the extremal values) as in binary, in radix 2^k, we need around $\log_2(k)$ fewer bits for representing the exponent. These saved bits can, however, be used for the significands. Hence, for a similar total number of bits (sign, exponent, and significand) for the representation of floating-point numbers, a fair comparison between radices 2, 4, and 16 is obtained by taking a value of w_s larger by one unit for radix 4, and two units for radix 16, so that we compare number systems with similar dynamic ranges, and the same value of $w_s + w_e$.

Table 2.2 gives some values of MRRE and ARRE for various formats. From that table, one can infer that radix 2 with implicit bit convention is the best choice from the point of view of the relative representation error. Since radix-2 floating-point arithmetic is also the easiest to implement using digital logic, this explains why it is predominant on current systems.

2.5.2   A case for radix 10

Representation error, however, is not the only issue to consider. A strong argument in favor of radix 10 is that we humans are working, reading, and writing in that radix. The following section will show how to implement the best possible conversions between radices 2 and 10, but such conversions may sometimes entail errors that, in some applications, are unacceptable. A typical example is banking. An interest rate is written on a contract in decimal, and the computer at the bank is legally mandated to conduct interest computations using the exact decimal value of this rate, not a binary approximation to it.

As another example, when some European countries abandoned their local currencies in favor of the euro, the conversion rate was defined by law as a decimal number (e.g., 1 euro = 6.55957 French francs), and the way this conversion had to be implemented was also defined by law. Using any other conversion value, such as 6.559569835662841796875, the binary32 number nearest to the legal value, was simply illegal.

Format	MRRE	ARRE
$\beta = 2, w_s = 64$ first bit stored	$5.421010862 \times 10^{-20}$	$1.955216373 \times 10^{-20}$
$\beta = 2, w_s = 64$ first bit hidden	$2.710505431 \times 10^{-20}$	$9.776081860 \times 10^{-21}$
$\beta = 4, w_s = 65$	$5.421010862 \times 10^{-20}$	$1.466412280 \times 10^{-20}$
$\beta = 8, w_s = 65$	$1.084202172 \times 10^{-19}$	$2.281085767 \times 10^{-20}$
$\beta = 8, w_s = 66$	$5.421010862 \times 10^{-20}$	$1.140542884 \times 10^{-20}$
$\beta = 16, w_s = 66$	$1.084202172 \times 10^{-19}$	$1.833015349 \times 10^{-20}$

Table 2.2: ARRE and MRRE of various formats of comparable dynamic range. The cases $\beta = 2, 4$, and 16 can be directly compared. In the case $\beta = 8$, one cannot get the same dynamic range exactly.

Colishaw [90] gives other examples and also shows how pervasive decimal computing is in business applications.

As the current technology is fundamentally binary, radix 10 will intrinsically be less efficient than radix 2, and indeed even hardware implementations of decimal floating-point are much slower than their binary counterparts (see Table 3.26, page 108 for an example).

However, this is at least partially compensated by other specificities of the financial application domain. In particular, in accounting applications, the floating point in most of the additions and subtractions is actually fixed. Indeed, one adds cents to cents. The good news is that this common case of addition is much easier to implement than the general case:

- first, the significands need not be shifted as in the general case [90] (see Chapters 8 and 9);

- second, such fixed-point additions and subtractions will be exact (entailing no rounding error).

Rounding does occur in financial applications; for instance, when applying a sales tax or an interest rate. However, from an accounting point of view, it is best managed in such a way that one eventually adds only numbers which have already been rounded to the nearest cent.

The reader should have in mind these peculiarities when reading about decimal formats and operations in this book. Most of the intricacies of the decimal part of the IEEE 754-2008 floating-point standard are directly motivated by the needs of accounting applications.

2.6   Tools for Manipulating Floating-Point Errors

2.6.1   The ulp function

In numerical analysis, errors are very often expressed in terms of relative errors. And yet, when we want to express the errors of "nearly atomic" functions (arithmetic operations, elementary functions, small polynomials, sums, and dots products, etc.), it is more adequate (and frequently more accurate!) to express errors in terms of what we would intuitively define as the "weight of the last bit of the significand." Let us define that notion more precisely. The term ulp (acronym for *unit in the last place*) was coined by William Kahan in 1960. The original definition was as follows [209]:

> ulp(x) is the gap between the two floating-point numbers nearest to x, even if x is one of them.

When x is a floating-point number (except, possibly, when x is a power of the radix; see below), we would like ulp(x) to be equal to the *quantum* of x. And yet, it is frequently useful to define that function for other numbers too.

   Several slightly different definitions of ulp(x) appear in the literature [148, 168, 270, 191, 320, 209]. They all coincide as soon as x is not extremely close to a power of the radix. They have properties that differ to a small degree. A good knowledge of these properties may be important, for instance, for anyone who wants to prove sure yet tight bounds on the errors of atomic computations. For instance, we frequently hear or read that correctly rounding to nearest is equivalent to having an error less than 0.5 ulp. This might be true, depending on the radix, on what we define as an ulp (especially near the powers of the radix), and depending on whether we consider function ulp to be taken at the real value being approximated, or at the floating-point value that approximates it.

   Consider first the following definition of the ulp function, due to John Harrison [168, 171].

Definition 3 (Harrison). ulp(x) *is the distance between the closest straddling floating-point numbers a and b (i.e., those with $a \leq x \leq b$ and $a \neq b$), assuming that the exponent range is not upper-bounded.*

   Figure 2.5 shows the values of Harrison's ulp near 1. One can easily find that, in radix β floating-point arithmetic, if x is a floating-point number then Harrison's ulp of x and the quantum of x have the same value, except if x is an integer power of β. Goldberg [148] gives another definition of function ulp.

Definition 4 (Goldberg). *If the FP number $d_0.d_1d_2d_3d_4\ldots d_{p-1}\beta^e$ is used to represent x, it is in error by*

$$\left| d_0.d_1d_2d_3d_4\ldots d_{p-1} - \frac{x}{\beta^e} \right|$$

units in the last place.

Figure 2.5: The values of $\mathrm{ulp}(x)$ *near 1, assuming a binary floating-point system with precision* p, *according to Harrison's definition.*

Figure 2.6: The values of $\mathrm{ulp}(x)$ *near 1, assuming a binary floating-point system with precision* p, *according to Definition 5 (Goldberg's definition extended to the reals). Notice that this definition and Harrison's definition only differ when* x *is a power of the radix.*

This definition does not define ulp as a function of x, since the value depends on which floating-point number approximates x. However, it clearly defines a function $\mathrm{ulp}(X)$, for a *floating-point* number $X \in [\beta^e, \beta^{e+1})$, as β^{e-p+1} (or, more precisely, as $\beta^{\max(e,e_{\min})-p+1}$, if we want to handle the subnormal numbers properly). Hence, a natural generalization to real numbers is the following, which is equivalent to the one given by Cornea, Golliver, and Markstein[15] [86, 270].

Definition 5 (Goldberg's definition, extended to reals). *If* $x \in [\beta^e, \beta^{e+1})$, *then* $\mathrm{ulp}(x) = \beta^{\max(e,e_{\min})-p+1}$.

When x is a floating-point number, this definition coincides with the *quantum* of x. Figure 2.6 shows the values of ulp near 1 according to Definition 5.

Let us now examine some properties of these definitions (see [292] for comments and some proofs). In the following, x is a real number and X is a radix-β, precision-p, floating-point number, $\mathrm{HarrisonUlp}(x)$ is $\mathrm{ulp}(x)$ according to Harrison's definition, and $\mathrm{GenGoldbergUlp}(x)$ is $\mathrm{ulp}(x)$ according to

---

[15]They gave it in radix 2, but generalization to radix β is straightforward.

Goldberg's definition extended to the reals. We assume that $|x|$ is less than the largest representable number, $\Omega = (\beta - \beta^{1-p}) \cdot \beta^{e_{\max}}$.

Property 2. *In radix* 2,

$$|X - x| < \frac{1}{2}\,\mathrm{HarrisonUlp}(x) \Rightarrow X = \mathrm{RN}(x).$$

It is important to notice that Property 2 is not true in radices greater than or equal to 3. Figure 2.7 gives a counterexample in radix 3.

Figure 2.7: This example shows that Property 2 is not true in radix 3. *Here, x satisfies $1 < x < 1 + \frac{1}{2}3^{-p}$ and $X = 1^- = 1 - 3^{-p}$ (if v is a floating-point number, v^- denotes its predecessor, namely, the largest floating-point number less than v, and v^+ denotes its successor). We have $\mathrm{HarrisonUlp}(x) = 3^{-p+1}$, and $|x - X| < 3^{-p+1}/2$, so that $|x - X| < \frac{1}{2}\,\mathrm{HarrisonUlp}(x)$. However, $X \neq \mathrm{RN}(x)$.*

If, instead of considering ulps of the "exact" value x, we consider ulps of the floating-point value X, we have a property that is very similar to Property 2, with the interesting difference that now it holds for any value of the radix.

Property 3. *For any value of the radix β,*

$$|X - x| < \frac{1}{2}\,\mathrm{HarrisonUlp}(X) \Rightarrow X = \mathrm{RN}(x).$$

Now, still with Harrison's definition, we might be interested in knowing if the converse property holds; that is, if having $X = \mathrm{RN}(x)$ implies that X is within $\frac{1}{2}\,\mathrm{HarrisonUlp}(x)$ or $\frac{1}{2}\,\mathrm{HarrisonUlp}(X)$. For the first case, we have the following.

Property 4. *For any radix,*

$$X = \mathrm{RN}(x) \Rightarrow |X - x| \leq \frac{1}{2} \, \mathrm{HarrisonUlp}(x).$$

On the other hand, there is no similar property for the second case: $X = \mathrm{RN}(x)$ does not imply $|X - x| \leq \frac{1}{2} \, \mathrm{HarrisonUlp}(X)$. For example, assume radix 2. Any number x strictly between $1 + 2^{-p-1}$ and $1 + 2^{-p}$ will round to 1, but it will be at a distance from 1 larger than $\frac{1}{2} \, \mathrm{HarrisonUlp}(1) = 2^{-p-1}$.

Concerning Goldberg's definition extended to the reals, we have very similar properties.

Property 5. *In radix 2,*

$$|X - x| < \frac{1}{2} \, \mathrm{GenGoldbergUlp}(x) \Rightarrow X = \mathrm{RN}(x).$$

Property 5 is not true in higher radices: The example of Figure 2.7, designed as a counterexample to Property 2, is also a counterexample to Property 5.

Also, Property 5 does not hold if we replace $\mathrm{GenGoldbergUlp}(x)$ by $\mathrm{GenGoldbergUlp}(X)$. Indeed, $|X - x| < \frac{1}{2} \, \mathrm{GenGoldbergUlp}(X)$ does not imply $X = \mathrm{RN}(x)$ (it suffices to consider x very slightly above $1^- = 1 - \beta^{-p}$, the floating-point predecessor of 1: x will be within $\frac{1}{2} \, \mathrm{GenGoldbergUlp}(1)$ from 1, and yet $\mathrm{RN}(x) = 1^-$). In a way, this kind of counterexample is the only one; see Property 6.

Property 6. *For any radix, if X is not an integer power of β,*

$$|X - x| < \frac{1}{2} \, \mathrm{GenGoldbergUlp}(X) \Rightarrow X = \mathrm{RN}(x).$$

We also have the following.

Property 7. *For any radix,*

$$X = \mathrm{RN}(x) \Rightarrow |X - x| \leq \frac{1}{2} \, \mathrm{GenGoldbergUlp}(x).$$

Property 8. *For any radix,*

$$X = \mathrm{RN}(x) \Rightarrow |X - x| \leq \frac{1}{2} \, \mathrm{GenGoldbergUlp}(X).$$

After having considered properties linked to the round-to-nearest mode, we can try to consider properties linked to the directed rounding modes (i.e., rounding toward $\pm\infty$ and rounding toward zero). One can show the following properties (still assuming $|x|$ is less than the largest representable number).

Property 9. *For any value of the radix β,*

$$X \in \{\mathrm{RD}(x), \mathrm{RU}(x)\} \Rightarrow |X - x| < \mathrm{HarrisonUlp}(x).$$

Note that the converse is not true. There are values X and x for which $|X - x| < \mathrm{HarrisonUlp}(x)$, and X is not in $\{\mathrm{RD}(x), \mathrm{RU}(x)\}$. It suffices to consider the case x slightly above 1 and X equal to $1^- = 1 - \beta^{-p}$, the floating-point predecessor of 1.

Property 10. *For any value of the radix β,*

$$|X - x| < \mathrm{HarrisonUlp}(X) \Rightarrow X \in \{\mathrm{RD}(x), \mathrm{RU}(x)\}.$$

But the converse is not true: $X \in \{\mathrm{RD}(x), \mathrm{RU}(x)\}$ does not imply $|X - x| \leq \mathrm{HarrisonUlp}(X)$.

Property 11.

$$X \in \{\mathrm{RD}(x), \mathrm{RU}(x)\} \Rightarrow |X - x| \leq \mathrm{GenGoldbergUlp}(x).$$

The converse is not true: $|X - x| < \mathrm{GenGoldbergUlp}(x)$ does not imply $X \in \{\mathrm{RD}(x), \mathrm{RU}(x)\}$. It suffices to consider $X = 1^- = 1 - \beta^{-p}$, the floating-point predecessor of 1, and x slightly above 1.

Property 12.

$$X \in \{\mathrm{RD}(x), \mathrm{RU}(x)\} \Rightarrow |X - x| \leq \mathrm{GenGoldbergUlp}(X).$$

Again, the converse is not true: $|X - x| < \mathrm{GenGoldbergUlp}(X)$ does not imply $X \in \{\mathrm{RD}(x), \mathrm{RU}(x)\}$.

After this examination of the properties of these two definitions of the ulp function, which one is to be chosen? A good definition of function ulp:

- should (of course) agree with the "intuitive" notion when x is not in an "ambiguous area" (i.e., x is not very near a power of the radix);

- should be *useful*: after all, for a binary format with precision p, defining $\mathrm{ulp}(1)$ as 2^{-p} (i.e., $1 - 1^-$) or 2^{-p+1} (i.e., $1^+ - 1$) are equally legitimate from a theoretical point of view. What matters is which choice is helpful (i.e., which choice will preserve in "ambiguous areas" properties that are true when we are far enough from them).

From that point of view, it is still not very easy to decide between Definitions 3 and 5. Both preserve interesting properties, yet also set some traps (e.g., the fact that $X = \mathrm{RN}(x)$ does not imply $|X - x| \leq \frac{1}{2} \mathrm{HarrisonUlp}(X)$, or the fact that $|X - x| < \mathrm{GenGoldbergUlp}(x)$ does not imply $X \in \{\mathrm{RD}(x), \mathrm{RU}(x)\}$). These traps sometimes make the task of proving properties of arithmetic algorithms a difficult job when some operand can be very near a power of the radix.

In the remainder of this book, $\mathrm{ulp}(x)$ *will be* $\mathrm{GenGoldbergUlp}(x)$. That is, we will follow Definition 5, not because it is the best (as we have seen, it is difficult to tell which one is the best), but because it is the most used.

2.6.2 Errors in ulps and relative errors

It is important to be able to establish links between errors expressed in ulps, and relative errors. Inequalities 2.3 and 2.5 exhibit such links when X is a normal floating-point number and x is a real number of the same sign.

Converting from errors in ulps to relative errors

First, let us convert from errors in ulps to relative errors. Assume that $|x - X| = \alpha \operatorname{ulp}(x)$. Assuming no underflow, we easily get

$$\left| \frac{x - X}{x} \right| \le \alpha \times \beta^{-p+1}. \tag{2.3}$$

Converting from relative errors to errors in ulps

Now, let us convert from relative errors to errors in ulps. A relative error

$$\epsilon_r = \left| \frac{x - X}{x} \right| \tag{2.4}$$

implies an error in ulps bounded by

$$|x - X| \le \epsilon_r \beta^p \operatorname{ulp}(x). \tag{2.5}$$

Hence, one can easily switch from an error in ulps to a relative error, and conversely. This is convenient, since for the correctly rounded arithmetic operations and functions, we have an error in ulps, whereas it is generally much easier to deal with relative errors for performing error calculations.

2.6.3 An example: iterated products

A typical example is iterated multiplications. Assume that we compute the product of the floating-point numbers x_1, x_2, ..., x_n in binary, precision-p, rounded to nearest floating-point arithmetic. That is, we perform

$P \leftarrow x_1$
for $i = 2$ to n **do**
    $P \leftarrow \operatorname{RN}(P \times x_i)$
end for
return P

Each multiplication is correctly rounded, leading to an error less than or equal to $0.5 \operatorname{ulp}$. And yet, reasoning in terms of ulps will lead to calculations that are

Figure 2.8: Conversion from ulps to relative errors. Assume we know that an oper-
ation is correctly rounded: the computed result X is within 0.5 ulp from the exact
result x. This implies that $|x - X|$ is below the bold (noncontinuous) curve. Con-
verted in terms of relative errors, this information becomes $X = x(1 + \epsilon)$, with
$|\epsilon| \leq 2^{-p}$, i.e., $|x - X|$ is below the dashed curve. This last property is less accurate.

much too complex. Whereas, if we assume that no underflows occur, a simple reasoning with relative errors shows that the final value of P satisfies

$$P = x_1 x_2 x_3 \ldots x_n \times K,$$

where

$$\left(1 - 2^{-p}\right)^{n-1} \leq K \leq \left(1 + 2^{-p}\right)^{n-1},$$

which means that the relative error of the result is upper-bounded by

$$\left(1 + 2^{-p}\right)^{n-1} - 1,$$

which is close to $(n - 1) \times 2^{-p}$ as long as $n \ll 2^p$.

And yet, one must keep in mind that each time we switch from one form of error to the other one, we lose some information. For instance, a correctly rounded to nearest operation returns a result X within 0.5 ulp from the exact value x. This implies a relative error bounded by 2^{-p}. Figure 2.8 shows, in the interval $[-1/2, 8]$, the bound on distance between x and X one can infer from the information in terms of ulps and the information in terms of relative errors. We immediately see that we have lost some information in the conversion.

When converting from relative errors to errors in ulps, some information is lost too. This is illustrated by Figure 2.9.

Figure 2.9: Conversion from relative errors to ulps. Assume we have a bound on the relative error between an exact value x and a floating-point approximation X (dashed curve). From it, we can infer an error in ulps that implies that X is below the bold curve. This last property is less accurate.

2.6.4   Unit roundoff

A useful notion, closely related to the notion of ulp, is the notion of *unit round-off*, also sometimes called *machine epsilon*:

Definition 6 (Unit roundoff). *The* unit roundoff **u** *of a radix-β, precision-p, floating-point system is defined as*

$$
\mathbf{u} = \begin{cases} \dfrac{1}{2}\,\mathrm{ulp}(1) & = & \dfrac{1}{2}\beta^{1-p} \text{ in round-to-nearest mode,} \\[2ex] \mathrm{ulp}(1) & = & \beta^{1-p} \text{ in directed rounding modes.} \end{cases}
$$

That notion is widespread in the analysis of numerical algorithms. See for instance the excellent book by Higham [182]. For any arithmetic operation $T \in \{+, -, \times, \div\}$, for any rounding mode $\circ \in \{\mathrm{RN}, \mathrm{RD}, \mathrm{RU}, \mathrm{RZ}\}$, and for all floating-point numbers a, b such that aTb does not underflow or overflow, we have

$$
\circ(aTb) = (aTb)(1 + \epsilon_1) = (aTb)/(1 + \epsilon_2),
$$

with $|\epsilon_1|, |\epsilon_2| \leq \mathbf{u}$. This property eases the computation of error bounds [182, 266]. See Section 2.2.3, and Chapter 6.

2.7 Note on Radix Conversion

2.7.1 Conditions on the formats

When the radix of the floating-point system is 2,[16] conversions from and to radix 10 must be provided, since humans read and write numbers in decimal. Early works on radix conversion were done by Goldberg [149] and by Matula [272]. Accurate algorithms for input and output radix conversion can be found in the literature [62, 72, 73, 349, 386], and are now implemented in most compilers. It is important to understand that radix conversion is not a fully innocuous operation.

- Some numbers that have a finite radix-10 representation do not have a finite binary one. A simple example is $0.1_{10} = 1/10$, whose binary representation is

  $$0.000110011001100110011001100110011001100110011001100110011001100110\cdots.$$

- Although all numbers with a finite radix-2 representation also have a finite decimal representation,[17] the number of digits of that decimal representation might sometimes be too large to be convenient. Consider for instance the following floating-point binary number:

  $$0.11111111111111111111111_2 \times 2^{-126} = 2^{-126} - 2^{-149}.$$

  That number is a single-precision number of the IEEE 754-1985 standard (see Chapter 3), or a binary32 number of IEEE 754-2008. Its exact decimal representation

  $$1.17549421069244107548702944484928734882705242874589333385717$$
  $$45305715888704756189042655023513361811637878417\,96875 \times 10^{-38}$$

  is too large to be convenient for all applications. This is the worst case in single precision. If the two extremal exponents $e_{\min}$ and $e_{\max}$ of the binary format satisfy $e_{\min} \approx -e_{\max}$ (which holds for all usual formats) and if p_2 is its precision, then the largest width of a decimal significand we can obtain by exact conversion is[18]

  $$-e_{\min} + p_2 + \left\lfloor (e_{\min} + 1)\log_{10}(2) - \log_{10}\left(1 - 2^{-p_2}\right)\right\rfloor \text{ digits.}$$

---

[16]A very similar study can be done when it is a power of 2.

[17]A necessary and sufficient condition for all numbers representable in radix β with a finite number of digits to be representable in radix γ with a finite number of digits is that β should divide an integer power of γ.

[18]That formula is valid for all *possible* values of p_2 and $e_{\min}$ (provided $e_{\min} \approx -e_{\max}$). And yet, for all *usual* formats, it can be simplified: a simple continued fraction argument (see Section 16.1, page 521) shows that for $p_2 \geq 16$ and $e_{\min} \geq -28000$, it is equal to

$$-e_{\min} + p_2 + \left\lfloor (e_{\min} + 1)\log_{10}(2)\right\rfloor.$$

For instance, for the various basic binary formats of the new IEEE 754-2008 standard (see Table 3.13, page 81), this gives 112 digits for binary32, 767 digits for binary64, and 11563 digits for binary128.

Hence, during radix conversions, numbers must be *rounded*. We assume here that we want to minimize the corresponding rounding errors (i.e., to round numbers to the nearest value in the target format whenever possible).

Other methods should be used when directed rounding modes are at stake, since an experienced user will choose these rounding modes to get sure lower or upper bounds on a numerical value. Therefore, it would be clumsy to carefully design a numerical program so that the finally obtained binary value is a certain lower bound on the exact result, and then to have that binary value rounded up during the radix conversion.

A question that naturally arises is: for a given binary format, which decimal format is preferable if the user does not specify something?

Assuming an internal binary format of precision p_2, the first idea that springs to mind would be to have an input/output decimal format whose precision would be the integer that is nearest to

$$p_2 \frac{\log(2)}{\log(10)}.$$

This would for instance give a decimal precision equal to 16 for the double-precision binary format ($p = 53$).

And yet, this is not the best idea, for the following reason. It is common practice to write a floating-point value in a file, and to read it later, or (equivalently) to re-enter on the keyboard the result of a previous computation. One would like this operation (let us call it a "write-read cycle") to be *error-free*: when converting a binary floating-point number x to the external decimal format, and back-converting the obtained result to binary, one would like to find x again, without any error. Of course, this is always possible by performing an "exact" conversion to decimal, using for the decimal representation of x a large number of digits, but we are going to see that an exact conversion is not required. Furthermore, there is an important psychological point: as pointed out by Steele and White [386], if a system prints too many decimal digits, the excess digits will seem to reflect more information than the number actually contains.

Matula [272] shows the following result.

Theorem 1 (Base conversion). *Assume we convert a radix-β, precision-p floating-point number to the nearest number in a radix-γ, precision-q format, and then convert back the obtained value to the nearest number in the initial radix-β, precision-p format. If there are no positive integers i and j such that $\beta^i = \gamma^j$, then a necessary and sufficient condition for this operation to be the identity (provided no underflow/overflow occurs) is*

$$\gamma^{q-1} > \beta^p.$$

Let us explain Matula's result in the case of a write-read cycle (that is, the values β and γ of Theorem 1 are 2 and 10, respectively). Let p_2 be the precision of the "internal" binary format and p_{10} be the precision of the "external" radix-10 format. We will assume in the following that the conversions will be correctly rounded, in round-to-nearest mode. Hence, our problem is to find conditions on p_{10} to make sure that a write-read cycle is error-free.

Figure 2.10 shows that if p_{10} is not large enough, then after a write-read cycle we may end up with a binary number slightly different from the initial one.

binary floating-point numbers

decimal floating-point numbers

Figure 2.10: In this example, the binary number b_1 will be converted to the decimal number d, and d will be converted to the binary number b_2.

In the neighborhood of the binary floating-point number x to be converted, let us call ϵ_2 the distance between two consecutive binary numbers (that is, $\epsilon_2 = \mathrm{ulp}(x)$), and ϵ_{10} the distance between two consecutive decimal floating-point numbers of the "external" format.

- When x is converted to a decimal floating-point number x', since we assume round-to-nearest mode, this implies that $|x - x'|$ is less than or equal to $\epsilon_{10}/2$.

- When x' is back-converted to a binary floating-point number x'', this implies that $|x' - x''|$ is less than or equal to $\epsilon_2/2$.

To always have $x'' = x$ therefore requires that

$$\epsilon_{10} < \epsilon_2. \tag{2.6}$$

Now, let us see what this constraint means in terms of p_2 and p_{10}. Consider numbers that are between two consecutive powers of 10, say, 10^r and 10^{r+1} (see Figure 2.11). In that domain,

$$\epsilon_{10} = 10^{r-p_{10}+1},$$

also, that domain contains at most four consecutive powers of 2, say, 2^q, 2^{q+1}, 2^{q+2}, and 2^{q+3}, so that the binary ulp ϵ_2 varies from 2^{q-p_2} to 2^{q-p_2+4}. Therefore, condition (2.6) becomes

$$10^r \cdot 10^{-p_{10}+1} < 2^q \cdot 2^{-p_2}. \tag{2.7}$$

Figure 2.11: Various possible values of the (binary) ulp function between two consecutive powers of 10. One can easily show that between two consecutive powers of 10 there are at most four consecutive powers of 2.

Now, since 2^q is larger than 10^r (yet, it can be quite close to 10^r), condition (2.7) will be satisfied if

$$2^{p_2} \le 10^{p_{10}-1}. \tag{2.8}$$

Notice that this condition is equivalent to $2^{p_2} < 10^{p_{10}-1}$, since $2^{p_2} = 10^{p_{10}-1}$ is impossible.

Therefore, the most convenient choice for p_{10} is the smallest integer for which (2.8) holds, namely,

$$p_{10} = 1 + \lceil p_2 \log_{10}(2) \rceil. \tag{2.9}$$

Table 2.3 gives such values p_{10} for various frequently used values of p_2.

2.7.2 Conversion algorithms

Output conversion: from radix 2 to radix 10

The results given in the previous section correspond to *worst cases*. For many binary floating-point numbers x, the number of radix-10 digits that should be

p_2	24	53	64	113
p_{10}	9	17	21	36

Table 2.3: For various values of the precision p_2 of the internal binary format, minimal values of the external decimal precision p_{10} such that a write-read cycle is error-free, when the conversions are correctly rounded to nearest.

used to guarantee error-free write-read cycles will be less than what is given in Table 2.3. Consider for instance the number

$$x = 5033165 \times 2^{-24} = 0.300000011920928955078125.$$

It is exactly representable in the single-precision format of the IEEE 754-1985 standard ($p = 24$). Hence, we know from the study of the previous section and from Table 2.3 that if we convert x to the 9-digit decimal number $x^{(1)} = 0.300000012$, and convert back that decimal number to single-precision binary arithmetic, we will find x again. And yet, once converted to single-precision arithmetic, the 1-digit decimal number $x^{(2)} = 0.3$ also gives x. Hence, in that particular case, an error-free write-read cycle is possible with precision $p_{10} = 1$. One could object that $x^{(1)}$ is as legitimate as $x^{(2)}$ to "represent" x, but there is an important psychological aspect here, that should not be neglected. Someone entering 0.3 on a keyboard (hence, getting x after conversion) will not like to see it displayed as 0.300000012.

This leads to a strategy suggested by Steele and White [385, 386]: when the output format is not specified, use for each binary floating-point number x the *smallest* number of radix-10 significand digits that allows for an error-free write-read cycle. Steele and White designed an algorithm for that. Their algorithm was later improved by Burger and Dybvig [62] , and by Gay [145]. Gay's code is available for anyone to use, and is very robust.[19] In the following, we present Burger and Dybvig's version of the algorithm.

We assume that the internal floating-point system is binary[20] and of precision p. If x is a binary floating-point number, we denote x^- and x^+ its floating-point predecessor and successor, respectively. In the following, we assume that the internal binary number to be converted is positive. The algorithm uses exact rational arithmetic (Burger and Dybvig also give a more complex yet more efficient algorithm that only uses high-precision integer arithmetic and an efficient scale-factor estimator; see [62] for details). The basic principle of Algorithm 2.2 for converting the binary number $x = X \times 2^{e-p+1}$ is quite simple:

- we scale x until it is between $1/10$ and 1, i.e., until it can be written $0.d_1 d_2 d_3 d_4 \cdots$ in decimal;

---

[19]As we are writing this book, it can be obtained at `http://www.netlib.org/fp/` (file `dtoa.c`).

[20]The algorithm works for other radices. See [62] for details.

- the first digit d_1 is obtained by multiplying the scaled value by 10 and taking the integer part. The fractional part is used to compute the subsequent digits in a similar fashion.

Algorithm 2.2 Conversion from radix 2 to radix 10 [62]. The input value is a precision-p binary floating-point number x, and the output value is a decimal number $V = 0.d_1 d_2 \cdots d_n \times 10^k$, where n is the smallest integer such that 1) $(x^- + x)/2 < V < (x + x^+)/2$, i.e., the floating-point number nearest to V is x, regardless of how the input rounding algorithm breaks ties (x^- is the floating-point predecessor of x, and x^+ is its floating-point successor); and 2) $|V - x| \leq 10^{k-n}/2$, i.e., V is correctly rounded in the precision-n output decimal format. Here $\{t\}$ denotes the fractional part of t.

$\ell \leftarrow (x^- + x)/2$
$u \leftarrow (x + x^+)/2$
find the smallest k such that $u \leq 10^k$
$W \leftarrow x/10^{k-1}$
$d_1 \leftarrow \lfloor W \rfloor$
$W \leftarrow \{W\}$
$n \leftarrow 1$
while $0.d_1 d_2 d_3 \cdots d_n \times 10^k \leq \ell$ and $\left(0.d_1 d_2 d_3 \cdots d_n + \frac{1}{10^n}\right) \times 10^k \geq u$ **do**
  $n \leftarrow n + 1$
  $d_n \leftarrow \lfloor 10 \times W \rfloor$
  $W \leftarrow \{10 \times W\}$
end while
if $0.d_1 d_2 d_3 \cdots d_n \times 10^k > \ell$ and $\left(0.d_1 d_2 d_3 \cdots d_n + \frac{1}{10^n}\right) \times 10^k \geq u$ **then**
  return $0.d_1 d_2 d_3 \cdots d_n \times 10^k$
else if $0.d_1 d_2 d_3 \cdots d_n \times 10^k \leq \ell$ and $\left(0.d_1 d_2 d_3 \cdots d_n + \frac{1}{10^n}\right) \times 10^k < u$ **then**
  return $\left(0.d_1 d_2 d_3 \cdots d_n + \frac{1}{10^n}\right) \times 10^k$
else
  return the value closest to x among $0.d_1 d_2 d_3 \cdots d_n \times 10^k$ and $\left(0.d_1 d_2 d_3 \cdots d_n + \frac{1}{10^n}\right) \times 10^k$
end if

Input conversion: from radix 10 to radix 2

The input conversion problem is very different from the previous one, primarily because the input decimal numbers may not have a predefined, bounded size. The number of input digits that need to be examined to decide which is the binary floating-point number nearest to the input decimal number may be arbitrarily large. Consider the following example. Assume that the internal format is the IEEE 754-1985 single-precision format (also called binary32 format in IEEE 754-2008, see Chapter 3), and that the

rounding mode is round to nearest even. If the input number is

$$1.000000005960464477539062500000000000000000000000000000000000$$
$$=\ 1 + 2^{-24},$$

then the conversion algorithm should return 1, whereas if the input number is

$$1.000000005960464477539062500000000000000000000000000000000001$$
$$=\ 1 + 2^{-24} + 10^{-60},$$

the conversion algorithm should return the floating-point successor of 1, namely $1 + 2^{-23}$.

The first efficient and accurate input conversion algorithms were introduced by Rump [349] and Clinger [72, 73]. Later on, Gay suggested improvements [145]. As for output conversion, Gay's code is available for anyone to use, and is very robust.[21] Let us describe Gay's version of Clinger's algorithm.

We assume that the floating-point number system being used is a binary system of precision p. When converting an input decimal number x to binary, the best result we can provide is a correctly rounded result: in that case, the obtained value v is $\circ(x)$, where $\circ$ is the chosen rounding mode. Notice that what the IEEE 754-1985 standard for floating-point arithmetic requires is not that strong.[22] Here, we assume that we want to correctly round to nearest (similar work can be done with the other rounding modes).

The input value d is an n-digit decimal number:

$$\begin{aligned} d\ &=\ 10^k \times [d_0.d_1d_2\cdots d_{n-1}]_{10} \\ &=\ \sum_{i=0}^{n-1} d_i 10^{k-i}. \end{aligned}$$

We want to return a precision-p binary floating-point number $b = \mathrm{RN}(d)$, where RN stands for "round to nearest even" (i.e., we return the floating-point number nearest to d, and if there are two such numbers, we return the one whose integral significand is an even number. See Section 3.1.3, page 61, for explanations). For simplicity, we assume that $d > 0$, and that no underflow or overflow will occur, that is:

$$2^{e_{\min}} \le d \le 2^{e_{\max}} \left(2 - 2^{1-p}\right),$$

---

[21]As we are writing this book, it can be obtained at `http://www.netlib.org/fp/` (file `dtoa.c`).

[22]In round-to-nearest modes, it requires that the error introduced by the conversion should be at most 0.97 ulps (see what this notation means in Section 2.6.1). The major reason for this somewhat weak requirement is that the conversion algorithms presented here were not known at the time the standard was designed.

$$b^- = b - 2^{e-p} \quad b = 2^e \qquad\qquad b^+ = b + 2^{e-p+1}$$

$2^e = \mathrm{RN}(d)$ means that d
lies in this area

Figure 2.12: Illustration of the conditions (2.10) in the case $b = 2^e$.

where $e_{\min}$ and $e_{\max}$ are the extremal exponents of the binary floating-point format. Our problem consists in finding an exponent e and a significand $b_0.b_1 b_2 \cdots b_{p-1}$, with $b_0 \neq 0$, such that the number

$$
\begin{aligned}
b &= 2^e \times [b_0.b_1 b_2 \cdots b_{p-1}]_2 \\
&= \sum_{i=0}^{p-1} b_i 2^{e-i}
\end{aligned}
$$

satisfies

$$
\begin{cases}
\text{if } b = 2^e \text{ exactly} & \text{then } -2^{e-p-1} \leq d - b \leq 2^{e-p} \\
\text{otherwise} & |b - d| \leq 2^{e-p}, \text{ and } |b - d| = 2^{e-p} \Rightarrow b_{p-1} = 0.
\end{cases}
\tag{2.10}
$$

Figure 2.12 helps us to understand these conditions in the case $b = 2^e$.

First, notice that some cases (in practice, those that occur most often!) are very easily handled. Denote

$$D = \frac{d}{10^{k-n+1}},$$

that is, D is the integer whose decimal representation is

$$d_0 d_1 d_2 \cdots d_{n-1}.$$

1. If $10^n \leq 2^p$ and $10^{|k-n+1|} \leq 2^p$, then the integers D and $10^{|k-n+1|}$ are exactly representable in the binary floating-point system. In that case, it suffices to compute D exactly as[23]

$$(\cdots(((d_0 \times 10 + d_1) \times 10 + d_2) \times 10 + d_3)\cdots) \times 10 + d_{n-1},$$

---

[23] Another solution consists in using a precomputed table of powers of 10 in the binary format.

and to compute $K = 10^{|k-n+1|}$ by iterative multiplications. We then get b by performing one floating-point multiplication or division:

$$b = \begin{cases} \text{RN}(D \times K) & \text{if } k - n + 1 \geq 0 \\ \text{RN}(D/K) & \text{otherwise.} \end{cases}$$

2. Even if the above conditions are not satisfied, if there exists an integer j, $1 \leq j < k$, such that the integers 10^{k-j} and $10^{n+j} [d_0.d_1 d_2 \cdots d_{n-1}]_{10}$ are less than or equal to $2^p - 1$, then 10^{k-j} and $10^{n+j} [d_0.d_1 d_2 \cdots d_{n-1}]_{10}$ are exactly representable and easily computed, and their floating-point product is b.

Now, if we are not in these "easy cases," we build a series of "guesses"

$$b^{(1)}, b^{(2)}, b^{(3)} \ldots$$

and stop at the first guess $b^{(m)}$ such that $b^{(m)} = b$. Let us now show how these guesses are built, and how we can check if $b^{(j)} = b$.

1. The first guess is built using standard floating-point arithmetic in the target format.[24] One can find $b^{(1)}$ such that

$$\left| d - b^{(1)} \right| < c \cdot 2^{e_b - p + 1},$$

where c is a small constant and e_b is an integer. Let us give a possible solution to do that. We assume $d_0 \neq 0$. Let j be the smallest integer such that $10^j \geq 2^p$. Define

$$\begin{aligned} D^* &= d_0 d_1 \cdots d_{\min\{n-1,j\}}.d_{\min\{n-1,j\}+1} \cdots d_{n-1} \\ &= \sum_{m=0}^{n-1} d_m 10^{\min\{n-1,j\}-m}. \end{aligned}$$

Also define

$$\hat{D} = \lfloor D^* \rfloor = d_0 d_1 \cdots d_{\min\{n-1,j\}}.$$

If we compute in standard floating-point arithmetic an approximation to $\hat{D}$ using the sequence of operations

$$(\cdots ((d_0 \times 10 + d_1) \times 10 + d_2) \cdots) \times 10 + d_{\min\{n-1,j\}},$$

then all operations except possibly the last multiplication and addition are performed exactly. A simple analysis shows that the computed result, say $\tilde{D}$, satisfies

$$\tilde{D} = \hat{D}(1 + \epsilon_1),$$

---

[24]If a wider internal format is available, one can use it and possibly save one step.

with $|\epsilon_1| \leq 2^{-p+1} + 2^{-2p}$. This gives

$$\tilde{D} = D^*(1 + \epsilon_1)(1 + \epsilon_2),$$

where $|\epsilon_2| \leq 10^{-j} \leq 2^{-p}$.[25] Now, we want to get a binary floating-point approximation to

$$d = D^* \times 10^{k-\min\{n-1,j\}}.$$

To approximate $K^* = 10^{k-\min\{n-1,j\}}$, several solutions are possible. We can assume that the best floating-point approximations to the powers of 10 are precomputed and stored in a table. An alternative solution [145] is to compute K^* on the fly (assuming we have stored the first powers of 10, and powers of the form $10^{(2^i)}$, to save time and accuracy). For simplicity, let us assume here that we get from a table the best floating-point approximation to K^*, i.e., that we get a binary floating-point number $\tilde{K}$ that satisfies

$$\tilde{K} = K^*(1 + \epsilon_3),$$

where $|\epsilon_3| \leq 2^{-p}$. We finally compute

$$b^{(1)} = \text{RN}(\tilde{K}\tilde{D}) = \tilde{K}\tilde{D}(1 + \epsilon_4),$$

with $|\epsilon_4| \leq 2^{-p}$. Therefore, we get

$$
\begin{aligned}
b^{(1)} &= d \times (1 + \epsilon_1)(1 + \epsilon_2)(1 + \epsilon_3)(1 + \epsilon_4) \\
&= d \times (1 + \epsilon),
\end{aligned}
$$

with

$$|\epsilon| \leq 5 \cdot 2^{-p} + 10 \cdot 2^{-2p} + 10 \cdot 2^{-3p} + 5 \cdot 2^{-4p} + 2^{-5p},$$

which gives $|\epsilon| \leq 5.0000006 \times 2^{-p}$ as soon as $p \geq 24$.

From this we deduce

$$|b^{(1)} - d| \leq 5.0000006 \times 2^{e_b - p + 1}.$$

2. Once we have an approximation $b^{(j)}$ of exponent e_j, as said above, for $b^{(j)}$ to be equal to b, it is necessary that

$$|d - b^{(j)}| \leq 2^{e_j - p}. \qquad (2.11)$$

Furthermore, if $b^{(j)} = 2^{e_j}$ exactly, it is also necessary that

$$d - b^{(j)} \geq -\frac{1}{2}2^{e_j - p}. \qquad (2.12)$$

---

[25]A straightforward analysis of the error induced by the truncation of the digit chain D^* would give $|\epsilon_2| \leq 10^{-\min\{n-1,j\}}$, but when $j \geq (n-1)$, $D^* = \hat{D}$ and there is no truncation error at all.

We will focus on condition (2.11) and show how Gay handles it. Define

$$M = \max\left\{1, 2^{p-e_j-1}\right\} \times \max\left\{1, 10^{n-k-1}\right\}.$$

Condition (2.11) is equivalent to

$$|2M(d - b^{(j)})| \leq M \times 2^{e_j-p}, \tag{2.13}$$

but since $2Md$, $2Mb^{(j)}$, and $M \times 2^{e_j-p}$ are integers, condition (2.13) can easily be checked using multiple-precision integer arithmetic.

If

$$|2M(d - b^{(j)})| < M \times 2^{e_j-p},$$

then $b^{(j)}$ is equal to $\text{RN}(d)$.

If

$$|2M(d - b^{(j)})| = M \times 2^{e_j-p},$$

then $\text{RN}(d)$ is $b^{(j)}$ if the integral significand of $b^{(j)}$ is even (i.e., if the last bit of the significand of $b^{(j)}$ is a zero), and the floating-point number adjacent to $b^{(j)}$ in the direction of d otherwise.

If

$$|2M(d - b^{(j)})| > M \times 2^{e_j-p},$$

we must find a closer floating-point approximation, $b^{(j+1)}$, to d.

3. $b^{(j+1)}$ can be built as follows. Define

$$\delta^{(j)} = \frac{(d - b^{(j)})}{2^{e_j-p+1}}.$$

That value will be computed as the ratio of the multiple-precision integers used in the previous test, as

$$\delta^{(j)} = \frac{1}{2} \times \frac{2M \times (d - b^{(j)})}{M \times 2^{e_j-p+1}}.$$

We have $d = b^{(j)} + \delta^{(j)} 2^{e_j-p+1}$: this means that $\delta^{(j)}$ is the number of units in the last place (ulps, see Section 2.6.1) that should be added to $b^{(j)}$ to get d. In most cases, b will be obtained by adding to $b^{(j)}$, in floating-point arithmetic, a floating-point approximation to that correcting term $\delta^{(j)}$. Hence, once floating-point approximations to $M(d - b^{(j)})$ and $M \times 2^{e_j-p+1}$ are computed (from the integers computed for checking $b^{(j)}$), we compute $\tilde{\delta}_j$ as the quotient of these approximations, and we compute

$$b^{(j+1)} = \text{RN}(b^{(j)} + \tilde{\delta}_j 2^{e_j-p+1}).$$

Some care is necessary to avoid loops (if $b^{(j+1)} = b^{(j)}$), see [145] for details on how to handle these cases. Gay [145] shows that the number of steps needed to have $b^{(m)} = b$ is at most 3. In most cases, b is $b^{(1)}$ or $b^{(2)}$. Indeed, the only cases for which $m = 3$ are those for which $|b^{(2)} - b| = 2^{e-p+1}$.

2.8 The Fused Multiply-Add (FMA) Instruction

The FMA instruction was introduced in 1990 on the IBM RS/6000 processor to facilitate correctly rounded software division and to make some calculations (especially dot products and polynomial evaluations) faster and more accurate.

Definition 7 (FMA instruction). *Assume that the rounding mode is $\circ$, and that a, b, and c are floating-point numbers. FMA(a, b, c) is $\circ(a \cdot b + c)$.*

Some algorithms facilitated by the availability of that instruction are presented in Chapter 5. A brief discussion on current implementations is given in Section 3.5.2, page 104.

The new IEEE 754-2008 standard for floating-point arithmetic specifies the FMA instruction.

2.9 Interval Arithmetic

Interval arithmetic [284, 285, 300, 216, 165, 286, 352], in its simplest form, is a means for computing guaranteed enclosures of real-valued expressions. This arithmetic manipulates connected closed subsets of the real numbers and its operations are defined in such a way that they satisfy the *inclusion property*. Given two intervals U and V and a mathematical operation on real numbers $\diamond \in \{+, -, \times, \div, \cdots \}$, the interval result $U \diamond V$ shall satisfy

$$\forall u \in U, \ \forall v \in V, \quad u \diamond v \in U \diamond V.$$

If the expressions u and v are enclosed in the intervals $[\underline{u}, \overline{u}]$ and $[\underline{v}, \overline{v}]$, then the following properties can be deduced from the monotonicity properties of the arithmetic operations on real numbers:

$$
\begin{aligned}
-u &\in [-\overline{u}, -\underline{u}] \\
\sqrt{u} &\in [\sqrt{\underline{u}}, \sqrt{\overline{u}}] \quad \text{if } \underline{u} \geq 0 \\
u^{-1} &\in [\overline{u}^{-1}, \underline{u}^{-1}] \quad \text{if both bounds have the same sign} \\
u + v &\in [\underline{u} + \underline{v}, \overline{u} + \overline{v}] \\
u \times v &\in [\min(\underline{u} \times \underline{v}, \overline{u} \times \underline{v}, \underline{u} \times \overline{v}, \overline{u} \times \overline{v}), \\
&\qquad \max(\underline{u} \times \underline{v}, \overline{u} \times \underline{v}, \underline{u} \times \overline{v}, \overline{u} \times \overline{v})]
\end{aligned}
\tag{2.14}
$$

These properties show that guaranteed enclosures can be obtained easily, by computing on interval bounds.

Note that, while the bounds computed by naive interval arithmetic are guaranteed, they are not necessarily tight. Consider an expression x enclosed in the interval $[0, 1]$. By interval arithmetic, the expression $x - x$ is known to be contained in $[0, 1] - [0, 1] = [0 - 1, 1 - 0] = [-1, 1]$, which is much wider than the tightest enclosure $x - x \in [0, 0]$. This explains why the development of algorithms computing tight interval enclosures is an active research field.

2.9.1   Intervals with floating-point bounds

Formulas (2.14) depend on the arithmetic on real numbers. However, it is possible to design formulas that use floating-point numbers, and still satisfy the inclusion property. Indeed, directed rounding modes in floating-point arithmetic provide an efficient implementation of interval arithmetic. For instance,

$$\sqrt{[\underline{u}, \overline{u}]} \quad := \quad [\mathrm{RD}(\sqrt{\underline{u}}), \mathrm{RU}(\sqrt{\overline{u}})] \quad \text{if } \underline{u} \geq 0$$
$$[\underline{u}, \overline{u}] + [\underline{v}, \overline{v}] \quad := \quad [\mathrm{RD}(\underline{u} + \underline{v}), \mathrm{RU}(\overline{u} + \overline{v})]$$
$$[\underline{u}, \overline{u}] - [\underline{v}, \overline{v}] \quad := \quad [\mathrm{RD}(\underline{u} - \overline{v}), \mathrm{RU}(\overline{u} - \underline{v})]$$

Thanks to the properties of RD and RU (see Section 2.2), if the input intervals have finite bounds, the interval result is guaranteed to contain the exact, mathematical value, even if the operations on the bounds are inexact.

Moreover, the lower bound can be replaced by $-\infty$ in order to represent an interval unbounded on the negative numbers. Similarly, $+\infty$ can be used for the upper bound.[26] Then, thanks to the properties of the IEEE 754 arithmetic with respect to infinite values, the inclusion property is still valid on the final result even when overflows to infinities occur during intermediate computations. Notice that the operations $\infty - \infty$ can never happen in these formulas, as long as the inputs are valid.

Floating-point arithmetic also makes it possible to handle the empty set. For instance, it can be represented by the pair $[\mathrm{NaN}, \mathrm{NaN}]$, which will properly propagate when given to the preceding formulas.

For multiplication, the situation is slightly more complicated. Consider the product of the intervals $[0, 7]$ and $[10, +\infty]$. (The $+\infty$ bound may have been obtained by overflow, if the real bound happens to be too large to be representable by a finite floating-point number.) Assume that, as in the case with real bounds, the lower bound is computed by taking the minimum of the four products $\mathrm{RD}(0 \times 10) = 0$, $\mathrm{RD}(7 \times 10) = 70$, $\mathrm{RD}(0 \times \infty) = \mathrm{NaN}$, and $\mathrm{RD}(7 \times \infty) = +\infty$. A NaN datum is obtained for a product whose result would be zero if there had been no overflow. Therefore, when the underlying floating-point arithmetic is compliant to IEEE 754, some special care is needed to prevent propagating incorrect bounds [179].

2.9.2   Optimized rounding

On some floating-point environments, performing computations with various rounding modes can be much costlier than performing all the floating-point computations with the same rounding mode. In that case, relying on the symmetry property of the rounding modes can be of help. Indeed, the

---

[26]An interval with floating-point bounds $[x, +\infty]$ contains all of the real numbers greater than or equal to x. In the basic interval model, intervals are just sets of reals; infinite bounds are not part of them.

identity $\forall x \; \mathrm{RD}(-x) = -\,\mathrm{RU}(x)$ makes it possible to use one single rounding direction for most of the arithmetic operations.

For instance, let us assume that interval operations should only use RU. Addition and subtraction can be rewritten as

$$[\underline{u}, \overline{u}] + [\underline{v}, \overline{v}] \quad := \quad [-\,\mathrm{RU}((-\underline{u}) - \underline{v}), \mathrm{RU}(\overline{u} + \overline{v})]$$
$$[\underline{u}, \overline{u}] - [\underline{v}, \overline{v}] \quad := \quad [-\,\mathrm{RU}(\overline{v} - \underline{u}), \mathrm{RU}(\overline{u} - \underline{v})]$$

The computation of the upper bound is left unchanged, but the lower bound now requires some (cheap) sign flipping. In order to avoid these extra operations, we can store the lower bound with its sign bit already flipped. Let us denote X^* an interval stored using this convention; addition and subtraction then become

$$[\underline{u}, \overline{u}]^* + [\underline{v}, \overline{v}]^* \quad := \quad [\mathrm{RU}(\underline{u} + \underline{v}), \mathrm{RU}(\overline{u} + \overline{v})]^*$$
$$[\underline{u}, \overline{u}]^* - [\underline{v}, \overline{v}]^* \quad := \quad [\mathrm{RU}(\underline{u} + \overline{v}), \mathrm{RU}(\overline{u} + \underline{v})]^*$$

Notice that, if intervals are now considered as length-2 floating-point vectors, interval addition is just a vector addition with rounding toward $+\infty$. Interval subtraction is also a vector addition, but the components of the second interval have to be exchanged first. Similar optimizations can be applied to other arithmetic operations; they lead to efficient implementations of floating-point interval arithmetic on SIMD architectures [237, 154].

Some floating-point environments may also have issues with subnormal results (see Section 2.1). The hardware may not support them directly; gradual underflow is then handled either by microcode or by software trap, which may incur some slowdowns. Abrupt underflow alleviates this issue at the expense of some properties mandated by IEEE 754. As long as this abrupt underflow is not implemented by a flush to zero,[27] the inclusion property is still satisfied. Therefore, even though abrupt underflow may cause the enclosures to be a bit wider, they are still guaranteed.

---

[27]That is, a positive subnormal result is rounded to the smallest positive normal floating-point when rounding toward $+\infty$, and to zero when rounding toward $-\infty$.

Chapter 3

Floating-Point Formats and Environment

O UR MAIN FOCUS IN THIS CHAPTER is the IEEE[1] 754-1985 Standard for Floating-Point Arithmetic [10], and its recent revision [187]. A paper written in 1981 by Kahan, *Why Do We Need a Floating-Point Standard?* [202], depicts the rather messy situation of floating-point arithmetic before the 1980s. Anybody who estimates that the current standards are too constraining and that circuit and system manufacturers could build much more efficient machines without them should read that paper and think about it. Even if there were at that time a few reasonably good environments, the various systems available then were so different that writing portable yet reasonably efficient numerical software was extremely difficult.

The IEEE 754-1985 Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Std 754-1985) was released in 1985, but the first meetings of the working group started more than eight years before [207]. William Kahan, a professor at the University of California at Berkeley, played a leading role in the development of the standard. We encourage the reader to look at Kahan's *Lecture Notes on the Status of IEEE-754* [205].

IEEE 754-1985 drastically changed the world of numerical computing. Two years later, another standard, the IEEE 854-1987 Standard for "Radix-Independent" (in fact, radix 2 or 10) Floating-Point Arithmetic was released. It generalized to radix 10 the main ideas of IEEE 754-1985. IEEE 754-1985 is also known as IEC 60559:1989 (or IEC 559), *Binary floating-point arithmetic for microprocessor systems* [188].

Some languages, such as Java and ECMAScript, are based on IEEE 754-1985. The ISO C99 standard (released in 1999) for the C language has optional support for IEEE 754-1985 in its normative annex F. Details will be given in Chapter 7.

---

[1] IEEE is an acronym for the Institute of Electrical and Electronics Engineers. For more details, see `http://www.ieee.org/web/aboutus/home/index.html`.

IEEE 754-1985 had been under revision since 2000. The working group recommended a draft to the IEEE Microprocessor Standards Committee in September 2006. After some tuning of the draft, the new standard was adopted in June 2008. In the following, it will be called IEEE 754-2008. In the literature, IEEE 754-1985 and its new revision are frequently called IEEE 754 and IEEE 754-R, respectively.

The description of the IEEE standards given in this chapter is not exhaustive: the standards are big documents that contain many details. Anyone who wants to implement a floating-point arithmetic function compliant to IEEE 754-2008 must carefully read that standard.

3.1   The IEEE 754-1985 Standard

3.1.1   Formats specified by IEEE 754-1985

The IEEE 754-1985 standard specifies *binary* floating-point arithmetic only: in this section, the radix β will always be equal to 2.

As explained in Chapter 2, in radix 2, the first, leftmost bit of the significand of a finite, nonzero floating-point number is always a "1" if it is a normal number, and a "0" if it is a subnormal number. Hence, provided we have a special encoding that tells us if a number is normal or subnormal, there is no need to store the first bit of its significand. This *hidden bit convention* is required for most formats specified by the IEEE 754-1985 standard, and what is actually stored is the *trailing significand*, also called *fraction*, namely the least $p - 1$ significant bits of the significand.

The standard defines two *basic formats*: *single precision* and *double precision*. The availability of single precision is mandatory. To each basic format is associated an *extended format*. Table 3.1 gives the main parameters of the formats specified by the IEEE 754-1985 standard. The major motivation for the extended formats is that, when implementing some function, they could be used to carry out intermediate computations in order to return a final result in the associated basic formats:

- the wider precision makes it possible to get a result that will almost always be significantly more accurate than that obtained with the basic formats only;

- and the wider range will drastically limit the occurrences of "apparent under/overflow" (that is, cases where there is an underflow or overflow in an intermediate result, whereas the final value would have been in the range of the basic format).

The standard recommends an extended format for the widest basic format supported only. Hence, in practice, the single-extended precision is not

implemented: when double precision is available, it fulfills all the purposes of a single-extended format.

Format	Hidden bit?	p	$e_{\min}$	$e_{\max}$
Single precision	yes	24	-126	127
Double precision	yes	53	-1022	1023
Single-extended	optional	≥ 32	≤ -1022	≥ 1023
Double-extended	optional	≥ 64	≤ -16382	≥ 16383
Double-extended (IA32)	no	64	-16382	16383

Table 3.1: Main parameters of the formats specified by the IEEE 754-1985 standard [10] (©IEEE, 1985, with permission). The single-extended format is not implemented in practice. The last line describes the double-extended format introduced by Intel in the 387 FPU, and available in subsequent IA32 compatible processors by Intel, Cyrix, AMD and others.

Table 3.2 gives the widths of the various fields (whole representation, significand, exponent) of these formats. The ordering of bits in the encodings is as follows. The most significant bit is the sign (0 for positive values, 1 for negative ones), followed by the exponent (represented as explained below), followed by the significand (with the hidden bit convention for the single- and double-precision formats: what is actually stored is the trailing significand). This ordering allows one to compare floating-point numbers as if they were sign-magnitude integers.

Format	word size	sign	exponent	significand	exponent bias b
Single precision	32	1	8	23	127
Double precision	64	1	11	52	1023
Double-extended (IA32)	80	1	15	64	16383

Table 3.2: Sizes of the various fields in the formats specified by the IEEE 754-1985 standard, and values of the exponent bias. Note that for the single- and double-precision formats, the size of the significand field is equal to $p-1$, where p is the precision. This is due to the hidden bit convention.

The exponents are represented using a *bias*. Assume the exponent is stored with W_E bits, and regard these bits as the binary representation of

an unsigned integer N_e. Unless $N_e = 0$ (which corresponds to *subnormal* numbers and the two signed zeros, see below), the (real) exponent of the floating-point representation is $N_e - b$, where $b = 2^{W_E - 1} - 1$ is the *bias*. The value of that bias b is given in Table 3.2. N_e is called the *biased exponent*. This means (see Tables 3.1 and 3.2) that all actual exponents from $e_{\min}$ to $e_{\max}$ are represented by N_e between 1 and $2^{W_E} - 2 = 1111 \cdots 110_2$. With W_E bits, one could represent integers from 0 to $2^{W_E} - 1 = 1111 \cdots 111_2$. The two extremal values 0 and $2^{W_E} - 1$, not needed for representing normal numbers, are used as follows.

- The extremal value 0 is reserved for subnormal numbers and ± 0 (the motivation for subnormal numbers and signed zeros was discussed in Section 2.1, pages 15 and 19 respectively). The bit encoding for a zero is the appropriate sign (0 for $+0$ and 1 for -0), followed by a string of zeros in the exponent field as well as in the significand field.

- The extremal value $2^{W_E} - 1$ is reserved for infinities and NaNs:[2]

  - The bit encoding for infinities is the appropriate sign, followed by $N_e = 2^{W_E} - 1$ (i.e., a string of ones) in the exponent field, followed by a string of zeros in the significand field.

  - The bit encoding for NaNs is an arbitrary sign, followed by $2^{W_E} - 1$ (i.e., a string of ones) in the exponent field, followed by any bit string different from $000 \cdots 00$ in the significand field. Hence, there are several possible encodings for NaNs. This allows the implementer to distinguish between *quiet* and *signaling* NaNs (see Section 3.1.6 for an explanation of the difference between these two kinds of NaNs) and to put possible diagnosis information in the significand field. In IEEE 754-2008, that information is called the *payload* of the NaN.

That choice of using biased representations for the exponents makes it possible to represent positive as well as negative exponents. Other solutions would have been possible, e.g., to represent exponents using two's complement or sign-magnitude representations [224, 126], but this would have made comparison of floating-point numbers slightly harder. Also, it has a nice property that is useful for implementers: One obtains the floating-point successor of a floating-point number by considering its binary representation as the binary representation of an integer, and adding one to that integer (see Section 8.2.1, page 241). From another point of view, positive floating-point numbers (including $+0$ and $+\infty$) are ordered like their binary representation, the latter considered as an integer.

Table 3.3 gives examples of the binary encoding of various floating-point values in single precision. Let us now detail two examples.

---

[2]NaN means *Not a Number*. See Section 2.3 page 25 and Section 3.1.5.

Datum	Sign	Biased exponent	Trailing significand
-0	1	00000000	00000000000000000000000
$+0$	0	00000000	00000000000000000000000
$-\infty$	1	11111111	00000000000000000000000
$+\infty$	0	11111111	00000000000000000000000
NaN	0	11111111	nonzero string
5	0	10000001	01000000000000000000000

Table 3.3: Binary encoding of various floating-point data in single precision.

Example 1 (Binary encoding of a normal number)**.** *Consider the single-precision number x whose binary encoding is*

<div align="center">

sign exponent trailing significand

0	01101011	01010101010101010101010

</div>

- *the bit sign of x is a zero, which indicates that $x \geq 0$;*

- *the biased exponent is neither* 00000000 *nor* 11111111, *which indicates that x is a normal number. It is $01101011_2 = 107_{10}$, hence, since the bias in single precision is 127, the real exponent of x is $107 - 127 = -20$;*

- *by placing the hidden bit (which is a 1, since x is not subnormal) at the left of the trailing significand, we get the significand of x:*

$$1.01010101010101010101010_2 = \frac{5592405}{4194304};$$

- *hence, x is equal to*

$$
\begin{aligned}
\frac{5592405}{4194304} \times 2^{-20} &= \frac{5592405}{4398046511104} \\
&= 0.00000127156567941710818558931350708007 8125.
\end{aligned}
$$

Example 2 (Binary encoding of a subnormal number)**.** *Consider the single-precision number x whose binary encoding is*

<div align="center">

sign exponent trailing significand

1	00000000	01100000000000000000000

</div>

- *the bit sign of x is a one, which indicates that $x \leq 0$;*

- *the biased exponent is* 00000000, *which indicates that x is a subnormal number. It is not a zero, since the significand field is not a string of zeros. Hence, the real exponent of x is $e_{\min} = -126$;*

- *by placing the hidden bit (which is a 0, since x is subnormal) at the left of the trailing significand, we get the significand of x:*

$$0.0110000000000000000000000_2 = \frac{3}{8};$$

- *hence, x is equal to*

$$-\frac{3}{8} \times 2^{-126} = -\frac{3}{680564733841876926926749214863536422912}$$

$$= -4.4081038155835781548827620145834212918199958378953282056578188985440647229552268981933 59375 \times 10^{-39}.$$

Biased exponent N_e	Trailing significand $t_1 t_2 \ldots t_{p-1}$	Value represented
$111 \cdots 1$	$\neq 000 \cdots 0$	NaN
$111 \cdots 1$	$000 \cdots 0$	$(-1)^s \times \infty$
$000 \cdots 0$	$000 \cdots 0$	$(-1)^s \times 0$
$000 \cdots 0$	$\neq 000 \cdots 0$	$(-1)^s \times 0.t_1 t_2 \ldots t_{p-1} \times 2^{e_{\min}}$
$0 < N_e < 2^{W_E} - 1$	any	$(-1)^s \times 1.t_1 t_2 \ldots t_{p-1} \times 2^{N_e - b}$

Table 3.4: How to interpret the binary encoding (sign s, biased exponent, trailing significand) of an IEEE 754-1985 floating-point number [10]. In single precision, $e_{\min} = -126$, $W_E = 8$, and $b = 127$, and in double precision, $e_{\min} = -1022$, $W_E = 11$, and $b = 1023$.

Table 3.4 sums up the way floating-point data are encoded in the IEEE 754-1985 standard, and Table 3.5 presents some extremal values (smallest subnormal, smallest normal, largest finite) in the various formats of the standard.

3.1.2 Little-endian, big-endian

The IEEE 754-1985 standard specifies how floating-point data are encoded, but only as a sequence of bits. How such a sequence of bits is ordered in the memory depends on the platform. In general, the bits are grouped into bytes,

Format	Smallest subnormal $2^{e_{\min}+1-p}$	Smallest normal $2^{e_{\min}}$	Largest finite $2^{e_{\max}}(2-2^{1-p})$
single precision	$2^{-126-23}$ $\approx 1.401 \times 10^{-45}$	2^{-126} $\approx 1.175 \times 10^{-38}$	$(2-2^{-23}) \times 2^{127}$ $\approx 3.403 \times 10^{38}$
double precision	$2^{-1022-52}$ $\approx 4.941 \times 10^{-324}$	2^{-1022} $\approx 2.225 \times 10^{-308}$	$(2-2^{-52}) \times 2^{1023}$ $\approx 1.798 \times 10^{308}$
IA32 double extended	$2^{-16382-63}$ $\approx 3.645 \times 10^{-4951}$	2^{-16382} $\approx 3.362 \times 10^{-4932}$	$(2-2^{-63}) \times 2^{16383}$ $\approx 1.190 \times 10^{4932}$

Table 3.5: Extremal values in the IEEE 754-1985 standard.

and these bytes are ordered according to what is called the *endianness*[3] of the platform.

For instance, the double-precision number that is closest to $-7.0868766365730135 \times 10^{-268}$ is encoded by the sequence of bytes 11 22 33 44 55 66 77 88 in memory (from the lowest address to the highest one) on x86 and Linux/IA-64 platforms (they are said to be *little-endian*) and by 88 77 66 55 44 33 22 11 on most PowerPC platforms (they are said to be *big-endian*). Some architectures, such as IA-64, ARM, and PowerPC are *bi-endian*, i.e., they may be either little-endian or big-endian depending on their configuration.

There exists an exception: some ARM-based platforms. ARM processors have traditionally used the *floating-point accelerator* (FPA) architecture, where the double-precision numbers are decomposed into two 32-bit words in the big-endian order and stored according to the endianness of the machine, i.e., little-endian in general, which means that the above number is encoded by the sequence 55 66 77 88 11 22 33 44. ARM has recently introduced a new architecture for floating-point arithmetic: *vector floating-point* (VFP), where the words are stored in the processor's native byte order.

3.1.3 Rounding modes specified by IEEE 754-1985

The IEEE 754-1985 standard defines four rounding modes. Basically, they are the same as those described in Chapter 2, Section 2.2, page 20: round toward $-\infty$ (RD), round toward $+\infty$ (RU), round toward zero (RZ), and round to nearest (RN). And yet, for the round-to-nearest mode, two special rules are worth mentioning: the way numbers larger than the largest finite floating-point number are handled, and the way numbers exactly halfway

---

[3]According to Wikipedia [432], endianness is the convention that two parties that wish to exchange information will use to send and receive this information when they need to cut the information down to pieces. The term *big-endian* comes from Jonathan Swift's book *Gulliver's Travels*.

between two consecutive floating-point numbers are rounded. More precisely, in round-to-nearest mode:

- a number of absolute value larger than or equal to $2^{e_{\max}}(2 - 2^{-p})$ will be rounded to infinity (with the appropriate sign). This of course is not what one would infer from a naive understanding of the words *round to nearest*, but the advantage is clear: when the result of an arithmetic operation is a normal number (including the largest one, $\Omega = 2^{e_{\max}}(2 - 2^{1-p})$), we know that the relative error induced by that operation is small. If huge numbers were rounded to the floating-point value that is really closest to them (namely, $\pm\Omega$), we would have no bound on the relative error induced by an arithmetic operation whose result is $\pm\Omega$;

- other numbers will be rounded to the nearest floating-point number of the format under consideration. In case of a tie (that is, when the exact result is exactly halfway between two consecutive floating-point numbers), the floating-point value whose last significand bit is a zero will be returned. Because of this, that rounding mode is frequently called *round to nearest even*.

The three *directed* rounding modes (toward $+\infty$, toward $-\infty$, and toward 0) behave as described in Section 2.2.

The special rule for round to nearest in case of a tie has several advantages:

- it is rather easily implementable;

- it has no statistical bias;

- Knuth ([222], Theorem D page 237) shows that using round to nearest even, we always have

$$\mathrm{RN}(\mathrm{RN}(\mathrm{RN}(\mathrm{RN}(a + b) - b) + b) - b) = \mathrm{RN}(\mathrm{RN}(a + b) - b),$$

which means that there is no "drift" when repeatedly adding and subtracting the same value.

3.1.4   Operations specified by IEEE 754-1985

Arithmetic operations and square root

The IEEE 754-1985 standard requires that addition, subtraction, multiplication, and division of operands of the same format be provided, for all supported formats, with correct rounding (with the four rounding modes presented above). It is also recommended that these operations be provided (still with correct rounding) for operands of different formats (in such a

case, the destination format must be at least as wide as the wider operand's format). Notice that when the sum or difference of two numbers is exactly zero, then the returned result is zero, with a "+" sign in the round-to-nearest, round-toward-zero, and round-toward $+\infty$ modes, and with a "$-$" in the round-toward $-\infty$ mode, except for $x + x$ and $x - (-x)$ with x being ± 0, in which case the result has the same sign as x.

The standard also requires a correctly rounded square root in all supported formats. The result is defined and has a positive sign for all input values greater than or equal to zero, with the exception[4] that $\sqrt{-0} = -0$.

Remainders

Remainders must also be provided. There are several different definitions of remainders [42]; here is the one chosen for the standard. If x is a finite floating-point number and y is a finite, nonzero floating-point number, then the remainder $r = x$ REM y is defined as

1. $r = x - y \times n$, where n is the integer nearest to the exact value x/y;

2. if x/y is an odd multiple of $1/2$ (i.e., there are two integers nearest to x/y), then n is even;

3. if $r = 0$, its sign is that of x.

A consequence of this definition is that remainders are always exactly representable, which implies that the returned result does not depend on the rounding mode.

Conversions to and from integer formats

It must be possible to convert between all supported integer formats and all supported floating-point formats. Conversion from floating-point formats to integers must be correctly rounded, and must follow the active rounding mode.

Conversions to and from decimal strings

Conversions to and from decimal strings are used very often; for instance, for reading numbers from a file, writing them in a file, or displaying them on screen. We have discussed some issues linked with conversions in Section 2.7. Note that at the time the IEEE 754-1985 standard was released, some of the

---

[4]This rule (that may help to implement complex functions [204]) may seem strange, but the most important point is that any sequence of exact computations on real numbers will give the correct result, even when $\sqrt{-0}$ is involved. Also let us recall that -0 is regarded as a null value, not as a negative number.

algorithms presented in Section 2.7 were not known.[5] This explains why the requirements of the standard might appear somehow below what one could now expect.

The requirements of the standard are:

- conversions must be provided between decimal strings in at least one format and binary floating-point numbers in all basic floating-point formats, for numbers of the form

$$\pm M_{10} \times 10^{\pm E_{10}}$$

  with $E_{10} \geq 0$. On input, trailing zeros are appended to or stripped from M_{10} up to the limits specified in Table 3.6 in order to minimize E_{10};

- conversions must be correctly rounded for operands in the ranges specified in Table 3.7;

- when the operands are not in the ranges specified in Table 3.7:

    - in round-to-nearest mode, the conversion error cannot exceed $0.97\,\mathrm{ulp}$ of the target format;

    - in the directed rounding modes, the "direction" of the rounding must be followed (e.g., for round-toward $-\infty$ mode, the delivered result must be less than or equal to the initial value), and the rounding error cannot exceed $1.47\,\mathrm{ulp}$ of the target format;

- conversions must be *monotonic* (if $x \leq y$ before conversion, then $x \leq y$ after conversion);

- when rounding to nearest, as long as the decimal strings have at least 9 digits for single precision and 17 digits for double precision, conversion from binary to decimal and back to binary must produce the initial value exactly. This allows one to store intermediate results in files, and to read them later on, without losing any information, as explained in Chapter 2, Section 2.7.

Comparisons

It must be possible to compare two floating-point numbers, in all formats specified by the IEEE 754-1985 standard, even if their formats differ. This can be done either by means of a condition code identifying one of the four following conditions: *less than, equal, greater than*, and *unordered*; or as a Boolean

---

[5]We should mention that, at that time, Rump had already suggested algorithms for correctly rounded conversion [349].

	Decimal to binary		Binary to decimal	
	$M_{10,\max}^{(1)}$	$E_{10,\max}^{(1)}$	$M_{10,\max}^{(2)}$	$E_{10,\max}^{(2)}$
Single precision	$10^9 - 1$	99	$10^9 - 1$	53
Double precision	$10^{17} - 1$	999	$10^{17} - 1$	340

Table 3.6: The thresholds for conversion from and to a decimal string, as specified by the IEEE 754-1985 standard [10] (©IEEE, 1985, with permission).

	Decimal to binary		Binary to decimal	
	$M_{10,\max}^{(1)}$	$E_{10,\mathrm{corr}}^{(1)}$	$M_{10,\max}^{(2)}$	$E_{10,\mathrm{corr}}^{(2)}$
Single precision	$10^9 - 1$	13	$10^9 - 1$	13
Double precision	$10^{17} - 1$	27	$10^{17} - 1$	27

Table 3.7: Correctly rounded decimal conversion range, as specified by the IEEE 754-1985 standard [10] (©IEEE, 1985, with permission).

response to a predicate that gives the desired comparison. The *unordered* condition arises when at least one of its operands is a NaN: a NaN compares unordered with everything *including itself*. A consequence of this is that the test

$$x \neq x$$

returns **true** when x is a NaN. As pointed out by Kahan [205], this provides a way of checking if a floating-point datum is a NaN in languages that lack an instruction for doing that (assuming the test is not optimized out). The other tests involving a NaN will return **false**. Hence, the test

$$x \leq y$$

is not always equivalent to the test

$$\mathrm{not}(x > y).$$

If at least one of the two operands is a NaN, the first test will return *false* whereas the second one will return *true*.

Also, the test $+0 = -0$ must return *true*.

Users and especially compiler designers should be aware of these subtleties.

When implementations provide predicates, it is requested that the first six predicates of Table 3.8 (namely, $=, \neq, >, <, \geq, \leq$) be provided, and it is recommended that the seventh one (namely, *unordered*) be provided.

Predicates	Relations				Invalid if unordered ?
	greater than	less than	equal	unordered	
$=$	false	false	true	false	no
$\neq$	true	true	false	true	no
$>$	true	false	false	false	yes
$\geq$	true	false	true	false	yes
$<$	false	true	false	false	yes
$\leq$	false	true	true	false	yes
unordered	false	false	false	true	no

Table 3.8: Comparison predicates and the four relations [10] (©IEEE, 1985, with permission).

3.1.5   Exceptions specified by IEEE 754-1985

The five exceptions listed in Section 2.3 (invalid, division by zero, overflow, underflow, inexact) must be signaled when detected. This can be done by taking a *trap* (see below) or by setting a *status flag*. The default mode is *not* to use traps.

For each type of exception, a status flag must be provided: that status flag is set each time the corresponding exception occurs and no corresponding trap occurs. The status flags are "sticky," so that the user does not need to check them immediately, but after some sequence of operations, such as at the end of a function. A system that is compliant with the standard must provide the user with ways of resetting, testing, and altering the flags individually. The standard also recommends (yet does not request) that the user should be able to save and restore all the flags simultaneously.

Traps

The IEEE 754-1985 standard allows the user to choose what should be done when one of the five exceptions occurs by specifying a *trap handler* for that exception. He can choose to disable, save, or restore an existing trap.

- When a trap is disabled, the corresponding exception is handled according to the default mode.

- When an exception is signaled and the corresponding trap handler is enabled, the trap handler is activated. In some cases (see below), a result is delivered to the trap handler.

Now, we discuss the various cases that lead to an exception in the IEEE standard.

Invalid operation

The invalid operation exception is signaled:

- when one of the operands is a signaling NaN;

- when performing one of the following additions/subtractions: $(-\infty) - (-\infty)$, $(+\infty) - (+\infty)$, $(-\infty) + (+\infty)$, $(+\infty) + (-\infty)$;

- when performing multiplications of the form $(\pm 0) \times (\pm \infty)$;

- when performing divisions of the form $(\pm 0)/(\pm 0)$ or $(\pm \infty)/(\pm \infty)$;

- when computing remainder(x, y), where $y = \pm 0$ or $x = \pm \infty$;

- when computing $\sqrt{x}$ with $x < 0$;

- when converting a floating-point number to an integer or a decimal format when there is no satisfactory way of representing it in the target format. This can happen in case of overflow, or for converting infinity or NaN if the target format does not have representations for such data;

- when performing comparisons of unordered operands using predicates that are listed as invalid if unordered in Table 3.8.

If the exception occurs without a trap, the returned result will be a quiet NaN.

Division by zero

When computing x/y, if x is a nonzero finite number and y is zero, the division by zero exception is signaled. If no trap occurs, the result is infinity, with the correct sign.

Overflow

Let us call an *intermediate* result what would have been the rounded result if the exponent range were unbounded. The overflow exception is signaled when the absolute value of the intermediate result is strictly larger than the largest finite number,

$$\Omega = (2 - 2^{1-p}) \times 2^{e_{\max}},$$

or, equivalently, when the exponent of the intermediate result is strictly larger than $e_{\max}$.

When there is an overflow and no trap occurs, the returned result depends on the rounding mode:

- it will be $\pm \infty$ with the round-to-nearest mode, with the sign of the intermediate result;

- it will be $\pm\Omega$ with the round-toward-zero mode, with the sign of the intermediate result;

- it will be $+\Omega$ for a positive intermediate result and $-\infty$ for a negative one with the round-toward $-\infty$ mode;

- it will be $-\Omega$ for a negative intermediate result and $+\infty$ for a positive one with the round-toward $+\infty$ mode.

For instance, in round-to-nearest mode, there is an overflow when the absolute value of the exact result of an operation is larger than or equal to

$$\left(2 - 2^{-p}\right) \times 2^{e_{\max}} = \Omega + \frac{1}{2}\,\mathrm{ulp}(\Omega).$$

Let us now present what is done if a trap occurs.

If m is the width of the exponent field of the destination format, define

$$K = 2^{3 \times 2^{m-2}}.$$

For instance, K equals 2^{192} in single-precision, 2^{1536} in double-precision, and 2^{24576} in IA32 double-extended precision formats. In case of a trapped overflow, the result that must be delivered to the trap handler is what we would obtain by first dividing the exact result by K, and then rounding it according to the active rounding mode.

For instance, this mechanism allows us to evaluate an expression of the form $\sqrt{x^2 + y^2 + z^2}$ using the straightforward algorithm, without worrying much about possible overflows. If an overflow occurs, the trap handler will be able to perform scaled arithmetic.

The value suggested for that number K may seem strange. As pointed out by the standard [10], that scaling factor is chosen so that we obtain values around the middle of the exponent range, to limit the risk of having further exceptions.

Underflow

When a nonzero result of absolute value less than $2^{e_{\min}}$ is obtained (i.e., it is in the subnormal range), a significant loss of accuracy may occur. And yet, sometimes, such a result is exact (this is a frequent case: see Theorem 3, page 124). To warn the user when an inaccurate very small result is computed, the standard defines two events: *tininess* (a nonzero result of absolute value less than $2^{e_{\min}}$ is obtained), and loss of accuracy.

Concerning the detection of tininess, there is some ambiguity in the standard[6] (see the *Note on underflow*, in Section 2.1, page 18):

---

[6]And unfortunately, this ambiguity remains in the revised standard, see Section 3.4.10.

- tininess can be signaled either *before rounding*, that is, when the absolute value of the exact result is nonzero and strictly less than $2^{e\min}$;

- or it can be signaled *after rounding*, that is, when the absolute value of the nonzero result rounded as if the exponent range were unbounded is strictly less than $2^{e\min}$.

Also, loss of accuracy may be detected either when the result differs from what would have been obtained were exponent range unbounded, or when it differs from what would have been obtained were exponent range and precision unbounded.

If an underflow trap is not implemented or is not enabled (which is the default), the result is always correctly rounded and underflow is signaled only when both tininess and loss of accuracy have been detected.

When a trap has been implemented and is enabled, underflow is signaled when tininess is detected.

Very much like the overflow case, in case of a trapped underflow, the result that must be delivered to the trap handler is what we would obtain by first multiplying the exact result by K, and then rounding it according to the active rounding mode, where K is the same scaling factor

$$2^{3 \times 2^{m-2}}$$

as for overflow.

Inexact

If the result of an operation (after rounding) is not exact, or if it overflows without an overflow trap, then the inexact exception is signaled. The correctly rounded or overflowed result is returned (to the destination or to the trap handler, depending on whether an inexact trap in enabled or not).

3.1.6 Special values

NaN: Not a Number

The standard defines two types of NaNs:

- *signaling NaNs* (sNaNs) do not appear, in default mode, as the result of arithmetic operations. They signal the invalid operation exception whenever they appear as operands. For instance, they can be used for uninitialized variables;

- *quiet NaNs* (qNaNs) propagate through almost all operations without signaling exceptions. They can be used for debugging and diagnostic purposes. As stated above, a quiet NaN is returned whenever an invalid operation exception occurs with the corresponding trap disabled.

For example, qNaN×8 = qNaN, and when the trap for invalid operations has not been enabled, sNaN+5 = qNaN and $\sqrt{-2}$ = qNaN.

Arithmetic of infinities and zeros

The arithmetic of infinities and zeros follows the intuitive rules. For instance, $-1/(-0) = +\infty$, $-5/(+\infty) = -0$, $\sqrt{+\infty} = +\infty$ (the only somewhat counter intuitive property is $\sqrt{-0} = -0$). This very frequently allows one to get sensible results even when an underflow or an overflow has occurred. And yet, one should be cautious. Consider for instance, in round-to-nearest mode, the computation of

$$f(x) = \frac{x}{\sqrt{1 + x^2}},$$

for $\sqrt{\Omega} < x \leq \Omega$, where Ω is the largest finite floating-point number. The computation of x^2 will return an infinite result; hence, the computed value of $\sqrt{1 + x^2}$ will be $+\infty$. Since x is finite, by dividing it by an infinite value we will get $+0$. Therefore, the computed value of $f(x)$, for x large enough, will be $+0$, whereas the exact value of $f(x)$ is extremely close to 1.

3.2 The IEEE 854-1987 Standard

The IEEE 854-1987 standard [11] covers "radix-independent" floating-point arithmetic. This does not mean that all possible radices are considered: actually, that standard only focuses on radices 2 and 10. We will just present it briefly (it is now superseded by IEEE 754-2008 [187]).

Unlike IEEE 754-1985, the IEEE 854-1987 standard does not fully specify formats or internal encodings. It merely expresses constraints between the parameters β, $e_{\min}$, $e_{\max}$, and p of the various precisions provided by an implementation. It also says that for each available precision we must have two infinities, at least one signaling NaN and at least one quiet NaN (as in the IEEE 754-1985 standard). In the remainder of this section, β is equal to 2 or 10. The same radix must be used for all available precisions: an arithmetic system is either binary or decimal, but it cannot mix up the two kinds of representations.

3.2.1 Constraints internal to a format

A balance must be found between the precision p and the value of the extremal exponents $e_{\min}$ and $e_{\max}$. If p is too large compared to $|e_{\min}|$ and $e_{\max}$, then underflows or overflows may occur too often. Also, there must be some balance between $e_{\min}$ and $e_{\max}$: to avoid underflows or overflows when computing reciprocals of normalized floating-point numbers as much as possible, one might want $e_{\min} \approx -e_{\max}$. Since underflow (more precisely, *gradual underflow*, with subnormal numbers available) is less harmful than

overflow, it is preferable to have $e_{\min}$ very slightly above[7] $-e_{\max}$. Here are the constraints specified by the IEEE 854-1987 standard.

- We *must* have

$$\frac{e_{\max} - e_{\min}}{p} > 5,$$

  and it is *recommended* that

$$\frac{e_{\max} - e_{\min}}{p} > 10.$$

- We must have $\beta^{p-1} \geq 10^5$.

- $\beta^{e_{\max}+e_{\min}+1}$ should be the smallest power of β greater than or equal to 4 (which is a very complicated way of saying that $e_{\min}$ should be $1 - e_{\max}$ in radix 2 and $-e_{\max}$ in radix 10).

For instance, the single-precision format of IEEE 754-1985 satisfies these requirements: with $\beta = 2$, $p = 24$, $e_{\min} = -126$, and $e_{\max} = 127$, we have

$$\begin{cases} \dfrac{e_{\max} - e_{\min}}{p} &= & 10.54\cdots & > & 10; \\ \beta^{p-1} &= & 2^{23} & \geq & 10^5; \\ \beta^{e_{\max}+e_{\min}+1} &= & 2^2 & = & 4. \end{cases}$$

3.2.2 Various formats and the constraints between them

The narrowest supported format is called *single-precision*. When a second, wider basic format is supported, it is called *double-precision*. The required constraints between their respective parameters $e_{\min s}$, $e_{\max s}$, p_s and $e_{\min d}$, $e_{\max d}$, p_d are:

- $\beta^{p_d} \geq 10\beta^{2p_s}$;

- $e_{\max d} \geq 8e_{\max s} + 7$;

- $e_{\min d} \leq 8e_{\min s}$.

Extended precisions are also possible. For obvious reasons, the only extended precision that is recommended is the one associated to the widest supported basic precision. If $e_{\min}$, $e_{\max}$, and p are the extremal exponents and precision of that widest basic precision, the parameters $e_{\min e}$, $e_{\max e}$, and p_e of the corresponding extended precision must satisfy:

---

[7]We will see in the following pages that the revised standard IEEE 754-2008 will require $e_{\min} = 1 - e_{\max}$ for all formats.

- $e_{\mathrm{max}e} \geq 8e_{\mathrm{max}} + 7$;

- $e_{\mathrm{min}e} \leq 8e_{\mathrm{min}}$;

- if $\beta = 2$,

$$p_e \geq p + \lceil \log_2\left(e_{\mathrm{max}} - e_{\mathrm{min}}\right)\rceil ; \qquad (3.1)$$

- for all β, $p_e \geq 1.2p$.

It is also recommended that

$$p_e > 1 + p + \frac{\log\left[3\log(\beta)\left(e_{\mathrm{max}} + 1\right)\right]}{\log(\beta)}. \qquad (3.2)$$

The purpose of constraint (3.1) was to facilitate the support of conversion to and from decimal strings for the basic formats, using algorithms that were available at that time. The purpose of (3.2) was to make accurate implementation, in the basic formats, of the power function x^y simpler. Again, the rationale behind the existence of the extended formats is to allow for efficient implementations of various functions of basic format variables without having to worry too much about roundoff error propagation and possible over/underflow in the intermediate calculations.

3.2.3 Conversions between floating-point numbers and decimal strings

Very much as in IEEE 754-1985, conversions must be provided between decimal strings (in at least one format) and floating-point numbers in all supported basic precisions. As for IEEE 754-1985, these constraints might seem somewhat below what one could now expect: at the time the standard was released, some of the best conversion algorithms that can now be found in the literature were not published yet.

Consider decimal strings with values of the form $\pm M \times 10^{\pm N}$, with $0 \leq M \leq 10^D - 1$. When several representations are possible, the one with the smallest N is used in Tables 3.9 and 3.10. Conversions must be provided in the range specified in Table 3.9, and correctly rounded in the range specified in Table 3.10. The value e_m in these tables is

$$e_m = \max\left\{D + (p - 1 - e_{\mathrm{min}})\log_{10}(\beta), (e_{\mathrm{max}} + 1)\log_{10}(\beta) + 1 - D\right\}.$$

When conversion is not correctly rounded (i.e., outside the range given in Table 3.10) and $\beta = 2$, the error in the converted result must be less than $0.97\,\mathrm{ulp}$ of the target format in the round-to-nearest mode, and less than $1.47\,\mathrm{ulp}$ in the directed rounding modes. The direction of the directed rounding modes must be satisfied (e.g., when the active rounding mode is RD, the obtained result must be less than or equal to the exact result). These bounds of 0.97 and 1.47 ulps come from the best conversion algorithms that were available at that time [81].

β	Max D	Max N
2	$\lceil p \log_{10} 2 + 1 \rceil$	$10^{\lfloor \log_{10}(e_m) \rfloor + 1} - 1$
10	p	$10^{\lfloor \log_{10}(e_m) \rfloor + 1} - 1$

Table 3.9: Floating-point from/to decimal string conversion ranges in the IEEE 854-1987 standard [11] (©IEEE, 1987, with permission).

β	Max D	Max N
2	$\lceil p \log_{10} 2 + 1 \rceil$	$\lfloor p_e \log_5(2) \rfloor$
10	p	$10^{\lfloor \log_{10}(e_m) \rfloor + 1} - 1$

Table 3.10: Correctly rounded conversion ranges in the IEEE 854-1987 standard [11] (©IEEE, 1987, with permission). Variable p_e denotes the smallest permissible value as extended support for precision p.

3.2.4 Rounding

The IEEE 854-1987 standard requires that the arithmetic operations and the square root be correctly rounded. Exactly as for IEEE 754-1985, four rounding modes are specified: rounding toward $-\infty$, toward $+\infty$, toward 0, and to nearest. Round to nearest must be the default mode, with the following two characteristics:

- when rounding x, if the two floating-point numbers nearest to x are equally near, the one whose least significant digit is even is delivered (this is the *round-to-nearest-even mode*, as in IEEE 754-1985, but generalized for other radices);

- if the exact result has an absolute value larger than or equal to $\beta^{e_{\max}} \left(\beta - \frac{1}{2} \beta^{1-p} \right)$, then an infinite result (with the correct sign) is returned. Notice that when $\beta = 2$, this is what was already required in IEEE 754-1985 (see Section 3.1.5, page 67).

3.2.5 Operations

The arithmetic operations, the remainder operation, and the square root (including the $\sqrt{-0} = -0$ requirement) are defined very much as in IEEE 754-1985.

3.2.6   Comparisons

The comparisons are defined very much as in IEEE 754-1985. Especially, every NaN compares "unordered" with everything including itself: the test "$x \neq x$" must return **true** if x is a NaN.

3.2.7   Exceptions

The IEEE 754-1985 way of handling exceptions was also chosen for IEEE 854-1987. The only modifications come from the facts that two possible radices are considered and the formats are not fully specified. For instance, when a trap occurs in the case of an overflow or underflow, the scaling factor K used for scaling the result returned to the trap handler becomes β^{α}, where $\alpha \approx \frac{3}{4}\left(e_{\max} - e_{\min}\right)$, and α should be a multiple of 12.

3.3   The Need for a Revision

The IEEE 754-1985 standard was a huge improvement. It soon became implemented on most platforms of commercial significance. And yet, 15 years after its release, there was a clear need for a revision.

- Some features that had become common practice needed to be standardized: e.g., the "quadruple-precision" (i.e., 128-bit wide, binary) format, the fused multiply-add operator.

- Since 1985, new algorithms were published that allowed one to easily perform computations that were previously thought too complex. Typical examples are the radix conversion algorithms presented in Section 2.7: now, for an internal binary format, it is possible to have much stronger requirements on the accuracy of the conversions that must be done when reading or printing decimal strings. Another example is the availability of reasonably fast libraries for some correctly rounded elementary functions: the revised standard can now deal with transcendental functions and recommend that some should be correctly rounded.

- There were some ambiguities in IEEE 754-1985. For instance, when evaluating expressions, when a larger internal format is available in hardware, it was unclear in which format the implicit intermediate variables should be represented.

- As pointed out by David Hough, the various implementations of IEEE 754-1985 did not allow one to code the most arcane aspects of the standard in a portable way.

3.3.1 A typical problem: "double rounding"

The processor being used may offer an internal precision that is wider than the precision of the variables of a program (a typical example is the double-extended format available on Intel platforms, when the variables of the program are single-precision or double-precision floating-point numbers). This may sometimes have strange side effects, as we will see in this section.

Consider the C program (Program 3.1).

```c
#include <stdio.h>

int main(void)
{
  double a = 1848874847.0;
  double b = 19954562207.0;
  double c;
  c = a * b;
  printf("c = %20.19e\n", c);
  return 0;
}
```

Program 3.1: A C program that might induce a double rounding.

Tables 3.11 and 3.12 give some results returned by this program, depending on the processor and the compilation options. In order to really test the arithmetic of the machine, it is important that the compiler does not optimize the multiplication by performing it at compile time (one controls even less what occurs at compile time); by default, GCC does not do such an optimization. Notice that the double-precision number closest to the exact product is `3.6893488147419111424e+19`.

Switches on the GCC command line	Output
no switch (default)	c = 3.6893488147419103232e+19
-mfpmath=387	c = 3.6893488147419103232e+19
-march=pentium4 -mfpmath=sse	c = 3.6893488147419111424e+19

Table 3.11: Results returned by Program 3.1 on a Linux/Debian Etch 32-bit Intel platform, with GNU Compiler Collection (GCC) 4.1.2 20061115, depending on the compilation options. Notice that on the 32-bit platform, the default is to use the 387 registers.

Switches on the GCC command line	Output
no switch (default)	c = 3.6893488147419111424e+19
-mfpmath=387	c = 3.6893488147419103232e+19
-march=pentium4 -mfpmath=sse	c = 3.6893488147419111424e+19

Table 3.12: Results returned by Program 3.1 on a Linux/Debian Etch 64-bit Intel platform, with GCC 4.1.2 20061115, depending on the compilation options. Notice that on the 64-bit platform, the default is to use the SSE registers.

What happened? The exact value of `a*b` is 36893488147419107329, whose binary representation is

$$\underbrace{\overbrace{1000000000000000000000000000000000000000000000000000}_{53\ \text{bits}} 10000000000}^{64\ \text{bits}}\ 01$$

On the processor used, with the `-mfpmath=387` switch, the product is first rounded to the precision of the registers (namely, double-extended precision), which gives (in binary)

$$\overbrace{\underbrace{1000000000000000000000000000000000000000000000000000}_{53\ \text{bits}} 10000000000}^{64\ \text{bits}} \times 4$$

Then, that intermediate value is rounded to the double-precision destination format, which gives (using the round-to-nearest-even rounding mode)

$$\underbrace{10000000000000000000000000000000000000000000000000000}_{53\ \text{bits}} \times 2^{13}$$
$$= 36893488147419103232_{10},$$

whereas the product `a*b` correctly rounded to the nearest double-precision number is

$$\underbrace{10000000000000000000000000000000000000000000000000001}_{53\ \text{bits}} \times 2^{13}$$
$$= 36893488147419111424_{10}.$$

The `-march=pentium4 -mfpmath=sse` compilation switches force the product to be stored in the 64-bit Streaming SIMD Extension (SSE) registers. In that case, it is directly rounded to double precision, so that we get the expected result.

The problem we have faced here is called "double rounding." In this example, it appears during a multiplication, but it may also appear during another arithmetic operation. Another example (still with double-precision input values) is the addition of

$$9223372036854775808.0 = 2^{63}$$

and

$$1024.25.$$

Such examples are not so rare that they can be neglected. Assuming double-precision variables and a double-extended internal format, if the chosen compilation switches do not prevent the problem from occurring, the double rounding problem occurs when the binary expansion of the exact result of some operation is of the form

$$2^k \times \overbrace{1.xxxxx \cdots xx0}^{53 \text{ bits}} \; \overbrace{10000000000}^{11 \text{ bits}} \; 0 \; \overbrace{xxxxxxxxxxxxxxxxxxxxxx \cdots}^{\text{at least one 1 somewhere}}$$

or

$$2^k \times \overbrace{1.xxxxx \cdots xx1}^{53 \text{ bits}} \; \overbrace{01111111111}^{11 \text{ bits}} \; 1 \; \overbrace{xxxxxxxxxxxxxxxxxxxxxx \cdots}^{\text{at least one 0 somewhere}}.$$

Assuming equal probabilities of occurrence for the zeros and ones in the binary expansion of the result of an arithmetic operation,[8] the probability of a double rounding is $2^{-12} = 1/4096$, which means that without care with the compilation options, double roundings will occur in any computation of significant size.

We must emphasize that this might be a problem with some very specific algorithms only (such as those presented in Chapter 4), but with most calculations, it will be unnoticed.

3.3.2 Various ambiguities

In [280], Monniaux gives some very convincing examples of consequences of "ambiguities." The examples shown here were obtained on a Pentium processor, under Linux in 32 bits, with GCC 4.0.1.

Consider Program 3.2.

What happened? Although in double-precision arithmetic, in round-to-nearest (i.e., the default) mode, the multiplication v * v should have returned $+\infty$, the implicit variable representing that product was actually stored in a double-extended precision register of the processor. And since the product v * v is much below the overflow threshold in double-extended

---

[8]Which is not very realistic but suffices to get a rough estimate of the frequency of occurrences of double roundings.

```
    #include <stdio.h>
    int main(void)
    {
      double v = 1E308;
      double x = (v * v) / v;
      printf("%g\n",x);
      return 0;
    }
```

Program 3.2: This example is due to David Monniaux [280]. Compiled with GCC under Linux/x86, we get 1e+308, *whereas if all computations had been performed in double-precision arithmetic, we should have obtained* $+\infty$.

precision, the stored value was not $+\infty$, but the double-extended number closest to the exact product. The result $+\infty$ can be obtained with recent GCC versions and the -ffloat-store option, which forces the intermediate results to be spilt to memory,[9] in double precision.

It is important to notice that, in this case, the obtained result is *very accurate*, which is not that surprising: in most cases, using a larger internal precision for intermediate calculations leads to better calculations. What matters then is not to forbid that, but to allow programmers to decide if they want all intermediate calculations to be performed in the format of the operands (which enhances portability and provability and is necessary for safely using most of the small algorithms given in Chapter 4), or if they prefer these intermediate calculations to be performed in a wider format (typically, the largest format available in hardware, which in general improves the accuracy of the results). A tradeoff is to be found between portability, accuracy, and (frequently) speed. Choosing which among these criteria is the most important should be the programmer's task, not the compiler's.

The following example is even more interesting. Consider Program 3.3.

Compiled with GCC under Linux without optimization, we get inf; compiled with optimization (option "-O"), we get 1e+308. Now, let us see what happens if we just insert a statement to print y just after its computation (see Program 3.4).

Compiled with GCC under Linux with optimization (option "-O"), we now get inf (whereas we got 1e+308 without the printf call). That statement forced y to be spilt to memory (hence, to be represented in the double-precision format), instead of staying in a double-extended precision register. This example shows that *even a statement that does not involve a numeric computation in the program (here, a call to the* printf *function) can change the final result*. This sometimes makes portable and provable numerical programs very difficult to design. Chapter 7 deals with these issues.

---

[9]However this is not guaranteed by the GCC documentation, and the effect of the -ffloat-store option may still depend on the GCC version.

```
#include <stdio.h>

static inline double f(double x)
{
  return x / 1E308;
}

double square(double x)
{
  double y = x * x;
  return y;
}

int main(void)
{
  printf("%g\n", f(square(1E308)));
  return 0;
}
```

Program 3.3: This example is due to David Monniaux [280]. Compiled with GCC under Linux without optimization, we get inf, *compiled with optimization (option "-O"), we get* 1e+308.

3.4 The New IEEE 754-2008 Standard

The IEEE 754-1985 standard has been revised from 2000 to 2006, and the revised standard was adopted in June 2008. Some of the various goals of the working group were as follows (see http://grouper.ieee.org/groups/754/revision.html):

- merging the 854-1987 standard into the 754-1985 standard;

- reducing the implementation choices;

- resolving some ambiguities in the 754-1985 standard (especially concerning expression evaluation and exception handling). The revised standard allows languages and users to focus on portability and reproducibility, or on performance;

- standardizing the fused multiply-add (FMA) operation, and

- including quadruple precision.

Also, the working group had to cope with a very strong constraint: the revised standard would rather not invalidate hardware that conformed to the old IEEE 754-1985 standard.

```
#include <stdio.h>

static inline double f(double x)
{
  return x / 1E308;
}

double square(double x)
{
  double y = x * x;
  printf("%g\n",y);
  return y;
}

int main(void)
{
  printf("%g\n", f(square(1E308)));
  return 0;
}
```

Program 3.4: This example is due to David Monniaux [280]. Compiled with GCC under Linux with optimization (option "-O"), we get inf.

3.4.1   Formats specified by the revised standard

The revised standard requires that the radix β should be 2 or 10, and that $e_{\min}$ should be $1 - e_{\max}$ for all formats. It defines two kinds of formats:

- *interchange formats*, whose encodings are fully specified as bit strings, and that allow data interchange between different platforms, provided that endianness problems (see Section 3.1.2) are resolved,[10] and

- *extended* and *extendable* precision formats,[11] whose encodings are not specified, but may match those of interchange formats.

   The standard requires that conversions between any two supported formats be implemented. Moreover, a format is said to be an *arithmetic format* if all the mandatory operations defined by the standard are supported by the format.

   Among the interchange formats, the standard defines five *basic formats*, which must also be arithmetic formats: the three binary formats on 32, 64, and 128 bits, and the two decimal formats on 64 and 128 bits. A conforming implementation must implement at least one of them.

---

[10]This is not much more difficult than with the integers, though. Alternatively character strings can be used.

[11]The revised standard [187] makes a distinction between an *extended format*, which extends a basic format with a wider precision and range, and is language defined or implementation defined, and an *extendable precision format*, whose precision and range are defined under program control.

Name	binary16	binary32 (basic)	binary64 (basic)	binary128 (basic)
p	11	24	53	113
$e_{\max}$	$+15$	$+127$	$+1023$	$+16383$
$e_{\min}$	-14	-126	-1022	-16382

Table 3.13: Main parameters of the binary interchange formats of size up to 128 bits specified by the 754-2008 standard [187].

Name	decimal32	decimal64 (basic)	decimal128 (basic)
p	7	16	34
$e_{\max}$	$+96$	$+384$	$+6144$
$e_{\min}$	-95	-383	-6143

Table 3.14: Main parameters of the decimal interchange formats of size up to 128 bits specified by the 754-2008 standard [187].

The main parameters of the interchange formats of size up to 128 bits are given in Tables 3.13 and 3.14.

3.4.2 Binary interchange format encodings

The binary interchange formats are very much like the formats of the IEEE 754-1985 standard. The floating-point numbers are encoded using a 1-bit sign, a W_E-bit exponent field, and a $(p-1)$-bit field for the trailing significand. This is illustrated in Figure 3.1.

Figure 3.1: Binary interchange floating-point formats [187] (©IEEE, 2008, with permission).

Define E as the integer whose binary representation consists of the bits of the exponent field, T as the integer whose representation consists of the bits of the trailing significand field, and S as the sign bit. The binary encoding (S, E, T), similar to that of IEEE 754-1985 (summarized in Table 3.4), should

be interpreted as follows [187]:

- if $E = 2^{W_E} - 1$ and $T \neq 0$, then a NaN, either quiet (qNaN) or signaling (sNaN), is represented. As in IEEE 754-1985, a quiet NaN is the default result of an invalid operation, and a signaling NaN will signal the invalid operation exception whenever it appears as an operand;

- if $E = 2^{W_E} - 1$ and $T = 0$, then $(-1)^S \times (+\infty)$ is represented;

- if $1 \leq E \leq 2^{W_E} - 2$, then the (normal) floating-point number being represented is

$$(-1)^S \times 2^{E-b} \times \left(1 + T \cdot 2^{1-p}\right),$$

where the *bias* $b = e_{\max} = 2^{W_E-1} - 1$ is equal to 15, 127, 1023, and 16383 in the binary16, binary32, binary64, and binary128 formats, respectively;

- if $E = 0$ and $T \neq 0$, then the (subnormal) number being represented is

$$(-1)^S \times 2^{e_{\min}} \times \left(0 + T \cdot 2^{1-p}\right);$$

- if $E = 0$ and $T = 0$, then the number being represented is the signed zero $(-1)^S \times (+0)$.

The sizes of the various fields are given in Table 3.15.

format	binary16	binary32	binary64	binary128
storage width	16	32	64	128
$p-1$, trailing significand width	10	23	52	112
W_E, exponent field width	5	8	11	15
b, bias	15	127	1023	16383

Table 3.15: Width (in bits) of the various fields in the encodings of the binary interchange formats of size up to 128 bits [187].

The binary32 and binary64 formats correspond to the single- and double-precision formats of the IEEE 754-1985 standard: the encodings are exactly the same.

3.4.3   Decimal interchange format encodings

The decimal format encodings are more complex than the binary ones, for several reasons.

- *Two* encoding systems are specified, called the *decimal* and *binary* encodings: the members of the revision committee could not agree on a single

encoding system. The reason for that is that the binary encoding makes a *software* implementation of decimal arithmetic easier, whereas the decimal encoding is more suited for a *hardware* implementation. And yet, despite this problem, one must understand that the set of representable floating-point numbers is *the same* for both encoding systems, so that this additional complexity will be transparent for most users. Also, a conforming implementation must provide conversions between these two encoding systems ([187, §5.5.2]).

- Contrary to the binary interchange formats, the sign, exponent, and (trailing) significand fields are not fully separated: to preserve as much accuracy as possible, some information on the significand is partly encoded in what used to be the exponent field and is hence called the *combination* field.

- In the decimal formats, the representations (M, e) are not normalized, so that a decimal floating-point number may have multiple valid representations. The set of the various representations of a same number is called a *cohort*. As a consequence, we will have to explain which exponent is *preferred* for the result of an arithmetic operation.

- Even if the representation itself (that is, values of the sign, exponent, and significand) of a number x (or an infinite, or a NaN) and the type of encoding (binary or decimal) are chosen, a same number (or infinite, or NaN) can still be encoded by different bit strings. One of them will be said to be *canonical*.

Roughly speaking, the difference between the decimal and binary encodings originates from a choice in the encoding of the significand. The integral significand is a non-negative integer less than or equal to $10^p - 1$. One can encode it either in binary (which gives the binary encoding) or in decimal (which gives the decimal encoding).

Concerning the decimal encoding, in the early days of computer arithmetic, people would use the binary coded decimal (BCD) encoding, where each decimal digit was encoded by four bits. That encoding was quite wasteful, since among the 16 possible values representable on four bits, only 10 were actually used. And yet, since $2^{10} = 1024$ is very close to 10^3 (and larger), one can design a much denser encoding by encoding three consecutive decimal digits by a 10-bit *declet*[68]. Many possible ways of performing that encoding are possible. The one chosen by the standard committee for the decimal encoding of decimal numbers is given in Tables 3.19 (declet to decimal) and 3.20 (decimal to declet). It was designed to facilitate conversions: all these tables have a straightforward hardware implementation and can be implemented in three gate levels [123]. Note that Table 3.19 has 1024 possible inputs and 1000 possible outputs (hence, there is some redundancy), and

Table 3.20 has 1000 possible inputs and outputs. This implies that there are 24 "noncanonical" bit patterns,[12] which are accepted in input values but cannot result from an arithmetic operation. An encoding that contains a noncanonical bit pattern is called *noncanonical*.

Let us explain more precisely why there is no clear separation between an exponent field and a significand field (as is the case in the binary formats). Consider as an example the decimal64 format (see Table 3.14). In that format, $e_{\max} = 384$ and $e_{\min} = -383$; therefore, there are 768 possible values of the exponent. Storing all these values in binary in an exponent field would require 10 bits. Since we can store 1024 possible values in a 10-bit field, that would be wasteful. This explains why it was decided to put all the information about the exponent plus some other information in a "combination field," where will be stored:

- "classification" information: Does the datum represent a finite number, or $\pm\infty$, or a NaN?

- the exponent (if the datum represents a finite number);

- the leading part of the significand (if the datum represents a finite number); more precisely, the leading decimal digit (if the *decimal* encoding is used) or 3 to 4 leading bits (if the *binary* encoding is used). The remaining significand bits/digits are stored in the trailing significand field.

MSB   LSB

S	G	T

1 bit

$\xleftarrow{\hspace{1cm}}$ $w + 5$ bits $\xrightarrow{\hspace{1cm}}$ $\xleftarrow{\hspace{2cm}}$ $t = J \times 10$ bits $\xrightarrow{\hspace{2cm}}$

Figure 3.2: Decimal interchange floating-point formats [187] (©IEEE, 2008, with permission).

The widths of the various fields are given in Table 3.16. It is important to notice that in this table the bias b is related to the *quantum exponent* (see Section 2.1), which means that if e is the exponent of x, if $q = e - p + 1$ is its quantum exponent, then the biased exponent E is

$$E = q + b = e - p + 1 + b.$$

The floating-point format illustrated in Figure 3.2, with a 1-bit sign, a $(w + 5)$-bit *combination* field, and a $t = (J \times 10)$-bit *trailing significand* field must be interpreted as follows [187]:

---

[12]Those of the form $01x11x111x$, $10x11x111x$, or $11x11x111x$.

	decimal32	decimal64	decimal128
storage width	32	64	128
$t = 10J$, trailing significand width	20	50	110
$w + 5$, combination field width	11	13	17
$b = E - (e - p + 1)$, bias	101	398	6176

Table 3.16: Width (in bits) of the various fields in the encodings of the decimal interchange formats of size up to 128 bits [187].

- if the five most significant bits of G (numbered from the left, G_0 to G_4) are all ones, then the datum being represented is a NaN. Moreover, if G_5 is 1, then it is an sNaN, otherwise it is a qNaN. In a canonical encoding of a NaN, the bits G_6 to G_{w+4} are all zeros;

- if the five most significant bits of G are 11110, then the value being represented is $(-1)^S \times (+\infty)$. Moreover, the canonical encodings of infinity have bits G_5 to G_{w+4} as well as trailing significand T equal to 0;

- if the four most significant bits of G, i.e., G_0 to G_3, are not all ones, then the value being represented is a finite number, equal to

$$(-1)^S \times 10^{E-b} \times C. \tag{3.3}$$

Here, the value $E - b$ is the *quantum exponent* (see Section 2.1), where b, the *exponent bias*, is equal to 101, 398, and 6176 for the decimal32, decimal64, and decimal128 formats, respectively. E and C are obtained as follows.

1. If the *decimal encoding* is used for the significand, then the least significant w bits of the biased exponent E are made up of the bits G_5 to G_{w+4} of G, whereas the most significant two bits of E and the most significant two digits of C are obtained as follows:

   - if the five most significant bits $G_0G_1G_2G_3G_4$ of G are of the form 110xx or 1110x, then the leading significand digit C_0 is $8 + G_4$ (which equals 8 or 9), and the leading biased exponent bits are G_2G_3;
   - if the five most significant bits of G are of the form 0xxxx or 10xxx, then the leading significand digit C_0 is $4G_2 + 2G_3 + G_4$ (which is between 0 and 7), and the leading biased exponent bits are G_0G_1.

   The $p - 1 = 3J$ decimal digits $C_1, \ldots, C_{p-1}$ of C are encoded by T, which contains J declets encoded in densely packed decimal (see Tables 3.19 and 3.20). Note that if the five most significant bits of G are 00000, 01000, or 10000, and $T = 0$, then the significand is 0 and the represented number is $(-1)^S \times (+0)$.

Table 3.17 summarizes these rules.

2. If the *binary encoding* is used for the significand, then

  – if G_0G_1 is 00, 01, or 10, then E is made up of the bits G_0 to G_{w+1}, and the binary encoding of the significand C is obtained by prefixing the last 3 bits of G (i.e., $G_{w+2}G_{w+3}G_{w+4}$) to T;

  – if G_0G_1 is 11 and G_2G_3 is 00, 01 or 10, then E is made up of the bits G_2 to G_{w+3}, and the binary encoding of the significand C is obtained by prefixing $100G_{w+4}$ to T.

  Remember that the maximum value of the integral significand is $10^p - 1 = 10^{3J+1} - 1$. If the value of C computed as above is larger than that maximum value, then the value used for C will be zero [187], and the encoding will not be canonical. Table 3.18 summarizes these rules.

A decimal software implementation of IEEE 754-2008, based on the binary encoding of the significand, is presented in [85, 87]. Interesting information on decimal arithmetic can be found in [90]. A decimal floating-point multiplier that uses the decimal encoding of the significand is presented in [129].

Example 3 (Finding the encoding of a decimal number assuming decimal encoding of the significands)**.** *Consider the number*

$$x = 3.141592653589793 \times 10^0 = 3141592653589793 \times 10^{-15}.$$

This number is exactly representable in the decimal64 format. Let us find its encoding, assuming decimal encoding of the significands.

- *First, the sign bit is 0;*

- *since the quantum exponent is -15, the biased exponent will be 383 (see Table 3.16), whose 10-bit binary representation is 0101111111. One should remember that the exponent is not directly stored in an exponent field, but combined with the most significant digit of the significand in a combination field G. Since the leading significand digit is 3, we are in the case*

  if the five most significant bits of G are of the form 0xxxx or 10xxx, then the leading significand digit C_0 is $4G_2 + 2G_3 + G_4$ (which is between 0 and 7), and the leading biased exponent bits are G_0G_1.

 Hence,

  – *G_0 and G_1 are the leading biased exponent bits, namely 0 and 1;*
  – *G_2, G_3, and G_4 are the binary encoding of the first significand digit 3, i.e., $G_2 = 0$, and $G_3 = G_4 = 1$; and*
  – *the bits G_5 to G_{12} are the least significant bits of the biased exponent, namely 01111111.*

G	T	Datum being represented
$11111\,0xxx\cdots x$	any	qNaN
$11111\,1xxx\cdots x$	any	sNaN
$11110\,xxxx\cdots x$	any	$(-1)^S \times (+\infty)$
$110xx\cdots$ or $1110x\cdots$	$T_0 T_1 \cdots T_{10J-1}$	$(-1)^S \times 10^{\overbrace{G_2 G_3 G_5 G_6 \cdots G_{w+4}}^{\text{binary}} - b} \times (8+G_4)\underbrace{C_1 C_2 \cdots C_{p-1}}_{\text{decimal}}$ with $C_{3j+1}C_{3j+2}C_{3j+3}$ deduced from $T_{10j}T_{10j+1}T_{10j+2}\cdots T_{10j+9}$ for $0 \le j < J$ using Table 3.19.
$0xxxx\cdots$ or $10xxx\cdots$	$T_0 T_1 \cdots T_{10J-1}$	$(-1)^S \times 10^{\overbrace{G_0 G_1 G_5 G_6 \cdots G_{w+4}}^{\text{binary}} - b} \times (4G_2 + 2G_3 + G_4)\underbrace{C_1 C_2 \cdots C_{p-1}}_{\text{decimal}}$ with $C_{3j+1}C_{3j+2}C_{3j+3}$ deduced from $T_{10j}T_{10j+1}T_{10j+2}\cdots T_{10j+9}$ for $0 \le j < J$ using Table 3.19.

Table 3.17: Decimal encoding of a decimal floating-point number (IEEE 754-2008).

G	T	Datum being represented
$11111\,0xxx\cdots x$	any	qNaN
$11111\,1xxx\cdots x$	any	sNaN
$11110\,xxxx\cdots x$	any	$(-1)^S \times (+\infty)$
$00xxx\cdots$ or $01xxx\cdots$ or $10xxx\cdots$	$T_0 T_1 \cdots T_{10J-1}$	$(-1)^S \times 10^{\overbrace{G_0 G_1 G_2 \cdots G_{w+1}}^{\text{binary}} - b} \times \underbrace{G_{w+2} G_{w+3} G_{w+4} T_0 T_1 \cdots T_{10J-1}}_{\text{binary}}$ if $G_{w+2} G_{w+3} G_{w+4} T_0 T_1 \cdots T_{10J-1} \leq 10^p - 1$, otherwise $(-1)^S \times (+0)$.
$1100xxx\cdots$ or $1101xxx\cdots$ or $1110xxx\cdots$	$T_0 T_1 \cdots T_{10J-1}$	$(-1)^S \times 10^{\overbrace{G_2 G_3 G_4 \cdots G_{w+3}}^{\text{binary}} - b} \times \underbrace{100 G_{w+4} T_0 T_1 \cdots T_{10J-1}}_{\text{binary}}$ if $100 G_{w+4} T_0 T_1 \cdots T_{10J-1} \leq 10^p - 1$, otherwise $(-1)^S \times (+0)$.

Table 3.18: Binary encoding of a decimal floating-point number (IEEE 754-2008).

$b_6 b_7 b_8 b_3 b_4$	d_0	d_1	d_2
0 x x x x	$4b_0 + 2b_1 + b_2$	$4b_3 + 2b_4 + b_5$	$4b_7 + 2b_8 + b_9$
1 0 0 x x	$4b_0 + 2b_1 + b_2$	$4b_3 + 2b_4 + b_5$	$8 + b_9$
1 0 1 x x	$4b_0 + 2b_1 + b_2$	$8 + b_5$	$4b_3 + 2b_4 + b_9$
1 1 0 x x	$8 + b_2$	$4b_3 + 2b_4 + b_5$	$4b_0 + 2b_1 + b_9$
1 1 1 0 0	$8 + b_2$	$8 + b_5$	$4b_0 + 2b_1 + b_9$
1 1 1 0 1	$8 + b_2$	$4b_0 + 2b_1 + b_5$	$8 + b_9$
1 1 1 1 0	$4b_0 + 2b_1 + b_2$	$8 + b_5$	$8 + b_9$
1 1 1 1 1	$8 + b_2$	$8 + b_5$	$8 + b_9$

Table 3.19: Decoding the declet $b_0 b_1 b_2 \cdots b_9$ of a densely packed decimal encoding to three decimal digits $d_0 d_1 d_2$ [187] (©IEEE, 2008, with permission).

$d_0^0\ d_1^0\ d_2^0$	$b_0\ b_1\ b_2$	$b_3\ b_4\ b_5$	b_6	$b_7\ b_8\ b_9$
0 0 0	$d_0^1\ d_0^2\ d_0^3$	$d_1^1\ d_1^2\ d_1^3$	0	$d_2^1\ d_2^2\ d_2^3$
0 0 1	$d_0^1\ d_0^2\ d_0^3$	$d_1^1\ d_1^2\ d_1^3$	1	$0\ 0\ d_2^3$
0 1 0	$d_0^1\ d_0^2\ d_0^3$	$d_2^1\ d_2^2\ d_1^3$	1	$0\ 1\ d_2^3$
0 1 1	$d_0^1\ d_0^2\ d_0^3$	$1\ 0\ d_1^3$	1	$1\ 1\ d_2^3$
1 0 0	$d_2^1\ d_2^2\ d_0^3$	$d_1^1\ d_1^2\ d_1^3$	1	$1\ 0\ d_2^3$
1 0 1	$d_1^1\ d_1^2\ d_0^3$	$0\ 1\ d_1^3$	1	$1\ 1\ d_2^3$
1 1 0	$d_2^1\ d_2^2\ d_0^3$	$0\ 0\ d_1^3$	1	$1\ 1\ d_2^3$
1 1 1	$0\ 0\ d_0^3$	$1\ 1\ d_1^3$	1	$1\ 1\ d_2^3$

Table 3.20: Encoding the three consecutive decimal digits $d_0 d_1 d_2$, each of them being represented in binary by four bits (e.g., d_0 is written in binary $d_0^0 d_0^1 d_0^2 d_0^3$), into a 10-bit declet $b_0 b_1 b_2 \cdots b_9$ of a densely packed decimal encoding [187] (©IEEE, 2008, with permission).

- *Now, the trailing significand field T is made up of the five declets of the densely packed decimal encoding of the trailing significand* 141592653589793:

  – *the 3-digit chain* 141 *is encoded by the declet* 0011000001, *according to Table 3.20;*
  – 592 *is encoded by the declet* 1010111010;
  – 653 *is encoded by the declet* 1101010011;
  – 589 *is encoded by the declet* 1011001111;
  – 793 *is encoded by the declet* 1110111011.

- *Therefore, the encoding of x is*

$$\underbrace{0}_{} \ \underbrace{0101101111111}_{\text{sign combination field}} \dots$$

$$\underbrace{0011000001101011101011010100111011001111111011011}_{\text{trailing significand field}}.$$

Example 4 (Finding an encoding of a decimal number assuming binary encoding of the significands)**.** *Consider the number*

$$x = 3.141592653589793 \times 10^0 = 3141592653589793 \times 10^{-15}.$$

(It is the same number as in Example 3, but now we consider binary encoding, in the decimal64 format.) The sign bit will be zero. Since 3141592653589793 *is a 16-digit integer that does not end with a 0, the quantum exponent can only be -15; therefore, the biased exponent E will be $398-15 = 383$, whose binary representation is* 101111111. *The binary representation of the integral significand of x is*

$$10 \underbrace{11001010010100001100001010001001010110100100001}_{t = 50 \text{ bits (trailing significand)}}.$$

The length of that bit string is 52, which is less than $t + 4 = 54$, hence we are not in the case

> *if G_0G_1 is 11 and G_2G_3 is 00, 01 or 10, then E is made up of the bits G_2 to G_{w+3}, and the binary encoding of the significand C is obtained by prefixing $100G_{w+4}$ to T,*

which means that we are in the case

> *if G_0G_1 is 00, 01, or 10, then E is made up of the bits G_0 to G_{w+1}, and the binary encoding of the significand C is obtained by prefixing the last 3 bits of G (i.e., $G_{w+2}G_{w+3}G_{w+4}$) to T.*

Therefore, $G_0G_1 \ldots G_9 = 0101111111$, $G_{10}G_{11}G_{12} = 010$ and

$$T = 1100101001010000110000101000100101010110110100100001.$$

Example 5 (Finding the value of a decimal floating-point number from its encoding, assuming decimal encoding of the significand)**.** *Consider the decimal32 number x whose encoding is*

$$\underbrace{1}_{sign} \quad \underbrace{11101101101}_{combination\ field\ G} \quad \underbrace{01101001101111000011}_{trailing\ significand\ field\ T} .$$

- *Since the bit sign is 1, we have $x \leq 0$;*

- *since the four most significant bits of G are not all ones, x is not an infinity or a NaN;*

- *by looking at the four most significant bits of G, we deduce that we are in the case*

  > *if the five most significant bits $G_0G_1G_2G_3G_4$ of G are of the form 110xx or 1110x, then the leading significand digit C_0 is $8 + G_4$ (which equals 8 or 9), and the leading biased exponent bits are G_2G_3.*

  Therefore, the leading significand bit C_0 is $8 + G_4 = 9$, and the leading biased exponent bits are 10. The least significant bits of the exponent are 101101; therefore, the biased exponent is $10101101_2 = 173_{10}$. Hence, the (unbiased) quantum exponent of x is $173 - 101 = 72$;

- *the trailing significand field T is made up of two declets, 0110100110 and 1111000011. According to Table 3.19,*

  - *the first declet encodes the 3-digit chain 326;*
  - *the second declet encodes 743.*

- *Therefore, x is equal to*

  $$-9326743 \times 10^{72} = -9.326743 \times 10^{78}.$$

Example 6 (Finding the value of a decimal floating-point number from its encoding, assuming binary encoding of the significand)**.** *Consider the decimal32 number x whose encoding is*

$$\underbrace{1}_{sign} \quad \underbrace{11101101101}_{combination\ field\ G} \quad \underbrace{01101001101111000011}_{trailing\ significand\ field\ T} .$$

(It is the same bit string as in Example 5, but now we consider binary encoding.)

- *Since the bit sign is 1, we have $x \leq 0$;*

- *since the four most significant bits of G are not all ones, x is not an infinity or a NaN;*

- *since $G_0 G_1 = 11$ and $G_2 G_3 = 10$, we are in the case*

  > *if $G_0 G_1$ is 11 and $G_2 G_3$ is 00, 01, or 10, then E is made up of the bits G_2 to G_{w+3}, and the binary encoding of the significand C is obtained by prefixing $100 G_{w+4}$ to T.*

  Therefore, the biased exponent E is $10110110_2 = 182_{10}$, which means that the quantum exponent of x is $182 - 101 = 71$, and the integral significand of x is

  $$100101101001101111000011_2 = 9870275_{10}.$$

- *Therefore, x is equal to*

  $$-9870275 \times 10^{71} = -9.870275 \times 10^{77}.$$

3.4.4 Larger formats

The IEEE 754-2008 standard also specifies larger interchange formats for widths that are multiples of 32 bits of at least 128 bits. Their parameters are given in Table 3.21, and examples are given in Tables 3.22 and 3.23. This allows one to define "big" (yet, fixed) precisions. A format is fully defined from its radix (2 or 10) and size: the various parameters (precision, $e_{\min}$, $e_{\max}$, bias, etc.) are derived from them, using the formulas given in Table 3.21. Hence, binary1024 or decimal512 will mean the same thing on all platforms.

3.4.5 Extended and extendable precisions

Beyond the interchange formats, the IEEE 754-2008 standard partially specifies the parameters of possible *extended precision* and *extendable precision* formats. These formats are optional, and their binary encoding is not specified.

- An *extended precision format* extends a basic format with a wider precision and range, and is either language defined or implementation defined. The constraints on these wider precisions and ranges are given by Table 3.24. The basic idea behind these formats is that they should be used to carry out intermediate computations, in order to return a final result in the associated basic formats. The wider precision makes it possible to get a result that will generally be more accurate than that obtained with the basic formats only, and the wider range will drastically limit the cases of "apparent under/overflow" (that is, cases where there is an underflow or overflow in an intermediate result, whereas the final value would have been representable).

Parameter	Binary$k$ format	Decimal$k$ format
	(k is a multiple of 32)	
k	≥ 128	≥ 32
p	$k - \lfloor 4\log_2(k) \rceil + 13$	$9 \times \frac{k}{32} - 2$
t	$p - 1$	$(p - 1) \times 10/3$
w	$k - t - 1$	$k - t - 6$
$e_{\max}$	$2^{w-1} - 1$	$3 \times 2^{w-1}$
$e_{\min}$	$1 - e_{\max}$	$1 - e_{\max}$
b	$e_{\max}$	$e_{\max} + p - 2$

Table 3.21: Parameters of the interchange formats. $\lfloor u \rceil$ is u rounded to the nearest integer, t is the trailing significand width, w is the width of the exponent field for the binary formats, and the width of the combination field minus 5 for the decimal formats, and b is the exponent bias [187], (©IEEE, 2008, with permission).

Format	p	t	w	$e_{\min}$	$e_{\max}$	b
binary256	237	236	19	-262142	$+262143$	262143
binary1024	997	996	27	-67108862	$+67108863$	67108863

Table 3.22: Parameters of the binary256 and binary1024 interchange formats deduced from Table 3.21. Variables p, t, w, $e_{\min}$, $e_{\max}$, and b are the precision, the trailing significant field length, the exponent field length, the minimum exponent, the maximum exponent, and the exponent bias, respectively.

- An *extendable precision format* is a format whose precision and range are defined under user or program control. The standard says that language standards supporting extendable precision shall allow users to specify p and $e_{\max}$ (or, possibly, p only with constraints on $e_{\max}$), and define $e_{\min} = 1 - e_{\max}$.

3.4.6   Attributes

The proposed revision of the standard defines *attributes* as parameters, attached to a program block, that specify some of its numerical and exception semantics. The availability of *rounding direction attributes* is mandatory, whereas the availability of *alternate exception-handling attributes*, *preferred width attributes*, *value-changing optimization attributes*, and *reproducibility attributes* is recommended only. Language standards must provide for constant

Format	p	t	$w+5$	$e_{\max}$	b
decimal256	70	230	25	+1572864	1572932
decimal512	142	470	41	+103079215104	103079215244

Table 3.23: Parameters of the decimal256 and decimal512 interchange formats deduced from Table 3.21. $e_{\min}$ (not listed in the table) equals $1 - e_{\max}$. Variables $p, t, w, e_{\min}, e_{\max}$, and b are the precision, the combination field length, the exponent field length, the minimum exponent, the maximum exponent, and the exponent bias, respectively.

Parameter	Extended formats associated with:				
	binary32	binary64	binary128	decimal64	decimal128
$p \geq$	32	64	128	22	40
$e_{\max} \geq$	1023	16383	65535	6144	24576
$e_{\min} \leq$	-1022	-16382	-65534	-6143	-24575

Table 3.24: Extended format parameters in the IEEE 754-2008 standard [187] (©IEEE, 2008, with permission).

specification of the attributes, and should also allow for dynamic-mode specification of them.

Rounding direction attributes

- The *directed rounding attributes* correspond to the directed rounding modes of IEEE 754-1985: the roundTowardPositive attribute corresponds to the round-toward $+\infty$ mode of IEEE 754-1985, the round-TowardNegative attribute corresponds to the round-toward $-\infty$ mode, and the roundTowardZero attribute corresponds to the round-toward-zero mode.

- Concerning rounding to nearest, the situation is somewhat different. There are two *rounding direction attributes to nearest*, which differ in the way of handling the case when an exact result is halfway between two consecutive floating-point numbers:

  - roundTiesToEven attribute: if the two nearest floating-point numbers bracketing the exact result are equally near, the one whose least significant significand digit is even is delivered. This corresponds to the *round-to-nearest-even mode* of IEEE 754-1985 (in binary) and IEEE 854-1987. The case where these floating-point numbers both have an odd least significant significand digit (this can occur in precision 1 only, possibly when converting a number

such as 9.5 into a decimal string for instance) has been forgotten in the standard, but for the next revision, it has been proposed[13] to deliver the one larger in magnitude;

– roundTiesToAway attribute: in the same case as above, the value whose magnitude is larger is delivered.

For instance, in the decimal64 format ($p = 16$), if the exact result of some arithmetic operation is 1.2345678901234565, then the returned result should be 1.234567890123456 with the roundTiesToEven attribute, and 1.234567890123457 with the roundTiesToAway attribute.

The standard requires that an implementation (be it binary or decimal) provide the roundTiesToEven and the three directed rounding attributes. A *decimal* implementation must also provide the roundTiesToAway attribute (this is not required for binary implementations).

Having roundTiesToEven as the default rounding direction attribute is mandatory for binary implementations and recommended for decimal implementations. Whereas roundTiesToEven has several advantages (see [222]), roundTiesToAway is useful for some accounting calculations. This is why it is required for radix-10 implementations only, the main use of radix 10 being financial calculations. For instance, the European Council Regulation No. 1103/97 of 17 June 1997 on certain provisions relating to the introduction of the Euro sets out a number of rounding and conversion rules. Among them,

If the application of the conversion rate gives a result which is exactly half-way, the sum shall be rounded up.

Alternate exception-handling attributes

It is recommended (yet not required) that language standards define means for programmers to possibly associate alternate exception-handling attributes with a block. The alternate exception handlers will define lists of exceptions (invalid operation, division by zero, overflow, underflow, inexact, all exceptions) and specify what should be done when each of these exceptions is signaled. If no alternate exception-handling attribute is associated with a block, the exceptions are treated as explained in Section 3.4.10 (default exception handling).

Preferred width attributes

Consider an expression of the form

$$((a + b) \times c + (d + e)) \times f,$$

---

[13]See http://speleotrove.com/misc/IEEE754-errata.html.

where a, b, c, d, e, and f are floating-point numbers, represented in the same radix, but possibly with different formats. Variables a, b, c, d, e, and f are *explicit*, but during the evaluation of that expression, there will also be *implicit* variables; for instance, the result r_1 of the calculation of $a + b$, and the result r_2 of the calculation of $r_1 \times c$. When more than one format is available on the considered system, an important question arises: *In which format should be represented these intermediate values?* That point was not very clear in IEEE 754-1985. Many choices are possible for the "destination width" of an implicit variable. For instance:

- one might prefer to always have these implicit variables in the largest format provided in hardware. This choice will generally lead to more accurate computations (although it is quite easy to build counterexamples for which this is not the case);

- one might prefer to clearly specify a destination format. If that format is available on all used platforms, this will increase the portability of the program being written;

- one might require the implicit variables to be in the same format as the operands (and, if the operands are of different formats, to be of the widest format among the operands). This also will improve the portability of programs and will ease the use of smart algorithms such as those presented in Chapters 4, 5, and 6.

The revised standard recommends (yet does not require) that the following preferredWidthNone and preferredWidthFormat attributes should be defined by language standards.

preferredWidthNone attribute: When the user specifies a preferredWidthNone attribute for a block, the destination width of an operation is the maximum of the operand widths.

preferredWidthFormat attributes: When the user specifies a preferredWidthFormat attribute for a block, the destination width is the maximum of the width of the preferredWidthFormat and the operand widths.

Value-changing optimization attributes

Some optimizations (e.g., generation of FMAs, use of distributive and associative laws) can enhance performance in terms of speed, and yet seriously hinder the portability and reproducibility of results. Therefore, it makes sense to let the programmer decide whether to allow them or not. The value-changing optimization attributes are used in this case. The standard recommends that language standards should clearly define what is called

the "literal meaning" of the source code of a program (that is, the order of the operations and the destination formats of the operations). By default, the implementations should preserve the literal meaning. Language standards should define attributes for allowing or disallowing value-changing optimizations such as:

- applying relations such as $x \cdot y + x \cdot z = x \cdot (y + z)$ (distributivity), or $x + (y + z) = (x + y) + z$ (associativity);

- using FMAs for replacing, e.g., an expression of the form $a \cdot b + c \cdot d$ by $\mathrm{FMA}(a, b, c \cdot d)$;

- using larger formats for storing intermediate results.

Reproducibility attributes

The standard requires that conforming language standards should define ways of expressing when reproducible results are required. To get reproducible results, the programs must be translated into an unambiguous sequence of reproducible operations in reproducible formats. As explained in the standard [187], when the user requires reproducible results:

- the execution behavior must preserve what the standard calls the *literal meaning* of the source code;[14]

- conversions from and to external character strings must not bound the value of the maximum precision H (see Section 3.4.9) of these strings;

- when the reproducibility of some operation is not guaranteed, the user must be warned;

- only default exception handling is allowed.

3.4.7 Operations specified by the standard

Preferred exponent for arithmetic operations in the decimal format

Let $Q(x)$ be the quantum exponent of a floating-point number x. Since some numbers in the decimal format have several possible representations (the set of their representations is a *cohort*), the standard specifies for each operation which exponent is preferred for representing the result of a calculation. The rule to be followed is:

- if the result of an operation is inexact, the cohort member of smallest exponent is used;

---

[14]This implies that the language standards must specify what that literal meaning is: order of operations, destination formats of operations, etc.

- if the result of an operation is exact, then if the result's cohort includes a member with the preferred exponent (see below), that very member is returned; otherwise, the member with the exponent closest to the preferred exponent is returned.

The preferred quantum exponents for the most common operations are:

- $x + y$ and $x - y$: $\min(Q(x), Q(y))$;

- $x \times y$: $Q(x) + Q(y)$;

- x/y: $Q(x) - Q(y)$;

- FMA(x, y, z) (i.e., $xy + z$ using an FMA): $\min(Q(x) + Q(y), Q(z))$;

- $\sqrt{x}$: $\lfloor Q(x)/2 \rfloor$.

scaleB and logB

For designing fast software for the elementary functions, or for efficiently scaling variables (for instance, to write robust code for computing functions such as $\sqrt{x^2 + y^2}$), it is sometimes very useful to have functions $x \cdot \beta^n$ and $\lfloor \log_\beta |x| \rfloor$, where β is the radix of the floating-point system, n is an integer, and x is a floating-point number. This is the purpose of the functions scaleB and logB:

- scaleB(x, n) is equal to $x \cdot \beta^n$, correctly rounded[15] (following the rounding direction attribute);

- when x is finite and nonzero, logB(x) equals $\lfloor \log_\beta |x| \rfloor$. When the output format of logB is a floating-point format, logB(NaN) is NaN, logB$(\pm\infty)$ is $+\infty$, and logB(± 0) is $-\infty$.

Operations with NaNs

We have seen in Sections 3.4.2 and 3.4.3 that in the binary interchange formats, the $p - 2$ least significant bits of a NaN are not defined, and that in the decimal interchange formats, the trailing significand bits of a NaN are not defined. These bits can be used for encoding the *payload* of the NaN, i.e., some information that can be transmitted through the arithmetic operation for diagnosis purposes. To preserve that diagnosis information, it is required that for an operation with quiet NaN inputs, other than minimum or maximum operations, the returned result should be one of these input NaNs. Also, the sign of a NaN is not interpreted.

---

[15] In most cases, $x \cdot \beta^n$ is exactly representable so that there is no rounding at all, but requiring correct rounding is the simplest way of defining what should be returned if the result is outside the normal range.

Miscellaneous

The standard defines many very useful operations, see [187]. Examples are nextUp(x) (smallest floating-point number in the format of x that is greater than x), maxNum(x, y) (maximum of x and y), and class(x) (tells if x is a signaling NaN, a quiet NaN, $-\infty$, a negative normal number, a negative subnormal number, -0, $+0$, a positive subnormal number, a positive normal number, or $+\infty$), etc.

3.4.8 Comparisons

Floating-point data represented in different formats specified by the standard must be comparable *if these formats have the same radix*: the standard does not require that comparing a decimal and a binary number should be possible without a preliminary conversion.[16] Exactly as in IEEE 754-1985, four relations are possible: *less than, equal, greater than,* and *unordered,* and a comparison is delivered either as one of these four relations, or as a Boolean response to some predicate that gives the desired comparison.

3.4.9 Conversions

Concerning input and output conversions (that is, conversions between an external decimal or hexadecimal character sequence and an internal binary or decimal format), the new standard has requirements that are much stronger than those of IEEE 754-1985. They are described as follows.

1. Conversions between an external decimal character sequence and a supported decimal format: Input and output conversions are correctly rounded (according to the applicable rounding direction).

2. Conversions between an external hexadecimal character sequence and a supported binary format: Input and output conversions are also correctly rounded (according to the applicable rounding direction), but such conversions are optional. They have been specified to allow any binary number to be represented exactly by a finite character sequence.

3. Conversions between an external decimal character sequence and a supported binary format: first, for each supported binary format, define a value p_{10} as the minimum number of decimal digits in the decimal external character sequence that allows for an error-free write-read

---

[16]Such comparisons appear extremely rarely in programs designed by sensible beings, and would be very tricky to implement without preliminary conversion. Also, if we really need such a comparison, we do not lose much information by performing a preliminary conversion. Assume that the binary and decimal numbers to be compared are x_2 (in format F_2) and y_{10} (in format F_{10}). Define y_2 as y_{10} correctly rounded to format F_2. Then $x_2 \geq y_{10}$ implies $x_2 \geq y_2$, and $x_2 \leq y_{10}$ implies $x_2 \leq y_2$.

format	binary32	binary64	binary128
p_{10}	9	17	36

Table 3.25: Minimum number of decimal digits in the decimal external character sequence that allows for an error-free write-read cycle, for the various basic binary formats of the standard. See Section 2.7 page 40 for further explanation.

cycle, as explained in Section 2.7. Table 3.25, which gives the value of p_{10} from the various basic binary formats of the standard, is directly derived from Table 2.3 (page 44).

Then, define a value H so that H is preferably unbounded, and in any case, H is larger than or equal to 3 plus the largest value of p_{10} for all supported binary formats.

The conversions must be correctly rounded to and from external character sequences with any number of significant digits between 1 and H (which implies that these conversions must always be correctly rounded if H is unbounded).

For output conversions, if the external decimal format has more than H significant digits, then the binary value is correctly rounded to H decimal digits and trailing zeros are appended to fill the output format. For input conversions, if the external decimal format has more than H significant digits, then the internal binary number is obtained by first correctly rounding the value to H significant digits (according to the applicable rounding direction), then by correctly rounding the resulting decimal value to the target binary format (with the applicable rounding direction). In the directed rounding directions, these rules allow intervals to be respected.

More details are given in the standard [187].

3.4.10 Default exception handling

The revised standard supports the same five exceptions listed in Sections 2.3 and 3.1.5, with minor differences for the underflow.

Invalid operation

This exception is signaled each time there is no satisfactory way of defining the result of some operation. The default result of such an operation is a quiet NaN, and it is recommended that its payload contain some diagnostic information. The operations that lead to an invalid operation exception are:

- an operation on a signaling NaN (except for some conversions);

- a multiplication of the form $0 \times \infty$ or $\infty \times 0$;

- an FMA of the form FMA$(0, \infty, x)$ (i.e., $0 \times \infty + x$) or FMA$(\infty, 0, x)$, unless x is a quiet NaN (in that last case, whether the invalid operation exception is signaled is implementation defined);

- additions/subtractions of the form $(-\infty) + (+\infty)$ or $(+\infty) - (+\infty)$;

- FMAs that lead to the subtraction of infinities of the same sign (e.g., FMA$(+\infty, -1, +\infty)$);

- divisions of the form $0/0$ or ∞/∞;

- remainder$(x, 0)$, where x is not a NaN;

- remainder(∞, y), where y is not a NaN;

- conversion of a floating-point number x to an integer, where x is $\pm\infty$, or a NaN, or when the result would lie outside the range of the chosen integer format;

- comparison using unordered-signaling predicates (called in the standard compareSignalingEqual, compareSignalingGreater, compareSignalingGreaterEqual, compareSignalingLess, compareSignalingLessEqual, compareSignalingNotEqual, compareSignalingNotGreater, compareSignalingLessUnordered, compareSignalingNotLess, and compareSignalingGreaterUnordered), when the operands are unordered;

- logB(x) where x is NaN or ∞;

- logB(0) when the output format of logB is an integer format (when it is a floating-point format, the value to be returned is $-\infty$).

Division by zero

The words "division by zero" are misleading, since this exception is signaled whenever an *exact* infinite result is obtained from an operation on *finite* operands. The most frequent case, of course, is the case of a division by zero, but this can also appear, e.g., when computing the logarithm of zero or the arctanh of 1. An important case is logB(0) when the output format of logB is a floating-point format.

Overflow

Let us call an *intermediate* result what would have been the rounded result if the exponent range were unbounded. The overflow exception is signaled when the absolute value of the intermediate result is strictly larger than the

largest finite number $\Omega = (\beta - \beta^{1-p}) \cdot \beta^{e_{\max}}$. When an overflow occurs, the returned result depends on the rounding direction attribute:

- it will be $\pm\infty$ with the two "round-to-nearest" attributes, namely roundTiesToEven and roundTiesToAway, with the sign of the intermediate result;

- it will be $\pm\Omega$ with the roundTowardZero attribute, with the sign of the intermediate result;

- it will be $+\Omega$ for a positive intermediate result and $-\infty$ for a negative one with the roundTowardNegative attribute;

- it will be $-\Omega$ for a negative intermediate result and $+\infty$ for a positive one with the roundTowardPositive attribute.

Furthermore, the overflow flag is raised and the inexact exception is signaled. It is important to understand three consequences of these rules:

- with the two "round-to-nearest" attributes, if the absolute value of the exact result of an operation is greater than or equal to

$$\beta^{e_{\max}} \cdot \left(\beta - \frac{1}{2}\beta^{1-p} \right) = \Omega + \frac{1}{2}\operatorname{ulp}(\Omega),$$

  then an infinite result is returned, which is not what one could expect from a naive interpretation of the words "round to nearest";

- "overflow" is *not* equivalent to "infinite result returned";

- with the roundTowardZero attribute, "overflow" is *not* equivalent to "$\pm\Omega$ is returned": if the absolute value of the exact result of some operation is larger than or equal to Ω, and strictly less than $\beta^{e_{\max}}$, then $\pm\Omega$ is returned, and yet there is no overflow.

Underflow

The underflow exception is signaled when a nonzero result whose absolute value is strictly less than $\beta^{e_{\min}}$ is computed.

- For *binary formats*, unfortunately, there remains some ambiguity in the revised standard[17] (the same as in IEEE 754-1985). See the *Note on underflow* in Section 2.1, page 18, for more explanation. The underflow can be signaled either *before rounding*, that is, when the absolute value of the exact result is nonzero and strictly less than $2^{e_{\min}}$, or *after rounding*,

---

[17]This unfortunate choice probably results from the desire to keep existing implementations conforming to the standard.

that is, when the absolute value of a nonzero result computed as if the exponent range were unbounded is strictly less than $2^{e_{\min}}$. In rare cases, this can make a difference, for instance, when computing

$$\text{FMA}\left(-\beta^{e_{\min}}, \beta^{-p-1}, \beta^{e_{\min}}\right)$$

in rounding to nearest, an underflow will be signaled if this is done before rounding, but not if it is done after rounding.

- For *decimal formats*, there is no ambiguity and the underflow result is signaled *before rounding*, i.e., when the absolute value of the exact result is nonzero and strictly less than $10^{e_{\min}}$.

The result is always correctly rounded: the choice (in the binary case) of how the underflow is detected (that is, before or after rounding) has no influence on the delivered result.

In case of underflow, if the result is inexact, then the underflow flag is raised and the inexact exception is signaled. If the result is exact, then the underflow flag is *not* raised. This might sound strange, but this was a clever choice of the floating-point working group: the major use of the underflow flag is for warning that the result of some operation might not be very accurate—in terms of relative error. Thus, raising it when the operation is *exact* would be a needless warning. This should not be thought of as an *extremely rare* case: indeed, Theorem 3 page 124 shows that with any of the two round-to-nearest rounding direction attributes, whenever an addition or subtraction underflows, it is performed exactly.

Inexact

If the result of an operation differs from the exact result, then the inexact exception is signaled. The correctly rounded result is returned.

3.4.11 Recommended transcendental functions

The revised standard recommends (yet does not require) that the following functions should be correctly rounded: e^x, $e^x - 1$, 2^x, $2^x - 1$, 10^x, $10^x - 1$, $\ln(x)$, $\log_2(x)$, $\log_{10}(x)$, $\ln(1+x)$, $\log_2(1+x)$, $\log_{10}(1+x)$, $\sqrt{x^2+y^2}$, $1/\sqrt{x}$, $(1+x)^n$, x^n, $x^{1/n}$ (n is an integer), $\sin(\pi x)$, $\cos(\pi x)$, $\arctan(x)/\pi$, $\arctan(y/x)/\pi$, $\sin(x)$, $\cos(x)$, $\tan(x)$, $\arcsin(x)$, $\arccos(x)$, $\arctan(x)$, $\arctan(y/x)$, $\sinh(x)$, $\cosh(x)$, $\tanh(x)$, $\sinh^{-1}(x)$, $\cosh^{-1}(x)$, $\tanh^{-1}(x)$.

See Chapter 12 for an introduction to the various issues linked with the correct rounding of transcendental functions.

3.5   Floating-Point Hardware in Current Processors

Virtually all recent computers are able to support the IEEE 754-1985 standard efficiently through a combination of hardware and software.

3.5.1   The common hardware denominator

Current processors of desktop computers offer hardware double-precision (or binary64) operators for floating-point addition, subtraction, and multiplication and at least hardware assistance for division and square root. Peak performance is typically between 2 and 4 double-precision floating-point operations per clock cycle for $+$, $-$, and $\times$, with much slower division and square root [310]. However, most processors go beyond this common denominator and offer larger precision and/or faster operators. The following sections detail these extensions.

3.5.2   Fused multiply-add

The IBM Power/PowerPC, HP/Intel IA-64, and HAL/Fujitsu SPARC64 VI instruction sets define a *fused multiply-add* (FMA) instruction, which performs the operation $a \times b + c$ with only one rounding error with respect to the exact result (see Section 2.8 page 51).[18] This is actually a family of instructions that includes useful variations such as fused multiply-subtract.

These operations are compatible with the FMA defined by IEEE 754-2008. As far as this operator is concerned, IEEE 754-2008 standardized already existing practice.

The processors implementing these instruction sets (the IBM POWER family, PowerPC processors from various vendors, the HP/Intel Itanium family for IA-64) provide hardware FMA operators, with latencies comparable to classical $+$ and $\times$ operators. For illustration, the FMA latency is 4 cycles in Itanium2, 7 cycles on Power6, and both processors are capable of launching 2 FMA operations at each cycle.

There should soon be FMA hardware in the processors implementing the IA-32 instruction set: they are defined in the SSE5 extensions announced by AMD and in the AVX extensions announced by Intel.

3.5.3   Extended precision

The legacy x87 instructions of the IA-32 instruction set can operate on a *double-extended* precision format with 64 bits of significand and 15 bits of

---

[18]Warning! The instructions called `FMADDs` and so on from SPARC64 V, which share the same name and the same encoding with SPARC64 VI, are *not* real FMA instructions as they perform two roundings. [140, page 56]

exponent. The corresponding floating-point operators can be instructed to round to single, to double, or to double-extended.

The IA-64 instruction set also defines several double-extended formats, including one 80-bit format compatible with IA-32 and one 82-bit format with a 64-bit significand and a 17-bit exponent. The two additional exponent bits are designed to avoid intermediate overflows in certain computations on 80-bit operands.

As we write this book, no processors have full hardware support for the binary128 format. Some instruction sets (SPARC, POWER) have instructions operating on binary128 data, but on current hardware these instructions trap to software emulation.

3.5.4 Rounding and precision control

In most processor instruction sets, including IA-32 (whose floating-point specification was designed at the same time as the IEEE 754-1985 standard), both the rounding precision (single, double, double-extended if available) and the rounding direction attributes are specified via a global status/control register (called FPSR, *Floating-Point Status Register* on IA-32). This global register defines the behavior of all the floating-point instructions.

Changing the value of this control word, however, is extremely costly on recent processors. First, it requires at least one instruction. More importantly, it has the effect of flushing the floating-point pipeline: before launching any new floating-point instruction with the new value of the control register, all current floating-point instructions have to terminate with the old value. Unfortunately, some applications, such as interval arithmetic [284], need frequent rounding direction changes. This performance issue could not be anticipated in 1985, when processor architectures were not pipelined yet. It also affects most processor instruction sets designed in the 1980s and 1990s.

More recent instruction sets (most notably HP/Intel IA-64 [88] and Sun Microsystems' VIS extension to the SPARC instructions set [397]) permit changing the rounding direction attribute on a per-instruction basis without any performance penalty. Technically, the rounding direction attribute is defined in the instruction word, not in a global control register. Unfortunately, the rounding direction specification in the IEEE 754-1985 standard (and hence in the language standards that were later designed to implement it) reflects the notion of a global status word. This means in practice that per-instruction rounding specification cannot be accessed from current high-level languages in a standard, portable way. The new IEEE 754-2008 standard corrects this, but it will take some time to percolate in programming languages. This issue will be addressed in more detail in Chapter 7, Languages and Compilers.

Note that both with VIS and IA-64, it is still possible to specify that the rounding direction is taken from a global status word.

3.5.5 SIMD instructions

Most recent instruction sets also offer single instruction, multiple data (SIMD) instructions. One such instruction applies the same operation to all the elements of a vector of data kept in a wide register (64 to 256 bits currently).

Such a wide register can be considered as a vector of 8-bit, 16-bit, or 32-bit integers. SIMD instructions operating on such integer vectors are often referred to as multimedia instructions, because typical applications include image processing (where the color of one pixel may be defined by three 8-bit integers giving the intensity of the red, green, and blue components), and sound processing (where sound data is commonly represented by 16-bit sound samples).

A wide register may also be considered as a vector of 16-bit, 32-bit, or 64-bit floating-point numbers (the 16-bit formats are used for graphics and gaming, so that the binary16 format in IEEE 754-2008 actually standardized existing practice). Examples include AltiVec for the POWER/PowerPC family, and for the IA-32 instruction set, 3DNow! (64-bit vector), then SSE to SSE5 (128-bit vector), then AVX (256-bit vector). Each of these extensions comes with too many new instructions to be detailed here (not only arithmetic operations but also data movement inside a vector, and complex operations such as scalar products or sums of absolute values of differences). In addition, in the IA-32 family, some extensions have been announced by AMD and some by Intel, and both vendors eventually implement a common subset.

As we write this book, and as far as floating-point instructions are concerned, this common subset on 64-bit IA-32 processors is the SSE2 extension. It defines sixteen 128-bit registers (called XMM0 to XMM15), each of which can be considered either as a vector of four binary32 or as a vector of two binary64 numbers. SSE2 instructions are fully IEEE 754-1985 compliant.

The most recently announced extensions (SSE5 on the AMD side and AVX on the Intel side) include FMA instructions. As we write this book, no processors implement these extensions yet.

3.5.6 Floating-point on x86 processors: SSE2 versus x87

The Intel 8087 co-processor was a remarkable achievement when it was first produced. Twenty years later, however, the floating-point instructions it defined are showing their age.

- There are only 8 floating-point registers, and their organization as a stack leads to data movement inefficiencies.

- The risk of double rounding has been exposed in Section 3.3.1.

- The dynamic rounding precision can introduce bugs in modern software, which is almost always made up of several components (dynamic libraries, plug-ins). For instance, the following bug in Mozilla's

Javascript engine was discovered in 2006: if the rounding precision was reduced to single precision by some plug-in, then the `js_dtoa` function (double-to-string conversion) could overwrite memory, making the application behave erratically, e.g., crash. The cause was the loop exit condition being always false due to an unexpected floating-point error.[19]

• Another subtle issue has not been mentioned yet. The x87 FPSR register defines the rounding precision (the significand size) but not the exponent size, which is always 15 bits. Even when instructed to round to single precision, the floating-point unit (FPU) will signal overflows or underflows only for numbers out of the *double-extended* precision exponent range. True single-precision or double-precision overflow/underflow detection is performed only when writing the content of a floating-point register to memory. This two-step overflow/underflow detection can lead to subtle software problems, just like double rounding. It may be avoided only by writing all the results to memory, unless the compiler can prove in advance that there will be no overflows.

The SSE2 instructions were designed more recently. They may result in computations less accurate than with the legacy x87 instructions, as they do not offer extended precision. However, in addition to the obvious performance advantage, they are fully IEEE 754-1985 compliant, and they permit better reproducibility (thanks to the static rounding precision) and portability with other platforms. Extended precision is still possible since the legacy x87 unit is still available. Moreover x87-only meant that one had the almost exclusive choice between portability (the processor being configured in double precision) and (in general) better accuracy, with the risk of breaking some software components when changing the x87 rounding precision.

With both SSE2 and x87 available, SSE2 can be used for double-precision computations and the x87 can be configured in extended precision in order to have higher precision for platform-specific applications. This is the choice made by GCC and GNU/Linux for the `x86_64` architecture, as SSE2 is always available on this architecture.

3.5.7 Decimal arithmetic

As we write this book, only high-end processors from IBM include hardware decimal support. For illustration, each POWER6 processor core includes one decimal FPU capable of decimal128 computations, in addition to its two binary64 FMA units and its SIMD VMX unit capable of 4 parallel binary32 operations [123]. However, decimal operations are much slower than binary ones, with variable latencies of several tens of cycles (see Table 3.26) compared to the fixed latency of 7 cycles for the binary64 FMA.

---

[19]CVE-2006-6499 / `https://bugzilla.mozilla.org/show_bug.cgi?id=358569`

Cycles	decimal64 operands	decimal128 operands
addition/subtraction	9 to 17	11 to 19
multiplication	$19 + N$	$21 + 2N$
division	82	154

Table 3.26: Execution times of decimal operations on POWER6, from [123]. N is the number of digits in the first operand, excluding leading zeros.

3.6 Floating-Point Hardware in Recent Graphics Processing Units

Graphics processing units (GPUs), initially highly specialized integer only processors, have evolved in recent years towards more and more programmability and increasingly powerful arithmetic capabilities.

Binary floating-point units appeared in 2002-2003 in the GPUs of the two main vendors, ATI (with a 24-bit format in the R300 series) and Nvidia (with a 32-bit format in the NV30 series). In both implementations, addition and multiplication were incorrectly rounded: according to a study by Collange et al. [79], instead of rounding the exact sum or product, these implementations typically rounded a $p + 2$-bit result to the output precision of p bits.

Still, these units fueled interest in GPUs for general-purpose computing (GPGPU), as the theoretical floating-point performance of a GPU is up to two orders of magnitude that of a processor (at least in binary32). In parallel, programmability was also improved, notably to follow the evolution to version 10 of Microsoft's DirectX application programming interface. Specific development environments also appeared: first Nvidia's C-based CUDA, soon followed by the Khronos Group's OpenCL.

Between 2007 and 2009, both ATI (now AMD) and Nvidia introduced new GPU architectures with, among other things, improved floating-point support. Addition and multiplication are now correctly rounded, the precisions supported are binary32 and binary64, and the Nvidia GT200 architecture even offers correctly rounded binary64 FMAs—which are also supported by the OpenCL programming environment—with subnormal support and the four IEEE 754-1985 rounding modes. As we write this book, the latest AMD GPUs (RV770) are a bit behind, with no subnormal support, no FMA, and round to nearest only. It is worth mentioning that GPUs also include hardware acceleration of some elementary functions [311].

Full IEEE-754 compliance in hardware is still not there, though. Some issues are inherited from older architectures: for instance, on the GT200, binary32 support is less compliant than binary64, lacking subnormal support, the four rounding modes, and hardware FMA. Division and square root are still incorrectly rounded in some cases. Flags and exceptions are not supported.

However, as we write this book, it is clear that the trend in GPUs is no longer to sacrifice IEEE 754 compliance to performance. With the availability of IEEE 754-2008, this trend toward better quality floating-point in GPUs is expected to continue.

3.7 Relations with Programming Languages

The IEEE 754-1985 standard was targeted mainly to processor vendors and did not focus on programming languages. In particular, it did not define bindings (i.e., how the IEEE 754 standard is to be implemented in the language), such as the mapping between native types of the language and the formats of IEEE 754 and the mapping between operators/functions of the language and the operations defined by IEEE 754. The IEEE 754-1985 standard did not even deal with what a language should specify or what a compiler is allowed to do. This has led to many misinterpretations, with users often thinking that the processor will do exactly what they have written in the programming language. Chapter 7 will survey in more detail floating-point issues in mainstream programming languages.

The IEEE 754-2008 standard clearly improves the situation, mainly in its clauses 10 (*Expression evaluation*) and 11 (*Reproducible floating-point results*). For instance, it deals with the double-rounding problem (observed on x87, described in Section 3.5.6): "Language standards should disallow, or provide warnings for, mixed-format operations that would cause implicit conversion that might change operand values."

3.7.1 The Language Independent Arithmetic (LIA) standard

Before these shortcomings were tackled in the revision of the IEEE 754 standard, a new series of standards, *Language Independent Arithmetic*, had been developed to fill this gap. This series focuses on the properties of the arithmetic together with the language and its implementation, and does not define formats. It consists of three parts.

- LIA-1 (ISO/IEC 10967-1:1994) [189] defines properties of integer and floating-point arithmetic. It does not go beyond the four arithmetic operations ($+$, $-$, $\times$, and $/$). However the floating-point system can be rather general: the radix is any integer larger than or equal to 2 (but it *should* be even), the precision is any integer larger than or equal to 2, and the minimum and maximum exponents must satisfy some loose bounds. A parameter iec_559 can be set to **true** if the arithmetic conforms to the IEEE 754-1985 standard (a.k.a. IEC 559).

- LIA-2 (ISO/IEC 10967-2:2001) [191] adds support for elementary floating-point functions. It still has a parameter for IEEE 754-1985 support. For instance, LIA-2 lets ties in the rounding-to-nearest mode be implementation defined (it just requires them to be sign symmetric), but if the parameter is true, then ties must be rounded to even.

- LIA-3 (ISO/IEC 10967-3:2006) [193] adds support for complex integer and floating-point arithmetic, and complex elementary functions.

Each part provides examples of bindings for various programming languages (e.g., Ada, Basic, C, Common Lisp, FORTRAN, Modula-2, Pascal, PL/I). But it is up to languages to define the bindings, such as what the ISO C99 standard does in its LIA-1 compatibility annex.

3.7.2   Programming languages

The requirements of the IEEE 754 and LIA standards do not depend on particular languages. Languages can specify how they conform to some standard by providing *bindings*, which can depend on the implementation.

For instance, it has commonly been believed that the `double` type of the ISO C language must correspond everywhere to the double-precision/binary64 binary format of the IEEE 754 standard, but this property is implementation defined, and behaving differently is not a bug. Indeed the *destination* (as defined by the IEEE 754 standard) does not necessarily correspond to the C floating-point type associated with the value, and a C implementation must provide a macro `FLT_EVAL_METHOD`, which determines how operations and constants are evaluated. This is the reason why both implementations mentioned in Section 3.5.6 are valid (assuming the value of `FLT_EVAL_METHOD` is correct).

The consequences are that one can get different results on different platforms. But even when dealing with a single platform, one can also get unintuitive results, as shown in Goldberg's article with the addendum *Differences Among IEEE 754 Implementations* [148] or in Chapter 7. More details will be given in this chapter. But the bottom line is that the reader should be aware that a language will not necessarily follow standards as expected *a priori*. Implementing the algorithms given in this book may require special care in some environments (languages, compilers, platforms, and so on), at least until the *reproducibility attributes* from the IEEE 754-2008 standard (see Section 3.4.6) are supported.

3.8   Checking the Environment

Checking a floating-point environment (for instance, to make sure that some compiler optimization option is compliant with one of the IEEE standards)

may be important for critical applications. Circuit manufacturers increasingly use formal proofs to make sure that their arithmetic algorithms are correct [283, 355, 356, 357, 169, 170]. Also, when the algorithms used by some environment are known, it is possible to design test vectors that allow one to explore every possible branching. Typical examples are methods for making sure that every element of the table of an SRT division or square root algorithm is checked. Checking the environment is more difficult for the end user, who generally does not have any access to the algorithms that have been used. When we check some environment as a "black box" (that is, without knowing the code, or the algorithms used in the circuits) there is no way of being absolutely sure that the environment will *always* follow the standards. Just imagine a buggy multiplier that always returns the right result but for one couple of input operands. The only way of detecting that would be to check all possible inputs, which, in a pinch, is possible in the binary32 format, but certainly not in the binary64 or binary128 formats. This is not pure speculation: in single precision, the divider of the first version of the Pentium circuit would produce a wrong quotient with probability around 2.5×10^{-11} [290, 122], assuming random inputs.

Since the 1980s, various programs have been designed for determining the basic parameters of a floating-point environment and assessing its quality.

3.8.1 MACHAR

MACHAR was a program, written in FORTRAN by W. Cody [78], whose purpose was to determine the main parameters of a floating-point format (radix, "machine epsilon," etc.). This was done using algorithms similar to the one we give in Section 4.1.1 for finding the radix β of the system. Now, it is interesting for historical purposes only.

In their book on elementary functions, Cody and Waite [75] also gave methods for estimating the quality of an elementary function library. Their methods were based on mathematical identities such as

$$\sin(3x) = 3\sin(x) - 4\sin^3(x). \tag{3.4}$$

These methods were useful at the time they were published. And yet, they can no longer be used with current libraries. Recent libraries are either correctly rounded or have a maximal error close to $\frac{1}{2}$ ulp. Hence, they are far more accurate than the methods that are supposed to check them.

3.8.2 Paranoia

Paranoia [215] is a program originally written in Basic by W. Kahan, and translated to Pascal by B.A. Wichmann and to C by T. Sumner and D. Gay in the 1980s, to check the behavior of floating-point systems. It finds the main properties of a floating-point system (such as its precision and its exponent

range), and checks if underflow is gradual, if the arithmetic operations are properly implemented, etc. It can be obtained at `http://www.netlib.org/paranoia/`. Below is an example of Paranoia's output. The program was run on the following environment:

- hardware: Intel 32 bits Xeon;

- operating system: Debian/Linux Etch;

- compiler: GCC 4.1.2 20061115 (as per "gcc -v");

- compilation command line: `gcc -o paranoia paranoia.c -lm`.

The produced code uses the 387 (i.e., double-extended precision) registers (see below). Notice that no optimization switch was set (default is `-O0`). Using more aggressive compilation options (`-O1`, `-O2`, etc.) only leads to more failures, defects, and flaws. Since Paranoia is rather verbose, we suppressed some parts of the output.[20] We interleave comments in Paranoia's output.

```
(...)

Running this program should reveal these characteristics:
     Radix = 1, 2, 4, 8, 10, 16, 100, 256 ...
     Precision = number of significant digits carried.
     U2 = Radix/Radix^Precision = One Ulp
(OneUlpnit in the Last Place) of 1.000xxx .
     U1 = 1/Radix^Precision = One Ulp of numbers a little
     less than 1.0 .
     Adequacy of guard digits for Mult., Div. and Subt.
     Whether arithmetic is chopped, correctly rounded, or
     something else
for Mult., Div., Add/Subt. and Sqrt.
     Whether a Sticky Bit used correctly for rounding.
     UnderflowThreshold = an underflow threshold.
     E0 and PseudoZero tell whether underflow is abrupt,
     gradual, or fuzzy.
     V = an overflow threshold, roughly.
     V0  tells, roughly, whether  Infinity  is represented.
     Comparisons are checked for consistency with subtraction
and for contamination with pseudo-zeros.
     Sqrt is tested.  Y^X is not tested.

(...)

The program attempts to discriminate among
   FLAWs, like lack of a sticky bit,
   Serious DEFECTs, like lack of a guard digit, and
   FAILUREs, like 2+2 == 5 .
```

---

[20]They are indicated by "(...)".

```
Failures may confound subsequent diagnoses.

(...)

BASIC version of this program (C) 1983 by Prof. W. M. Kahan;
see source comments for more history.

(...)

Program is now RUNNING tests on small integers:
-1, 0, 1/2, 1, 2, 3, 4, 5, 9, 27, 32 & 240 are O.K.

Searching for Radix and Precision.
Radix = 2.000000 .
Closest relative separation found is U1 = 1.1102230e-16 .

(...)

The number of significant digits of the Radix is 53.000000 .
```

Paranoia detects that the arithmetic used has radix 2, precision 53 (it is the IEEE 754-1985 double-precision format).

```
Some subexpressions appear to be calculated extra
precisely with about 11 extra B-digits, i.e.
roughly 3.31133 extra significant decimals.
That feature is not tested further by this program.
```

Interestingly enough, Paranoia detects that subexpressions are evaluated with 11 extra binary digits. This corresponds to the 64-bit double-extended precision significands of the 387 registers. This message disappears when the -march=pentium4 -mfpmath=sse switches are used on the gcc command line as, instead, they trigger the SSE2 64-bit floating-point registers with 53-bit significands. More information on compilation options will be given in Chapter 7.

On the corresponding 64-bit platform (an Intel Core 2 Quad processor, matching 64-bit versions of the system and the compiler), defaults are different: one has to use the -mfpmath=387 switch to get the same message about extra bits.

```
(...)

Subtraction appears to be normalized, as it should be.
Checking for guard digit in *, /, and -.
    *, /, and - appear to have guard digits, as they should.
Checking rounding on multiply, divide and add/subtract.
* is neither chopped nor correctly rounded.
/ is neither chopped nor correctly rounded.
Addition/Subtraction neither rounds nor chops.
Sticky bit used incorrectly or not at all.
```

```
FLAW:  lack(s) of guard digits or failure(s) to correctly
round or chop (noted above) count as one flaw in the
final tally below.
```

This FLAW, certainly due to double roundings, disappears when the
-march=pentium4 -mfpmath=sse switches are used on the gcc command line
as, instead, they trigger the SSE2 64-bit floating-point registers with a 53-bit
significand.

```
Does Multiplication commute?  Testing on 20 random pairs.
    No failures found in 20 integer pairs.


Running test of square root(x).
Testing if sqrt(X * X) == X for 20 Integers X.
Test for sqrt monotonicity.
sqrt has passed a test for Monotonicity.
Testing whether sqrt is rounded or chopped.
Square root is neither chopped nor correctly rounded.
Observed errors run from -5.0000000e-01 to 5.0000000e-01 ulps.
```

Note that the errors, although larger than $1/2$ ulp (otherwise, square root
would be correctly rounded!) are so close to $1/2$ ulp that, when printed in
Paranoia's output format, they cannot be differentiated from $1/2$ ulp. This
warning disappears when moving to the corresponding 64-bit platform (an
Intel Core 2 Quad processor, with 64-bit versions of the system and the com-
piler, but unchanged compiler switches).

```
Testing powers Z^i for small Integers Z and i.
... no discrepancies found.


Seeking Underflow thresholds UfThold and E0.
Smallest strictly positive number found is E0 = 4.94066e-324 .
Since comparison denies Z = 0, evaluating (Z + Z) / Z should
be safe.
What the machine gets for (Z + Z) / Z is
2.00000000000000000e+00 .
This is O.K., provided Over/Underflow has NOT just been
signaled.
Underflow is gradual; it incurs Absolute Error =
(roundoff in UfThold) < E0.
The Underflow threshold is 2.22507385850720188e-308,  below
which calculation may suffer larger Relative error than
merely roundoff.
Since underflow occurs below the threshold
UfThold = (2.00000000000000000e+00)^(-1.02200000000000000e+03)
only underflow should afflict the expression
(2.00000000000000000e+00) ^ (-2.04400000000000000e+03);
actually calculating yields: 0.00000000000000000e+00 .
This computed value is O.K.


Testing X^((X + 1) / (X - 1)) vs. exp(2) =
```

```
7.38905609893065218e+00 as X -> 1.
DEFECT:  Calculated 7.38905609548934539e+00 for
(1 + (-1.11022302462515654e-16) ^ (-1.80143985094819840e+16);
differs from correct value by -3.44130679508225512e-09 .
This much error may spoil financial
calculations involving tiny interest rates.
```

Again, this DEFECT disappears when moving to the corresponding 64-bit platform (an Intel Core 2 Quad processor, with 64-bit versions of the system and the compiler, but unchanged compiler switches).

```
Testing powers Z^Q at four nearly extreme values.
 ... no discrepancies found.
(...)
Searching for Overflow threshold:
This may generate an error.
Can 'Z = -Y' overflow?
Trying it on Y = -inf .
Seems O.K.
Overflow threshold is V  = 1.79769313486231571e+308 .
Overflow saturates at V0 = inf .
No Overflow should be signaled for
V * 1 = 1.79769313486231571e+308
nor for V / 1 = 1.79769313486231571e+308 .
Any overflow signal separating this * from the one
above is a DEFECT.
(...)

What message and/or values does Division by Zero produce?
This can interupt your program.  You can skip this part if
you wish.
Do you wish to compute 1 / 0?
O.K.

Do you wish to compute 0 / 0?
O.K.
(...)
The number of  DEFECTs  discovered =        1.
The number of  FLAWs  discovered =        1.

The arithmetic diagnosed may be Acceptable
despite inconvenient Defects.
```

3.8.3 UCBTest

UCBTest can be obtained at `http://www.netlib.org/fp/ucbtest.tgz`. It is a collection of programs whose purpose is to test certain difficult cases of the IEEE floating-point arithmetic. Paranoia is included in UCBTest. The "difficult cases" for multiplication, division, and square root (i.e., almost hardest-to-round cases: input values for which the result of the operation is very near

a breakpoint of the rounding mode) are built using algorithms designed by Kahan, such as those presented in [324].

3.8.4   TestFloat

J. Hauser designed a software implementation of the IEEE 754-1985 floating-point arithmetic. The package is named SoftFloat and can be downloaded at `http://www.jhauser.us/arithmetic/SoftFloat.html`. He also designed a program, TestFloat, aimed at testing whether a system conforms to IEEE 754-1985. TestFloat compares results returned by the system to results returned by SoftFloat.

3.8.5   IeeeCC754

UCBTest focuses on the precisions specified by the IEEE 754-1985 standard. Verdonk, Cuyt and Verschaeren [418, 419] present a new tool, IeeeCC754, acronym for *IEEE 754 Compliance Checker*, that is precision and range independent. It is based on a huge set of precision- and range-independent test vectors. It can be downloaded at `http://www.cant.ua.ac.be/old/ieeecc754.html`.

3.8.6   Miscellaneous

SRTEST is a FORTRAN program written by Kahan for checking implementation of SRT [125, 126] division algorithms. It can be accessed on Kahan's web page, at `http://www.cs.berkeley.edu/~wkahan/srtest/`. Some useful software, written by Nelson H.F. Beebe, can be found at `http://www.math.utah.edu/~beebe/software/ieee/`. MPCHECK is a program written by Revol, Pélissier, and Zimmermann. It checks mathematical function libraries (for correct rounding, monotonicity, symmetry, and output range). It can be downloaded at `http://www.loria.fr/~zimmerma/mpcheck/` or `https://gforge.inria.fr/projects/mpcheck/`.

Part II

Cleverly Using Floating-Point Arithmetic

Chapter 4

Basic Properties and Algorithms

I N THIS CHAPTER, we present some short yet useful algorithms and some basic properties that can be derived from specifications of floating-point arithmetic systems, such as the ones given in the various successive IEEE standards. Thanks to these standards, we now have an accurate definition of floating-point formats and operations. The behavior of a sequence of operations becomes at least partially[1] predictable (see Chapter 7 for more details on this). We therefore can build algorithms and proofs that use these specifications.

This also allows the use of formal proofs to verify pieces of mathematical software. For instance, Harrison uses HOL Light to formalize floating-point arithmetic [168, 171] and check floating-point trigonometric functions [169] for the Intel-HP IA-64 architecture. Russinoff [355] used the ACL2 prover to check the AMD-K7 floating-point multiplication, division, and square root instructions. Boldo, Daumas, and Théry use the Coq proof assistant to formalize floating-point arithmetic and prove properties of some arithmetic algorithms [33, 256].

4.1 Testing the Computational Environment

4.1.1 Computing the radix

The various parameters (radix, significand and exponent widths, rounding modes, etc.) of the floating-point arithmetic used in a computing system may strongly influence the result of a numerical program. Indeed, very simple and short programs that only use floating-point operations can find these parameters. An amusing example of this is the C program (Listing 4.1), given by Malcolm [146, 269], that returns the radix β of the floating-point system. It works if the active rounding mode is one of the four rounding modes of IEEE

---

[1]In some cases, for instance, intermediate calculations may be performed in a wider internal format. Some examples are given in Section 3.3.

754-1985 (or one of the rounding direction attributes of IEEE 754-2008 [187]).
It is important to make sure that a zealous compiler does not try to "simplify"
expressions such as $(A+1.0)-A$. See Chapter 7 for more information on how
languages and compilers handle floating-point arithmetic.

C listing 4.1 Malcolm's algorithm (Algorithm 4.1, see below), written in C.

```c
#include <stdio.h>
#include <math.h>

#pragma STDC FP_CONTRACT OFF

int main (void)
{
  double A, B;

  A = 1.0;
  while ((A + 1.0) - A == 1.0)
    A *= 2.0;
  B = 1.0;
  while ((A + B) - A != B)
    B += 1.0;
  printf ("Radix B = %g\n", B);
  return 0;
}
```

Let us describe the corresponding algorithm more precisely. Let ∘ be the
active rounding mode. The algorithm is

Algorithm 4.1 Computing the radix of a floating-point system.

$A \leftarrow 1.0$
$B \leftarrow 1.0$
while $\circ(\circ(A + 1.0) - A) = 1.0$ **do**
    $A \leftarrow \circ(2 \times A)$
end while
while $\circ(\circ(A + B) - A) \neq B$ **do**
    $B \leftarrow \circ(B + 1.0)$
end while
return B

Incidentally, this example shows that analyzing algorithms sometimes
depends on the whole specification of the arithmetic operations, and espe-
cially the fact that they are correctly rounded:

- If one assumes that the operations are exact, then one erroneously con-
  cludes that the first loop never ends (or ends with an error due to an
  overflow on variable A).

- If one tries to analyze this algorithm just by assuming that $\circ(x + y)$ is $(x + y)(1 + \epsilon)$ where $|\epsilon|$ is bounded by some tiny value, it is impossible to prove anything. For instance, $\circ(\circ(A + 1.0) - A)$ is just 1 plus some "noise."

And yet, assuming correctly-rounded operations, it is easy to show that the final value of B is the radix of the floating-point system being used, as we show now.

Proof. Define A_i as the value of A after the i-th iteration of the loop:

$$\textbf{while } \circ(\circ(A + 1.0) - A) = 1.0.$$

Let β be the radix of the floating-point system and p its precision. One easily shows by induction that if $2^i \leq \beta^p - 1$, then A_i equals 2^i exactly. In such a case, $A_i + 1 \leq \beta^p$, which implies that $\circ(A_i + 1.0) = A_i + 1$. Therefore, one deduces that $\circ(\circ(A_i + 1.0) - A_i) = \circ((A_i + 1) - A_i) = 1$. Hence, while $2^i \leq \beta^p - 1$, we stay in the first loop.

Now, consider the first iteration j, such that $2^j \geq \beta^p$. We have $A_j = \circ(2A_{j-1}) = \circ(2 \times 2^{j-1}) = \circ(2^j)$. Since $\beta \geq 2$, we deduce

$$\beta^p \leq A_j < \beta^{p+1}.$$

This implies that the floating-point successor of A_j is $A_j + \beta$. Therefore, depending on the rounding mode, $\circ(A_j + 1.0)$ is either A_j or $A_j + \beta$, which implies that $\circ(\circ(A_j + 1.0) - A_j)$ is either 0 or β. In any case, this value is different from 1.0, so we exit the first loop.

So we conclude that, at the end of the first **while** loop, variable A satisfies $\beta^p \leq A < \beta^{p+1}$.

Now, let us consider the second **while** loop. We have seen that the floating-point successor of A is $A + \beta$. Therefore, while $B < \beta$, $\circ(A + B)$ is either A or $A + \beta$, which implies that $\circ(\circ(A + B) - A)$ is either 0 or β. In any case, this value is different from B, which implies that we stay in the loop.

Now, as soon as $B = \beta$, $\circ(A + B)$ is exactly equal to $A + B$; hence, $\circ(\circ(A + B) - A) = B$. We therefore exit the loop when $B = \beta$. $\qquad\square$

4.1.2 Computing the precision

Algorithm 4.2, also introduced by Malcolm [269], is very similar to Algorithm 4.1 [269]. It computes the precision p of the floating-point system being used.

Algorithm 4.2 Computing the precision of a floating-point system. It requires the knowledge of the radix of the system, and that radix can be given by Algorithm 4.1.

Input: B (the radix of the FP system)
$i \leftarrow 0$
$A \leftarrow 1.0$
while $\circ(\circ(A + 1.0) - A) = 1.0$ **do**
   $A \leftarrow \circ(B \times A)$
   $i \leftarrow i + 1$
end while
return i

The proof is very similar to the proof of Algorithm 4.1, so we omit it.

Similar—yet more sophisticated—algorithms are used in inquiry programs such as Paranoia [215], which provide a means for examining your computational environment (see Section 3.8, page 111).

4.2   Exact Operations

Although most floating-point operations involve some sort of rounding, there are some cases when a single operation will be *exact*, i.e., without rounding error. Knowing these cases allows an experienced programmer to use them in critical algorithms. Typical examples of such algorithms are elementary function programs [95]. Many other examples will follow throughout this book.

What are the exact operations? The IEEE standards state that, if the infinitely-precise result of an arithmetic operation is a floating-point number, then this number should be returned, whatever the rounding mode. Therefore, the exact operations are those for which one may prove that the result belongs to the set of floating-point numbers of the considered format.

4.2.1   Exact addition

An important result, frequently used when designing or proving algorithms, is Sterbenz's lemma.

Lemma 2 (Sterbenz [392])**.** *In a radix-β floating-point system with subnormal numbers available, if x and y are finite floating-point numbers such that*

$$\frac{y}{2} \le x \le 2y,$$

then $x - y$ is exactly representable.

Sterbenz's lemma implies that, with any of the four rounding modes, if x and y satisfy the preceding conditions, then when computing $x - y$ in floating-point arithmetic, the obtained result is exact.

Proof. For reasons of symmetry, we can assume that $x \geq 0$, $y \geq 0$, and $y \leq x \leq 2y$. Let M_x and M_y be the integral significand of x and y, and e_x and e_y be their exponents (if x and y have several floating-point representations, we choose the ones with the smallest exponents). We have

$$x = M_x \times \beta^{e_x - p + 1},$$

and

$$y = M_y \times \beta^{e_y - p + 1},$$

with

$$\begin{cases} e_{\min} \leq e_x \leq e_{\max} \\ e_{\min} \leq e_y \leq e_{\max} \\ 0 \leq M_x \leq \beta^p - 1 \\ 0 \leq M_y \leq \beta^p - 1. \end{cases}$$

From $y \leq x$, we easily deduce $e_y \leq e_x$. Define $\delta = e_x - e_y$. We get

$$x - y = \left(M_x \beta^\delta - M_y \right) \times \beta^{e_y - p + 1}.$$

Define $M = M_x \beta^\delta - M_y$. We have

- $x \geq y$ implies $M \geq 0$;

- $x \leq 2y$ implies $x - y \leq y$, hence $M\beta^{e_y - p + 1} \leq M_y \beta^{e_y - p + 1}$; therefore,

$$M \leq M_y \leq \beta^p - 1.$$

Therefore, $x - y$ is equal to $M \times \beta^{e - p + 1}$ with $e_{\min} \leq e \leq e_{\max}$ and $|M| \leq \beta^p - 1$. This shows that $x - y$ is a floating-point number, which implies that it is exactly computed. $\qquad\square$

It is important to notice that, in our proof, the only thing we have shown is that $x - y$ is representable with an integral significand M whose absolute value is less than or equal to $\beta^p - 1$. We have not shown (and it is *not* possible to show) that it can be represented with an integral significand of absolute value larger than or equal to β^{p-1}. Indeed, $|x - y|$ can be less than $\beta^{e_{\min}}$. In such a case, the availability of subnormal numbers is required for Sterbenz's lemma to be applicable (it suffices to consider the example given by Figure 2.1 page 18).

A slightly more general result is that if the exponent of $x-y$ is less than or equal to the minimum of e_x and e_y, then the subtraction is exactly performed.

Sterbenz's lemma might seem strange to those who remember their early lectures on numerical analysis: It is common knowledge that subtracting numbers that are very near may lead to very inaccurate results. This kind of numerical error is called a *cancellation*, or a *catastrophic cancellation* when almost all digits of the result are lost. There is no contradiction: The subtraction of two floating-point numbers that are very near does not introduce any error in itself (since it is an exact operation), yet it *amplifies* a *pre-existing error*. Consider the following example in IEEE 754-1985 single-precision arithmetic and round-to-nearest mode:

- $A = 10000$ and $B = 9999.5$ (they are exactly representable);

- $C = \mathrm{RN}(1/10) = 13421773/134217728$;

- $A' = \mathrm{RN}(A + C) = 5120051/512$;

- $\Delta = \mathrm{RN}(A' - B) = 307/512 = 0.599609375$.

Sterbenz's Lemma implies that Δ is exactly equal to $A' - B$. And yet, the computation $A' = \mathrm{RN}(A + C)$ introduced some error: A' is slightly different from $A+C$. This suffices to make Δ a rather bad approximation to $(A+C)-B$, since $(A + C) - B \approx 0.6000000015$.

In this example, the subtraction $A' - B$ was errorless, but it somehow amplified the error introduced by the computation of A'.

Hauser [176] gives another example of exact additions. It shows, incidentally, that when a gradual underflow occurs (that is, the obtained result is less than $\beta^{e_{\min}}$ but larger than or equal to the smallest subnormal number $\alpha = \beta^{e_{\min}-p+1}$), this does not necessarily mean an inaccurate result.

Theorem 3 (Hauser). *If x and y are radix-β floating-point numbers, and if the number $\mathrm{RN}(x + y)$ is subnormal, then $\mathrm{RN}(x + y) = x + y$ exactly.*

Proof. It suffices to notice that x and y (as all floating-point numbers) are multiples of the smallest nonzero floating-point number $\alpha = \beta^{e_{\min}-p+1}$. Hence, $x + y$ is a multiple of α. If it is a subnormal number, then it is less than $\beta^{e_{\min}}$. This implies that it is exactly representable.   □

4.2.2   Exact multiplications and divisions

Some multiplications and divisions are exactly performed. A straightforward example is multiplication or division by a power of the radix: As soon as there is no overflow or underflow,[2] the result is exactly representable.

---

[2]Beware! We remind the reader that by "no underflow" we mean that the absolute value of the result (before or after rounding, this depends on the definition) is not less than the smallest *normal* number $\beta^{e_{\min}}$. When subnormal numbers are available, as requested by the IEEE standards, it is possible to represent smaller nonzero numbers, but with a precision that does not always suffice to represent the product exactly.

Another example is multiplication of numbers with known zero bits at the lower-order part of the significand. For instance, assume that x and y are floating-point numbers whose significands have the form

$$\underbrace{x_0.x_1x_2x_3\cdots x_{k_x}}_{k_x \text{ digits}}\ \underbrace{000\cdots 0}_{p-k_x \text{ zeros}}$$

for x, and

$$\underbrace{y_0.y_1y_2y_3\cdots y_{k_y}}_{k_y \text{ digits}}\ \underbrace{000\cdots 0}_{p-k_y \text{ zeros}}$$

for y. If $k_x + k_y \le p$ then the product of the significands of x and y fits in p radix-β digits. This implies that the product xy is exactly computed if no overflow nor underflow occurs. This property is at the heart of Dekker's multiplication algorithm (see Section 4.4.2, page 135). It is also very useful for reducing the range of inputs when evaluating elementary functions (see Section 11.1, page 379).

4.3   Accurate Computations of Sums of Two Numbers

Let a and b be radix-β precision-p floating-point numbers. Let s be $\mathrm{RN}(a+b)$, i.e., $a+b$ correctly rounded to the nearest precision-p floating-point number, with any choice here in case of a tie. It can easily be shown that, if the computation of s does not overflow, then the error of the floating-point addition of a and b, namely $t = (a+b) - s$, is exactly representable in radix β with p digits. It is important to notice that this property can be false with other rounding modes. For instance, in a radix-2 and precision-p arithmetic, assuming rounding toward $-\infty$, if $a = 1$ and $b = -2^{-3p}$, then

$$
\begin{aligned}
s &= \mathrm{RD}(a+b) = 0.\underbrace{111111\cdots 11}_{p}\\
&= 1 - 2^{-p},
\end{aligned}
$$

and

$$t - s = \underbrace{1.1111111111\cdots 11}_{2p}\times 2^{-p-1},$$

which cannot be exactly represented with precision p (it would require precision $2p$).

In the following sections, we present two algorithms for computing t. They are useful for performing very accurate sums of many numbers. They also are of interest for very careful implementation of mathematical functions [95].

4.3.1 The Fast2Sum algorithm

The Fast2Sum algorithm was introduced by Dekker [108] in 1971, but the three operations of this algorithm already appeared in 1965 as a part of a summation algorithm, called "Compensated sum method," due to Kahan [201] (Algorithm 6.6, page 192). The name "Fast-Two-Sum" seems to have been coined by Shewchuk [377].

Theorem 4 (Fast2Sum algorithm). *([108], and Theorem C of [222], page 236). Assume the floating-point system being used has radix $\beta \leq 3$, subnormal numbers available, and provides correct rounding with rounding to nearest.*

Let a and b be floating-point numbers, and assume that the exponent of a is larger than or equal to that of b (this condition might be difficult to check, but of course, if $|a| \geq |b|$, it will be satisfied). Algorithm 4.3 computes two floating-point numbers s and t that satisfy the following:

- *$s + t = a + b$ exactly;*

- *s is the floating-point number that is closest to $a + b$.*

Algorithm 4.3 The Fast2Sum algorithm [108].

$s \leftarrow \mathrm{RN}(a + b)$
$z \leftarrow \mathrm{RN}(s - a)$
$t \leftarrow \mathrm{RN}(b - z)$

(We remind the reader that $\mathrm{RN}(x)$ means x rounded to nearest.)

Notice that if a wider internal format is available (one more digit of precision is enough), and if the computation of z is carried on using that wider format, then the condition $\beta \leq 3$ is no longer necessary [108]. This may be useful when working with decimal arithmetic. The Fast2Sum algorithm is simpler when written in C, as all rounding functions are implicit, as one can see in Listing 4.2 (yet, it requires round-to-nearest mode, which is the default rounding mode).

In that program it is assumed that all the variables are declared to be of the same floating-point format, say all `float` or all `double`, and the system is set up to ensure that all the computations are done in this format. A compiler that is compliant with the C99 standard will not attempt to simplify these operations.

Let us now give Dekker's proof for this algorithm.

Proof. Let e_a, e_b, and e_s, be the exponents of a, b, and s. Let M_a, M_b, and M_s, be their integral significands, and let p be the precision of the floating-point format being used. We recall that the integral significands have absolute values less than $\beta^p - 1$.

C listing 4.2 Fast2Sum.

```c
/* fast2Sum.c */

#include <stdio.h>
#include <stdlib.h>

void fast2Sum(double a, double b, double *s, double *t)
{
  double dum;
  double z;
  /* Branching below may hinder performance */
  /* Suppress if we know in advance that a >= b */

  if (b > a) { dum = a; a = b; b = dum; }

  *s =  a + b;
   z = *s - a;
  *t =  b - z;
}

int main(int argc, char **argv)
{
  double a;
  double b;
  double s;
  double t;

  /* The inputs are read on the command line. */
  a = strtod(argv[1], NULL);
  b = strtod(argv[2], NULL);

  fprintf(stdout, "a = %1.16g\n", a);
  fprintf(stdout, "b = %1.16g\n", b);

  fast2Sum(a, b, &s, &t);

  printf("s = %1.16g\n", s);
  printf("t = %1.16g\n", t);

  return(0);
}
```

First, let us show that $s - a$ is exactly representable. Notice that, since $e_b \leq e_a$, s can be represented with an exponent less than or equal to $e_a + 1$. This comes from

$$a + b \leq 2(\beta^p - 1)\beta^{e_a - p + 1} \leq (\beta^p - 1)\beta^{e_a - p + 2}.$$

1. **If $e_s = e_a + 1$.**

   Define $\delta = e_a - e_b$. We have

   $$M_s = \left\lceil \frac{M_a}{\beta} + \frac{M_b}{\beta^{\delta + 1}} \right\rfloor,$$

   where $\lceil u \rfloor$ is the integer that is nearest to u (when u is an odd multiple of $1/2$, there are two integers that are nearest to u, and we choose the one that is even).

   Define $\mu = \beta M_s - M_a$. We easily find

   $$\frac{M_b}{\beta^\delta} - \frac{\beta}{2} \leq \mu \leq \frac{M_b}{\beta^\delta} + \frac{\beta}{2}.$$

   Since μ is an integer and $\beta \leq 3$, this gives

   $$|\mu| \leq |M_b| + 1.$$

   Therefore, since $|M_b| \leq \beta^p - 1$, either $|\mu| \leq \beta^p - 1$ or $|\mu| = \beta^p$. In both cases, since $s - a = \mu\beta^{e_a - p + 1}$, $s - a$ is exactly representable.[3]

2. **If $e_s \leq e_a$.**

   Define $\delta_1 = e_a - e_b$. We have

   $$a + b = \left(\beta^{\delta_1} M_a + M_b \right) \beta^{e_b - p + 1}.$$

   If $e_s \leq e_b$ then $s = a - b$, since $a + b$ is a multiple of $\beta^{e_b - p + 1}$, and s is obtained by rounding $a + b$ to the nearest multiple of $\beta^{e_s - p + 1} \leq \beta^{e_b - p + 1}$. This implies that $s - a = b$ is exactly representable. If $e_s > e_b$, then define $\delta_2 = e_s - e_b$. We have

   $$s = \left\lceil \beta^{\delta_1 - \delta_2} M_a + \beta^{-\delta_2} M_b \right\rfloor \beta^{e_s - p + 1},$$

   which implies

   $$\left(\beta^{-\delta_2} M_b - \frac{1}{2} \right) \beta^{e_s - p + 1} \leq s - a \leq \left(\beta^{-\delta_2} M_b + \frac{1}{2} \right) \beta^{e_s - p + 1}.$$

---

[3]When $|\mu| \leq \beta^p - 1$, $s - a$ is representable with exponent e_a, but not necessarily in normal form. This is why the availability of subnormal numbers is necessary.

Hence,

$$|s - a| \leq \left(\beta^{-\delta_2} |M_b| + \frac{1}{2} \right) \beta^{e_s - p + 1},$$

and $s - a$ is a multiple of $\beta^{e_s - p + 1}$, which gives $s - a = K \beta^{e_s - p + 1}$, with

$$|K| \leq \beta^{-\delta_2} |M_b| + \frac{1}{2} \leq \beta^p - 1,$$

which implies that $s - a$ is exactly representable.

Therefore, in all cases, $z = \text{RN}(s - a) = s - a$ exactly.

Second, let us show that $b - z$ is exactly representable. From $e_a \geq e_b$, we deduce that a and b are both multiples of $\beta^{e_b - p + 1}$. This implies that s (obtained by rounding $a + b$), $s - a$, $z = s - a$, and $b - z$, are multiples of $\beta^{e_b - p + 1}$. Moreover,

$$|b - z| \leq |b|. \tag{4.1}$$

This comes from $|b - z| = |a + b - s|$: If $|a + b - s|$ was larger than $|b| = |a + b - a|$, then a would be a better floating-point approximation to $a + b$ than s.

From (4.1) and the fact that $b - z$ is a multiple of $\beta^{e_b - p + 1}$, we deduce that $b - z$ is exactly representable, which implies $t = b - z$.

Now, the theorem is easily obtained. From $t = b - z$ we deduce

$$t = b - (s - a) = (a + b) - s.$$

$\square$

4.3.2 The 2Sum algorithm

The Fast2Sum algorithm, presented in the previous section, requires a preliminary knowledge of the orders of magnitude of the two operands (since we must know which of them has the largest exponent).[4]

The following 2Sum algorithm (Algorithm 4.4), due to Knuth [222] and Møller [279], requires 6 consecutive floating-point operations instead of 3 for Fast2Sum, but does not require a preliminary comparison of a and b. Notice that on modern processors the penalty due to a wrong branch prediction when comparing a and b costs much more than 3 additional floating-point operations. Also, unless a wider format is available, Fast2Sum does not work in radices greater than 3, whereas 2Sum works in any radix. This is of interest when using a radix-10 system.

The name "TwoSum" seems to have been coined by Shewchuk [377].

---

[4]Do not forget that $|a| \geq |b|$ implies that the exponent of a is larger than or equal to that of b. Hence, it suffices to compare the two variables.

Theorem 6. *In double-precision/binary64 floating-point arithmetic, an RN-addition algorithm without branching that computes the same results as 2Sum has depth at least* 5.

These theorems show that, among the RN-addition algorithms without branching, if we do not have any information on the ordering of $|a|$ and $|b|$, 2Sum is optimal both in terms of number of arithmetic operations and in terms of depth. The proof was obtained by enumerating all possible RN-addition algorithms without branching that use 5 additions or less, or that have depth 4 of less. Each of these algorithms was run with a few well-chosen floating-point entries. None of the enumerated algorithms gave the same results as 2Sum for all chosen entries.

4.3.3 If we do not use rounding to nearest

The Fast2Sum and 2Sum algorithms rely on rounding to nearest. The example given at the beginning of Section 4.3 shows that, if s is computed as $\mathrm{RD}(a+b)$ or $\mathrm{RU}(a+b)$, then $s - (a+b)$ may not be a floating-point number, hence the algorithms would fail to return an approximate sum and the error term.

Nonetheless, in his Ph.D. dissertation [337], Priest gives a longer algorithm that only requires faithful arithmetic. From two floating-point numbers a and b, it deduces two other floating-point numbers c and d such that

- $c + d = a + b$, and

- either $c = d = 0$, or $|d| < \mathrm{ulp}(c)$.

In particular, Algorithm 4.5 works if ○ is any of the four rounding modes of IEEE 754-1985, provided no underflow or overflow occurs.

Algorithm 4.5 Priest's Sum and Roundoff error algorithm. It only requires faithful arithmetic.

> **if** $|a| < |b|$ **then**
> $\quad$ swap(a, b)
> **end if**
> $c \leftarrow \circ(a + b)$
> $e \leftarrow \circ(c - a)$
> $g \leftarrow \circ(c - e)$
> $h \leftarrow \circ(g - a)$
> $f \leftarrow \circ(b - h)$
> $d \leftarrow \circ(f - e)$
> **if** $\circ(d + e) \neq f$ **then**
> $\quad c \leftarrow a$
> $\quad d \leftarrow b$
> **end if**
> **return** c, d

4.4   Computation of Products

In the previous section, we have seen that, under some conditions, the error of a floating-point addition is a floating-point number that can be computed using a few operations. The same holds for floating-point multiplication:

- If x and y are radix-β precision-p floating-point numbers, whose exponents e_x and e_y satisfy $e_x + e_y \geq e_{\min} + p - 1$ (where $e_{\min}$ is the minimum exponent of the system being considered), and

- if r is $\circ(xy)$, where $\circ$ is one of the four rounding modes of the IEEE 754-1985 standard,

then $t = xy - r$ is a radix-β precision-p floating-point number.

Actually, computing t is very easy if a fused multiply-add (FMA) operator is available; This will be described in Chapter 5. Without an FMA, the best-known algorithm for computing t is Dekker's algorithm [108]. Roughly speaking, it consists in first splitting each of the operands x and y into two floating-point numbers, the significand of each of them being representable with $\lfloor p/2 \rfloor$ or $\lceil p/2 \rceil$ digits only. The underlying idea is that (using a property given in Section 4.2.2) the pairwise products of these values should be exactly representable, which is not always possible if p is odd, since the product of two $\lceil p/2 \rceil$-digit numbers does not necessarily fit on p digits.[6]

Then these pairwise products are added.

Let us now present that algorithm with more detail. We first show how to perform the splitting, using floating-point addition and multiplication only, by means of an algorithm due to Veltkamp [416, 417].

4.4.1   Veltkamp splitting

Before examining how we can compute exact products without an FMA, we need to see how we can "split" a precision-p radix-β floating-point number x into two floating-point numbers x_h and x_ℓ such that, for a given $s < p$, the significand of x_h fits in $p - s$ digits, the significand of x_ℓ fits in s digits, and $x = x_h + x_\ell$ exactly.

This is done using Veltkamp's algorithm (Algorithm 4.6). It uses a floating-point constant C equal to $\beta^s + 1$.

---

[6]In radix 2, we will use the fact that a $2g+1$-bit number can be split into two g-bit numbers. This explains why (see Section 4.4.2) Dekker's algorithm work if the precision is even or if the radix is 2.

Algorithm 4.6 Split(x,s): Veltkamp's algorithm.

Require: $C = \beta^s + 1$

   $\gamma \leftarrow \text{RN}(C \cdot x)$
   $\delta \leftarrow \text{RN}(x - \gamma)$
   $x_h \leftarrow \text{RN}(\gamma + \delta)$
   $x_\ell \leftarrow \text{RN}(x - x_h)$

Dekker [108] proves this algorithm in radix 2, with the implicit assumption that no overflow or underflow occurs. Boldo [30] shows that for any radix β and any precision p, provided that $C \cdot x$ does not overflow, the algorithm works. More precisely:

- if $C \cdot x$ does not overflow, no other operation will overflow;

- there is no underflow problem: If x_ℓ is subnormal, the result still holds.

Another property of Veltkamp's splitting that will be important for analyzing Dekker's multiplication algorithm is the following: If $\beta = 2$, the significand of x_ℓ actually fits in $s - 1$ bits.

Before giving a proof of Veltkamp's splitting algorithm, let us give an example.

Example 7 (Veltkamp's splitting). *Assume a radix-10 system, with precision 8. We want to split the significands of the floating-point numbers into parts of equal width; that is, we choose $s = 4$. This gives $C = 10001$. Assume $x = 1.2345678$. We successively find:*

- *$C \cdot x = 12346.9125678$, therefore*

$$\gamma = \text{RN}(C \cdot x) = 12346.913;$$

- *$x - \gamma = -12345.6784322$, therefore*

$$\delta = \text{RN}(x - \gamma) = -12345.678;$$

- *$\gamma + \delta = 1.235$, therefore*

$$x_h = \text{RN}(\gamma + \delta) = 1.235;$$

- *$x - x_h = -0.0004322$, therefore*

$$x_\ell = \text{RN}(x - x_h) = -0.0004322.$$

One can easily check that 1.2345678 equals $1.235 - 0.0004322$. In this example, the last two arithmetic operations are exact; that is, $x_h = \gamma + \delta$ and $x_\ell = x - x_h$ (no rounding error occurs). We will see in the proof that this is always true.

Now, let us give a proof of Algorithm 4.6. For simplicity, we assume that the radix is 2, that $s \geq 2$, and that no underflow nor overflow occurs. For a more general and very rigorous proof, see the remarkable paper by Boldo [30].

Proof. Since all variables in the algorithm are scaled by a factor 2^k, if we multiply x by 2^k, we can assume that $1 \leq x < 2$ without loss of generality. Furthermore, since the behavior of the algorithm is fairly obvious if $x = 1$, we assume $1 < x < 2$, which gives (since x is a precision-p floating-point number)

$$1 + 2^{-p+1} \leq x \leq 2 - 2^{-p+1}.$$

We now consider the four successive operations in Algorithm 4.6.

Computation of γ. $Cx = (2^s + 1)x$ implies $2^s + 1 \leq Cx < 2^{s+2}$. Therefore,

$$2^{s-p+1} \leq \mathrm{ulp}(Cx) \leq 2^{s-p+2}.$$

This gives

$$\begin{cases} \gamma = (2^s + 1)x + \epsilon_1, \text{ with } |\epsilon_1| \leq 2^{s-p+1}, \\ \gamma \text{ is a multiple of } 2^{s-p+1}. \end{cases}$$

Computation of δ. We have $x - \gamma = -2^s x - \epsilon_1$. From this we deduce

$$|x - \gamma| \leq 2^s(2 - 2^{-p+1}) + 2^{s-p+1} = 2^{s+1}.$$

This implies that $\delta = \mathrm{RN}(x - \gamma) = x - \gamma + \epsilon_2 = -2^s x - \epsilon_1 + \epsilon_2$, with $|\epsilon_2| \leq 2^{s-p}$.

Also, $|x - \gamma| \geq 2^s(1 + 2^{-p+1}) - 2^{s-p+1} \geq 2^s$, which implies that δ is a multiple of 2^{s-p+1}.

Computation of x_h. Now, $-\delta = 2^s x + \epsilon_1 - \epsilon_2$ and $\gamma = 2^s x + x + \epsilon_1$ are quite close together. As soon as $s \geq 2$, they are within a factor of 2 from each other. So Lemma 2 (Sterbenz's lemma) can be applied to deduce that $\gamma + \delta$ is computed exactly. Therefore,

$$x_h = \gamma + \delta = x + \epsilon_2.$$

Also, x_h is a multiple of 2^{s-p+1} (since $x_h = \gamma + \delta$, and γ and δ are multiples of 2^{s-p+1}). From $x < 2$ and $x_h = x - \epsilon_2$ one deduces $x_h < 2 + 2^{s-p}$, but the only multiple of 2^{s-p+1} between 2 and $2 + 2^{s-p}$ is 2, so $x_h \leq 2$.

Computation of x_ℓ. Since $x_h = x + \epsilon_2$ and x are very near, we can use Lemma 2 again to show that $x - x_h$ is computed exactly. Therefore,

$$x_\ell = x - x_h = \epsilon_2.$$

As a consequence, $|x_\ell| = |\epsilon_2|$ is less than or equal to 2^{s-p}. Moreover, x_ℓ is a multiple of 2^{-p+1}, since x and x_h are multiples of 2^{-p+1}.

Thus, we have written x as the sum of two floating-point numbers x_h and x_ℓ. Moreover,

- $x_h \leq 2$ and x_h is a multiple of 2^{s-p+1} imply that x_h fits in $p - s$ bits;

- $x_\ell \leq 2^{s-p}$ and x_ℓ is a multiple of 2^{-p+1} imply that x_ℓ fits in $s - 1$ bits.

$\square$

4.4.2   Dekker's multiplication algorithm

Algorithm 4.7 was discovered by Dekker, who presented it and proved it in radix 2, yet seemed to assume that it would work as well in higher radices. Later on, it was analyzed by Linnainmaa [263], who found the necessary condition in radices different from 2 (an even precision), and by Boldo [30], who examined the difficult problem of possible overflow or underflow in the intermediate operations, and gave a formal proof in Coq.

Here, we present the algorithm using Boldo's notation. We assume a floating-point arithmetic with radix β, subnormal numbers, and precision p. From two finite floating-point numbers x and y, the algorithm returns two floating-point numbers r_1 and r_2 such that $xy = r_1 + r_2$ exactly, under conditions that will be made explicit below.

Algorithm 4.7 Dekker product.

Require: $s = \lceil p/2 \rceil$
  $(x_h, x_\ell) \leftarrow \mathrm{Split}(x, s)$
  $(y_h, y_\ell) \leftarrow \mathrm{Split}(y, s)$
  $r_1 \leftarrow \mathrm{RN}(x \cdot y)$
  $t_1 \leftarrow \mathrm{RN}(-r_1 + \mathrm{RN}(x_h \cdot y_h))$
  $t_2 \leftarrow \mathrm{RN}(t_1 + \mathrm{RN}(x_h \cdot y_\ell))$
  $t_3 \leftarrow \mathrm{RN}(t_2 + \mathrm{RN}(x_\ell \cdot y_h))$
  $r_2 \leftarrow \mathrm{RN}(t_3 + \mathrm{RN}(x_\ell \cdot y_\ell))$

Listing 4.3 presents the same algorithm, written in C.

Here is Boldo's version [30] of the theorem that gives the conditions under which Dekker's multiplication algorithm returns a correct result.

C listing 4.3 Dekker product.

```c
/* Dekker (exact) double multiplication */

#include <stdlib.h>
#include <stdio.h>
#define SHIFT_POW 27 /* 53 / 2  for double precision. */
void dekkerMult(double a, double b, double *p, double *t);
void veltkampSplit(double x, int sp, double *x_high, double *x_low);
int main(int argc, char **argv)
{
  double x;
  double y;
  double r_1;
  double r_2;
  x = strtod(argv[1], NULL);
  y = strtod(argv[2], NULL);
  printf("x        = %1.16a\n", x);
  printf("y        = %1.16a\n", y);
  dekkerMult(x, y, &r_1, &r_2);
  printf("r_1      = %1.16a\n", r_1);
  printf("r_2      = %1.16a\n", r_2);
  return 0;
}

void dekkerMult(double x, double y, double *r_1, double *r_2)
{
  double x_high, x_low;
  double y_high, y_low;
  double t_1;
  double t_2;
  double t_3;
  veltkampSplit(x, SHIFT_POW, &x_high, &x_low);
  veltkampSplit(y, SHIFT_POW, &y_high, &y_low);
  printf("x_high    = %1.16a\n", x_high);
  printf("x_low     = %1.16a\n", x_low);
  printf("y_high    = %1.16a\n", y_high);
  printf("y_low     = %1.16a\n", y_low);
  *r_1 = x * y;
  t_1  = -*r_1 + x_high * y_high  ;
  t_2  =    t_1 + x_high * y_low;
  t_3  =    t_2 + x_low  * y_high;
  *r_2 =    t_3 + x_low  * y_low;
}

void veltkampSplit(double x, int sp, double *x_high, double *x_low)
{
  unsigned long C = (1UL << sp) + 1;
  double gamma = C * x;
  double delta = x - gamma;
  *x_high      = gamma + delta;
  *x_low       = x - *x_high;
}
```

Theorem 7. *Assume the minimal exponent $e_{\min}$ and the precision p satisfy[7] $p \geq 3$ and $\beta^{e_{\min}-p+1} \leq 1$. Let e_x and e_y be the exponents of the floating-point numbers x and y. If $\beta = 2$ or p is even, and if there is no overflow in the splitting of x and y or in the computation of $r_1 = \mathrm{RN}(x \cdot y)$ and $\mathrm{RN}(x_h \cdot y_h)$, then the floating-point numbers r_1 and r_2 returned by Algorithm 4.7 satisfy:*

1. *if $e_x + e_y \geq e_{\min} + p - 1$ then $xy = r_1 + r_2$ exactly;*

2. *in any case,*

$$|xy - (r_1 + r_2)| \leq \frac{7}{2}\beta^{e_{\min}-p+1}.$$

As pointed out by Boldo, the "7/2" in Theorem 7 could probably be sharpened. In particular, if the radix is 2, that coefficient can be reduced to 3.

Notice that the condition "$e_x + e_y \geq e_{\min} + p - 1$" on the exponents is a necessary and sufficient condition for the error term of the product to be always representable (see [31]), whatever the significands of x and y might be. Condition "$\beta = 2$ or p is even" might seem strange at first glance, but is easily understood by noticing that:

- $x_h \cdot y_h$ is exactly representable with a $2 \times \lfloor p/2 \rfloor$-digit significand, hence, it is representable in precision p:

$$\mathrm{RN}(x_h \cdot y_h) = x_h \cdot y_h;$$

- $x_h \cdot y_\ell$ and $x_\ell \cdot y_h$ are exactly representable with a $\lfloor p/2 \rfloor + \lceil p/2 \rceil = p$-digit significand;

- and yet, in many cases, $x_\ell \cdot y_\ell$ will be a $2 \times \lceil p/2 \rceil$-digit number.

Therefore, if the precision p is even, then $2 \times \lceil p/2 \rceil = p$, so that $x_\ell \cdot y_\ell$ is exactly representable. And if $\beta = 2$, then we know (see Section 4.4.1) that, even if p is odd, x_ℓ and y_ℓ actually fit on $\lceil p/2 \rceil - 1$ bits, so their product fits in p bits.

For instance, with the decimal formats specified by the new standard IEEE 754-2008 (see Chapter 3), Algorithm 4.7 will not always work in the decimal32 interchange format ($p = 7$), and yet it can be used safely in the decimal64 ($p = 16$) and decimal128 ($p = 34$) interchange formats.

Conditions on the absence of overflow for $r_1 = \mathrm{RN}(x \cdot y)$ and $\mathrm{RN}(x_h \cdot y_h)$ might seem redundant: Since these values are very close, in general they will overflow simultaneously. And yet, it is possible to build tricky cases where one of these computations will overflow, and not the other one. Condition $\beta^{e_{\min}-p+1} \leq 1$ is always satisfied in practice.

---

[7]These assumptions hold on any "reasonable" floating-point system.

Proof. For a full proof, see [30]. Here, we give a simplified proof, assuming radix 2, and assuming that no underflow/overflow occurs.

First, since the splittings have been performed to make sure that $x_h y_h$, $x_\ell y_h$, $x_h y_\ell$, and $x_\ell y_\ell$ should be exactly representable, we have

$$\mathrm{RN}(x_h y_h) = x_h y_h, \mathrm{RN}(x_\ell y_h) = x_\ell y_h, \mathrm{RN}(x_h y_\ell) = x_h y_\ell, \text{ and } \mathrm{RN}(x_\ell y_\ell) = x_\ell y_\ell.$$

Without loss of generality, we can assume $1 \le x < 2$ and $1 \le y < 2$. From the proof of Veltkamp's algorithm, we know that $x_h \le 2$, $y_h \le 2$, and that $|x - x_h| \le 2^{s-p}$ and $|y - y_h| \le 2^{s-p}$. From

$$(xy - x_h y_h) = (x - x_h)y + (y - y_h)x_h,$$

we deduce

$$|xy - x_h y_h| \le 2^{s-p+2}.$$

Since we also have

$$|xy - r_1| \le \frac{1}{2}\mathrm{ulp}(xy) \le 2^{-p+1},$$

we get

$$|r_1 - x_h y_h| \le 2^{-p+1} + 2^{s-p+2}.$$

This shows that r_1 and $\mathrm{RN}(x_h y_h) = x_h y_h$ are very close, so that Sterbenz's lemma (Lemma 2) can be applied to their subtraction. As a consequence,

$$t_1 = -r_1 + x_h y_h.$$

Now, $xy - x_h y_h = x_h y_\ell + x_\ell y_h + x_\ell y_\ell$, so that

$$\begin{aligned}|t_1 + x_h y_\ell| &= |-r_1 + x_h y_h + x_h y_\ell| \\ &= |-r_1 + xy + (x_h y_h + x_h y_\ell - xy)| \\ &\le |-r_1 + xy| + |x_\ell(y_h + y_\ell)| \\ &\le 2^{-p+1} + 2^{s-p+1} < 2^{s-p+2}.\end{aligned}$$

Since x_h is a multiple of 2^{s-p+1} and y_ℓ is a multiple of 2^{-p+1}, $x_h y_\ell$ is a multiple of 2^{s-2p+2}. This implies that $t_1 + x_h y_\ell$ is a multiple of 2^{s-2p+2}. This and $|t_1 + x_h y_\ell| < 2^{s-p+2}$ imply that $t_1 + x_h y_\ell$ is exactly representable. Therefore,

$$t_2 = -r_1 + x_h y_h + x_h y_\ell.$$

Now, $t_2 + x_\ell y_h = (-r_1 + xy) + x_\ell y_\ell$; therefore,

$$\begin{aligned}|t_2 + x_\ell y_h| &\le |-r_1 + xy| + |x_\ell y_\ell| \\ &\le 2^{-p+1} + 2^{2s-2p}.\end{aligned}$$

From $s = \lceil p/2 \rceil$, we deduce $2s - 2p = -p$ or $-p + 1$; therefore,

$$|t_2 + x_\ell y_h| \leq 2^{-p+2}.$$

This and the fact that $t_2 + x_\ell y_h$ is a multiple of 2^{s-2p+2} imply that $t_2 + x_\ell y_h$ is exactly representable; therefore,

$$t_3 = t_2 + x_\ell y_h = -r_1 + x_h y_h + x_h y_\ell + x_\ell y_h.$$

Lastly, $|t_3 + x_\ell y_\ell| = |-r_1 + xy| \leq 2^{-p+1}$, and it is a multiple of 2^{-2p+2}, thus $t_3 + x_\ell y_\ell$ is exactly computed. Therefore,

$$r_2 = xy - r_1.$$

<div align="right">□</div>

Dekker's multiplication algorithm requires 17 floating-point operations: 7 multiplications, and 10 additions/subtractions. This may seem a lot, compared to the 6 floating-point additions/subtractions required by the 2Sum algorithm (Algorithm 4.4). Yet an actual implementation of Dekker's algorithm will not be 17/6 times slower than an actual implementation of 2Sum. Indeed, since many operations in Dekker's algorithm are independent, they can be performed in parallel or in pipeline if the underlying architecture supports it. In the summary given in Figure 4.1, all the operations on a same line can be performed in parallel.

$$
\begin{array}{lll}
\gamma_x \leftarrow \text{RN}(Cx) & \gamma_y \leftarrow \text{RN}(Cy) & r_1 \leftarrow \text{RN}(xy) \\
\delta_x \leftarrow \text{RN}(x - \gamma_x) & \delta_y \leftarrow \text{RN}(y - \gamma_y) & \\
x_h \leftarrow \text{RN}(\gamma_x + \delta_x) & y_h \leftarrow \text{RN}(\gamma_y + \delta_y) & \\
x_\ell \leftarrow \text{RN}(x - x_h) & y_\ell \leftarrow \text{RN}(y - y_h) & \alpha_{11} \leftarrow \text{RN}(x_h y_h) \\
t_1 \leftarrow \text{RN}(-r_1 + \alpha_{11}) & \alpha_{12} \leftarrow \text{RN}(x_h y_\ell) & \alpha_{21} \leftarrow \text{RN}(x_\ell y_h) \quad \alpha_{22} \leftarrow \text{RN}(x_\ell y_\ell) \\
t_2 \leftarrow \text{RN}(t_1 + \alpha_{12}) & & \\
t_3 \leftarrow \text{RN}(t_2 + \alpha_{21}) & & \\
r_2 \leftarrow \text{RN}(t_3 + \alpha_{22}) & &
\end{array}
$$

Figure 4.1: Summary of the various floating-point operations involved in the Dekker product of x and y. The operations on a same line can be performed in parallel.

4.5 Complex numbers

If a and b are two floating-point numbers, the pair (a, b) can be used to represent the complex number $z = a + ib$. This is called the "Cartesian

representation" of z. Another possible representation of a complex number is its "polar" form $z = r \cdot e^{i\phi}$, but the polar form is less suited for the addition. Here we will concentrate on the Cartesian representation. Notice that accurate conversions can be performed between the two representations once accurate trigonometric functions and their inverses are implemented. We refer to Chapters 11 and 12 for these issues.

Adding two floating-point complex numbers $z_0 = a_0 + ib_0$ and $z_1 = a_1 + ib_1$ is done easily by computing the coordinate-wise sum:

$$(\mathrm{RN}(a_0 + a_1), \mathrm{RN}(b_0 + b_1)).$$

Things become more complicated with multiplication (Section 4.5.2), division (Section 4.5.3), and square root (Section 4.5.4). For instance, the C99 standard says that *the usual mathematical formulas for complex multiply, divide, and absolute value are problematic because of their treatment of infinities and because of undue overflow and underflow. The CX_LIMITED_RANGE pragma can be used to inform the implementation that (where the state is "on") the usual mathematical formulas are acceptable.*

4.5.1   Various error bounds

Two notions of relative error are often considered while performing approximate computations on complex numbers. Suppose we have a floating-point computation producing a complex number $\bar{z} = \bar{a} + i\bar{b}$ which is meant to approximate a complex number $z = a + ib$. One sometimes requires component-wise relative error bounds, which corresponds to bounding both

$$\frac{|\bar{a} - a|}{|a|} \quad \text{and} \quad \frac{|\bar{b} - b|}{|b|}.$$

This is intrinsically linked with the Cartesian interpretation of complex numbers. Sometimes one wants a bound on the quantity

$$\frac{|\bar{z} - z|}{|z|},$$

which is intrinsically related to the polar interpretation of complex numbers. The latter is usually referred to as a normwise error bound, as $|z|$ is the Euclidean norm of the two-dimensional vector (a, b). Having a bound of the first kind implies having a bound of the second kind, but the reverse is incorrect, as detailed in the following lemma.

Lemma 8. *Let $z = a + ib$ and $z' = a' + ib'$ be two complex numbers. Suppose that $|a' - a| \leq \epsilon|a|$ and $|b' - b| \leq \epsilon|b|$, for some $\epsilon > 0$. Then*

$$|z' - z| \leq \epsilon|z|.$$

The converse does not hold: We may simultaneously have $|z' - z| \leq \epsilon|z|$ and an arbitrarily large quantity $\frac{|a'-a|}{|a|}$.

Proof. We have the following relations:

$$\begin{aligned} |z' - z|^2 &= (a' - a)^2 + (b' - b)^2 \\ &\leq \epsilon^2 |a|^2 + \epsilon^2 |b|^2 \\ &= \epsilon^2 |z|^2. \end{aligned}$$

For the second part of the lemma, consider $z = t + i$ and $z' = t + \sqrt{t} + i$, with $t \in (0, 1)$. Then

$$\frac{|a' - a|}{|a|} = \frac{1}{\sqrt{t}}$$

can be made arbitrarily large (when t is close to 0), while

$$\frac{|z' - z|}{|z|} = \sqrt{\frac{t}{1 + t^2}} \leq \sqrt{t} \leq 1.$$

$\square$

4.5.2 Error bound for complex multiplication

Suppose we are given two complex numbers $z_0 = a_0 + ib_0$ and $z_1 = a_1 + ib_1$. We want to compute in floating-point arithmetic an approximation to their product

$$z_0 z_1 = (a_0 a_1 - b_0 b_1) + i(a_0 b_1 + b_0 a_1).$$

We first consider the "naive algorithm" which consists in computing the quantities $\mathrm{RN}(\mathrm{RN}(a_0 a_1) - \mathrm{RN}(b_0 b_1))$ and $\mathrm{RN}(\mathrm{RN}(a_0 b_1) + \mathrm{RN}(b_0 a_1))$. Error bounds for complex multiplication based on the naive algorithm can be found in [436, 317]. Here we show the following bound, a proof of which can also be found in [182].

Lemma 9. *Consider a floating-point arithmetic with rounding to nearest, and assume that no underflow/overflow occurs in the computation*[8]. *Let ϵ be $\frac{1}{2}\,\mathrm{ulp}(1)$. Let a_0, a_1, b_0, b_1 be four floating-point numbers, and define two complex numbers $z_0 = a_0 + ib_0$ and $z_1 = a_1 + ib_1$. Let*

$$z = a + ib = \mathrm{RN}(\mathrm{RN}(a_0 a_1) - \mathrm{RN}(b_0 b_1)) + i\,\mathrm{RN}(\mathrm{RN}(a_0 b_1) + \mathrm{RN}(b_0 a_1)).$$

Then

$$|z - z_0 z_1| \leq \sqrt{2}(2 + \epsilon)\epsilon |z_0||z_1|.$$

---

[8]We remind the reader that when a subnormal number is returned, we consider that there is an underflow.

Proof. We first consider the real part of z. We have

$$
\begin{aligned}
&\left|a - (a_0a_1 - b_0b_1)\right| \\
\leq\ & \left|a - (\mathrm{RN}(a_0a_1) - \mathrm{RN}(b_0b_1))\right| + \left|\,\mathrm{RN}(a_0a_1) - a_0a_1\right| + \left|\,\mathrm{RN}(b_0b_1) - b_0b_1\right| \\
\leq\ & \epsilon\left|\,\mathrm{RN}(a_0a_1) - \mathrm{RN}(b_0b_1)\right| + \left|\,\mathrm{RN}(a_0a_1) - a_0a_1\right| + \left|\,\mathrm{RN}(b_0b_1) - b_0b_1\right| \\
\leq\ & \epsilon|a_0a_1 - b_0b_1| + (1+\epsilon)|\,\mathrm{RN}(a_0a_1) - a_0a_1| + (1+\epsilon)|\,\mathrm{RN}(b_0b_1) - b_0b_1| \\
\leq\ & \epsilon|a_0a_1 - b_0b_1| + (1+\epsilon)\epsilon|a_0a_1| + (1+\epsilon)\epsilon|b_0b_1| \\
\leq\ & (2+\epsilon)\epsilon[|a_0a_1| + |b_0b_1|] \\
\leq\ & (2+\epsilon)\epsilon|z_0||z_1|.
\end{aligned}
$$

Similarly, we have $|b - (a_0b_1 + b_0a_1)| \leq (2+\epsilon)\epsilon|z_0||z_1|$. As a consequence,

$$|z - z_0z_1|^2 \leq 2(2+\epsilon)^2\epsilon^2|z_0|^2|z_1|^2,$$

which provides the result. □

Notice that the error bound given above is not component-wise. A relative component-wise error bound cannot exist since complex multiplication can be used to perform subtraction of two floating-point numbers x and y. For instance, consider the real part of z_0z_1, with $z_0 = 1 + i$ and $z_1 = x + iy$.

Furthermore, the error bound given in Lemma 9 is not tight. One cannot build floating-point numbers a_0, a_1, b_0, b_1 for which $|z - z_0z_1|$ is indeed close to $\sqrt{8}\epsilon|z_0z_1|$. This was discovered experimentally by Percival [330], who gave a first proof that $\sqrt{8}$ can in fact be replaced by $\sqrt{5}$. Unfortunately, his proof was flawed, and a corrected proof was published by Brent, Percival, and Zimmermann [46]. More precisely, they showed the following.

Theorem 10. *Consider a floating-point arithmetic with rounding to nearest, and assume that no underflow/overflow arises in the computation. Let ϵ be $\frac{1}{2}\,\mathrm{ulp}(1)$. Suppose that $\epsilon \leq 2^{-5}$. Let a_0, a_1, b_0, b_1 be four floating-point numbers, and define $z_0 = a_0 + ib_0$ and $z_1 = a_1 + ib_1$. Let*

$$z = a + ib = \mathrm{RN}(\mathrm{RN}(a_0a_1) - \mathrm{RN}(b_0b_1)) + i\,\mathrm{RN}(\mathrm{RN}(a_0b_1) + \mathrm{RN}(b_0a_1)).$$

Then

$$|z - z_0z_1| \leq \sqrt{5}\epsilon|z_0||z_1|.$$

The proof is lengthy, and we refer to [46] for the details. Notice first that both bounds (that we obtained in the proof of Lemma 9)

$$|a - (a_0a_1 - b_0b_1)| \leq (2+\epsilon)\epsilon|z_0||z_1|$$

and

$$|b - (a_0b_1 + b_0a_1)| \leq (2+\epsilon)\epsilon|z_0||z_1|$$

are essentially sharp when considered *independently*. The proof of Brent, Percival, and Zimmermann exploits the fact that they cannot be tight

Assumption	Bound on $|a - (a_0a_1 - b_0b_1)|$
$\mathrm{ulp}(b_0b_1) \leq \mathrm{ulp}(a_0a_1) \leq u$	$\epsilon(2a_0a_1 - b_0b_1) + 2\epsilon^2(a_0a_1 + b_0b_1)$
$\mathrm{ulp}(b_0b_1) < u < \mathrm{ulp}(a_0a_1)$	$\frac{7}{4}\epsilon a_0a_1$
$u \leq \mathrm{ulp}(b_0b_1) < \mathrm{ulp}(a_0a_1)$	$\frac{3}{2}\epsilon a_0a_1$
$u < \mathrm{ulp}(b_0b_1) = \mathrm{ulp}(a_0a_1)$	$\epsilon(a_0a_1 + b_0b_1)$

Table 4.1: The four cases of Brent, Percival, and Zimmermann.

simultaneously. By considering multiplications by i and -1 and by taking complex conjugates, one can see that it is sufficient to prove that the bound holds when $a_0, a_1, a_2, a_3 \geq 0$ and $b_0b_1 \leq a_0a_1$.

By distinguishing the cases when

$$\mathrm{ulp}(a_0b_1 + b_0a_1) < \mathrm{ulp}(\mathrm{RN}(a_0b_1) + \mathrm{RN}(b_0a_1))$$

and

$$\mathrm{ulp}(a_0b_1 + b_0a_1) \geq \mathrm{ulp}(\mathrm{RN}(a_0b_1) + \mathrm{RN}(b_0a_1)),$$

the authors prove that

$$|b - (a_0b_1 + b_0a_1)| \leq 2\epsilon(a_0b_1 + b_0a_1),$$

which is better than in the proof of Lemma 9 only by a factor of $(1 + \epsilon/2)$.

Then they consider four cases (see Table 4.1) for the real part of z, by looking at the respective values of $\mathrm{ulp}(b_0b_1)$, $\mathrm{ulp}(a_0a_1)$, and

$$u = \mathrm{ulp}(\mathrm{RN}(a_0a_1) - \mathrm{RN}(b_0b_1)).$$

One of these cases must occur since we assumed that $b_0b_1 \leq a_0a_1$.

Then each bound is combined independently with the previously obtained error bound on the imaginary part.

Their analysis shows that the bound

$$|z - z_0z_1| \leq \sqrt{5}\epsilon|z_0||z_1|$$

can be close to tight only in the fourth case of Table 4.1. In [46], the authors also provide worst cases for the single precision/binary32 and double precision/binary64 formats of the IEEE 754-1985 and IEEE 754-2008 standards. For $\beta = 2$, $t = 24$, and $\epsilon = 2^{-24}$ (single precision), the largest value of the quantity

$$\frac{|z - z_0z_1|}{|z_0z_1|}$$

is reached for

$$z_0 = \frac{3}{4} + i\frac{3}{4}(1 - 4\epsilon) \quad \text{and} \quad z_1 = \frac{2}{3}(1 + 11\epsilon) + i\frac{2}{3}(1 + 5\epsilon),$$

with

$$\frac{|z - z_0 z_1|}{|z_0 z_1|} \simeq \epsilon\sqrt{5 - 168\epsilon}.$$

For $\beta = 2$, $t = 53$, and $\epsilon = 2^{-53}$ (double precision), the worst case is reached for

$$z_0 = \frac{3}{4}(1 + 4\epsilon) + i\frac{3}{4} \quad \text{and} \quad z_1 = \frac{2}{3}(1 + 7\epsilon) + i\frac{2}{3}(1 + \epsilon),$$

with

$$\frac{|z - z_0 z_1|}{|z_0 z_1|} \simeq \epsilon\sqrt{5 - 96\epsilon}.$$

On systems for which multiplication of floating-point numbers is significantly more expensive than addition and subtraction (a typical example is multiple-precision systems), it is worth considering Karatsuba's algorithm [214] for complex multiplication. It performs 3 floating-point multiplications and 5 additions/subtractions instead of 4 multiplications and 2 additions/subtractions. Suppose we are multiplying the complex numbers $z_0 = a_0 + ib_0$ and $z_1 = a_1 + ib_1$. Then Karatsuba's algorithm is as shown in Algorithm 4.8.

Algorithm 4.8 Karatsuba's complex multiplication of $a_0 + ib_0$ and $a_1 + ib_1$

$p_1 \leftarrow \mathrm{RN}(\mathrm{RN}(a_0 + b_0) \cdot \mathrm{RN}(a_1 + b_1))$
$p_2 \leftarrow \mathrm{RN}(a_0 \cdot a_1)$
$p_3 \leftarrow \mathrm{RN}(b_0 \cdot b_1)$
$a \leftarrow \mathrm{RN}(p_2 - p_3)$
$b \leftarrow \mathrm{RN}(\mathrm{RN}(p_1 - p_2) - p_3)$

It is argued in [46] that the norm-wise relative error induced by Karatsuba's algorithm can be bounded by approximately 8ϵ, but this bound may be pessimistic.

4.5.3   Complex division

Complex division is not as well understood as complex multiplication. It seems harder to obtain tight worst-case norm-wise error bounds. Moreover, the influence of underflows and overflows is more complicated.

We now give an error bound similar to the one of Higham [182] in the case where no underflow/overflow occurs when performing division using the straightforward algorithm. Then we will review some results explaining how to handle underflows and overflows.

Lemma 11. *Consider a floating-point arithmetic with rounding to nearest, and assume that no underflow/overflow occurs in the computation. Define $\epsilon = \frac{1}{2}\,\mathrm{ulp}(1)$. Suppose that $\epsilon \leq 2^{-3}$. Let a_0, a_1, b_0, b_1 be four floating-point numbers, and define $z_0 = a_0 + ib_0$ and $z_1 = a_1 + ib_1$. Let*

$$z = a + ib$$

$$= \mathrm{RN}\left(\frac{\mathrm{RN}(\mathrm{RN}(a_0 a_1) + \mathrm{RN}(b_0 b_1))}{\mathrm{RN}(\mathrm{RN}(a_1^2) + \mathrm{RN}(b_1^2))}\right) + i\,\mathrm{RN}\left(\frac{\mathrm{RN}(\mathrm{RN}(b_0 a_1) - \mathrm{RN}(a_0 b_1))}{\mathrm{RN}(\mathrm{RN}(a_1^2) + \mathrm{RN}(b_1^2))}\right).$$

Then

$$\left| z - \frac{z_0}{z_1} \right| \leq 5\sqrt{2}(1 + 6\epsilon)\epsilon \frac{|z_0|}{|z_1|}.$$

Proof. We first consider the denominator. The quantity $\left||z_1|^2 - \mathrm{RN}(\mathrm{RN}(a_1^2) + \mathrm{RN}(b_1^2))\right|$ is less than or equal to:

$$|\,\mathrm{RN}(\mathrm{RN}(a_1^2) + \mathrm{RN}(b_1^2)) - (\mathrm{RN}(a_1^2) + \mathrm{RN}(b_1^2))|$$

$$+|a_1^2 - \mathrm{RN}(a_1^2)| + |b_1^2 - \mathrm{RN}(b_1^2)|$$

$$\leq \quad \epsilon|\,\mathrm{RN}(a_1^2) + \mathrm{RN}(b_1^2)| + |a_1^2 - \mathrm{RN}(a_1^2)| + |b_1^2 - \mathrm{RN}(b_1^2)|$$

$$\leq \quad \epsilon(1 + \epsilon)(a_1^2 + b_1^2) + \epsilon a_1^2 + \epsilon b_1^2 = (2 + \epsilon)\epsilon|z_1|^2.$$

Similar to the proof of Lemma 9, we have

$$|\,\mathrm{RN}(\mathrm{RN}(a_0 a_1) + \mathrm{RN}(b_0 b_1)) - (a_0 a_1 + b_0 b_1)| \quad \leq \quad (2 + \epsilon)\epsilon|z_0||z_1|,$$

$$|\,\mathrm{RN}(\mathrm{RN}(b_0 a_1) - \mathrm{RN}(a_0 b_1)) - (b_0 a_1 - a_0 b_1)| \quad \leq \quad (2 + \epsilon)\epsilon|z_0||z_1|.$$

Using the triangular inequality, this gives us that the quantity

$$\left| \frac{\mathrm{RN}(\mathrm{RN}(a_0 a_1) + \mathrm{RN}(b_0 b_1))}{\mathrm{RN}(\mathrm{RN}(a_1^2) + \mathrm{RN}(b_1^2))} - \frac{a_0 a_1 + b_0 b_1}{a_1^2 + b_1^2} \right|$$

is less than or equal to

$$\frac{|\,\mathrm{RN}(\mathrm{RN}(a_0 a_1) + \mathrm{RN}(b_0 b_1)) - (a_0 a_1 + b_0 b_1)|}{a_1^2 + b_1^2}$$

$$+|\,\mathrm{RN}(\mathrm{RN}(a_0 a_1) + \mathrm{RN}(b_0 b_1))|\frac{|\,\mathrm{RN}(\mathrm{RN}(a_1^2) + \mathrm{RN}(b_1^2)) - (a_1^2 + b_1^2)|}{\mathrm{RN}(\mathrm{RN}(a_1^2) + \mathrm{RN}(b_1^2))(a_1^2 + b_1^2)}$$

$$\leq \quad (2 + \epsilon)\epsilon\frac{|z_0|}{|z_1|} + (1 + (2 + \epsilon)\epsilon)\frac{(2 + \epsilon)\epsilon}{1 - (2 + \epsilon)\epsilon}\frac{|z_0|}{|z_1|}$$

$$= \quad \frac{2(2 + \epsilon)\epsilon}{1 - (2 + \epsilon)\epsilon}\frac{|z_0|}{|z_1|}.$$

We can then bound the quantity

$$\left| \text{RN} \left(\frac{\text{RN}(\text{RN}(a_0 a_1) + \text{RN}(b_0 b_1))}{\text{RN}(\text{RN}(a_1^2) + \text{RN}(b_1^2))} \right) - \frac{a_0 a_1 + b_0 b_1}{a_1^2 + b_1^2} \right|$$

by:

$$\epsilon \left| \frac{\text{RN}(\text{RN}(a_0 a_1) + \text{RN}(b_0 b_1))}{\text{RN}(\text{RN}(a_1^2) + \text{RN}(b_1^2))} \right| + \frac{2(2+\epsilon)\epsilon}{1 - (2+\epsilon)\epsilon} \frac{|z_0|}{|z_1|}$$

$$\leq \quad \epsilon \left(1 + (1+\epsilon) \frac{2(2+\epsilon)}{1 - (2+\epsilon)\epsilon} \right) \frac{|z_0|}{|z_1|}.$$

Since $\epsilon \leq 1/8$, we have

$$(2+\epsilon)\epsilon \leq \frac{17}{8}\epsilon \leq 1/2$$

and

$$\frac{(1+\epsilon)(2+\epsilon)}{1 - (2+\epsilon)\epsilon} \leq (1+\epsilon)(2+\epsilon)\left(1 + \frac{17}{4}\epsilon \right) \leq 2 + 14\epsilon.$$

Overall, this gives that:

$$\left| RN \left(\frac{\text{RN}(\text{RN}(a_0 a_1) + \text{RN}(b_0 b_1))}{\text{RN}(\text{RN}(a_1^2) + \text{RN}(b_1^2))} \right) - \frac{a_0 a_1 + b_0 b_1}{a_1^2 + b_1^2} \right| \leq \epsilon(5 + 28\epsilon)\frac{|z_0|}{|z_1|}.$$

The latter also holds for the imaginary part, and overall this provides

$$\left| z - \frac{z_0}{z_1} \right|^2 \leq 2\epsilon^2(5 + 28\epsilon)^2 \frac{|z_0|^2}{|z_1|^2},$$

which gives the result. $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \square$

Overflows and underflows may occur while performing a complex division even if the result lies well within the limits. Consider for example the quotient of $z_0 = 1$ and $z_1 = 2^{600}$ in the IEEE 754-1985 double-precision format. The denominator $|z_1|^2$ is evaluated to $+\infty$, so that the result of the computation is 0, which is far away from the correct result 2^{-600}. The most famous method to work around such harmful intermediate under-flows/overflows is due to Smith [381]. It consists in first comparing $|a_1|$ and $|b_1|$. After that,

- if $|a_1| \geq |b_1|$, one considers the formula

$$z = \frac{a_0 + b_0(b_1 a_1^{-1})}{a_1 + b_1(b_1 a_1^{-1})} + i\frac{b_0 - a_0(b_1 a_1^{-1})}{a_1 + b_1(b_1 a_1^{-1})},$$

  and

- if $|a_1| \leq |b_1|$, one considers the formula

$$z = \frac{a_0(a_1 b_1^{-1}) + b_0}{a_1(a_1 b_1^{-1}) + b_1} + i\frac{b_0(a_1 b_1^{-1}) - a_0}{a_1(a_1 b_1^{-1}) + b_1}.$$

Doing this requires 1 comparison, 3 floating-point divisions, and 3 floating-point multiplications, instead of 2 divisions and 6 multiplications. Note that if dividing is much more expensive than multiplying, then it is worth starting by inverting the common denominator of both the real and imaginary parts, and then multiplying the result by the numerators. This leads to 1 comparison, 2 divisions, and 5 multiplications for Smith's algorithm, and 1 division and 8 multiplications for the naive algorithm. In any case, Smith's algorithm can thus be significantly slower if (as it frequently happens) comparing and dividing are more expensive than multiplying.

Stewart [393] improves the accuracy of Smith's algorithm by performing a few additional comparisons. More precisely, he suggests computing the products $a_0 b_1 a_1^{-1}$ and $b_0 a_1 b_1^{-1}$ in a way that prevents harmful underflows and overflows during that particular step: when computing $a \cdot b \cdot c$, if the result does not underflow or overflow, then it is safe to first compute the product of the two numbers of extremal magnitudes and then multiply the obtained result by the remaining term. However, Stewart's variant of Smith's algorithm can still suffer from harmful intermediate overflows and underflows. For instance, suppose that $0 < b_1 < a_1$ and that both are close to the overflow limit. Then the denominator $a_1 + b_1(b_1 a_1^{-1})$ will overflow. Suppose furthermore that $a_0 \approx a_1$ and $b_0 \approx b_1$. Then the result of Stewart's algorithm will be NaN, although the result of the exact division is close to 1. Li et al. [257, Appendix B] show how to avoid harmful underflows and overflows in Smith's algorithm, by scaling the variables a_0, a_1, b_0 and b_1 beforehand. We do not describe their method, since Priest's algorithm, given below, achieves the same result at a smaller cost.

Priest [338] uses a two-step scaling of the variables to prevent harmful overflows and underflows in the naive complex division algorithm. The scalings are particularly efficient because the common scaling factor s is a power of 2 that can be quickly determined from the input operands.

Algorithm 4.9 Priest's algorithm for computing $\frac{a_0 + ib_0}{a_1 + ib_1}$, using a scaling factor s

---

$a_1' \leftarrow s \times a_1$
$b_1' \leftarrow s \times b_1$
$t \leftarrow 1/(a_1' \times a_1' + b_1' \times b_1')$
$a_1'' \leftarrow s \times a_1'$
$b_1'' \leftarrow s \times b_1'$
$x \leftarrow (a_0 \times a_1'' + b_0 \times b_1'') \times t$
$y \leftarrow (b_0 \times a_1'' - a_0 \times b_1'') \times t$
return $x + iy$

In Algorithm 4.9, the scaling factor s is first applied to a_1 and b_1 to allow for a safe computation of (a scaling of) the denominator. It is applied a second time to allow for a safe computation of the numerators. Priest proves that one can always choose a scaling factor s that depends on the inputs (taking s close to $|a_1 + ib_1|^{-3/4}$ suffices for most values of $a_0 + ib_0$), such that the following properties hold:

- If an overflow occurs in the computation, at least one of the two components of the exact value of

$$\frac{a_0 + ib_0}{a_1 + ib_1}$$

  is above the overflow threshold or within a few ULPs of it.

- Any underflow occurring in the first five steps of Algorithm 4.9 or in the computations of $a_0 \times a_1'' + b_0 \times b_1''$ and $b_0 \times a_1'' - a_0 \times b_1''$ is harmless, i.e., its contribution to the relative error

$$\frac{|(x + iy) - z_0/z_1|}{|z_0/z_1|}$$

  is at most a few ULPs.

- Any underflow occurring in the multiplications by t in the last two steps of Algorithm 4.9 incurs at most a few $\mathrm{ulp}(\lambda)$'s to the absolute error $|(x + iy) - z_0/z_1|$, where λ is the smallest positive normal floating-point number.

The above properties imply that an error bound similar to that of Lemma 11 holds for Priest's algorithm. Furthermore, the restriction that no underflow nor overflow should occur may then be replaced by the weaker restriction that none of the components of the exact value of

$$\frac{a_0 + ib_0}{a_1 + ib_1}$$

should lie above the overflow threshold nor within a few ULPs of it.

In [338], Priest also provides a very efficient way of computing a scaling parameter s that satisfies the constraints for the above properties to hold, for the particular situation of IEEE-754 binary double precision arithmetic. In [204], Kahan describes a similar scaling method, but it requires possibly slow library functions to find and apply the scaling.

4.5.4 Complex square root

The simplest algorithm for evaluating a complex square root $u + iv$ of $x + iy$, based on real square roots, consists in successively computing

$$
\begin{aligned}
\ell &= \sqrt{x^2 + y^2} \\
u &= \sqrt{(\ell + x)/2} \\
v &= \pm\sqrt{(\ell - x)/2}
\end{aligned}
\qquad (4.2)
$$

with $\operatorname{sign}(v) = \operatorname{sign}(y)$. Ahrendt [3] shows that this algorithm is optimal in the *algebraic* sense, i.e., in the number of exact operations $+, -, \times, \div, \sqrt{}$. However, it suffers from a major drawback: $x^2 + y^2$ can overflow and underflow, even if the exact square root is representable, leading to very poor results.

Another solution [135] is to first compute

$$
w = \begin{cases}
0 & \text{if} \quad x = y = 0 \\[2ex]
\sqrt{|x|}\sqrt{\dfrac{1 + \sqrt{1 + (y/x)^2}}{2}} & \text{if} \quad |x| \geq |y| \\[3ex]
\sqrt{|y|}\sqrt{\dfrac{|x/y| + \sqrt{1 + (x/y)^2}}{2}} & \text{if} \quad |x| < |y|
\end{cases}
$$

and then obtain

$$
u + iv = \sqrt{x + iy} = \begin{cases}
0 & \text{if} \quad w = 0 \\[2ex]
w + i\dfrac{y}{2w} & \text{if} \quad w \neq 0 \text{ and } x \geq 0 \\[2ex]
\dfrac{|y|}{2w} + iw & \text{if} \quad w \neq 0 \text{ and } x < 0 \text{ and } y \geq 0 \\[2ex]
\dfrac{|y|}{2w} - iw & \text{if} \quad w \neq 0 \text{ and } x < 0 \text{ and } y < 0.
\end{cases}
$$

This allows one to avoid intermediate overflows at the cost of more computation including several tests, divisions, and real square roots which make the complex square root evaluation quite slow compared to a single arithmetic instruction. Also, estimating the final accuracy seems very difficult.

Kahan [204] gives a better solution that also correctly handles all special cases (infinities, zeros, NaNs, etc.), also at the cost of much more computation than the naive method (4.2).

Chapter 5

The Fused Multiply-Add Instruction

THE FUSED MULTIPLY-ADD (FMA) instruction makes it possible to evaluate $ab + c$, where a, b, and c are floating-point numbers, with one final rounding only. That is, it computes

$$\circ(ab + c),$$

where $\circ$ is the active rounding mode (see Section 2.2).

FMA was introduced in 1990 on the IBM RS/6000 processor [183, 281]. The instruction allows for faster and, in general, more accurate dot products, matrix multiplications, and polynomial evaluations. As noticed for instance by Markstein [271], it also makes it possible to design fast algorithms for correctly rounded division and square root, as we will see later in this chapter. This might be the most interesting property of the FMA instruction, and it explains why, on current chips offering such an instruction, there is no hardwired division and/or square root operator. An FMA also simplifies the design of an accurate range reduction algorithm for the trigonometric functions [256].

After the IBM RS/6000, FMA units were implemented in many commercial, general-purpose processors. Examples are the IBM PowerPC [199], the HP PA-8000 [212, 236], and the HP/Intel Itanium [88]. An interesting survey on FMA architectures, along with suggestions for new architectures, is presented in [339].

The FMA instruction is included in the new IEEE 754-2008 standard for floating-point arithmetic. As a consequence, within a few years, this instruction will probably be available on most general-purpose processors.

The aim of this chapter is to show examples of calculations that are facilitated (and sometimes made possible) when an FMA instruction is available. We will start with the very simple yet useful example of the evaluation of the error of floating-point multiplication.

5.1   The 2MultFMA Algorithm

In Chapter 4 (Section 4.4.2, page 135), we have studied an algorithm due to Dekker that allows one to deduce, from two floating-point numbers x_1 and x_2, two other floating-point numbers r_1 and r_2 such that (under some conditions)

$$r_1 + r_2 = x_1 \cdot x_2$$

exactly, and

$$r_1 = \mathrm{RN}(x_1 \cdot x_2).$$

That is, r_2 is the error of the floating-point multiplication of x_1 by x_2. Dekker's multiplication algorithm requires 17 floating-point operations. It only works if the radix is 2 or the precision is even (see Chapter 4, Theorem 7, page 137), and if no overflow occurs in the intermediate calculations.

   If an FMA instruction is available, we can design a much simpler algorithm, which only requires two consecutive operations, and works for any radix and precision, provided the product $x_1 \cdot x_2$ does not overflow and $e_{x_1} + e_{x_2} \geq e_{\min} + p - 1$, where e_{x_1} and e_{x_2} are the exponents of x_1 and x_2. Although we present it for round-to-nearest mode, it works as well for the other rounding modes. See Algorithm 5.1.

Algorithm 5.1 2MultFMA(x_1, x_2).

$r_1 \leftarrow \mathrm{RN}(x_1 \cdot x_2)$
$r_2 \leftarrow \mathrm{RN}(x_1 \cdot x_2 - r_1)$

   Notice that condition $e_{x_1} + e_{x_2} \geq e_{\min} + p - 1$ cannot be avoided: if it is not satisfied, the product may not be representable as the exact sum of two floating-point numbers (r_2 would be below the underflow threshold). Consider for instance the following example, in the decimal64 format of the IEEE 754-2008 standard ($\beta = 10$, $p = 16$, $e_{\min} = -383$).

- $x_1 = 3.141592653589793 \times 10^{-190}$;

- $x_2 = 2.718281828459045 \times 10^{-190}$;

- the floating-point number closest to $x_1 \cdot x_2$ is $r_1 = 8.539734222673566 \times 10^{-380}$;

- the floating-point number closest to $x_1 \cdot x_2 - r_1$ is subnormal. Its value is $-0.000000000000322 \times 10^{-383}$, which differs from the exact value of $x_1 \cdot x_2 - r_1$, namely $-0.322151269472315 \times 10^{-395}$.

5.2 Computation of Residuals of Division and Square Root

As we will see, the availability of an FMA instruction simplifies the implementation of correctly rounded division and square root. The following two theorems have been known for a long time (see for instance [29]). We prefer here to give the recent presentation by Boldo and Daumas [31], since it fully takes into account the possibilities of underflow. Assume a radix-β, precision-p, floating-point system with extremal exponents $e_{\min}$ and $e_{\max}$.

In these theorems, we will call a *representable pair* for a floating-point number x a pair (M, e) of integers[1] such that $x = M \cdot \beta^{e-p+1}$, $|M| \leq \beta^p - 1$, and $e_{\min} \leq e$ (such pairs are "floating-point representations" that are not necessarily normal, without upper constraint on the exponents).

Theorem 12 (Exact residual for division [31]). *Let x and y be floating-point numbers in the considered format. Let q be $\circ(x/y)$, where $\circ$ is round-to-nearest, or a directed mode (see Section 2.2). If q is neither an infinity nor a Not a Number (NaN) datum, then*

$$x - qy$$

is a floating-point number if and only if there exist two representable pairs (M_y, e_y) and (M_q, e_q) that represent y and q such that

- *$e_y + e_q \geq e_{\min} + p - 1$ and*

- *$q \neq \alpha$ or $\alpha/2 \leq |x/y|$,*

where $\alpha = \beta^{e_{\min}-p+1}$ is the smallest positive subnormal number.

Theorem 13 (Exact residual for square root [31]). *Let x be a floating-point number in the considered format. Let σ be x rounded to a nearest floating-point value. If σ is neither an infinity nor a NaN, then*

$$x - \sigma^2$$

is representable if and only if there exists a representable pair (M_σ, e_σ) that represents σ such that

$$2e_\sigma \geq e_{\min} + p - 1.$$

See [31] for the proofs of these theorems. Consider the following example, which illustrates that if the conditions of Theorem 12 are not satisfied, then $x - qy$ is not exactly representable.

Example 8 (A case where $x - qy$ is not exactly representable). *Assume $\beta = 2$, $p = 24$, and $e_{\min} = 1 - e_{\max} = -126$ (single-precision format of the IEEE 754-1985*

---

[1]Beware: M is not necessarily the integral significand of x.

standard, binary32 format of IEEE 754-2008). Let x and y be floating-point numbers defined as

$$x = 2^{-104} + 2^{-105} = (1.1 \overbrace{0000000000000000000000}^{22 \ zeros})_2 \times 2^{-104}$$

and

$$y = 2^{-21} + 2^{-44} = (1.\overbrace{00000000000000000000000}^{22 \ zeros} 1)_2 \times 2^{-21}.$$

The floating-point number that is nearest to x/y is

$$q = (1.0 \underbrace{1111111111111111111111}_{22 \ ones})_2 \times 2^{-83},$$

and the exact value of $x - qy$ is

$$x - qy = -(1.\underbrace{111111111111111111111}_{21 \ ones})_2 \times 2^{-129},$$

which is not exactly representable. The (subnormal) floating-point number obtained by rounding $x - qy$ to nearest even is -2^{-128}.

In the example, $e_y + e_q = -104$, and $e_{\min} + p - 1 = -103$: condition "$e_y + e_q \geq e_{\min} + p - 1$" of Theorem 12 is not satisfied.

An important consequence of Theorem 12 is the following result, which will make it possible to perform correctly rounded divisions using Newton–Raphson iterations, provided that an FMA instruction is available (see Section 5.3).

Corollary 1 (Computation of division residuals using an FMA). *Assume x and y are precision-p, radix-β, floating-point numbers, with $y \neq 0$ and $|x/y|$ below the overflow threshold. If q is defined as*

- *x/y if it is exactly representable;*

- *one of the two floating-point numbers that surround x/y otherwise;[2]*

then

$$x - qy$$

is exactly computed using one FMA instruction, with any rounding mode, provided that

$$e_y + e_q \geq e_{\min} + p - 1,$$

$$and \tag{5.1}$$

$$q \neq \alpha \ or \ |x/y| \geq \tfrac{\alpha}{2},$$

---

[2]Or q is the largest finite floating-point number Ω, in the case where x/y is between that number and the overflow threshold (the same thing applies on the negative side).

where e_y and e_q are the exponents of y and q. In the frequent case $1 \leq x < \beta$ and $1 \leq y < \beta$ (straightforward separate handling of the exponents in the division algorithm), condition 5.1 is satisfied as soon as

$$e_{\min} \leq -p,$$

which holds in all usual formats.

Similarly, from Theorem 13, we deduce the following result.

Corollary 2 (Computation of square root residuals using an FMA)**.** *Assume x is a precision-p, radix-β, positive floating-point number. If σ is $\sqrt{x}$ rounded to a nearest floating-point number then*

$$x - \sigma^2$$

is exactly computed using one FMA instruction, with any rounding mode, provided that

$$2e_\sigma \geq e_{\min} + p - 1, \tag{5.2}$$

where e_σ is the exponent of σ. In the frequent case $1 \leq x < \beta^2$ (straightforward separate handling of the exponent in the square root algorithm), condition 5.2 is satisfied as soon as

$$e_{\min} \leq 1 - p,$$

which holds in all usual formats.

Corollary 2 is much weaker than Corollary 1: the "correcting term" $x-\sigma^2$ may not be exactly representable when σ is not a floating-point number nearest to x, even if σ is one of the two floating-point numbers that surround x. An example, in radix-2, precision-5 arithmetic is $x = 11110_2$ and $\sigma = 101.10_2$.

5.3 Newton–Raphson-Based Division with an FMA

Before using some of the results presented in the previous section to build algorithms for correctly rounded division, let us recall some variants of the Newton–Raphson iteration for reciprocation and division.

5.3.1 Variants of the Newton–Raphson iteration

Assume we wish to compute an approximation to b/a in a binary floating-point arithmetic of precision p. We will first present some classical iterations, all derived from the Newton–Raphson root-finding iteration, that are used in several division algorithms [271, 86, 270, 88]. Some of these iterations make it possible to directly compute b/a, yet most algorithms first compute $1/a$: A multiplication by b followed by a possible correcting step is necessary.

For simplicity, we assume that a and b satisfy

$$1 \leq a, b < 2,$$

which is not a problem in binary floating-point arithmetic (they are significands of floating-point numbers).

The Newton–Raphson iteration is a well-known and useful technique for finding roots of functions. It was introduced by Newton around 1669 [301] to solve polynomial equations (without explicit use of the derivative) and generalized by Raphson a few years later [372].

For finding roots of function f, the iteration is

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}. \tag{5.3}$$

If x_0 is close enough to a root α of f, if f has a second derivative, and if $f'(\alpha) \neq 0$, then the iteration (5.3) converges *quadratically* to α. By "quadratic convergence" we mean that the distance between x_{n+1} and α is proportional to the square of the distance between x_n and α. If $\alpha \neq 0$, this implies that the number of common digits between x_n and α roughly doubles at each iteration. For computing $1/a$, we look for the root of function $f(x) = 1/x - a$, which gives

$$x_{n+1} = x_n(2 - ax_n). \tag{5.4}$$

That iteration converges to $1/a$ for any $x_0 \in (0, 2/a)$. This is easy to see in Figure 5.1. And yet, of course, fast convergence requires a value of x_0 close to $1/a$.

In the case of iteration (5.4), we easily get

$$x_{n+1} - \frac{1}{a} = -a \left(x_n - \frac{1}{a} \right)^2, \tag{5.5}$$

which illustrates the quadratic convergence.

Beware: for the moment, we only deal with mathematical, "exact" iterations. When rounding errors are taken into account, the formulas become more complicated. Table 5.1 gives the first values x_n in the case $a = 1.5$ and $x_0 = 1$.

Figure 5.1: Iteration (5.4). We easily see that it converges to $1/a$ for $0 < x_0 < 2/a$.

n	x_n
1	0.5
2	0.625
3	0.6640625
4	0.666656494140625
5	0.66666666651144623756408691 40625
6	0.6666666666666666663052659425048318553308490663 76686 $\cdots$
7	0.666666666666666666666666666666666666666647075094152961 $\cdots$

Table 5.1: First values x_n given by iteration (5.4), in the case $a = 1.5$, $x_0 = 1$. The quadratic convergence is displayed by the fact that the number of common digits between x_n and the limit roughly doubles at each iteration.

Iteration (5.4) has a drawback: the three floating-point operations (two if an FMA instruction is available) it requires are dependent, i.e., no parallelism is available. This is a significant penalty, for instance, if the floating-point operators are implemented with a pipeline of large depth. And yet, a great advantage of that iteration is that it is "self-correcting": small errors (e.g., rounding errors) when computing x_n do not change the value of the limit.

Although the iterations would converge with $x_0 = 1$ (they converge for $0 < x_0 < 2/a$, and we have assumed $1 \leq a < 2$), in practice, we drastically reduce their number by starting with a value x_0 close to $1/a$, obtained either by looking up an approximation to $1/a$ in a table[3] addressed by a few most significant bits of a, or by using a polynomial approximation of very small degree to the reciprocal function. Many papers address the problem of cleverly designing a table that returns a convenient value x_0 [322, 360, 361, 362, 394, 106, 229]—see Section 9.2.8 page 286 for a review.

Now, by defining $\epsilon_n = 1 - ax_n$, one obtains the following iteration:

$$
\begin{cases}
\epsilon_n & = & 1 - ax_n \\
x_{n+1} & = & x_n + x_n\epsilon_n
\end{cases}
. \tag{5.6}
$$

That iteration was implemented on the Intel Itanium processor [86, 270, 88]:

- it still has the "self-correcting" property;

- it is as sequential as iteration (5.4), since there is a dependency between x_{n+1} and ϵ_n;

- however, it has a nice property; under the conditions of Corollary 1 (that is, roughly speaking, if x_n is within one ulp from $1/a$), the "residual" $\epsilon_n = 1 - ax_n$ will be exactly computed with an FMA. As we will see later on, this is a key feature that allows for correctly rounded division.

Another *apparently different* way of devising fast division algorithms is to use the power series

$$
\frac{1}{1 - \epsilon} = 1 + \epsilon + \epsilon^2 + \epsilon^3 + \cdots \tag{5.7}
$$

with $\epsilon = 1 - a$. Using (5.7) and the factorization

$$
1 + \epsilon + \epsilon^2 + \epsilon^3 + \cdots = (1 + \epsilon)(1 + \epsilon^2)(1 + \epsilon^4)(1 + \epsilon^8)\cdots,
$$

one can get a "new" fast iteration. By denoting

$$
\epsilon_n = \epsilon^{2^n},
$$

we get $\epsilon_{n+1} = \epsilon_n^2$, and

$$
\frac{1}{1 - \epsilon} = (1 + \epsilon_0)(1 + \epsilon_1)(1 + \epsilon_2)(1 + \epsilon_3)\cdots,
$$

---

[3]For instance, on the Intel/HP Itanium, there is an instruction, `frcpa`, that returns approximations to reciprocals, with at least 8.886 valid bits.

denoting $x_n = (1 + \epsilon_0)(1 + \epsilon_1) \cdots (1 + \epsilon_n)$, we get the following iteration:

$$\begin{cases} x_{n+1} & = & x_n + x_n \epsilon_n \\ \epsilon_{n+1} & = & \epsilon_n^2. \end{cases} \tag{5.8}$$

As for the Newton–Raphson iteration, one can significantly accelerate the convergence by starting from a value x_0 close to $1/a$, obtained from a table. It suffices then to choose $\epsilon_0 = 1 - ax_0$. Now, it is important to notice that the variables ϵ_n of (5.8) and (5.6) are the same: from $\epsilon_n = 1 - ax_n$, one deduces

$$\begin{aligned} \epsilon_n^2 & = & 1 - 2ax_n + a^2 x_n^2 \\ & = & 1 - ax_n(2 - ax_n) \\ & = & 1 - ax_{n+1} \\ & = & \epsilon_{n+1}. \end{aligned}$$

Hence, from a *mathematical* point of view, iterations (5.6) and (5.8) are equivalent: we have found again the same iteration through a totally different method. However, from a *computational* point of view, they are quite different:

- the computations of x_{n+1} and ϵ_{n+1} in (5.8) can be done in parallel, which is a significant improvement in terms of performance on most platforms;

- however, variable a no longer appears in (5.8). A consequence of this is that rounding errors in the computations will make information on the input operand disappear progressively—this iteration is *not* self-correcting.

Of course, one can mix these various iterations. For instance, we may choose to compute ϵ_n as $1 - ax_n$ during the last iterations, because accuracy becomes crucial, whereas it may be preferable to compute it as ϵ_{n-1}^2 during the first iterations, because this can be done in parallel with the computation of x_n.

Another feature of iteration (5.8) is that one can directly compute b/a instead of first computing $1/a$ and then multiplying by b, *but this is not necessarily efficient (see below)*. This is done by defining a new variable,

$$y_n = bx_n,$$

which gives

$$\begin{cases} y_{n+1} & = & y_n + y_n \epsilon_n \\ \epsilon_{n+1} & = & \epsilon_n^2. \end{cases} \tag{5.9}$$

The difficult point, however, is to get a sensible starting value y_0:

- either we start the iterations with $y_0 = b$ (which corresponds to $x_0 = 1$) and $\epsilon_0 = 1 - a$, and in such a case, we may need many iterations if a is not very close to 1 (i.e., if x_0 is far from $1/a$);

- or we try to start the iterations with the (hidden) variable x_0 equal to a close approximation a^* to $1/a$, and $\epsilon_0 = 1 - aa^*$. In such a case, we must have $y_0 = ba^*$.

Now, one can design another iteration by defining

$$\begin{cases} r_n & = & 1 - \epsilon_n \\ K_{n+1} & = & 1 + \epsilon_n, \end{cases} \tag{5.10}$$

which leads to the well-known *Goldschmidt iteration* [151]

$$\begin{cases} y_{n+1} & = & K_{n+1}y_n \\ r_{n+1} & = & K_{n+1}r_n \\ K_{n+1} & = & 2 - r_n. \end{cases} \tag{5.11}$$

This iteration, still mathematically equivalent to the previous ones, also has different properties from a computational point of view. The computations of y_{n+1} and r_{n+1} are independent and hence can be done in parallel, and the computation of K_{n+1} is very simple (it is done by two's complementing r_n). In case of a hardware implementation, since both multiplications that appear in the iteration are by the same value K_{n+1}, some optimizations—such as a common Booth recoding [37, 126]—are possible. However, unfortunately, the iteration is not self-correcting: after the first rounding error, exact information on a is lost forever.

From (5.9) or (5.6), by defining

$$\begin{cases} y_n & = & bx_n \\ \delta_n & = & b\epsilon_n, \end{cases} \tag{5.12}$$

one gets

$$\begin{cases} \delta_n & = & b - ay_n \\ y_{n+1} & = & y_n + \delta_n x_n. \end{cases} \tag{5.13}$$

This last iteration is used in Intel and HP's algorithms for the Itanium: once a correctly rounded approximation x_n to $1/a$ is obtained from (5.6)—we will see later how it can be correctly rounded—one computes a first approximation y_n to b/a by multiplying x_n by b. Then, this approximation is improved by applying iteration (5.13).

5.3.2 Using the Newton–Raphson iteration for correctly rounded division

In this section, we assume that we wish to compute $\circ(b/a)$, where a and b are floating-point numbers of the same format, and $\circ$ is one of the four following

rounding modes: round to nearest even, round toward $-\infty$, round toward $+\infty$, round toward zero. We will use some of the iterations presented in the previous section. If the radix of the floating-point system is 2, we do not have to worry about how values halfway between two consecutive floating-point numbers are handled in the round-to-nearest mode. This is due to the following result.

Lemma 14 (Size of quotients in prime radices, adapted from [270]). *Assume that the radix β of the floating-point arithmetic is a prime number. Let $q = b/a$, where a and b are two floating-point numbers of precision p:*

- *either q cannot be exactly represented with a finite number of radix-β digits;*

- *or q is a floating-point number of precision p (assuming unbounded exponent range).*

Proof. Assume that q is representable with a finite number of radix-β digits, but not with p digits or less. This means that there exist integers Q and e_q such that

$$q = Q \cdot \beta^{e_q - p + 1},$$

where $Q > \beta^p$, and Q is not a multiple of β.

Let A and B be the integral significands of a and b, and let e_a and e_b be their exponents. We have

$$\frac{B \cdot \beta^{e_b - p + 1}}{A \cdot \beta^{e_a - p + 1}} = Q \cdot \beta^{e_q - p + 1}.$$

Therefore, there exists an integer e, $e \geq 0$, such that

$$B = AQ \cdot \beta^e$$

or

$$B \cdot \beta^e = AQ.$$

This and the primality of β imply that B is a multiple of Q, which implies $B > \beta^p$. This is not possible since b is a precision-p floating-point number. $\square$

In Lemma 14, the fact that the radix should be a prime number is necessary. For instance, in radix 10 with $p = 4$, 2.005 and 2.000 are floating-point numbers, and their quotient 1.0025 has a finite representation, but cannot be represented with precision 4. A consequence of Lemma 14 is that, in radix 2, a quotient is never exactly halfway between two consecutive floating-point numbers. Notice that in prime radices greater than 2, Lemma 14 does not imply that quotients exactly halfway between two consecutive floating-point numbers do not occur.[4]

---

[4]When the radix is an odd number, values exactly halfway between two consecutive floating-point numbers are represented with infinitely many digits.

An interesting consequence of Lemma 14, since there is a finite number of quotients of floating-point numbers, is that there exists an *exclusion zone* around middles of consecutive floating-point numbers, where we cannot find quotients. There are several, slightly different, "exclusion theorems." This is one of them:

Lemma 15 (Exclusion lemma [88]). *Assume radix 2 and precision p, and let b be a normal floating-point number, and a a nonzero floating-point number. If c is either a floating-point number or the exact midpoint between two consecutive floating-point numbers, then we have*

$$\left| \frac{b}{a} - c \right| > 2^{-2p-2} \frac{b}{a}.$$

Another one is the following.

Lemma 16 (A slightly different exclusion lemma). *Assume radix 2 and precision p, and let b be a normal floating-point number, and a a nonzero floating-point number. If c is the exact midpoint between two consecutive floating-point numbers, then we have*

$$\left| \frac{b}{a} - c \right| > 2^{-p-1} \operatorname{ulp}\left(\frac{b}{a} \right).$$

Let us give a simple proof for Lemma 16.

Proof. Let A and B be the integral significands of a and b. We have

$$a = A \cdot 2^{e_a - p + 1}$$

and

$$b = B \cdot 2^{e_b - p + 1}.$$

Define

$$\delta = \begin{cases} 0 & \text{if } B \geq A \\ 1 & \text{otherwise.} \end{cases}$$

We have

$$\operatorname{ulp}\left(\frac{b}{a} \right) = 2^{e_b - e_a - p + 1 - \delta}.$$

Also, the middle c of two consecutive floating-point numbers around b/a is of the form

$$c = (2C + 1) \cdot 2^{e_b - e_a - p - \delta},$$

where C is an integer, $2^{p-1} \leq C \leq 2^p - 1$.

Therefore,

$$\begin{aligned} \frac{b}{a} - c &= \frac{B}{A} \cdot 2^{e_b - e_a} - (2C + 1) \cdot 2^{e_b - e_a - p - \delta} \\ &= \frac{1}{2} \operatorname{ulp}\left(\frac{b}{a} \right) \cdot \left(\frac{B}{A} \cdot 2^{p+\delta} - (2C + 1) \right) \\ &= \frac{1}{2A} \operatorname{ulp}\left(\frac{b}{a} \right) \cdot \left(B \cdot 2^{p+\delta} - (2C + 1)A \right). \end{aligned}$$

Since $1/A > 2^{-p}$, and since $B \cdot 2^{p+\delta} - (2C+1)A$ is a nonzero integer, we deduce

$$\left| \frac{b}{a} - c \right| > 2^{-p-1} \operatorname{ulp}\left(\frac{b}{a} \right).$$

$\square$

When the processor being used has an internal precision that is significantly wider than the "target" precision, a careful implementation of iterations such as (5.6) and (5.13) will allow one to obtain approximations to b/a accurate enough so that Lemma 15 or a variant can be applied to show that, once rounded to the target format, the obtained result will be the correctly rounded (to the nearest) quotient. The main difficulty is when we want to compute quotients of floating-point numbers in a precision that is the widest available. In such a case, the following result is extremely useful.

Theorem 17 (Peter Markstein [270]). *Assume a precision-p binary floating-point arithmetic, and let a and b be normal numbers. If*

- *q is a faithful approximation to b/a, and*

- *y approximates $1/a$ with a relative error less than 2^{-p}, and*

- *the calculations*

$$r = \circ(b - aq), \qquad q' = \circ(q + ry)$$

  are performed using a given rounding mode $\circ$, taken among round to nearest even, round toward zero, round toward $-\infty$, round toward $+\infty$,

then q' is exactly $\circ(b/a)$ (that is, b/a rounded according to the same rounding mode $\circ$).

Markstein also shows that the last FMA raises the inexact exception if and only if the quotient was inexact. Notice that if y approximates $1/a$ with an error less than or equal to $\frac{1}{2} \operatorname{ulp}(1/a)$, (that is, if y is $1/a$ rounded to nearest), then it approximates $1/a$ with a relative error less than 2^{-p}, so that Theorem 17 can be applied.

Proof. Let us prove Theorem 17 in the case of round-to-nearest even mode (the other cases are quite similar). First, notice that from Theorem 12, r is computed exactly.[5] Also, we will use the fact that if a is positive and is not a subnormal number,[6] then

$$\frac{a}{2^p - 1} \le \operatorname{ulp}(a) \le \frac{a}{2^{p-1}}. \tag{5.14}$$

---

[5]As soon as $e_a + e_q \ge e_{\min} + p - 1$, which will hold, in practice, if either we perform some "prescaling" or handle the "difficult" cases separately.

[6]Again, such "difficult" cases can be processed separately.

We assume that a, b, and q are not in the subnormal range; we also assume $a > 0$, $b > 0$ (and hence $q > 0$). Figure 5.2 illustrates the various cases that may occur, if we wish to return b/a rounded to nearest:

- if $q - \frac{1}{2}\operatorname{ulp}\left(\frac{b}{a}\right) < \frac{b}{a} < q + \frac{1}{2}\operatorname{ulp}\left(\frac{b}{a}\right)$, we must return q;

- if $\frac{b}{a} > q + \frac{1}{2}\operatorname{ulp}\left(\frac{b}{a}\right)$, we must return $q + \operatorname{ulp}\left(\frac{b}{a}\right)$;

- if $\frac{b}{a} < q - \frac{1}{2}\operatorname{ulp}\left(\frac{b}{a}\right)$, we must return $q - \operatorname{ulp}\left(\frac{b}{a}\right)$.

Notice that the case $\frac{b}{a} = q \pm \frac{1}{2}\operatorname{ulp}\left(\frac{b}{a}\right)$ cannot occur from Lemma 14. The returned value q' is obtained by adding a correcting term ry to q, and rounding the obtained result to nearest. One can easily find that to make sure that $q' = \operatorname{RN}(b/a)$:

- if $q - \frac{1}{2}\operatorname{ulp}\left(\frac{b}{a}\right) < \frac{b}{a} < q + \frac{1}{2}\operatorname{ulp}\left(\frac{b}{a}\right)$, $|ry|$ must be less than $\frac{1}{2}\operatorname{ulp}\left(\frac{b}{a}\right)$;

- if $\frac{b}{a} > q + \frac{1}{2}\operatorname{ulp}\left(\frac{b}{a}\right)$, ry must be larger than $\frac{1}{2}\operatorname{ulp}\left(\frac{b}{a}\right)$ and less than $\frac{3}{2}\operatorname{ulp}\left(\frac{b}{a}\right)$;

- if $\frac{b}{a} < q - \frac{1}{2}\operatorname{ulp}\left(\frac{b}{a}\right)$, ry must be larger than $-\frac{3}{2}\operatorname{ulp}\left(\frac{b}{a}\right)$ and less than $-\frac{1}{2}\operatorname{ulp}\left(\frac{b}{a}\right)$.
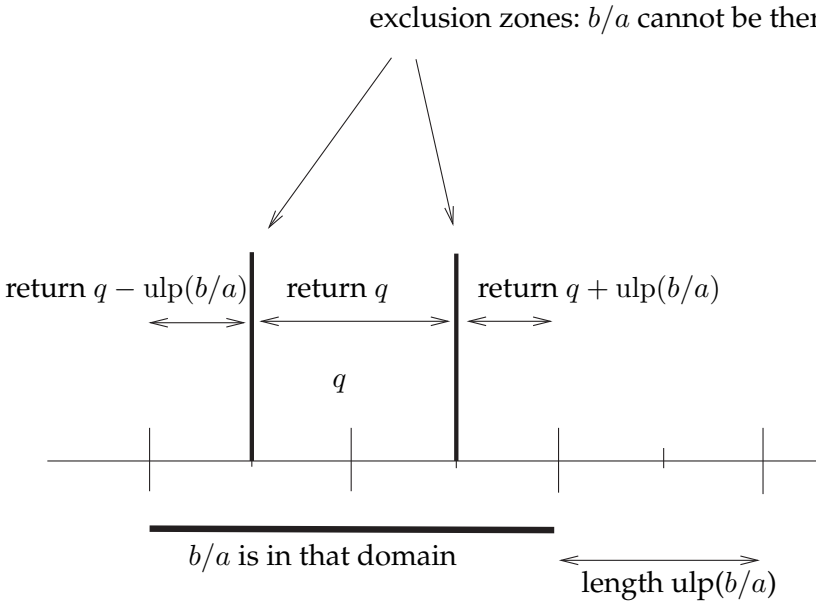
Figure 5.2: The various values that should be returned in round-to-nearest mode, assuming q is within one $\operatorname{ulp}(b/a)$ *from b/a.*

Since y approximates $1/a$ with relative error less than 2^{-p}, we have

$$\frac{1}{a} - \frac{1}{2^p a} < y < \frac{1}{a} + \frac{1}{2^p a}. \qquad (5.15)$$

1. If $q - \frac{1}{2} \operatorname{ulp}\left(\frac{b}{a}\right) < \frac{b}{a} < q + \frac{1}{2} \operatorname{ulp}\left(\frac{b}{a}\right)$. Assume $\frac{b}{a} - q \geq 0$ (the other case is symmetrical). We have

$$0 \leq b - aq < \frac{a}{2} \operatorname{ulp}\left(\frac{b}{a}\right);$$

   therefore, since $b - aq$ and $\frac{a}{2} \operatorname{ulp}\left(\frac{b}{a}\right)$ are floating-point numbers,

$$r = b - aq \leq \frac{a - \operatorname{ulp}(a)}{2} \operatorname{ulp}\left(\frac{b}{a}\right).$$

   Using (5.14), we get

$$0 \leq r \leq \left(\frac{a}{2} - \frac{a}{2^{p+1} - 2}\right) \operatorname{ulp}\left(\frac{b}{a}\right).$$

   This, along with (5.15), gives an upper bound on $|ry|$:

$$|ry| < \left(\frac{a}{2} - \frac{a}{2^{p+1} - 2}\right)\left(\frac{1}{a} + \frac{1}{2^p a}\right) \operatorname{ulp}\left(\frac{b}{a}\right),$$

   from which we deduce

$$|ry| < \left(\frac{1}{2} - \frac{1}{2^{p-1}(2^{p+1} - 2)}\right) \operatorname{ulp}\left(\frac{b}{a}\right) < \frac{1}{2} \operatorname{ulp}\left(\frac{b}{a}\right).$$

   Therefore, $\operatorname{RN}(q + ry) = q$, which is what we wanted to show.

2. If $\frac{b}{a} > q + \frac{1}{2} \operatorname{ulp}\left(\frac{b}{a}\right)$ (the case $\frac{b}{a} < q - \frac{1}{2} \operatorname{ulp}\left(\frac{b}{a}\right)$ is symmetrical). We have

$$\frac{a}{2} \operatorname{ulp}\left(\frac{b}{a}\right) < b - aq < a \operatorname{ulp}\left(\frac{b}{a}\right);$$

   therefore, since $b - aq$, $\frac{a}{2} \operatorname{ulp}\left(\frac{b}{a}\right)$ and $a \operatorname{ulp}\left(\frac{b}{a}\right)$ are floating-point numbers,

$$\left(\frac{a + \operatorname{ulp}(a)}{2}\right) \operatorname{ulp}\left(\frac{b}{a}\right) \leq r = b - aq \leq (a - \operatorname{ulp}(a)) \operatorname{ulp}\left(\frac{b}{a}\right).$$

   Using (5.14), we get

$$\left(\frac{a + \frac{a}{2^p - 1}}{2}\right) \operatorname{ulp}\left(\frac{b}{a}\right) \leq r \leq \left(a - \frac{a}{2^p - 1}\right) \operatorname{ulp}\left(\frac{b}{a}\right).$$

   This, along with (5.15), gives

$$\left(\frac{a + \frac{a}{2^p - 1}}{2}\right)\left(\frac{1}{a} - \frac{1}{2^p a}\right) \operatorname{ulp}\left(\frac{b}{a}\right)$$

$$< ry$$

$$< \left(a - \frac{a}{2^p - 1}\right)\left(\frac{1}{a} + \frac{1}{2^p a}\right) \operatorname{ulp}\left(\frac{b}{a}\right),$$

from which we deduce

$$\frac{1}{2}\operatorname{ulp}\left(\frac{b}{a}\right) < ry < \left(1 - \frac{2^{1-p}}{2^p - 1}\right)\operatorname{ulp}\left(\frac{b}{a}\right) < \operatorname{ulp}\left(\frac{b}{a}\right).$$

Therefore, $\operatorname{RN}(q + ry) = q + \operatorname{ulp}\left(\frac{b}{a}\right)$, which is what we wanted to show.

$\square$

Hence, a careful use of Theorem 17 makes it possible to get a correctly rounded quotient b/a, once we have computed a very accurate approximation to the reciprocal $1/a$. Let us therefore now focus on the computation of reciprocals. More precisely, we wish to always get $\operatorname{RN}(1/a)$. The central result, due to Peter Markstein, is the following one.

Theorem 18 (Markstein [270]). *In precision-p binary floating-point arithmetic, if y is an approximation to $1/a$ with an error less than $1\operatorname{ulp}(1/a)$, and the calculations*

$$r = \circ(1 - ay), \; y' = \circ(y + ry)$$

are performed using round-to-nearest-even mode, then y' is exactly $1/a$ rounded to nearest even, provided that the integral significand of a, namely $A = a/\operatorname{ulp}(a)$, is different from $2^p - 1 = 11111\cdots11_2$.

A division algorithm can therefore be built as follows.

- First, $\operatorname{RN}(1/a)$ is computed. In general, it is wise to use iteration (5.8) for the first steps because it is faster, and iteration (5.6) for the last steps because it is more accurate (both in round-to-nearest mode). A *very careful error analysis* must be performed to make sure that, after these iterations, we get an approximation to $1/a$ that is within 1 ulp from $1/a$. That error analysis depends on the accuracy of the table (or approximation of any kind, e.g., by a polynomial of small degree) that gives x_0, on the precision of the input and output values, on the available internal precision, etc.). A way to perform (and to automate) that error analysis is presented by Panhaleux [321].

- Then, Theorem 18 is applied, to get $\operatorname{RN}(1/a)$ (except, possibly in the case where the integral significand of a is $2^p - 1 = 11111\cdots11_2$. That case is easily handled separately [270]).

- A first approximation to the quotient is computed, by multiplying the previously obtained value $\operatorname{RN}(1/a)$ by b.

- That approximation to the quotient is refined, in round-to-nearest mode, using iteration (5.13). Again, a careful error analysis is required to make sure that we get an approximation to b/a that is within $\operatorname{ulp}(b/a)$.

- Finally, Theorem 17 is applied to get b/a correctly rounded in the desired rounding mode.

Several variants of division algorithms (depending on the required and internal precisions, depending on whether we wish to optimize the throughput or to minimize the latency) are given by Markstein in his excellent book [270].

5.4 Newton–Raphson-Based Square Root with an FMA

The Newton–Raphson iteration can also be used to evaluate square roots. Again, the availability of an FMA instruction allows for rather easily obtained correctly rounded results. We will not present the methods in detail here (one can find them in [270, 86, 88]): we will focus on the most important results only.

5.4.1 The basic iterations

From the general Newton–Raphson iteration (5.3), one can derive two classes of algorithms for computing the square root of a positive real number a.

- If we look for the positive root of function $f(x) = x^2 - a$, we get

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right).$$ (5.16)

  This "Newton–Raphson" square root iteration goes back to much before Newton's time. Al-Khwarizmi mentions this method in his arithmetic book [94]. Furthermore, it was already used by Heron of Alexandria (which explains why it is frequently named "the Heron iteration"), and seems to have been known by the Babylonians 2000 years before Heron [138]. One can easily show that if $x_0 > 0$, then x_n goes to $\sqrt{a}$. This iteration has a drawback: it requires a division at each step. Also, guaranteeing correct rounding does not seem to be a simple task.

- If we look for the positive root of function $f(x) = 1/x^2 - a$, we get

$$x_{n+1} = x_n(3 - ax_n^2)/2.$$ (5.17)

  This iteration converges to $1/\sqrt{a}$, provided that $x_0 \in (0, \sqrt{3}/\sqrt{a})$. To get a first approximation to $\sqrt{a}$ it suffices to multiply the obtained result by a. And yet, this does not always give a correctly rounded result: some refinement is necessary. To obtain fast, quadratic, convergence, the first point x_0 must be a close approximation to $1/\sqrt{a}$, read from a table or obtained using a polynomial approximation of small degree. Iteration (5.17) still has a division, but that division (by 2) is very simple, especially in radix 2.

5.4.2   Using the Newton–Raphson iteration for correctly rounded square roots

From Equation (5.17), and by defining a "residual" ϵ_n as $1 - ax_n^2$, one gets

$$\begin{cases} \epsilon_n & = \ 1 - ax_n^2 \\ x_{n+1} & = \ x_n + \frac{1}{2}\epsilon_n x_n. \end{cases} \tag{5.18}$$

To decompose these operations in terms of FMA instructions, Markstein [270] defines new variables:

$$\begin{cases} r_n & = \ \frac{1}{2}\epsilon_n \\ g_n & = \ ax_n \\ h_n & = \ \frac{1}{2}x_n. \end{cases} \tag{5.19}$$

From (5.18) and (5.19), one finds the following iteration:

$$\begin{cases} r_n & = \ \frac{1}{2} - g_n h_n \\ g_{n+1} & = \ g_n + g_n r_n \\ h_{n+1} & = \ h_n + h_n r_n. \end{cases} \tag{5.20}$$

Variable h_n goes to $1/(2\sqrt{a})$, and variable g_n goes to $\sqrt{a}$. Iteration (5.20) is easily implemented with an FMA instruction. Some parallelism is possible since the computations of g_{n+1} and h_{n+1} can be performed simultaneously.

Exactly as for the division iteration, a very careful error analysis is needed, and the iterations are performed as well as a final refinement step. Here are some results that make it possible to build refinement techniques. See Markstein's book [270] for more details.

Theorem 19. *In any radix, the square root of a floating-point number cannot be the exact midpoint between two consecutive floating-point numbers.*

Proof. Assume that r is the exact middle of two consecutive radix-β, precision-p floating-point numbers, and assume that it is the square-root of a floating-point number x. Without loss of generality we can assume that r has the form

$$r = (r_0.r_1 r_2 \cdots r_{p-1})_\beta + \frac{1}{2}\beta^{-p+1};$$

i.e., that $1 \leq r < \beta$. Let $R = (r_0 r_1 r_2 \cdots r_{p-1})_\beta$ be the integral significand of r. We have

$$2r\beta^{p-1} = 2R + 1;$$

i.e.,

$$4r^2\beta^{2p-2} = (2R+1)^2.$$

Since r^2 is a floating-point number between 1 and β^2, it is a multiple of β^{-p+1}, which implies that $r^2\beta^{2p-2}$ is an integer. Thus, $4r^2\beta^{2p-2}$ is a multiple of 4. This contradicts the fact that it is equal to the square of the odd number $2R+1$. $\square$

Theorem 20. *If the radix of the floating-point arithmetic is* 2, *then the square root reciprocal of a floating-point number cannot be the exact midpoint between two consecutive floating-point numbers.*

The proof is very similar to the proof of Theorem 19.

In non-binary radices, Theorem 20 may not hold, even if the radix is prime. Consider for example a radix-3 arithmetic, with precision $p = 6$. The number

$$x = 324_{10} = 110000_3$$

is a floating-point number, and the reader can easily check that

$$\frac{1}{\sqrt{x}} = \frac{1}{3^8} \cdot \left(111111_3 + \frac{1}{2}\right),$$

which implies that $1/\sqrt{x}$ is the exact midpoint between two consecutive floating-point numbers.

Also, in radix 10 with precision 16 (which corresponds to the decimal64 format of the IEEE 754-2008 standard), the square-root reciprocal of

$$\underbrace{70.36874417766400}_{16 \text{ digits}}$$

is

$$0.\underbrace{11920928955078125}_{17 \text{ digits}},$$

which is the exact midpoint between two consecutive floating-point numbers.

The following *Tuckerman test* allows one to check if a floating-point number r is the correctly rounded-to-nearest square root of another floating-point number a. Markstein [270] proves the following theorem in prime radices, but it holds in any radix.

Theorem 21 (The Tuckerman test, adapted from Markstein's presentation [270])**.** *In radix β, if a and r are floating-point numbers, then r is $\sqrt{a}$ rounded to nearest if and only if*

$$r(r - \operatorname{ulp}(r^-)) < a \leq r(r + \operatorname{ulp}(r)) \tag{5.21}$$

where r^- is the floating-point predecessor of r.

Proof. We should first notice that Theorem 20 implies that $\sqrt{a}$ cannot be a "midpoint" (i.e., a value exactly halfway between two consecutive floating-point numbers). Therefore, we do not have to worry about tie-breaking rules. Also, if k is an integer such that $\beta^k r$ and $\beta^{2k} a$ do not overflow or underflow, then (5.21) is equivalent to

$$(\beta^k r)((\beta^k r) - \beta^k \operatorname{ulp}(r^-)) < \beta^{2k} a \leq (\beta^k r)((\beta^k r) + \beta^k \operatorname{ulp}(r));$$

which is equivalent, if we define $R = \beta^k r$ and $A = \beta^{2k} a$ to

$$R(R - \mathrm{ulp}(R^-)) < A \leq R(R + \mathrm{ulp}(R)).$$

Since $R = \mathrm{RN}(\sqrt{A})$ is straightforwardly equivalent to $r = \mathrm{RN}(\sqrt{a})$, we deduce that without loss of generality, we can assume that $1 \leq r < \beta$. Let us now consider two cases.

1. **If $r = 1$.** In this case (5.21) becomes $1 - \beta^{-p} < a \leq 1 + \beta^{-p+1}$; that is, since a is a floating-point number,

$$a \in \{1, 1 + \beta^{-p+1}\}. \tag{5.22}$$

   Since $\sqrt{a}$ cannot be a midpoint, $1 = \mathrm{RN}(\sqrt{a})$ is equivalent to

$$1 - \frac{1}{2}\beta^{-p} < \sqrt{a} < 1 + \frac{1}{2}\beta^{-p+1},$$

   which is equivalent to

$$1 - \beta^{-p} + \frac{1}{4}\beta^{-2p} < a < 1 + \beta^{-p+1} + \frac{1}{4}\beta^{-2p+2}. \tag{5.23}$$

   The only floating-point numbers lying in the real range defined by (5.23) are 1 and $1 + \beta^{-p+1}$, so that (5.23) is equivalent to (5.22).

2. **If $1 < r < \beta$.** In this case $\mathrm{ulp}(r^-) = \mathrm{ulp}(r) = \beta^{-p+1}$ and (5.21) is thus equivalent to

$$r(r - \beta^{-p+1}) < a \leq r(r + \beta^{-p+1}). \tag{5.24}$$

   Since $\sqrt{a}$ cannot be a midpoint, $r = \mathrm{RN}(\sqrt{a})$ is equivalent to

$$r - \frac{1}{2}\beta^{-p+1} < \sqrt{a} < r + \frac{1}{2}\beta^{-p+1}.$$

   By squaring all terms, this is equivalent to

$$r(r - \beta^{-p+1}) + \frac{1}{4}\beta^{-2p+2} < a < r(r + \beta^{-p+1}) + \frac{1}{4}\beta^{-2p+2}. \tag{5.25}$$

   Now, since r is a floating-point number between 1 and β, it is a multiple of β^{-p+1}. This implies that $r(r - \beta^{-p+1})$ and $r(r + \beta^{-p+1})$ are multiples of β^{-2p+2}.

   An immediate consequence is that there is no multiple of β^{-2p+2} between $r(r - \beta^{-p+1})$ and $r(r - \beta^{-p+1}) + \frac{1}{4}\beta^{-2p+2}$, or between $r(r + \beta^{-p+1})$ and $r(r + \beta^{-p+1}) + \frac{1}{4}\beta^{-2p+2}$.

   This implies that there is no *floating-point number* between $r(r - \beta^{-p+1})$ and $r(r - \beta^{-p+1}) + \frac{1}{4}\beta^{-2p+2}$, or between $r(r + \beta^{-p+1})$ and $r(r + \beta^{-p+1}) + \frac{1}{4}\beta^{-2p+2}$.

   As a consequence, since a is a floating-point number, (5.25) is equivalent to (5.24). q.e.d.   □

With an FMA instruction, and assuming that instructions for computing the floating-point predecessor and successor of r are available, the Tuckerman test is easily performed. For instance, to check whether

$$a \leq r(r + \mathrm{ulp}(r)),$$

it suffices to compute

$$\delta = \mathrm{RN}(r \times \mathrm{successor}(r) - a)$$

using an FMA, and to test the sign of δ.

5.5 Multiplication by an Arbitrary-Precision Constant

Many numerical algorithms require multiplications by constants that are not exactly representable in floating-point arithmetic. Typical examples of such constants are π, $1/\pi$, $\ln(2)$, e, as well as values of the form $\cos(k\pi/N)$ and $\sin(k\pi/N)$, which appear in fast Fourier transforms. A natural question that springs to mind is: Can we, at low cost, perform these multiplications with correct rounding?

The algorithm presented here was introduced by Brisebarre and Muller [51].

Assume that C is an arbitrary-precision constant. We want to design an algorithm that always returns $\mathrm{RN}(Cx)$, for any input floating-point number x of a given format. We want the algorithm to be very simple (two consecutive operations only, without any test). We assume that the "target" format is a binary floating-point format of precision p. Two possible cases are of interest: in the first case, all intermediate calculations are performed in the target format. In the second case, the intermediate calculations are performed in a somewhat larger format. A typical example is when the target precision is the double precision of IEEE 754-1985 (i.e., the binary64 format of IEEE 754-2008), and the internal precision is that of the double-extended precision format.

The algorithm requires that the two following floating-point numbers be pre-computed:

$$\begin{cases} C_h & = & \mathrm{RN}(C), \\ C_\ell & = & \mathrm{RN}(C - C_h). \end{cases} \tag{5.26}$$

Let us first present Algorithm 5.2.

Algorithm 5.2 Multiplication by C with a multiplication and an FMA [51].

From x, compute

$$\begin{cases} u_1 & = & \mathrm{RN}(C_\ell x), \\ u_2 & = & \mathrm{RN}(C_h x + u_1). \end{cases} \tag{5.27}$$

The result to be returned is u_2.

Beware: we do not claim that for all values of C, this algorithm will return $\mathrm{RN}(Cx)$ for all x. Indeed, it is quite simple to build counterexamples. What we claim is that

- we have reasonably simple methods that make it possible, for a given value of C, to check if Algorithm 5.2 will return $\mathrm{RN}(Cx)$ for all floating-point numbers x;

- in practice, for most usual values of C, Algorithm 5.2 returns $\mathrm{RN}(Cx)$ for all floating-point numbers x.

Note that without the use of an FMA instruction, Algorithm 5.2 would fail to always return a correctly rounded result for all but a few simple (e.g., powers of 2) values of C.

In this section, we will present a simple method that allows one to check, for a given constant C and a given format, if Algorithm 5.2 will return $\mathrm{RN}(Cx)$ for all x. That method is sometimes unable to give an answer. See [51] for a more sophisticated method that always either certifies that Algorithm 5.2 always returns a correctly rounded result, or gives all counterexamples. The method we are going to present is based on the continued fraction theory. Continued fractions are very useful in floating-point arithmetic (for instance, to get worst cases for range reduction of trigonometric functions, see Chapter 11 and [293]). We present some basic results on that theory in the appendix (Section 16.1, page 521).

5.5.1   Checking for a given constant C if Algorithm 5.2 will always work

Without loss of generality, we assume in the following that $1 < x < 2$ and $1 < C < 2$, that C is not exactly representable, and that $C - C_h$ is not a power of 2. Define $x_{\mathrm{cut}} = 2/C$. We will have to separate the cases $x < x_{\mathrm{cut}}$ and $x > x_{\mathrm{cut}}$, because the value of $\mathrm{ulp}(C \cdot x)$ is not the same for these two cases.

The middle of two consecutive floating-point numbers around $C \cdot x$ has the form

$$\frac{2A + 1}{2^p} \qquad \text{if } x < x_{\mathrm{cut}},$$

and

$$\frac{2A + 1}{2^{p-1}} \qquad \text{if } x > x_{\mathrm{cut}},$$

where A is an integer between 2^{p-1} and 2^p. Our problem is reduced to knowing if there can be such a midpoint between $C \cdot x$ and the value u_2 returned by Algorithm 5.2. If this is not the case, then, necessarily,

$$u_2 = \mathrm{RN}(C \cdot x).$$

Hence, first, we must bound the distance between u_2 and $C \cdot x$.

In this book, we will assume that the intermediate calculations are performed in the "target" format (see [51] for the general case).

One has the following property; see [51, Property 2]:

Property 13. *Define* $\epsilon_1 = |C - (C_h + C_\ell)|$.

- *If* $x < x_{cut} - 2^{-p+2}$, *then* $|u_2 - Cx| < \frac{1}{2}\operatorname{ulp}(u_2) + \eta$,

- *If* $x \geq x_{cut} + 2^{-p+2}$, *then* $|u_2 - Cx| < \frac{1}{2}\operatorname{ulp}(u_2) + \eta'$,

where

$$
\begin{cases}
\eta & = & \dfrac{1}{2}\operatorname{ulp}(C_\ell x_{cut}) + \epsilon_1 x_{cut}, \\
\eta' & = & \operatorname{ulp}(C_\ell) + 2\epsilon_1.
\end{cases}
$$

Property 13 tells us that u_2 is within $\frac{1}{2}\operatorname{ulp}(u_2) + \eta$ or $\frac{1}{2}\operatorname{ulp}(u_2) + \eta'$ from $C \cdot x$, where η and η' are very small. What does this mean?

- u_2 is within $\frac{1}{2}\operatorname{ulp}(u_2)$ from $C \cdot x$. In such a case,[7] $u_2 = \operatorname{RN}(C \cdot x)$, which is the desired result;

- or (see Figure 5.3), $C \cdot x$ is very close (within distance η or η') from the exact middle of two consecutive floating-point numbers. Depending on whether $x < x_{\text{cut}}$ or not, this means that $C \cdot x$ is very close to a number of the form $(2A + 1)/2^p$ or $(2A + 1)/2^{p-1}$, which, by defining

$$
X = 2^{p-1}x,
$$

means that $2C$ or C is very close to a rational number of the form

$$
\frac{2A+1}{X}.
$$

Hence, our problem is reduced to examining the best possible rational approximations to C or $2C$, with denominator bounded by $2^p - 1$. This is a typical continued fraction problem. Using Theorem 50 (in the Appendix, Chapter 16, page 523), one can prove the following result [51].

---

[7]Unless u_2 is a power of 2, but this case is easily handled separately.
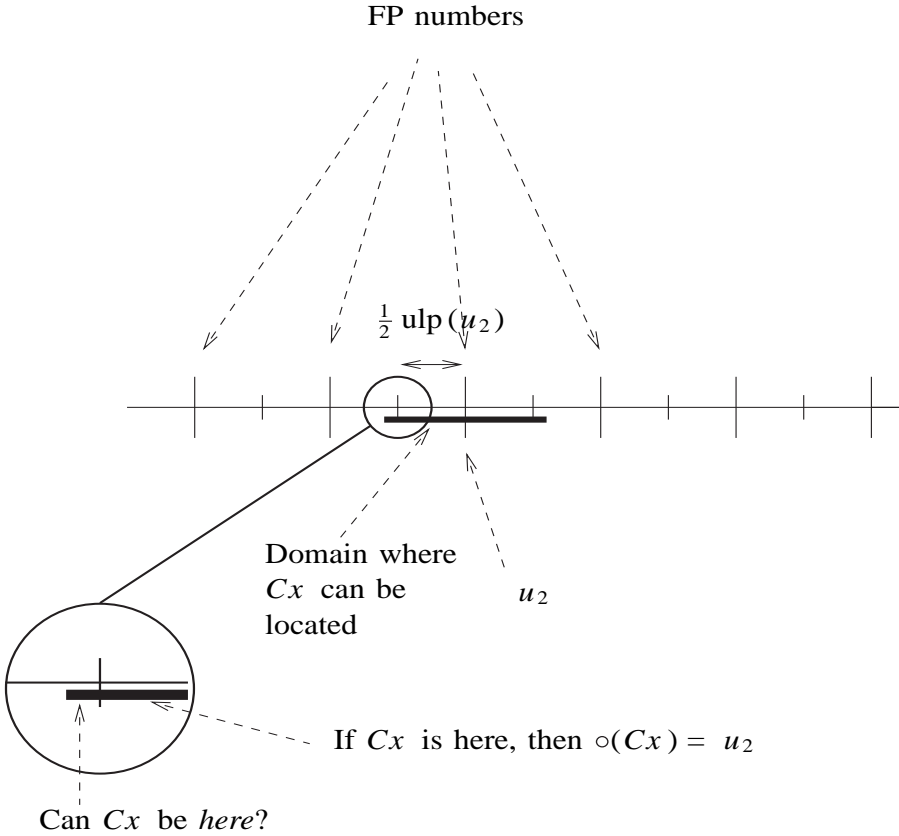
FP numbers

Figure 5.3: We know that Cx is within $\frac{1}{2}\operatorname{ulp}(u_2) + \eta$ (or η') from the floating-point (FP) number u_2, where η is less than 2^{-2p+1}. If we can show that Cx cannot be at a distance less than or equal to η (or η') from the midpoint of two consecutive FP numbers, then u_2 will be the FP number that is closest to Cx [51]. © IEEE, with permission.

Theorem 22 (Conditions on C and p). *Assume $1 < C < 2$. Let $x_{cut} = 2/C$ and $X_{cut} = \left\lfloor 2^{p-1}x_{cut} \right\rfloor$.*

- *If*

$$X = 2^{p-1}x \le X_{cut}$$

  and

$$\epsilon_1 x_{cut} + \frac{1}{2}\operatorname{ulp}(C_\ell x_{cut}) \le \frac{1}{2^{p+1}X_{cut}}$$

  (where ϵ_1 is defined in Property 13) then Algorithm 5.2 will always return a correctly rounded result, except possibly if X is a multiple of the denominator of a convergent n/d of $2C$ for which

$$|2Cd - n| < \frac{2^p}{\lceil 2^{p-1}/d \rceil}\left(\epsilon_1 x_{cut} + \frac{1}{2}\operatorname{ulp}(C_\ell x_{cut})\right);$$

- *If*

$$X = 2^{p-1}x > X_{cut}$$

  and

$$2^{2p+1}\epsilon_1 + 2^{2p-1}\operatorname{ulp}(2C_\ell) \le 1,$$

  then Algorithm 5.2 will always return a correctly rounded result, except possibly if X is a multiple of the denominator of a convergent n/d of C for which

$$|Cd - n| < \epsilon_1 d + \frac{2^{n-1}}{\lceil X_{cut}/d \rceil}\operatorname{ulp}(C_\ell).$$

Hence, to check whether Algorithm 5.2 will always return a correctly rounded result, it suffices to compute the first convergents of C and $2C$ (those of denominator less than 2^p).

Table 5.2 gives some results obtained using this method ("Method 2" in the table) and two other methods, presented in [51]. Method 3 is the most complex, but it always gives an answer (either it certifies, for a given C, that the algorithm will always return $\operatorname{RN}(C \cdot x)$, or it returns all the counterexamples). From Table 5.2, one can for instance deduce the following result.

Theorem 23 (Correctly rounded multiplication by π [51]). *Algorithm 5.2 always returns a correctly rounded result in the double-precision/binary64 format with $C = 2^j\pi$, where j is any integer, provided no under/overflow occurs.*

Hence, in this case, multiplying by π with correct rounding only requires two consecutive FMAs.

If a wider internal format of precision $p + g$ is available then it is possible to slightly modify Algorithm 5.2 to get an algorithm that works for more values of C. See [51], as well as `http://perso.ens-lyon.fr/jean-michel.muller/MultConstant.html`, for more details.

5.6 Evaluation of the Error of an FMA

We have previously seen that, under some conditions, the error of a floating-point addition or multiplication can be exactly representable using one floating-point number, and is readily computable using simple algorithms. When dealing with the FMA instruction, two natural questions arise:

- How many floating-point numbers are necessary for representing the error of an FMA?

- Can these numbers be easily calculated?

Boldo and Muller [35] studied that problem, in the case of radix-2 arithmetic and assuming rounding to nearest. They showed that two floating-point numbers always suffice for representing the error of an FMA, and

C	p	Method 1	Method 2	Method 3
π	8	Does not work for 226	Does not work for 226	AW unless $X =$ 226
π	24	unable	unable	AW
π	53	AW	unable	AW
π	64	unable	AW	AW
π	113	AW	AW	AW
$1/\pi$	24	unable	unable	AW
$1/\pi$	53	Does not work for 6081371451248382	unable	AW unless $X =$ 6081371451248382
$1/\pi$	64	AW	AW	AW
$1/\pi$	113	unable	unable	AW
$\ln 2$	24	AW	AW	AW
$\ln 2$	53	AW	unable	AW
$\ln 2$	64	AW	unable	AW
$\ln 2$	113	AW	AW	AW
$\frac{1}{\ln 2}$	24	unable	AW	AW
$\frac{1}{\ln 2}$	53	AW	AW	AW
$\frac{1}{\ln 2}$	64	unable	unable	AW
$\frac{1}{\ln 2}$	113	unable	unable	AW

Table 5.2: Some results obtained using the method presented here (Method 2), as well as Methods 1 and 3 of [51]. The results given for constant C hold for all values $2^{\pm j}C$. "AW" means "always works" and "unable" means "the method is unable to conclude." [51], © IEEE, 2008, with permission.

they gave an algorithm for computing these two numbers. That algorithm is Algorithm 5.3, given below. It uses Algorithm 5.1 (2MultFMA), presented at the beginning of this chapter, as well as Algorithms 4.3 (Fast2Sum) and 4.4 (2Sum) of Chapter 4. The total number of floating-point operations it requires is 20.

Algorithm 5.3 ErrFma(a,x,y).

$r_1 \leftarrow \text{RN}(ax + y)$;
$(u_1, u_2) \leftarrow 2\text{MultFMA}(a, x)$;
$(\alpha_1, \alpha_2) \leftarrow 2\text{Sum}(y, u_2)$;
$(\beta_1, \beta_2) \leftarrow 2\text{Sum}(u_1, \alpha_1)$;
$\gamma \leftarrow \text{RN}(\text{RN}(\beta_1 - r_1) + \beta_2)$;
$(r_2, r_3) \leftarrow \text{Fast2Sum}(\gamma, \alpha_2)$;

One can show that if no underflow or overflow occurs, then:

- $ax + y = r_1 + r_2 + r_3$ exactly;

- $|r_2 + r_3| \leq \frac{1}{2}\,\mathrm{ulp}(r_1)$;

- $|r_3| \leq \frac{1}{2}\,\mathrm{ulp}(r_2)$.

Recently, Boldo wrote a formal proof in Coq of that result.[8]

5.7 Evaluation of Integer Powers

Now, we describe a method due to Kornerup et al. for accurately evaluating powers to a positive integer n in binary floating-point arithmetic. We refer to the research report [225] for the proofs of the results that we claim here.

We assume that we use a radix-2 floating-point arithmetic that follows the IEEE 754-1985 or the IEEE 754-2008 standard, in round-to-nearest mode. We also assume that an FMA instruction is available.[9] An important case is when an internal format wider than the "target precision" is available. For example, when the target format is double precision (or, equivalently, binary64) and the internal format is the associated extended format, for "reasonable" values of the power n, we will be able to guarantee correctly rounded powers.

Algorithms Fast2Sum (Algorithm 4.3, page 126) and 2MultFMA (Algorithm 5.1, page 152) both provide exact results for computations of the form $x + y$ and $x \cdot y$. These exact results are represented by pairs (a_h, a_ℓ) of floating-point numbers such that $|a_\ell| \leq \frac{1}{2}\,\mathrm{ulp}(a_h)$. In the following we need products of such pairs. However, we do not need *exact* products: we will be satisfied with *approximations* to the products, obtained by discarding terms of the order of the product of the two low order terms. Given two double-precision operands (a_h, a_ℓ) and (b_h, b_ℓ), the algorithm DblMult (Algorithm 5.4) computes (x, y) such that

$$x + y = [(a_h + a_\ell)(b_h + b_\ell)](1 + \delta),$$

where the relative error δ is given by Theorem 24. Several slightly different versions of algorithm DblMult are possible.

---

[8] See http://lipforge.ens-lyon.fr/www/pff/FmaErr.html.

[9] As a matter of fact, the availability of an FMA is not strictly mandatory. And yet, the algorithm uses 2MultFMA products (Algorithm 5.1). If an FMA is not available, these products must be replaced by Dekker products (Algorithm 4.7), which are much more costly.

Algorithm 5.4 DblMult(a_h, a_ℓ, b_h, b_ℓ), Kornerup et al. [225].

$$
\begin{aligned}
t_{1h} &:= \mathrm{RN}(a_h b_h); \\
t_2 &:= \mathrm{RN}(a_h b_\ell); \\
t_{1\ell} &:= \mathrm{RN}(a_h b_h - t_{1h}); \\
t_3 &:= \mathrm{RN}(a_\ell b_h + t_2); \\
t_4 &:= \mathrm{RN}(t_{1\ell} + t_3); \\
c_h &:= \mathrm{RN}(t_{1h} + t_4); \\
t_5 &:= \mathrm{RN}(c_h - t_{1h}); \\
c_\ell &:= \mathrm{RN}(t_4 - t_5);
\end{aligned}
$$

The result to be returned is (c_h, c_ℓ).

Theorem 24 (Kornerup et al. [225]). *Let $\epsilon = 2^{-p}$, where $p \geq 3$ is the precision of the radix-2 floating-point system used. If $|a_\ell| \leq 2^{-p}|a_h|$ and $|b_\ell| \leq 2^{-p}|b_h|$, then the returned value (x, y) of DblMult(a_h, a_ℓ, b_h, b_ℓ) satisfies*

$$ x + y = (a_h + a_\ell)(b_h + b_\ell)(1 + \alpha), \quad with \quad |\alpha| \leq \eta, $$

where

$$ \eta := 7\epsilon^2 + 18\epsilon^3 + 16\epsilon^4 + 6\epsilon^5 + \epsilon^6. $$

Function DblMult uses 8 floating-point operations: 2 multiplications, 4 additions/subtractions, and 2 FMAs. DblMult is at the heart of the following powering algorithm (Algorithm 5.5).

Algorithm 5.5 IteratedProductPower(x, n), $n \geq 1$, Kornerup et al. [225].

$$
\begin{aligned}
&i := n; \\
&(h, \ell) := (1, 0); \\
&(u, v) := (x, 0); \\
&\textbf{while } i > 1 \textbf{ do} \\
&\quad \textbf{if } (i \bmod 2) = 1 \textbf{ then} \\
&\quad\quad (h, \ell) := \mathrm{DblMult}(h, \ell, u, v); \\
&\quad \textbf{end}; \\
&\quad (u, v) := \mathrm{DblMult}(u, v, u, v); \\
&\quad i := \lfloor i/2 \rfloor; \\
&\textbf{end do}; \\
&(h, \ell) := \mathrm{DblMult}(h, \ell, u, v);
\end{aligned}
$$

The number of floating-point operations used by the IteratedProduct-Power algorithm is between $8(1 + \lfloor \log_2(n) \rfloor)$ and $8(1 + 2\lfloor \log_2(n) \rfloor)$. One can show the following result.

Theorem 25. *The values h and ℓ returned by algorithm IteratedProductPower satisfy*

$$h + \ell = x^n(1 + \alpha), \quad with \quad |\alpha| \le (1 + \eta)^{n-1} - 1,$$

where $\eta = 7\epsilon^2 + 18\epsilon^3 + 16\epsilon^4 + 6\epsilon^5 + \epsilon^6$ is the same value as in Theorem 24.

From Theorem 25 one can deduce that if algorithm Iterated ProductPower is implemented in double-precision/binary64 arithmetic, then $\mathrm{RN}(h + \ell)$ is a *faithful* result (see Section 2.2) for x^n, as long as $n \le 2^{49}$.

To guarantee a correctly rounded result in double-precision/binary64 arithmetic, we must know how close x^n can be to the exact midpoint between two consecutive floating-point numbers. This problem is an instance of the *Table Maker's Dilemma*, which is the main topic of Chapter 12.

For instance, in double-precision arithmetic, the hardest-to-round case for function x^{952} corresponds to

$$x = (1.0101110001101001001000000010110101000110100000100001)_2,$$

for which we have

$$
\begin{aligned}
x^{952} = \; &(\underbrace{1.0011101110011001001111100000100010101010110100100110}_{53 \text{ bits}} 1 \\
&\underbrace{00000000 \cdots 00000000}_{63 \text{ zeros}} 1001 \cdots)_2 \times 2^{423}.
\end{aligned}
$$

For that example, x^n is extremely close to the exact middle of two consecutive double-precision numbers. There is a run of 63 consecutive zeros after the rounding bit. This case is the worst case for all values of n between 3 and 1035.

To get correctly rounded results in double-precision, we will need to run algorithm IteratedProductPower in double-extended precision. In the following RN_d means round to nearest in double precision. When implemented in double-extended precision, Algorithm 5.5 returns two double-extended numbers h and ℓ such that

$$h + \ell = x^n(1 + \alpha), \quad with \quad |\alpha| \le \alpha_{max},$$

where α_{max} is given by Theorem 25.

Using that bound and worst cases for the correct rounding of functions x^n that are presented in Section 12.5.2 (page 458), one can show the following result.

Theorem 26. *If algorithm IteratedProductPower is performed in double-extended precision, and if $3 \le n \le 1035$, then $\mathrm{RN}_d(h + \ell) = \mathrm{RN}_d(x^n)$: hence by rounding $h + \ell$ to the nearest double-precision number, we get a correctly rounded result.*

Chapter 6

Enhanced Floating-Point Sums, Dot Products, and Polynomial Values

IN THIS CHAPTER, we focus on the computation of sums and dot products, and on the evaluation of polynomials in IEEE 754 floating-point arithmetic.[1] Such calculations arise in many fields of numerical computing. Computing sums is required, e.g., in numerical integration and the computation of means and variances. Dot products appear everywhere in numerical linear algebra. Polynomials are used to approximate many functions (see Chapter 11).

Many algorithms have been introduced for each of these tasks (some of them will be presented later on in this chapter), usually together with some backward, forward, or running/dynamic error analysis. See for example [182, Chapters 3, 4, 5] and [222, 354, 266].

Our goal here is not to add to these algorithms but rather to observe how a floating-point arithmetic compliant with one of the IEEE standards presented in Chapter 3 can be used to provide validated *running error bounds* on and/or *improved accuracy* of the results computed by various algorithms. The consequence is enhanced implementations that need neither extended precision nor interval arithmetic but only the current working precision. In all that follows, we assume that the arithmetic is correctly rounded, and, more specifically, that it follows the IEEE 754-1985 or IEEE 754-2008 standard for floating-point arithmetic.

Providing a validated *running error bound* means being able to compute on the fly, during the calculation of a sum, a dot product, or a polynomial value, a floating-point number that is a mathematically true error bound on the result of that calculation, and that can be computed using only standard floating-point operations (just like the error terms of, say, $a + b$). Such bounds

---

[1]Section 9.7 will survey how these tasks may be accelerated using specific hardware.

follow from some basic properties of IEEE 754-2008 floating-point arithmetic, which we shall review first.

Providing *improved accuracy* means that we are able to return a useful result even if the problem is ill conditioned (e.g., when computing the sum $a_1 + a_2 + \cdots + a_n$, if $|\sum_{i=1}^{n} a_i|$ is very small in front of $\sum_{i=1}^{n} |a_i|$). More precisely, we wish to obtain results that are approximately as accurate as if the intermediate calculations were performed in, say, double-word or triple-word arithmetic (see Chapter 14), without having to pay the cost (in terms of computation time, of code size, and clarity) of such an arithmetic. To do that, we will frequently use "compensated algorithms": In Chapters 4 and 5, we have studied some tricks (2Sum, Fast2Sum, Dekker product, 2MultFMA) that allow one to retrieve the error of a floating-point addition or multiplication.[2] It is therefore tempting to use these tricks to somehow *compensate* for the rounding errors that occur in a calculation. The first compensated algorithm was due to Kahan (see Section 6.3.2).

Although we will only focus here on sums, dot products, and polynomials, compensated algorithms can be built for other numerical problems. Some numerical algorithms that are simple enough (we need some kind of "linearity") can be transformed into compensated algorithms automatically. This is the underlying idea behind Langlois's CENA method [241]. Rump and Böhm also suggested a way of automatically improving some numerical calculations [350].

In this chapter overflows are ignored and all input values are assumed to be exactly representable by floating-point numbers (which means that we do not include a possible preliminary rounding in the error analyses).

6.1   Preliminaries

We collect here some notation and basic facts needed later that follow from the definition of floating-point arithmetic given in Chapter 2. Most of these basic facts have already been mentioned in previous chapters, but we review them here, to make this chapter (almost) self-contained.

Recall that the smallest positive subnormal number is

$$\alpha = \beta^{e_{\min}-p+1},$$

that the smallest positive normal number is

$$\beta^{e_{\min}},$$

and that the largest finite floating-point number is

$$\Omega = (\beta - \beta^{1-p}) \cdot \beta^{e_{\max}}.$$

---

[2]Under some conditions. See Chapters 4 and 5 for more details.

Also (see Definition 6 in Chapter 2, page 39), we remind the reader that the *unit roundoff* **u** is

$$\mathbf{u} = \begin{cases} \dfrac{1}{2}\beta^{1-p} & \text{in round-to-nearest mode,} \\ \beta^{1-p} & \text{in the other rounding modes.} \end{cases}$$

6.1.1 Floating-point arithmetic models

Several models have been proposed in the literature to analyze numerical algorithms that use floating-point arithmetic [186, 55, 25, 60, 182]. A widely used property is the following. Let x and y be two floating-point numbers and let op $\in \{+, -, \times, /\}$. If $\beta^{e_{\min}} \le |x \text{ op } y| \le \Omega$ then no underflow/overflow[3] occurs when computing $x \text{ op } y$, and there exist some real numbers ϵ_1 and ϵ_2 such that

$$\begin{aligned} \circ(x \text{ op } y) &= (x \text{ op } y)(1 + \epsilon_1) & \text{(6.1a)} \\ &= (x \text{ op } y)/(1 + \epsilon_2), & |\epsilon_1|, |\epsilon_2| \le \mathbf{u}. & \text{(6.1b)} \end{aligned}$$

See Section 2.2.3, page 23, for an explanation. Identity (6.1a) corresponds to what is called the *standard model* of floating-point arithmetic (see [182, p. 40]). Identity (6.1b) proves to be extremely useful for running error analysis and exact error-bound derivation, as we will see below.

The identities in (6.1) assume that no underflow occurs. If we want to take into account the possibility of underflow, we must note that:

- if underflow occurs during addition/subtraction, then *the computed result is the exact result* (this is Theorem 3 in Chapter 4, page 124, see also [176], [182, Problem 2.19]). Thus, (6.1) still holds (indeed, with $\epsilon_1 = \epsilon_2 = 0$) for op $\in \{+, -\}$ in the case of underflows;

- however, if op $\in \{\times, /\}$ and underflow may occur, then the preceding model must be modified as follows: there exist some real numbers ϵ_1, ϵ_2, η_1, η_2 such that

$$\begin{aligned} \circ(x \text{ op } y) &= (x \text{ op } y)(1 + \epsilon_1) + \eta_1 & \text{(6.2a)} \\ &= (x \text{ op } y)/(1 + \epsilon_2) + \eta_2, & \text{(6.2b)} \end{aligned}$$

  with

$$|\epsilon_1|, |\epsilon_2| \le \mathbf{u}, \quad |\eta_1|, |\eta_2| \le \alpha, \quad \epsilon_1 \eta_1 = \epsilon_2 \eta_2 = 0. \quad \text{(6.2c)}$$

From now on we will restrict our discussion to (6.1) for simplicity, thus assuming that no underflow occurs. More general results that do take possible underflows into account using (6.2) can be found in the literature (see for instance [266, 354]).

---

[3]See the *note on underflow*, Section 2.1, page 18.

6.1.2   Notation for error analysis and classical error estimates

Error analysis using the "standard model" (6.1) often makes repeated use of factors of the form $1 + \epsilon_1$ or $1/(1 + \epsilon_2)$, with $|\epsilon_1|, |\epsilon_2| \leq \mathbf{u}$ (see for instance the example of iterated products, given in Section 2.6.3, page 37). A concise way of handling such terms is through the θ_n and γ_n notation defined by Higham in [182, p. 63]:

Definition 9 (θ_n and γ_n). *For ϵ_i such that $|\epsilon_i| \leq \mathbf{u}$, $1 \leq i \leq n$, and assuming $n\mathbf{u} < 1$,*

$$\prod_{i=1}^{n}(1 + \epsilon_i)^{\pm 1} = 1 + \theta_n,$$

where

$$|\theta_n| \leq \frac{n\mathbf{u}}{1 - n\mathbf{u}} =: \gamma_n.$$

Notice that if $n \ll 1/\mathbf{u}$, $\gamma_n \approx n\mathbf{u}$. Such quantities enjoy many properties, among which (see [182, p. 67]):

$$\gamma_n \leq \gamma_{n+1}. \tag{6.3}$$

Let us now see the kind of error bounds that can be obtained by combining the standard model with the above θ_n and γ_n notation, focusing for instance on the following three classical algorithms (Algorithms 6.1 through 6.3):[4]

Algorithm 6.1 Algorithm RecursiveSum(a).

$r \leftarrow a_1$
for $i = 2$ to n **do**
  $r \leftarrow \circ(r + a_i)$
end for
return r

Algorithm 6.1 is the straightforward algorithm for computing

$$a_1 + a_2 + \cdots + a_n.$$

More sophisticated algorithms will be given in Section 6.3. Similarly, Algorithm 6.2 is the straightforward algorithm for computing

$$a_1 \cdot b_1 + a_2 \cdot b_2 + \cdots + a_n \cdot b_n.$$

---
[4]These bounds are given in [182, p. 82, p. 63, p. 95].

Algorithm 6.2 Algorithm RecursiveDotProduct(a,b).

$r \leftarrow \circ(a_1 \times b_1)$
for $i = 2$ to n **do**
  $r \leftarrow \circ\big(r + \circ(a_i \times b_i)\big)$
end for
return r

Algorithm 6.3 computes $p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_0$ using Horner's rule. See Section 6.5 for a "compensated" Horner algorithm.

Algorithm 6.3 Algorithm Horner(p,x).

$r \leftarrow a_n$
for $i = n - 1$ downto 0 **do**
  $r \leftarrow \circ\big(\circ(r \times x) + a_i\big)$
end for
return r

Let us first consider recursive summation (Algorithm 6.1). In that algorithm, the first value of variable r (after the first iteration of the **for** loop) is

$$(a_1 + a_2)(1 + \epsilon_1),$$

with $|\epsilon_1| \le \mathbf{u}$. That is, that value of r can be rewritten as

$$(a_1 + a_2)(1 + \theta_1),$$

where the notation θ_1 is introduced in Definition 9. The second value of variable r (after the second iteration of the **for** loop) is

$$((a_1 + a_2)(1 + \epsilon_1) + a_3)(1 + \epsilon_2) \quad \text{with} \quad |\epsilon_2| \le \mathbf{u}$$
$$= (a_1 + a_2)(1 + \theta_2) + a_3(1 + \theta_1),$$

and, by a straightforward induction, the last value of r has the form

$$(a_1 + a_2)(1 + \theta_{n-1}) + a_3(1 + \theta_{n-2}) + a_4(1 + \theta_{n-3}) + \cdots + a_n(1 + \theta_1).$$

Using (6.3), we obtain the absolute forward error bound

$$\left| \text{RecursiveSum}(a) - \sum_{i=1}^{n} a_i \right| \le \gamma_{n-1} \sum_{i=1}^{n} |a_i|. \tag{6.4}$$

Similarly, it follows for Algorithm 6.2 that

$$\left| \text{RecursiveDotProduct}(a, b) - \sum_{i=1}^{n} a_i \cdot b_i \right| \le \gamma_n \sum_{i=1}^{n} |a_i \cdot b_i|, \tag{6.5}$$

and, for Algorithm 6.3, that

$$|\mathrm{Horner}(p, x) - p(x)| \leq \gamma_{2n} \sum_{i=0}^{n} |a_i| \cdot |x|^i. \tag{6.6}$$

Notice that if a fused multiply-add (FMA, see Section 2.8, page 51) instruction is available, Algorithms 6.2 and 6.3 can be rewritten so that they become simpler, faster (the number of operations is halved) and, in general, slightly more accurate (for polynomial evaluation, γ_{2n} is replaced by γ_n so that the error bound is roughly halved).[5]

    An important notion in numerical analysis is that of *backward error*, introduced by Wilkinson [435]. Assume we wish to compute $y = f(x)$. Instead of y we compute some value $\hat{y}$ (that we hope is close to y). In most cases, $\hat{y}$ is the exact value of f at some locally unique point $\hat{x}$ (that we hope is close to x).

- The *backward error* of this computation is

$$|x - \hat{x}|,$$

- and the *relative backward error* is

$$\left| \frac{x - \hat{x}}{x} \right|.$$

When there might be some ambiguity, the usual absolute and relative errors, namely

$$|y - \hat{y}|$$

and

$$|y - \hat{y}|/|y|$$

are called the *forward error* and *relative forward error*, respectively.

    In Equations (6.4), (6.5), and (6.6), the values γ_{n-1}, γ_n, and γ_{2n} are upper bounds on the backward relative error of the computation. These equations show that recursive summation and dot product have a small backward error if $n\mathbf{u} \ll 1$, as well as Horner's algorithm if $2n\mathbf{u} \ll 1$ (which holds in all practical cases: for instance, in double-precision/binary64 arithmetic, nobody evaluates a polynomial of degree 2^{50}).

---

[5]Which, of course, does not imply that the error itself is halved.

And yet, the *forward* relative errors of these algorithms can be arbitrarily large if the *condition numbers*

$$C_{\text{summation}} \;=\; \frac{\displaystyle\sum_{i=1}^{n} |a_i|}{\left| \displaystyle\sum_{i=1}^{n} a_i \right|} \qquad\qquad \text{(summation)}$$

$$C_{\text{dot product}} \;=\; \frac{2 \cdot \displaystyle\sum_{i=1}^{n} |a_i \cdot b_i|}{\left| \displaystyle\sum_{i=1}^{n} a_i \cdot b_i \right|} \qquad\qquad \text{(dot product)}$$

$$C_{\text{Horner}} \;=\; \frac{\displaystyle\sum_{i=0}^{n} |a_i| \cdot |x|^i}{\left| \displaystyle\sum_{i=0}^{n} a_i \cdot x^i \right|} \qquad\qquad \text{(polynomial evaluation)}$$

are too large.

6.1.3 Properties for deriving validated running error bounds

Together with the definition of $\mathbf{u}$, Equation (6.1) yields a number of properties that will prove useful for deriving validated error bounds.

Property 14. *Let x, y, z be non-negative floating-point numbers. If underflow does not occur, then $xy + z \le \circ(\circ(x \times y) + z)(1 + \mathbf{u})^2$.*

Proof. Applying (6.1b) to xy gives $xy + z = \circ(x \times y)(1 + \epsilon) + z$, $|\epsilon| \le \mathbf{u}$. Since $xy + z$, $\circ(x \times y)$, and z are all non-negative, we deduce that

$$
\begin{aligned}
xy + z &\le& \circ(x \times y)|1 + \epsilon| + z \\
&\le& \circ(x \times y)(1 + \mathbf{u}) + z \\
&\le& (\circ(x \times y) + z)(1 + \mathbf{u}).
\end{aligned}
$$

Applying (6.1b) to the sum $\circ(x \times y) + z$ gives further

$$
\begin{aligned}
\circ(x \times y) + z &=& \circ(\circ(x \times y) + z)(1 + \epsilon'), \quad |\epsilon'| \le \mathbf{u}, \\
&=& \circ(\circ(x \times y) + z)|1 + \epsilon'| \\
&\le& \circ(\circ(x \times y) + z)(1 + \mathbf{u}),
\end{aligned}
$$

which concludes the proof. $\square$

Property 15. *If the radix is even, $p \leq -e_{\min}$ (these two conditions always hold in practice), and if k is a positive integer such that $k\mathbf{u} < 1$, then $1 - k\mathbf{u}$ is a normal floating-point number.*

Proof. Recall that $\mathbf{u}$ is equal to $\frac{1}{2}\beta^{1-p}$ in round-to-nearest, and to β^{1-p} in the other rounding modes. Since $p > 0$ both k and $\mathbf{u}^{-1}$ are positive integers. Thus, $k\mathbf{u} < 1$ implies $\mathbf{u} \leq 1 - k\mathbf{u} < 1$. Since $p \leq -e_{\min}$ and $\beta \geq 2$ is even, both possible values for $\mathbf{u}$ are at least $\beta^{e_{\min}}$. Consequently, $\beta^{e_{\min}} \leq 1 - k\mathbf{u} < 1$.

Hence, there exist a real μ and an integer e such that

$$1 - k\mathbf{u} = \mu \cdot \beta^{e-p+1},$$

with $\beta^{p-1} \leq \mu < \beta^p$ and $e_{\min} \leq e < 0$. Writing $\mu = (\mathbf{u}^{-1} - k)\mathbf{u}\,\beta^{-e+p-1}$, it follows that μ is a positive integer as the product of the positive integers $\mathbf{u}^{-1} - k$ and $\mathbf{u}\,\beta^{-e+p-1}$. □

Property 16 is a direct consequence of [314, Lemma 2.3] and Property 15.

Property 16 (Rump et al. [314]). *Let k be a positive integer such that $k\mathbf{u} < 1$ and let x be a floating-point number such that $\beta^{e_{\min}} \leq |x| \leq \Omega$. Then*

$$(1 + \mathbf{u})^{k-1}|x| \leq \circ\left(\frac{|x|}{1 - k\mathbf{u}}\right).$$

6.2   Computing Validated Running Error Bounds

Equations (6.4), (6.5), and (6.6) give error bounds for three basic algorithms. These error bounds contain expressions such as $\gamma_{2n} \sum_{i=0}^{n} |a_i| \cdot |x|^i$ (for polynomial evaluation) that make them difficult to check using floating-point arithmetic only. Hence, it is interesting to design algorithms that compute a validated error bound on the fly.

Ogita, Rump, and Oishi give a compensated algorithm for the dot product with running error bound in [313]. Below, Algorithm 6.4 is a modification of [182, Algorithm 5.1] which evaluates a polynomial using Horner's rule and provides a validated running error bound.

Algorithm 6.4 This algorithm computes a pair (r, b) of floating-point numbers such that $r = \text{Horner}(p, x)$ and $|r - p(x)| \leq b$, provided no underflow or overflow occurs.

$r \leftarrow a_n;$
$s \leftarrow \circ(|a_n|/2);$
for $i = n - 1$ downto 1 **do**
$\quad r \leftarrow \circ\big(\circ(r \times x) + a_i\big);$
$\quad s \leftarrow \circ\big(\circ(s \times |x|) + |r|\big);$
end for
$r \leftarrow \circ\big(\circ(r \times x) + a_0\big);$
$b \leftarrow \circ\big(2 \times \circ(s \times |x|) + |r|\big);$
$b \leftarrow \mathbf{u} \times \circ\big(b/(1 - (2n - 1)\mathbf{u})\big);$
return $(r, b);$

If an FMA instruction is available, then the core of Horner's loop in Algorithm 6.4 obviously becomes $r \leftarrow \circ(r \times x + a_i)$. This results in a faster and slightly better algorithm.

Following the analysis of Horner's rule given in [182, page 95], we arrive at the result below.

Theorem 27. *If no underflow or overflow occurs, then Algorithm 6.4 computes in $4n + 1$ flops a pair of normal floating-point numbers (r, b) such that*

$$\left| r - \sum_{i=0}^{n} a_i x^i \right| \leq b.$$

Proof. Recalling that Horner's rule in degree n takes exactly $2n$ flops and ignoring operations such as absolute value and multiplication/division by 2, $\mathbf{u}$, or n, we deduce the flop count of $4n + 1$ for Algorithm 6.4.

Now, for the error bound, following [182, page 95], let r_i be the value of r after the loop of index i, so that

$$r_i = \circ\big(\circ(r_{i+1} \cdot x) + a_i\big)$$

for $0 \leq i < n$, and $r_n = a_n$. Using (6.1a) and (6.1b), we obtain

$$(1 + \epsilon_i)r_i = r_{i+1}x(1 + \delta_i) + a_i, \quad |\epsilon_i|, |\delta_i| \leq \mathbf{u}.$$

Now, for $0 \leq i \leq n$, define $q_i = \sum_{h=i}^{n} a_h x^{h-i}$ and $e_i = r_i - q_i$. We have $e_n = 0$ and, for $1 \leq i < n$, using $q_i = q_{i+1}x + a_i$ allows one to deduce from the above equation that

$$e_i = xe_{i+1} + \delta_i x r_{i+1} - \epsilon_i r_i.$$

Taking absolute values, we get $|e_i| \leq |x||e_{i+1}| + \mathbf{u}\big(|x||r_{i+1}| + |r_i|\big)$. Using $e_n = 0$ further leads to

$$\left| r - \sum_{i=0}^{n} a_i x^i \right| = |e_0| \leq \mathbf{u}E_0,$$

with E_0 given by the following recurrence:

$$E_n = 0 \quad \text{and, for } n > i \geq 0, \quad E_i = \big(E_{i+1} + |r_{i+1}|\big)|x| + |r_i|.$$

Therefore, $E_0 = |r_n x^n| + 2\sum_{i=1}^{n-1} |r_i x^i| + |r_0|$. Since $r_n = a_n$ and $r_0 = r$, this can be rewritten as

$$E_0 = 2S(|x|) \cdot |x| + |r|,$$

where

$$S(x) = \frac{|a_n|}{2}x^{n-1} + \sum_{i=0}^{n-2} |r_{i+1}|x^i.$$

Since S is a polynomial of degree $n-1$ with non-negative coefficients only, we deduce from Property 14 that $S(|x|) \leq s \cdot (1 + \mathbf{u})^{2n-2}$, where s is the floating-point number obtained at the end of Algorithm 6.4. The conclusion follows using Property 16 with $k = 2n - 1$.   □

Notice that the validated running error bound b computed by Algorithm 6.4 is obtained at essentially the same cost as the running error estimate of [182, Algorithm 5.1]. The paper by Ogita, Rump, and Oishi [313] and Louvet's Ph.D. dissertation [266] are good references for other examples of validated running error bounds.

6.3   Computing Sums More Accurately

As stated in the beginning of this chapter, many numerical problems require the computation of sums of many floating-point numbers. In [181] and later on in [182], Higham gives a survey on summation methods. Interesting information can also be found in [354]. Here, we will just briefly present the main results: the reader should consult these references for more details.

We will first deal with methods that generalize the straightforward RecursiveSum algorithm (Algorithm 6.1). After that, we will present some methods that use the Fast2Sum and 2Sum algorithms presented in Chapter 4, pages 126 and 130 (we remind the reader that these algorithms compute the error of a rounded-to-nearest floating-point addition. It is therefore tempting to use them to somehow compensate for the rounding errors).

In this section, we want to evaluate, as accurately as possible, the sum of n floating-point numbers, $x_1, x_2, \ldots, x_n$.

6.3.1   Reordering the operands, and a bit more

When considering the RecursiveSum algorithm (Algorithm 6.1), conventional methods for improving accuracy consist in preliminarily sorting the input values, so that

$$|x_1| \leq |x_2| \leq \cdots \leq |x_n|$$

(increasing order), or even sometimes

$$|x_1| \geq |x_2| \geq \cdots \geq |x_n|$$

(decreasing order). Another common strategy (yet expensive in terms of comparisons), called *insertion* summation, consists in first sorting the x_i's by increasing order of magnitude, then computing $\circ(x_1 + x_2)$, and inserting that result in the list x_3, x_4, ..., x_n, so that the increasing order is kept, and so on. We stop when there remains only one element in the list: that element is the approximation to $\sum_{1 \leq i \leq n} x_i$.

To analyze a large class of similar algorithms, Higham defines in [182, page 81] a general algorithm expressed as Algorithm 6.5.

Algorithm 6.5 General form for a large class of addition algorithms (Higham, Algorithm 4.1 of [182]).

Let $\mathcal{S} = \{x_1, x_2, \ldots, x_n\}$
while $\mathcal{S}$ contains more than one element **do**
    Remove two numbers x and y from $\mathcal{S}$ and add $\circ(x + y)$ to $\mathcal{S}$.
end while
Return the remaining element of $\mathcal{S}$.

Note that since the number of elements of $\mathcal{S}$ decreases by one at each iteration, this algorithm always performs $n - 1$ floating-point additions (the **while** loop can be replaced by a **for** loop).

If T_i is the result of the i-th addition of Algorithm 6.5, Higham shows that the final returned result, say $s = T_{n-1}$, satisfies

$$\left| s - \sum_{i=1}^{n} x_i \right| \leq \mathbf{u} \sum_{i=1}^{n-1} |T_i|, \tag{6.7}$$

where, as in the previous sections, $\mathbf{u}$ is the *unit roundoff* defined in Chapter 2, page 39 (Definition 6).

Hence, a good strategy is to minimize the terms $|T_i|$. This explains some properties:

- although quite costly, insertion summation is a rather accurate method (as pointed out by Higham, if all the x_i's have the same sign, this is the best method among those that are modeled by Algorithm 6.5);

- When all the x_i's have the same sign, ordering the input values in increasing order gives the smallest bound among the recursive summation methods.[6]

---

[6]It gives the smallest *bound*, which does not mean that it will always give the smallest *error*.

Also, when there is much cancellation in the computation (that is, when

$$\left|\sum_{i=1}^{n} x_i\right|$$

is much less than $\sum_{i=1}^{n} |x_i|$), Higham suggests that recursive summation with the x_i sorted by decreasing order is likely to give better results than that using increasing ordering (an explanation of this apparently strange phenomenon is Sterbenz's lemma, Chapter 4, page 122: when x is very close to y, the subtraction $x - y$ is performed exactly). Table 6.1 presents an example of such a phenomenon.

6.3.2 Compensated sums

The algorithms presented in the previous section could be at least partly analyzed just by considering that, when we perform an addition $a + b$ and no underflow occurs, the computed result is equal to

$$(a+b)(1+\epsilon),$$

with $|\epsilon| \leq \mathbf{u}$. Now, we are going to consider algorithms that cannot be so simply analyzed. They use the fact that when the arithmetic operations are correctly rounded (to the nearest), floating-point addition has specific properties that allow for the use of tricks such as Fast2Sum (Algorithm 4.3, page 126).

In 1965 Kahan suggested the following *compensated summation* algorithm (Algorithm 6.6) for computing the sum of n floating-point numbers. Babuška [18] independently found the same algorithm.

Algorithm 6.6 Original version of Kahan's summation algorithm.

```
$s \leftarrow x_1$
$c \leftarrow 0$
for $i = 2$ to $n$ do
    $y \leftarrow \circ(x_i - c)$
    $t \leftarrow \circ(s + y)$
    $c \leftarrow \circ(\circ(t - s) - y)$
    $s \leftarrow t$
end for
return $s$
```

Presented like this, Kahan's algorithm may seem very strange. But we note that in round-to-nearest mode the second and third lines of the **for** loop constitute the Fast2Sum algorithm (Algorithm 4.3, page 126), so that Kahan's algorithm can be rewritten as Algorithm 6.7.

Algorithm 6.7 Kahan's summation algorithm rewritten with a Fast2Sum.

$s \leftarrow x_1$
$c \leftarrow 0$
for $i = 2$ to n **do**
  $y \leftarrow \circ(x_i + c)$
  $(s, c) \leftarrow \text{Fast2Sum}(s, x_i)$
end for
return s

Can we safely use the Fast2Sum algorithm? Notice that the conditions of Theorem 4 (Chapter 4, page 126) are not necessarily fulfilled:[7]

- we are not sure that the exponent of s will always be larger than or equal to the exponent of y;

- furthermore, Algorithm 6.6 is supposed to be used with various rounding modes, not only round-to-nearest.

And yet, even if we cannot be certain (since the conditions of Theorem 4 may not hold) that after the line

$$(s, c) \leftarrow \text{Fast2Sum}(s, y),$$

the new value of s plus c will be *exactly* equal to the old value of s plus y, in practice, they will be quite close. Thus, $(-c)$ *is a good approximation to the rounding error committed when adding y to s*. The elegant idea behind Kahan's algorithm is therefore to subtract that approximation from the next term of the sum, in order to (at least partly) compensate for that rounding error.

Knuth and Kahan show that the final value s returned by Algorithm 6.6 satisfies

$$\left| s - \sum_{i=1}^{n} x_i \right| \leq \left(2\mathbf{u} + \mathcal{O}(n\mathbf{u}^2)\right) \sum_{i=1}^{n} |x_i|. \tag{6.8}$$

This explains why, in general, Algorithm 6.6 will return a very accurate result.

And yet, if there is much cancellation in the computation (that is, if $|\sum_{i=1}^{n} x_i| \ll \sum_{i=1}^{n} |x_i|$), the relative error on the sum can be very large. Priest gives the following example [337]:

- assume binary, precision-p, rounded-to-nearest arithmetic, with $n = 6$, and

---

[7]Indeed, when Kahan introduced his summation algorithm, Theorem 4 of Chapter 4 was not known: Dekker's paper was published in 1971. Hence, it is fair to say that the Fast2Sum algorithm first appeared in Kahan's paper, even if it was without the conditions that must be fulfilled for it to always return the *exact* error of a floating-point addition.

- set $x_1 = 2^{p+1}$, $x_2 = 2^{p+1} - 2$, and $x_2 = x_2 = \cdots = x_6 = -(2^p - 1)$.

The exact sum is 2, whereas the sum returned by Algorithm 6.6 is 3.

To deal with such difficulties, Priest [336, 337] comes up with another idea. He suggests to first sort the input numbers x_i in descending order of magnitude, then to perform the *doubly compensated summation algorithm* shown in Algorithm 6.8.

Algorithm 6.8 Priest's doubly compensated summation algorithm.

$s_1 \leftarrow x_1$
$c_1 \leftarrow 0$
for $i = 2$ to n **do**
$\quad y_i \leftarrow \circ(c_{i-1} + x_i)$
$\quad u_i \leftarrow \circ(x_i - \circ(y_i - c_{i-1}))$
$\quad t_i \leftarrow \circ(y_i + s_{i-1})$
$\quad v_i \leftarrow \circ(y_i - \circ(t_i - s_{i-1}))$
$\quad z_i \leftarrow \circ(u_i + v_i)$
$\quad s_i \leftarrow \circ(t_i + z_i)$
$\quad c_i \leftarrow \circ(z_i - \circ(s_i - t_i))$
end for

Again, the algorithm looks much less arcane if we rewrite it with Fast2Sums as shown in Algorithm 6.9.

Algorithm 6.9 Priest's doubly compensated summation algorithm, rewritten with Fast2Sums.

$s_1 \leftarrow x_1$
$c_1 \leftarrow 0$
for $i = 2$ to n **do**
$\quad (y_i, u_i) \leftarrow \text{Fast2Sum}(c_{i-1}, x_i)$
$\quad (t_i, v_i) \leftarrow \text{Fast2Sum}(s_{i-1}, y_i)$
$\quad z_i \leftarrow \circ(u_i + v_i)$
$\quad (s_i, c_i) \leftarrow \text{Fast2Sum}(t_i, z_i)$
end for

Priest shows the following result.

Theorem 28 (Priest [337]). *In radix-β, precision-p arithmetic, assuming round-to-nearest mode,[8] if $|x_1| \geq |x_2| \geq \cdots \geq |x_n|$ and $n \leq \beta^{p-3}$, then the floating-point number s_n returned by the algorithm satisfies*

$$\left| s_n - \sum_{i=1}^{n} x_i \right| \leq 2\mathbf{u} \left| \sum_{i=1}^{n} x_i \right|.$$

---

[8]Priest proves that result in a more general context, just assuming that the arithmetic is faithful and satisfies a few additional properties. See [337] for more details.

As soon as the x_i's do not all have the same sign, this is a significantly better bound than the one given by formula (6.8). Indeed, if n is not huge ($n \leq \beta^{p-3}$), then the relative error is bounded by $2\mathbf{u}$ even for an arbitrarily large condition number. On the other hand, due to the need for preliminarily sorted input values, this algorithm will be significantly slower than Kahan's algorithm: one should therefore reserve Priest's algorithm for cases where we need very accurate results and we know that there will be some cancellation in the summation (i.e., $|\sum x_i|$ is significantly less than $\sum |x_i|$).

In Kahan's algorithm (Algorithm 6.7), in many practical cases, c will be very small in front of x_i, so that when adding them, a large part of the information contained in variable c may be lost. Indeed, Priest's algorithm also compensates for the error of this addition, hence the name "doubly compensated summation."

To deal with that problem, Pichat [332] and Neumaier [299] independently found the same idea: at step i, the rounding error,[9] say e_i, is still computed due to the addition of x_i. However, instead of immediately subtracting e_i from the next operand, the terms e_k are added together, to get a correcting term e that will be added to s at the end of the computation. See Algorithm 6.10.

Algorithm 6.10 Pichat and Neumaier's summation algorithm [332, 299]. Notice, since Fast2Sum is used, that the radix of the floating-point system must be at most 3 (which means, in practice, that this algorithm should be used in radix 2 only).

$s \leftarrow x_1$
$e \leftarrow 0$
for $i = 2$ to n **do**
  if $|s| \geq |x_i|$ **then**
    $(s, e_i) \leftarrow$ Fast2Sum(s, x_i)
  else
    $(s, e_i) \leftarrow$ Fast2Sum(x_i, s)
  end if
  $e \leftarrow$ RN$(e + e_i)$
end for
return RN$(s + e)$

To avoid tests, the algorithm of Pichat and Neumaier can be rewritten using the 2Sum algorithm (it also has the advantage of working in any radix). This gives the *cascaded summation* algorithm of Rump, Ogita, and Oishi [354]. It always gives exactly the same result as Pichat and Neumaier's algorithm, but it does not need comparisons. See Algorithm 6.11.

---

[9]It was then possible to evaluate that error exactly: Dekker's result was known when Pichat published her paper.

Algorithm 6.11 By rewriting Pichat and Neumaier's summation algorithm with a 2Sum, we get the *cascaded summation* algorithm of Rump, Ogita, and Oishi [354].

$s \leftarrow x_1$
$e \leftarrow 0$
for $i = 2$ to n **do**
    $(s, e_i) \leftarrow 2\text{Sum}(s, x_i)$
    $e \leftarrow \text{RN}(e + e_i)$
end for
return $\text{RN}(s + e)$

Rump, Ogita, and Oishi show the following result.

Theorem 29 (Rump, Ogita, and Oishi [354]). *If Algorithm 6.11 is applied to floating-point numbers x_i, $1 \leq i \leq n$, and if $n\mathbf{u} < 1$, then, even in the presence of underflow, the final result σ returned by the algorithm satisfies*

$$\left| \sigma - \sum_{i=1}^{n} x_i \right| \leq \mathbf{u} \left| \sum_{i=1}^{n} x_i \right| + \gamma_{n-1}^2 \sum_{i=1}^{n} |x_i|.$$

The same result also holds for Algorithm 6.10, since both algorithms output the very same value.

Rump, Ogita, and Oishi generalize their method, by reusing the same algorithm for summing the e_i, in a way very similar to what Pichat suggested in [332]. To present their *K-fold* algorithm, we first modify Algorithm 6.11 as shown in Algorithm 6.12.

Algorithm 6.12 VecSum(x) [354]. Here, p and x are vectors of floating-point numbers: $x = (x_1, x_2, \ldots, x_n)$ represents the numbers to be summed. If we compute $p = \text{Vecsum}(x)$, then p_n is the final value of variable s in Algorithm 6.11, and p_i (for $1 \leq i \leq n - 1$) is variable e_i of Algorithm 6.11.

$p \leftarrow x$
for $i = 2$ to n **do**
    $(p_i, p_{i-1}) \leftarrow 2\text{Sum}(p_i, p_{i-1})$
end for
return p

Then, here is Rump, Ogita, and Oishi's K-fold summation algorithm (Algorithm 6.13).

Algorithm 6.13 K-fold algorithm [354]. It takes a vector $x = (x_1, x_2, \ldots, x_n)$ of floating-point numbers to be added and outputs a result whose accuracy is given by Theorem 30.

> **for** $k = 1$ to $K - 1$ **do**
> $x \leftarrow \text{VecSum}(x)$
> **end for**
> $c = x_1$
> **for** $i = 2$ to $n - 1$ **do**
> $c \leftarrow c + x_i$
> **end for**
> return $x_n + c$

Rump, Ogita, and Oishi prove the following result.

Theorem 30 (Rump, Ogita, and Oishi [354])**.** *If Algorithm 6.13 is applied to floating-point numbers x_i, $1 \leq i \leq n$, and if $4n\mathbf{u} < 1$, then, even in the presence of underflow, the final result σ returned by the algorithm satisfies*

$$\left| \sigma - \sum_{i=1}^{n} x_i \right| \leq (\mathbf{u} + \gamma_{n-1}^2) \left| \sum_{i=1}^{n} x_i \right| + \gamma_{2n-2}^K \sum_{i=1}^{n} |x_i|.$$

Theorem 30 shows that the K-fold algorithm is almost as accurate as a conventional summation in precision Kp followed by a final rounding to precision p. Klein [221] suggests very similar algorithms.

Tables 6.1 and 6.2 give examples of errors obtained using some of the summation algorithms presented in this section. They illustrate the fact that Pichat and Neumaier's and Priest's algorithms give very accurate results. Although slightly less accurate, Kahan's compensated summation algorithm is still of much interest, since it is very simple and fast.

Incidentally, one could wonder whether it is possible to design a very simple summation algorithm that would always return correctly rounded sums. Kornerup, Lefèvre, Louvet, and Muller [226] have recently shown that, under simple conditions, an *RN-addition algorithm without branching* (that is, an algorithm that only uses rounded-to-nearest additions and subtractions, without any test; see Definition 8 in Chapter 4, page 130) cannot always return the correctly rounded-to-nearest sum of 3 or more floating-point numbers. This shows that an "ultimately accurate" floating-point summation algorithm cannot be very simple. And yet, if we accept tests and/or changes of rounding modes, getting the correctly rounded sum of several floating-point numbers is indeed possible, as we will see in Section 6.3.4 in the case of 3 numbers. Notice that Rump, Ogita and Oishi have designed an efficient algorithm, with tests, for the rounded-to-nearest sum of n numbers [353].

Method	Error in ulps
increasing order	18.90625
decreasing order	16.90625
compensated (Kahan)	6.90625
doubly compensated (Priest)	0.09375
Pichat and Neumaier; or Rump, Ogita, and Oishi	0.09375

Table 6.1: Errors of the various methods for $x_i = \mathrm{RN}(\cos(i))$, $1 \leq i \leq n$, with $n = 5000$ and binary32 arithmetic. Notice that all the x_i are exactly representable. The methods of Priest; Pichat and Neumaier; and Rump, Ogita, and Oishi give the best possible result (that is, the exact sum rounded to the nearest binary32 number). The recursive summation method with decreasing ordering is slightly better than the same method with increasing order (which is not surprising: there is much cancellation in this sum), and Kahan's compensated summation method is significantly better than the recursive summation.

Method	Error in ulps
increasing order	6.86
decreasing order	738.9
compensated (Kahan)	0.137
doubly compensated (Priest)	0.137
Pichat and Neumaier; or Rump, Ogita, and Oishi	0.137

Table 6.2: Errors of the various methods for $x_i = \mathrm{RN}(1/i)$, $1 \leq i \leq n$, with $n = 10^5$ and binary32 arithmetic. Notice that all the x_i are exactly representable. The methods of Kahan; Priest; Pichat and Neumaier; and Rump, Ogita, and Oishi give the best possible result (that is, the exact sum rounded to the nearest binary32 number). The recursive summation method with increasing ordering is much better than the same method with decreasing order, which is not surprising since all the x_i's have the same sign.

6.3.3   Implementing a "long accumulator"

Kulisch advocated augmenting the processors with a long accumulator that would enable exact accumulation and dot product [233, 234]. So far, processor vendors have not considered the benefits of this extension to be worth its cost. It will be reviewed, as well as other prospective hardware improvements, in Section 9.7, page 314.

6.3.4   On the sum of three floating-point numbers

Computing the correctly rounded sum of three numbers is sometimes useful. For instance, in the CRlibm elementary function library,[10] several calculations are done using a "triple-double" intermediate format (see Section 14.1, page 494), using functions due to Lauter [244]. To return a correctly rounded result in double-precision/binary64 arithmetic, one must convert a "triple-double" into a binary64 number: this reduces to computing the correctly rounded sum of three floating-point numbers.

For that purpose, Boldo and Melquiond [34] introduce a new rounding mode, *round-to-odd*, $\circ_{\mathrm{odd}}$, defined as follows:

- if x is a floating-point number, then $\circ_{\mathrm{odd}}(x) = x$;

- otherwise, $\circ_{\mathrm{odd}}(x)$ is the value among $\mathrm{RD}(x)$ and $\mathrm{RU}(x)$ whose integral significand is an odd integer.[11]

This rounding mode is not implemented on current architectures, but that could easily be done. Interestingly enough, Boldo and Melquiond show that in radix-2 floating-point arithmetic, using only one rounded-to-odd addition (and a few rounded-to-nearest additions/subtractions), one can easily compute

$$\mathrm{RN}(a + b + c),$$

where a, b, and c are floating-point numbers. Their algorithm is presented in Figure 6.1. Boldo and Melquiond also explain how to emulate rounded-to-odd additions (with a method that requires testing). Listing 6.1 presents a C program that implements their method in the particular case when the input operands are ordered.

Algorithm 6.14, introduced by Kornerup et al. [225], emulates the rounded-to-odd addition required by Boldo and Melquiond's algorithm displayed in Figure 6.1.

---

[10]See `http://lipforge.ens-lyon.fr/www/crlibm/`.
[11]This means, in radix 2, that the least significant bit of the significand is a one.
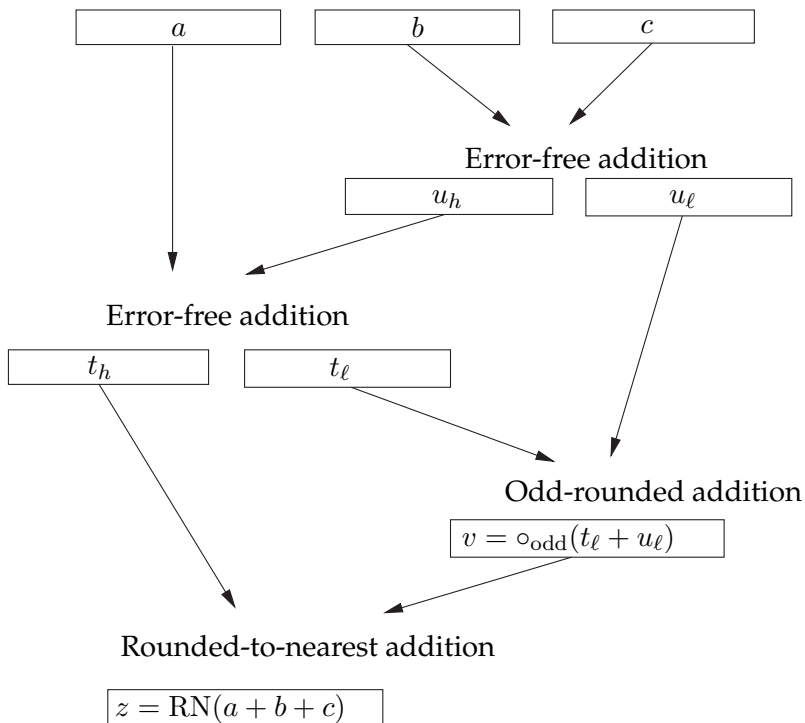
Figure 6.1: Boldo and Melquiond's algorithm [34] for computing $\mathrm{RN}(a + b + c)$ *in radix-2 floating-point arithmetic. It requires an "odd-rounded" addition. The error-free additions are performed using the 2Sum algorithm (unless we know for some reason the ordering of the magnitude of the variables, in which case the Fast2Sum algorithm can be used).* © *IEEE, with permission.*

Algorithm 6.14 OddRoundSum(a,b): computes $\circ_{\mathrm{odd}}(a + b)$ in radix-2 floating-point arithmetic.

$d \leftarrow \mathrm{RD}(a + b)$
$u \leftarrow \mathrm{RU}(a + b)$
$e' \leftarrow \mathrm{RN}(d + u)$
$e \leftarrow e' \times 0.5$
$z' \leftarrow u - e$
$z \leftarrow z' + d$
return z

C listing 6.1 Boldo and Melquiond's program [34] for computing the correctly rounded-to-nearest sum of three binary floating-point numbers x_h, x_m, and x_l, assuming $|x_h| \geq |x_m| \geq |x_l|$. The encoding of the double-precision/binary64 floating-point numbers specified by the IEEE 754 standard is necessary here: it is that encoding that ensures that `thdb.l++` is the floating-point successor of `thdb.l`.

```
double CorrectRoundedSum3(double xh,
double xm, double xl) {
double th, tl;
db_number thdb; // thdb.l is the binary
// representation of th
// Dekker's error-free adder of two ordered
// numbers
Add12(th, tl, xm, xl);
// round to odd th if tl is not zero
if (tl != 0.0) {
thdb.d = th;
// if the significand of th is odd, there is
// nothing to do
if (!(thdb.l & 1)) {
// choose the rounding direction
// depending on the signs of th and tl
if ((tl > 0.0) ^ (th < 0.0))
thdb.l++;
else
thdb.l--;
th = thdb.d;
}
}
// final addition rounded to the nearest
return xh + th;
}
```

6.4 Compensated Dot Products

The dot product of two vectors $[a_i]_{1 \leq i \leq n}$ and $[b_i]_{1 \leq i \leq n}$ is $\sum_{1 \leq i \leq n} a_i \cdot b_i$. When the condition number

$$C_{\text{dot product}} = \frac{2 \cdot \sum_{i=1}^{n} |a_i \cdot b_i|}{\left| \sum_{i=1}^{n} a_i \cdot b_i \right|}$$

is not too large, Algorithm 6.2 can be used safely. When this is not the case, one can design a compensated dot product algorithm.

Notice that one can easily reduce the problem of computing the dot product of two vectors of dimension n to the problem of computing the

sum of $2n$ floating-point numbers, since the Dekker product (Algorithm 4.7, page 135) and the 2MultFMA algorithm (Algorithm 5.1, page 152) make it possible (under some conditions) to deduce, from two floating-point numbers a and b, two floating-point numbers r_1 and r_2 such that

$$r_1 + r_2 = a \cdot b \qquad \text{and} \qquad |r_2| \leq \frac{1}{2}\,\mathrm{ulp}(r_1). \tag{6.9}$$

Hence, many methods presented in Section 6.3 can readily be adapted to the computation of dot products. And yet, by doing that, we do not use the fact that, in Equation (6.9), r_2 is very small in front of r_1: the sum we have to compute is not an arbitrary sum of $2n$ numbers, some are much smaller than others, and that property can be used to design faster algorithms.

Let us now give the compensated dot product algorithm introduced by Ogita, Rump, and Oishi [313]. See Algorithm 6.15. In that algorithm, 2Prod will denote either the 2MultFMA algorithm (if an FMA instruction is available), or the Dekker product. Remember that to be able to use the Dekker product we need the radix of the floating-point system to be equal to 2 or the precision p to be even. Remember also that Equation (6.9) holds if $e_a + e_b \geq e_{\min} + p - 1$, where e_a and e_b are the exponents of a and b, respectively (see Chapter 4 for more details).

Algorithm 6.15 Algorithm CompensatedDotProduct(a,b) [313]. It computes $a_1 \cdot b_1 + a_2 \cdot b_2 + \cdots + a_n \cdot b_n$.

$(s_1, c_1) \leftarrow 2\mathrm{Prod}(a_1, b_1)$
for $i = 2$ to n **do**
   $(p_i, \pi_i) \leftarrow 2\mathrm{Prod}(a_i, b_i)$
   $(s_i, \sigma_i) \leftarrow 2\mathrm{Sum}(p_i, s_{i-1})$
   $c_i \leftarrow \mathrm{RN}(c_{i-1} + \mathrm{RN}(\pi_i + \sigma_i))$
end for
return $\mathrm{RN}(s_n + c_n)$

Many variants of the algorithm can be designed, possibly inspired from the variants of the compensated summation algorithm presented in Section 6.3. Ogita, Rump, and Oishi have shown the following theorem, which says (it suffices to compare to Equation (6.5)) that, in precision p, the result of Algorithm 6.15 is as accurate as the value computed by the straightforward algorithm (Algorithm 6.2) in precision $2p$ and rounded back to the working precision p.

Theorem 31 (Ogita, Rump, and Oishi [313]). *If no underflow or overflow occurs, in radix* 2,

$$\left| CompensatedDotProduct(a, b) - \sum_{i=1}^{n} a_i \cdot b_i \right| \leq \mathbf{u} \left| \sum_{i=1}^{n} a_i \cdot b_i \right| + \gamma_n^2 \sum_{i=1}^{n} |a_i \cdot b_i|.$$

6.5 Compensated Polynomial Evaluation

Recently, Graillat, Langlois, and Louvet [155, 266] introduced a new *compensated algorithm* for polynomial evaluation. Let us present it briefly. Assume we wish to compute

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_0,$$

where x and all the a_i are floating-point numbers. In the following, 2Prod will denote the 2MultFMA algorithm (Algorithm 5.1, page 152) if an FMA instruction is available, and the Dekker product (Algorithm 4.7, page 135) otherwise. Graillat, Langlois, and Louvet first define the following "error-free transformation" (see Algorithm 6.16).

Algorithm 6.16 The Graillat–Langlois–Louvet error-free transformation [155, 266]. Input: a degree-n polynomial $p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_0$. Output: the same result r_0 as the conventional Horner's evaluation of p, and two degree-$(n-1)$ polynomials $\pi(x)$ and $\sigma(x)$, of degree-i coefficients π_i and σ_i, such that $p(x) = r_0 + \pi(x) + \sigma(x)$ exactly.

$r_n \leftarrow a_n$
for $i = n - 1$ downto 0 **do**
  $(p_i, \pi_i) \leftarrow 2\text{Prod}(r_{i+1}, x)$
  $(r_i, \sigma_i) \leftarrow 2\text{Sum}(p_i, a_i)$
end for
return r_0, $(\pi_0, \pi_1, \ldots, \pi_{n-1})$, $(\sigma_0, \sigma_1, \ldots, \sigma_{n-1})$

Define Horner(p, x) as the result returned by Horner's rule (Algorithm 6.3) with polynomial p and input variable x. Also, for a polynomial $q = \sum_{i=0}^{n} q_i x^i$, define

$$\tilde{q}(x) = \sum_{i=0}^{n} |q_i| x^i.$$

We have the following result.

Theorem 32 (Langlois, Louvet [242]). *The values r_0, $(\pi_0, \pi_1, \ldots, \pi_{n-1})$, and $(\sigma_0, \sigma_1, \ldots, \sigma_{n-1})$ returned by Algorithm 6.16 satisfy*

- $r_0 = Horner(p, x)$;

- $p(x) = r_0 + \pi(x) + \sigma(x)$;

- $\widetilde{(\pi + \sigma)}(|x|) \leq \gamma_{2n} \, \tilde{p}(|x|)$.

Theorem 32 suggests to transform Algorithm 6.16 into a compensated polynomial evaluation algorithm as shown in Algorithm 6.17.

Algorithm 6.17 The Graillat–Langlois–Louvet compensated polynomial evaluation algorithm [155, 266]. Input: a degree-n polynomial $p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_0$. Output: an approximation r to $p(x)$. In practice, the evaluation of $\mathrm{Horner}(q, x)$ wit $q(x) = \sum_{0 \le i < n} q_i x^i$ would be done in the **for** loop: we wrote it as shown here for the sake of clarity.

$r_n \leftarrow a_n$
for $i = n - 1$ downto 0 **do**
    $(p_i, \pi_i) \leftarrow 2\mathrm{Prod}(r_{i+1}, x)$
    $(r_i, \sigma_i) \leftarrow 2\mathrm{Sum}(p_i, a_i)$
    $q_i \leftarrow \mathrm{RN}(\pi_i + \sigma_i)$
end for
$r \leftarrow \mathrm{RN}(r_0 + \mathrm{Horner}(q, x))$
return r

An immediate consequence of Theorem 32 is the following.

Theorem 33 (Langlois, Louvet [242]). *The value r returned by Algorithm 6.17 satisfies*

$$|r - p(x)| \le \mathbf{u}|p(x)| + \gamma_{2n}^2\, \tilde{p}(|x|).$$

Note the similarity with Theorem 31. Juste like for dot products, the above result says that as soon as $\tilde{p}(|x|) = \sum_{i=0}^n |a_i| \cdot |x|^i$ is not huge in front of $|p(x)|$, Algorithm 6.17 will return a very accurate result (namely, for a working precision p, as accurate as if we had used Horner's rule in twice that working precision and then rounded back). If this is not the case, it is possible to define *K-fold compensated polynomial evaluation algorithms* by recursively using the same method for evaluating $\sigma(x)$ and $\pi(x)$. See Louvet's Ph.D. dissertation [266] for details.

Chapter 7

Languages and Compilers

THE PREVIOUS CHAPTERS have given an overview of interesting properties and algorithms that can be built on IEEE 754-compliant floating-point arithmetic. In this chapter, we discuss the practical issues encountered when trying to implement such algorithms in actual computers using actual programming languages. In particular, we discuss the relationship between standard compliance, portability, accuracy, and performance. This chapter is useful to a programmer wishing to obtain a standard-compliant behavior from his/her program, but it is also useful for understanding how performance may be improved by relaxing standard compliance and also what risks may be encountered.

7.1 A Play with Many Actors

Even with a computer supporting one of the IEEE 754 standards for floating-point arithmetic, it still requires some effort to be in control of the details of the floating-point computations of one's program (for instance, to ensure portability).

Most programming languages will allow one to write an expression such as a+b+c∗d, using variables a, b, c, and d, of some (possibly implicit) floating-point type. However, when the program is run, the sequence of floating-point operations that is actually executed differs widely, depending on the language, but also on the environment used, including:

- the compiler, and the optimization options that were passed to it;

- the processor, which may or may not have a given floating-point capability;

- the operating system, which is responsible for making the processor's capabilities available for user programs;

- the system libraries, for the mathematical functions (when they are not
  handled by the compiler).

Let us now review these elements to give a taste of what may happen.
The following sections will then detail in more depth the specifics of some
widely used programming environments.

7.1.1 Floating-point evaluation in programming languages

Consider the evaluation of a+b+c+d. Its semantics is a sequence of three
floating-point additions. The results of two of them have to be kept in tem-
porary variables or registers. This leads to several implementation choices:

Expression reordering

In which order should these operations be executed? In other terms, what is
the implicit parenthesizing used when evaluating a+b+c+d? Alternatives are,
on this example, ((a+b)+c)+d, (a+b)+(c+d), a+(b+(c+d)). A related question
is: Should addition be considered associative? Is (a+d)+(b+c) also an option?

There is a tradeoff here. On one side, it should be clear to the reader, after
reading the previous chapters, that floating-point addition is not associative.
A language should allow one to express useful algorithms such as the 2Sum
algorithm, which requires computing (a+b)-a (See Section 4.3.2, page 129).
On the other side, such situations are fairly rare and well identified. In more
general floating-point codes, rewriting freedom allows for many optimiza-
tions.

- (a+b)+(c+d) exposes more parallelism for processors able to compute
  two additions in parallel.

- In general, in a pipelined processor, the fastest parenthesizing depends
  on the order of availability of the four variables, which itself depends
  on previous computations.

- If a and d are compile-time constants, computing the sum a+d at com-
  pile time will save one addition at execution time.

- The parenthesizing may impact register allocation, etc.

Precision of intermediate computations

Many languages require the programmer to declare the variables of a given
type, say binary32 (`float` in C) or binary64 (`double` in C). However, they usu-
ally do not require the precision to be declared for each *operation* of the code
(assembly languages, of course, do). Deducing the precision of the operations
from the precision of the data is not straightforward. Consider, for instance,
the following situations.

- If a, b, c, and d are declared of binary32 type, and the hardware is able to compute on binary64 as fast as on binary32, shouldn't this "free" extra accuracy be used?

- In an assignment such as r=a+b+c+d, where r is declared as binary64 and the other variables are declared as binary32, should the computations be performed using binary32 addition or binary64 addition?

- If a and d are declared binary32, and b and c are declared binary64, what will be the precision of the operations? Note that this question makes sense only after a parenthesizing has been chosen.

These questions are not purely academic. For most applications, it makes perfect sense to use binary32 as a storage format, as it requires half the space of binary64 and holds more precision than most instruments can measure. And it makes sense to use binary64 to carry out computations that may involve millions of iterations.

In the new IEEE 754-2008 standard, there has been some effort to address this problem; see Section 3.4.6, page 93. Note that similar issues arise when one considers the active rounding mode.

Antagonistic answers

The languages C and FORTRAN, probably the two languages that are most used in numerical computing, offer perfectly antagonistic answers to the previous questions.

- In C, an expression of successive add or multiply operators is interpreted with left-associativity,[1] i.e., a+b+c+d is syntactically equivalent to ((a+b)+c)+d. Concerning the precision, each operation may well be performed in an internal precision wider than the precision of the type (we already discussed that problem in Section 3.3.1, page 75); the expression may also be contracted (see Section 7.2.3).

- In contrast, FORTRAN fixes the precision but does not guarantee the parenthesizing, so the expression a+b+c+d may validly be evaluated as (a+b)+(c+d) or (a+d)+(b+c).

Each language has a rationale for these choices, and we will explore it in the following sections.

---

[1]This requirement was introduced in C89 and kept in the current C99 standard. The original Kernighan and Ritchie C [217] allowed regrouping, even with explicit parentheses in expressions.

7.1.2   Processors, compilers, and operating systems

The compiler is in charge of translating the program into a succession of elementary processor instructions. Modern compilers spend most of the compilation time in optimization. We have seen some optimizations related to floating-point evaluation order and precision, but there also exist optimizations that are more directly related to the processor's available hardware.

As an example, consider a processor which offers hardware implementations of the fused multiply-add (FMA, see Section 2.8, page 51). To comply with the IEEE 754-1985 standard, a compiler should not generate FMA instructions for such processors. Additions should be implemented as $(a \times 1 + b)$ and multiplications as $(a \times b + 0)$. Of course, the default behavior of most compilers will be to try to fuse additions and multiplications (that is, to use FMA instructions), which usually improves both speed and accuracy. If one wants portability between, for instance, a platform without FMA and one with FMA, one has to find special directives (such as C's FP_CONTRACT pragma) or compiler options (such as GCC's -mno-fused-madd with processors that support this option) that prevent fusing $\times$ and $+$.

There is a similar tradeoff between portability and improved accuracy on processors which offer hardware binary80 arithmetic (see Section 3.5.3, page 104). Here the extended accuracy, although providing more accurate results in most cases, incurs additional risks, such as subtle bugs due to double rounding (see Section 3.3.1, page 75).

Again, much effort has been spent to address such issues in IEEE 754-2008. There was a clear consensus on the fact that programmers who want portability should be able to get it, while programmers who want performance also should be able to get it. The consensus was not so clear on what should be the default.

Finally, the *operating system* (kernel and libraries) is in charge of initializing the state of the processor prior to the program execution. For example, it will set the dynamic rounding precision on x87 hardware. Considering the previous tradeoff between accuracy and portability, different systems make different choices. For instance, the same conforming C program, compiled by the same compiler with the same options, may yield different results on the same hardware under OpenBSD and Linux. By default, OpenBSD chooses to enhance portability and configures the x86 traditional floating-point unit (FPU) to round to double precision. Linux, in contrast, favors better accuracy and configures the FPU to round to double-extended precision by default.

To summarize, the behavior of each of these actors may influence the others. A program may change the processor state because of an operating system call, for instance, to request rounding toward zero. An ill effect will often be that the mathematical library (a.k.a. libm, also a part of the operating system) no longer functions properly because it expects the processor to round to nearest.

7.1.3   In the hands of the programmer

So standard compliance enhances portability, but usually degrades performance, and sometimes even accuracy. For this reason, the default behavior of a computing system will usually be a compromise between performance, accuracy, and portability. A notable exception is Java, which was designed for portability from the ground up. Section 7.5 will show the difficulty of fulfilling this ambition.

However, recent versions of Java offer means to relax portability for performance under programmer control, while the C99 standard added pragmas to improve portability in C. This illustrates the consensus that a programmer should be given the ability to choose the behavior of the floating-point environment.

The important message in this chapter is that the floating-point behavior of a given program in a given computer is not arbitrary. It is usually well documented, although unfortunately in various places (language standards, compiler manuals, operating system specifications, web pages, and so on). It is thus possible for the programmer to control to the last bit the behavior of every last floating-point operation of his/her programs.

Let us now consider some mainstream programming environments in more detail.

7.2   Floating Point in the C Language

The C language was designed to replace the assembly language for rewriting an operating system (UNIX). This explains why C is very close to the hardware. Since it was not designed as a language for numerical computations, important issues such as reproducibility of the results in floating-point arithmetic were not given much attention. Floating-point code could behave very differently from one platform to another, or even from one compiler another on the same system. As time went by, C compilers were retrofitted with features that were common in languages like FORTRAN, and it was only in the C99 standard that support for IEEE 754-1985 (mostly overlooked in the 1989/1990 C standard revision) was paid some attention.

The remainder of this section describes floating-point features of the C99 standard. Programmers should be aware that C99 compliance may not be the default for all compilers. However, there is almost always a compiler option to enable it. On most POSIX systems, the `c99` utility can be used for this purpose.

7.2.1   Standard C99 headers and IEEE 754-1985 support

Three headers, `<float.h>`, `<math.h>`, and `<fenv.h>`, define the macros and functions that are necessary for dealing with floating-point numbers in C.

- The `<float.h>` header gathers the description of the characteristics of the floating-point environment. Indeed, the C standard requires very little.

  First, the radix for the standard floating-point types (`float`, `double`, and `long double`) is implementation defined, in other words not defined by the C standard. It can be obtained with the `FLT_RADIX` macro. In most cases, it is equal to 2. But particular platforms may choose another radix. For instance, radix 10 has been chosen by the TIGCC project for Texas Instruments calculators[2] (see Section 2.5, page 29 for a discussion on the choice of the radix). Interest in decimal floating-point arithmetic has increased even for desktop computers. An extension to C, bringing new decimal types, is currently being standardized [194].

  Other macros (such as `DBL_MAX`, `DBL_EPSILON`...) provide information on the range and precision of each standard floating-point type and on how rounding is performed.

- In `<math.h>`, one finds, apart from the expected mathematical functions (sin, cos...), most of the functions and predicates (isnan, isunordered...), that were recommended by the Appendix to the IEEE 754-1985/IEC 60559 standard and that can be found nowadays in Section 5.7.2 of the new IEEE 754-2008 standard. One also finds additional types and macros.

- In `<fenv.h>`, one finds the necessary tools for manipulating the floating-point environment (e.g., changing the rounding mode, accessing the status flags...).

Most of the material related to the support of IEEE 754 can be found in Annex F of the C99 standard [190]. Throughout that document, IEEE 754-1985 is referred to as IEC 60559. Recent compilers (IBM's XL C/C++ 9.0 Linux PowerPC64, Sun's C 5.9 Linux i386, Intel's icc 10.1 Linux IA-64, for instance) and C libraries (since 1997, the GNU C library, a.k.a. glibc, for instance) define the `__STDC_IEC_559__` macro, which guarantees their conformance to Annex F, even though, in practice, the conformance is known to be incomplete. For instance, the glibc defines `__STDC_IEC_559__` unconditionally, even when extended intermediate precision is used for the `double` type (see below).

7.2.2 Types

When `__STDC_IEC_559__` is defined, two of the basic binary formats (single-precision/binary32 and double-precision/binary64) are directly supported by the `float` and `double` types, respectively.

---
[2]`http://tigcc.ticalc.org/doc/float.html#FLT_RADIX`

What `long double` means is another story. In C99, whether `__STDC_IEC_559__` is defined or not, the `long double` type can be almost anything, provided that a `long double` has at least the precision and the range of a `double`. Note that these requirements are much weaker than those imposed on IEEE-754 extended formats (see Table 3.2, page 57, for the IEEE 754-1985 requirements). The format of the `long double` type and the associated arithmetic depend on both the processor and the operating system. Sometimes it can also be changed by compiler options.

A program can obtain information on the arithmetic behind `long double` through the macros `LDBL_MANT_DIG`, `LDBL_MIN_EXP`, and `LDBL_MAX_EXP` (defined in `<float.h>`), which respectively provide the number of digits of the significand of normal numbers, and the minimum and maximum possible exponents of normal numbers (the `FLT_RADIX` macro, already mentioned, gives the radix). Beware: The extremal exponents given by `LDBL_MIN_EXP` and `LDBL_MAX_EXP` do not correspond to our definition of the exponent (given in Section 2.1, page 13). There, we assumed significands of normal numbers in radix β to be between 1 and β, whereas these macros assume significands between $1/\beta$ and 1. Hence, having `LDBL_MAX_EXP` $= 1024$ corresponds, with our notation, to having $e_{\max} = 1023$.

For illustration, here are four arithmetics that have been found among various C implementations. The numbers in parentheses correspond to `LDBL_MANT_DIG`, `LDBL_MIN_EXP`, and `LDBL_MAX_EXP`, respectively.

Double precision (53 / −1021 / 1024): This arithmetic (the same as the one of the `double` type) has been found on ARM processors under Linux, and this is the choice originally made for the *ARM Developer Suite* [13, Section 3.3.2]. This choice could be surprising, as the floating-point accelerator (FPA) architecture (ARM's first floating-point implementation) supports extended precision [14, Section 2.9.2]. But ARM processors originally did not have floating-point hardware, many ARM processors still do not (thus floating-point arithmetic must be emulated in software), and the new Vector floating-point (VFP) architecture does not support extended precision [14, Section 2.9.1].

80-bit extended precision (64 / −16381 / 16384): This arithmetic has been found on `x86`, `x86_64`, and IA-64 architectures, because of hardware support.

Double-double arithmetic (106 / −968 / 1024): This arithmetic has been found on PowerPC, under both Darwin (Mac OS X) and Linux (with recent GCC/glibc), and comes from IBM's AIX operating system. A number is representable in that arithmetic if it can be written as the sum of two double-precision/binary64 floating-point numbers, that *very roughly* emulates a 106-bit precision. This is not a genuine floating-point arithmetic, but can be regarded as an extension of a floating-point

arithmetic whose precision and exponent-range parameters are the numbers given in parentheses (see Section 14.1, page 494 for information on "double-double"—more generally, "double-word"—numbers). This conforms to the C standard, which allows, in addition to normalized floating-point numbers, other kinds of floating-point data (such as the subnormal numbers, infinities, and NaNs of the IEEE 754 standard). The range of numbers that can be represented is roughly the same as the range of double-precision numbers. Therefore, this is a valid `long double` type. However, some properties requiring strict floating-point arithmetic (such as Sterbenz's lemma: Lemma 2 in Chapter 4, page 122) will not always be true for the `long double` type, and corresponding floating-point algorithms may no longer work.

Quadruple precision (113 / −16381 / 16384): This arithmetic has been found under HP-UX (HPPA and IA-64) and Solaris (SPARC only [396]), and is implemented in software.[3]

The arithmetic does not completely define the type. Indeed the encoding may depend on the platform (e.g., due to a different endianness). Even the type size (as per the `sizeof` operator) can vary, for alignment reasons (to provide better memory access speed). For instance, with GCC, the default width of a `long double` is 12 bytes long for the x86 (32-bit) application binary interface (ABI) and rises to 16 bytes long for the `x86_64` (64-bit) ABI. However, compiler options can alter this behavior. For instance, the `-m96bit-long-double` and `-m128bit-long-double` switches control the storage size of `long double` values. But note that this has no effect on the floating-point results: range and precision remain the same.

Infinities, NaNs, and signed zeros

Support for (signed or unsigned) infinities and signed zeros is optional. The `INFINITY` macro from `<math.h>` represents a positive or unsigned infinity, when available. However, a `HUGE_VAL` (for the `double` type) macro is always available, which typically is an infinity when supported (this is required by Annex F of the C99 standard).

Support of Not a Number (NaN, also optional) is limited to the quiet flavor (qNaN). Signaling NaNs (sNanN) were not included in the C99 standard since its authors felt it entailed a lot of trouble for a limited usefulness, and that qNaNs were sufficient for the closure of the floating-point arithmetic algebraic structure.

---

[3]The SPARC architecture has instructions for quadruple precision, but in practice, current processors generate traps to compute the results in software.

7.2.3 Expression evaluation

Except for assignment and cast, the C99 standard states [190, Section 5.2.4.2.2]:

> the values of operations with floating operands and values subject to the usual arithmetic conversions and of floating constants are evaluated to a format whose range and precision may be greater than required by the type.

While this will be a bonus in many situations, it may also break the assumptions founding some algorithms presented in this book. It may also lead to the "double rounding" problem depicted in Section 3.3.1, page 75, which can occur even with a single operation, such as `a = b + c`. Splitting an expression by using temporary variables and only one operation per statement will force any intermediate result to be in the required precision; this workaround does not avoid the "double rounding" problem, but one gets a faithful rounding (see Section 2.2, page 20), which may be sufficient for some algorithms.

The C99 standard provides a macro, `FLT_EVAL_METHOD`, whose value gives a clue about what is actually going on. Table 7.1 shows which intermediate type (both range and precision) is used for the evaluation of an operation of a given type.

FLT_EVAL_METHOD	float	double	long double
0	float	double	long double
1	double	double	long double
2	long double	long double	long double

Table 7.1: FLT_EVAL_METHOD macro values.

`FLT_EVAL_METHOD = 2` is typically used with x87 arithmetic (when the processor is configured to round in extended precision). Processors with static rounding format (range and precision) will generally use `FLT_EVAL_METHOD = 0`. In addition to these values, `FLT_EVAL_METHOD = −1` can be used when the evaluation method is not determined (e.g., because of some optimizations that can change the results).

Unfortunately, `FLT_EVAL_METHOD` is an information macro only. There is no standard way of changing the behavior it indicates. This does not mean that it cannot be changed, but it may require compiler switches (Section 3.3.1, page 75, contained some examples) or nonstandard features (e.g., operating system calls).

Operators and functions

The `+`, `-`, `*`, `/` operators and the `sqrt()` and `remainder()` functions provide the basic operations as expected in IEEE 754-1985. More functions and macros

cover conversions, comparison, and floating-point environment manipulation as well as the set of functions recommended in Annex A of 754-1985. Among others, we can highlight the `fma()` function (new in C99), which will allow one to "play around" with the FMA instruction even if it is not directly supported by the processor (none of the x86 chips does, up to year 2008) and test the nifty algorithms presented in the FMA chapter (see Chapter 5). However, beware: as we write this book, some software implementations of `fma()` do not comply with the requirements of the C99 and IEEE 754-2008 standards. For instance, the GNU C Library (glibc), at least until the current version 2.9, implements `fma()` as a multiplication followed by an addition on x86 processors; this means that two roundings are done instead of only one, and in particular, spurious overflow exceptions can occur (if the multiplication overflows but the mathematical result is in the range of the floating-point format).

Contracted expressions

As stated by the C99 standard, "a floating expression may be contracted, that is, evaluated as though it were an atomic operation, thereby omitting rounding errors implied by the source code and the expression evaluation method." This was meant to allow the use of mixed-format operations (with a single rounding, when supported by the processor) and hardware compound operators such as FMA. For instance, the default behavior of the main compilers (GCC, IBM's XL C/C++ 9.0 Linux PowerPC64 compiler, Intel's compiler icc, Microsoft Visual C/C++ compiler) is to contract `x * y + z` to `fma(x,y,z)` when a hardware FMA is available.

Most of the time, this is beneficial in terms of both performance and accuracy. However, it will break algorithms that rely on evaluation as prescribed by the code. For instance, `sqrt(a * a - b * b)` may be contracted using an FMA as if `sqrt(fma(a, a, - b * b))` were used, and if a and b are equal, the result can be nonzero because of the broken symmetry (see example below).

The `FP_CONTRACT` pragma (from `<math.h>`) gives some control on this issue to the programmer. When set to *on*, it allows contracting expressions. When set to *off*, it prevents it. As contracting expressions is potentially dangerous, a C implementation (compiler and associated libraries) must document the default state and the way in which expressions are contracted. Compilers may ignore this pragma, but in this case, they should behave as if it were *off* (disabled contraction) in order to preserve the semantics of the program. Again, your mileage may vary: At the time of writing this book, `gcc` does the opposite.

Following the same example, Listing 7.1 computes the result of the expression `a >= b ? sqrt (a * a - b * b) : 0` in the particular case of a = b. By default, contraction is disabled, but compiling this program with `-DFP_CONTRACT` sets the pragma to *on*, thus enabling contraction. For instance,

icc 10.1 on IA-64 gives on the inputs 1, 1.1, and 1.2:

```
Test of a >= b ? sqrt (a * a - b * b) : 0 with FP_CONTRACT OFF
test(1) = 0
test(1.1000000000000000888) = 0
test(1.1999999999999999556) = 0


Test of a >= b ? sqrt (a * a - b * b) : 0 with FP_CONTRACT ON
test(1) = 0
test(1.1000000000000000888) = 2.9802322387695326562e-09
test(1.1999999999999999556) = nan
```

C listing 7.1 Testing the effect of the contraction of a floating expression to FMA.

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#ifdef FP_CONTRACT
#undef FP_CONTRACT
#define FP_CONTRACT "ON"
#pragma STDC FP_CONTRACT ON
#else
#define FP_CONTRACT "OFF"
#pragma STDC FP_CONTRACT OFF
#endif

static double fct (double a, double b)
{
  return a >= b ? sqrt (a * a - b * b) : 0;
}

/* "volatile" and "+ 0.0" may be needed to avoid optimizations. */
static void test (volatile double x)
{
  printf ("test(%.20g) = %.20g\n", x, fct (x, x + 0.0));
}

int main (int argc, char **argv)
{
  int i;

  printf ("Test of a >= b ? sqrt (a * a - b * b) : 0 with FP_CONTRACT "
          FP_CONTRACT "\n");
  for (i = 1; i < argc; i++)
    test (atof (argv[i]));
  return 0;
}
```

Constant expressions, initialization, and exceptions

There are some issues specific to C regarding the relationship between translation (compilation) time and execution time, on the one hand, and exceptions, on the other hand. As stated by the C99 standard:

> the `FENV_ACCESS` pragma provides a means to inform the implementation when a program might access the floating-point environment to test floating-point status flags or run under non-default floating-point control modes.

When the state of this pragma is *off*, the compiler is allowed to do some optimizations that can have side effects, such as evaluating constant expressions. Otherwise, if the state is *on*, constant expressions should be evaluated at execution time (unless the compiler can deduce that a translation-time evaluation will not change the result, including exceptions). However, this does not affect initialization of objects with *static storage duration*, necessarily using constant expressions, which are evaluated at translation time and do not raise exceptions.

Special values of mathematical functions

The C99 standard also specifies (in its optional Annex F) the values of the elementary functions for particular arguments. Some of these values are different from those recommended by the new IEEE 754-2008 standard. Moreover, these definitions can cause some head scratching. Consider for instance, the following requirement for the power function: $\mathrm{pow}(-1, \pm\infty) = 1$. This may be a bit puzzling until one understands that any sufficiently large binary floating-point number is an even integer: hence, "by taking the limit," the prescribed value of 1. Other rules do not even have such a justification: $\mathrm{pow}(+1,x)$ and $\mathrm{pow}(x,\pm0)$ must return 1 for any x, even a NaN, which breaks with the usual NaN propagation rule.

   Concerning these special cases for pow, IEEE 754-2008 chose to remain compatible with C99, but added three more functions, powr (whose special cases are derived by considering that it is defined by $e^{y\log x}$), and pown/rootn (which deal with integral exponents only).

7.2.4   Code transformations

Many common transformations of a code into some *naively* equivalent code become impossible if one takes into account special values such as NaNs, signed zeros, and rounding modes. For instance, the expression `x + 0` is not equivalent to the expression `x` if `x` is `-0` and the rounding is to nearest.

   Similarly, some transformations of code involving relational operators become impossible due to the possibility of unordered values (see page 64). This is illustrated in Listing 7.2.

C listing 7.2 Strangely behaving relational operators (excerpt from Annex F of the C99 standard). These two pieces of code may seem equivalent, but behave differently if a and b are unordered.

```
// calls g and raises "invalid" if a and b are unordered
if (a < b)
  f();
else
  g();
// calls f and raises "invalid" if a and b are unordered
if (a >= b)
  g();
else
  f();
```

As sNaNs are not specified in C99, C implementations that support them must do so with special care. For instance, the transformation of 1 * x to x is valid in C99, but if x can be a sNaN, this transformation becomes invalid.

7.2.5 Enabling unsafe optimizations

Complex arithmetic in C99

The C99 standard defines another pragma allowing a more efficient implementation of some operations for multiplication, division, and absolute value of complex numbers, for which the usual, straightforward formulas can give incorrect results on infinities or spurious exceptions (overflow or underflow) on huge or tiny inputs (see Section 4.5, page 139). The programmer can set the CX_LIMITED_RANGE pragma to *on* if he or she knows that the straightforward mathematical formulas are acceptable, in which case the compiler can choose to use them instead of a code that would work on (almost) any input but which is slower. The default state of this pragma is *off*, for safe computation.

Range reduction for trigonometric functions

For ARM processors using the ARM Developer Suite or the RealView tools, the default trigonometric range reduction is inaccurate for very large arguments. This is valid for most programs: if a floating-point number is so large that the value of its ulp is several times the period 2π, it usually makes little sense to compute its sine accurately. Conversely, if the input to the sine is bound by construction to reasonably small values, the default range reduction is perfectly accurate. The situation is comparable to using the previous quick and unsafe complex operators: they are perfectly safe if the values that may appear in the program are under control. The big difference, however, is that here the default behavior is the unsafe one.

The accuracy of range reduction can be improved by the following pragma [13, Section 5.4]:

```
#pragma import (__use_accurate_range_reduction)
```

The more accurate range reduction is slower and requires more memory (this will be explained in Section 11.1, page 379). The ARM mainly focuses on embedded applications such as mobile devices, which are memory-limited.

Compiler-specific optimizations

Compilers can have their own directives to provide unsafe optimizations which may be acceptable for most common codes, e.g., assuming that no exceptions or special values occur, that mathematically associative operations can be regarded as associative in floating-point arithmetic, and so on. This is the case of GCC's generic `-ffast-math` option (and other individual options enabled by this one). Users should use such options with much care. In particular, using them on a code they have not written themselves is highly discouraged.

7.2.6   Summary: a few horror stories

As pointed out by David Goldberg [148] (in his edited reprint), all these uncertainties make it impossible, in many cases, to figure out the exact semantics of a floating-point C program just by reading its code. As a consequence, portability is limited, and moving a program from one platform to another may involve some rewriting. This also makes the automatic verification of floating-point computations very challenging, as noticed by Monniaux [280].

The following section describes a few traps that await the innocent programmer.

Printing out a variable changes its value

Even the crudest (and most common) debugging mode of all, printing out data, can be a trap. Consider the program given in Listing 7.3. With GCC version 4.1.2 20061115 on a 32-bit Linux platform and the default settings, the program will display:

```
9007199254740991.5 is strictly less than 9007199254740992
9007199254740992 is strictly less than 9007199254740992
```

While there is nothing wrong with the first line, the second is a bit more disturbing. For the former we have used the `long double` type, which, on this platform, maps to 80-bit x87 registers. These have enough room to store all the bits of the sum. To no one's surprise, the first test evaluates to `true`.

C listing 7.3 Strange behavior caused by spilling data to memory.

```
long double lda = 9007199254740991.0; // 2^53 - 1
double da = 9007199254740991.0;        // dito
if (lda + 0.5 < 9007199254740992.0)
{
  printf("%.70Lg is strictly less than %.70Lg\n",
         lda + 0.5,
         (long double) 9007199254740992.0);
}
if (da + 0.5 < 9007199254740992.0)
{
  printf("%.70g is strictly less than %.70g\n",
         da + 0.5,
         9007199254740992.0);
}
```

There is also enough room to store all the bits when, to call the `printf()` function, the register holding the sum is spilled out to memory (remember a `long double` is 12 bytes long on this platform). The printed message reflects what happens in the registers.

For the second line, while we are supposed to work with 64 bits, the addition and the test for inequality are also executed in the 80-bit x87 registers. The test evaluates again to `true` since, at register level, we are in the exact same situation. What changes is that, when the sum is spilled to memory, it is rounded to its "actual" 64-bit size. Using the rounding-to-nearest mode and applying the "even rule" to break ties leads us to the 9007199254740992 value, which is eventually printed out. By the way, it has nothing to do (as a suspicious reader might wonder) with the formats used in the `printf()` function. One may be convinced by trying the following:

- on the same platform, add to the command line the flags that require the use of 64-bit SSE registers instead of the 80-bit x87 ones (`-march=pentium4` and `-mfpmath=sse`);

- on the 64-bit corresponding platform, run GCC with the default settings (which are to use the SSE registers and not the x87).

The second line never prints out since the rounding of the sum takes place, this time, in the 64-bit registers *before* the comparison is executed.

A possible infinite loop in a sort function

It is difficult to implement a sorting algorithm without the basic hypothesis that if, at some point, two elements have compared as $a < b$, then in the future they will also compare as $b > a$. If this assumption is violated, the programmer of the sorting algorithm cannot be held responsible if the program goes into an infinite loop.

Now let us write a function that compares two `my_type` structures according to their radius.

C listing 7.4 A radius comparison function.

```
int compare_radius (const my_type *a, const my_type *b)
{
  double temp = a->x*a->x + a->y*a->y - b->x*b->x - b->y*b->y;
  if (temp > 0)
    return 1;
  else if (temp < 0)
    return -1;
  else
    return 0;
}
```

We see at least two ways things can go wrong in Listing 7.4.

- If `temp` is computed using an FMA, there are two different ways of computing each side of the subtraction, as we have seen in Section 7.2.3.

- Some of the intermediate results in the computation of `temp` may be computed to a wider precision, especially when using an x87 FPU.

In both cases, the net result is that the `compare_radius` function, although written in a symmetrical way, may be compiled into asymmetrical code: it may happen that `compare_radius(a,b)` returns 0 while `compare_radius(b,a)` returns 1, for instance. This is more than enough to break a sorting algorithm.

Getting to the root of such bugs is very difficult for several reasons. First, it will happen extremely rarely. Second, as we have just seen, entering debug mode or adding `printf`s is likely to make the bug vanish. Third, it takes a deep understanding of floating-point issues to catch (and fix) such a bug. We hope that our reader now masters this knowledge.

7.3 Floating-Point Arithmetic in the C++ Language

7.3.1 Semantics

The semantics of the C++ language is similar to that of the C language with respect to floating-point arithmetic. The parenthesizing order is the same and, as in C, intermediate expressions may use a bigger format since as per the C99 standard [190], section 6.3.1.8:

> The values of the floating operands and the results of floating expressions may be represented in greater precision and range than that required by the type; the types are not changed thereby.

While the C++ standard does not mention the C99 language,[4], there is no fundamental reason for floating-point arithmetic to behave differently between C99 and C++. Most of Section 7.2 should therefore apply to C++ as well.

7.3.2 Numeric limits

In addition to the macros inherited from C, the C++ standard library provides the template class `std::numeric_limits` in the `<limits>` header file to allow metaprogramming depending on the floating-point capabilities. For a floating-point type `T`, the class `std::numeric_limits<T>` provides the following static members.

1. Format:

   - `int radix`: the radix β, either 2 or 10 usually,
   - `bool is_iec559`: *true* if `T` is an IEEE-754 format and the operations on `T` are compliant.[5]

2. Special values:

   - `bool has_infinity`: *true* if `T` has a representation for $+\infty$,
   - `bool has_quiet_NaN`: *true* if `T` has a representation for a qNaN,
   - `bool has_signaling_NaN`: *true* if `T` has a representation for an sNaN,
   - `T infinity()`: representation of $+\infty$,
   - `T quiet_NaN()`: representation of a qNaN,
   - `T signaling_NaN()`: representation of a sNaN.

3. Range:

   - `T min()`: smallest positive normal number,
   - `T max()`: largest finite number,
   - `T lowest()`: negated `max()`,
   - `int min_exponent`: smallest integer k such that β^{k-1} is a normal number, e.g., -125 for binary32,
   - `int max_exponent`: largest integer k such that β^{k-1} is a normal number, e.g., 128 for binary32,

---

[4]The first version C++98, was published before the C99 standard. There was a minor revision in 2003, but which was only a corrected version of the 1998 standard. The next C++ standard, tentatively named C++0x, will take C99 into account.

[5]When `is_iec559` is *true*, the C++ standard mandates that infinities and NaNs are available.

- `int min_exponent10`: smallest integer k such that 10^k is a normal number, e.g., -37 for binary32,

- `int max_exponent10`: largest integer k such that 10^k is a normal number, e.g., 38 for binary32.

4. Subnormal numbers:

- `float_denorm_style has_denorm`: `denorm_present`, `denorm_-absent`, or `denorm_indeterminate`, depending on whether subnormal numbers are known to be supported or not,

- `bool has_denorm_loss`: *true* if inaccurate subnormal results are detected with an "underflow" exception (Section 3.1.5) instead of just an "inexact" exception,

- `T denorm_min()`: smallest positive subnormal number,

- `bool tinyness_before`: *true* if subnormal results are detected before rounding (see Definition 1 of Chapter 2, page 18).

5. Rounding mode and error:

- `T epsilon()`: the subtraction $1^+ - 1$ with 1^+ the successor of 1 in the format `T`,

- `T round_error()`: biggest relative error for normalized results of the four basic arithmetic operations, with respect to `epsilon()`, hence 0.5 when rounding to nearest,

- `float_round_style round_style`: `round_toward_zero`, `round_-to_nearest`, `round_toward_infinity`, `round_toward_neg_-infinity`, or `round_indeterminate`, depending on whether the rounding mode is known or not. [6]

7.3.3   Overloaded functions

In C++, functions from the `<cmath>` header have been overloaded to take argument types into account (`float` and `long double`). For instance, while `<math.h>` provides a `sinf` function for computing sine on `float`, `<cmath>` provides `float sin(float)` in the `std` namespace. In particular, it means that the following piece of code does not have the same semantics in C and in C++ (assuming the `std` namespace is in scope):

```
float a = 1.0f;
double b = sin(a);
```

---

[6] `round_style` is a constant member. Therefore, it may well be set to the default `round_to_nearest` style, even if the architecture allows us to change the rounding direction on the fly.

In C, the variable a will first be promoted to double. The double-precision sine will then be called and the double-precision result will be stored in b. In C++, the single-precision sine will be called and its single-precision result will then be promoted to double and stored in b. Of course, the first approach provides a more accurate result.

The C++0x standard (to be ratified in 2009) also provides utility functions that replace the C99 macros for classifying or ordering floating-point values:

```cpp
namespace std {
  template <class T> bool signbit(T x);
  template <class T> int fpclassify(T x);
  template <class T> bool isfinite(T x);
  template <class T> bool isinf(T x);
  template <class T> bool isnan(T x);
  template <class T> bool isnormal(T x);

  template <class T> bool isgreater(T x, T y);
  template <class T> bool isgreaterequal(T x, T y);
  template <class T> bool isless(T x, T y);
  template <class T> bool islessequal(T x, T y);
  template <class T> bool islessgreater(T x, T y);
  template <class T> bool isunordered(T x, T y);
}
```

7.4   FORTRAN Floating Point in a Nutshell

7.4.1   Philosophy

FORTRAN was initially designed as a FORmula TRANslator, and this explains most of its philosophy with respect to floating-point arithmetic. In principle, a FORTRAN floating-point program describes the implementation of a mathematical formula, written by a mathematician, an engineer, or a physicist, and involving real numbers instead of floating-point numbers. This is illustrated by the fact that a floating-point variable is declared with the REAL keyword or one of its variants. Compare this with the C float keyword which describes a machine implementation of the reals, following the C "close-to-the-metal" philosophy. FORTRAN also draws a clear line between integers and reals, and acknowledges them as fundamentally different mathematical objects.

In the compilation of a FORTRAN program, the formula provided by the user should be respected. In the translation process, a FORTRAN compiler is free to apply to the source code (formula) any mathematical identity that is valid over the reals, as long as it results in a mathematically equivalent formula. However, it gives little importance to the rounding errors involved in this process. They are probably considered unavoidable, and small anyway.

Here is a biased excerpt of the FORTRAN standard [192] that illustrates this.

> (...) the processor may evaluate any mathematically equivalent expression (...). Two expressions of a numeric type are mathematically equivalent if, for all possible values of their primaries, their mathematical values are equal. However, mathematically equivalent expressions of numeric type may produce different computational results.

Again, integers and reals are distinct objects, as illustrated by the following excerpt:

> Any difference between the values of the expressions `(1./3.)*3.` and `1.` is a computational difference, not a mathematical difference. The difference between the values of the expressions `5/2` and `5./2.` is a mathematical difference, not a computational difference.

Therefore, `(1./3.)*3.` may be quietly replaced by `1.`, but `5/2` and `5./2.` are not interchangeable.

However, the standard acknowledges that a programmer may sometimes want to impose a certain order to the evaluation of a formula. It therefore makes a considerable exception to the above philosophy: when parentheses are given, the compiler should respect them. Indeed, the first sentence of the first excerpt reads in full:

> (...) the processor may evaluate any mathematically equivalent expression, provided that the integrity of parentheses is not violated.

Then, another excerpt elaborates on this:

> In addition to the parentheses required to establish the desired interpretation, parentheses may be included to restrict the alternative forms that may be used by the processor in the actual evaluation of the expression. This is useful for controlling the magnitude and accuracy of intermediate values developed during the evaluation of an expression.

For example, an expression written

$$a/b * c/d$$

may be computed either as

$$(a/b) * (c/d), \tag{7.1}$$

or as

$$(a * c)/(b * d). \tag{7.2}$$

A FORTRAN compiler may choose the parenthesizing it deems the more efficient. These two expressions are mathematically equivalent but do not lead to the same succession of rounding errors, and therefore the results may differ. For instance, if a, b, c, and d are all equal and strictly larger than the square root of the largest representable finite floating-point number Ω, then choosing Equation (7.1) leads to the right result 1, whereas choosing Equation (7.2) leads to the quotient of two infinities (which gives a NaN in an arithmetic compliant with the IEEE 754-2008 standard). During the development of the LHC@Home project [103], such an expression, appearing identically in two points of a program, was compiled differently and—very rarely—gave slightly different results. This led to inconsistencies in the distributed simulation, and was solved by adding explicit parentheses on the offending expression.

Here are some more illustrations of the FORTRAN philosophy, also from the FORTRAN standard [192]. In Tables 7.2 and 7.3 , X, Y, Z are of arbitrary numeric types, A, B, C are reals or complex, and I, J are of integer type.

Expression	*Allowable* alternative
X+Y	Y+X
X*Y	Y*X
-X + Y	Y-X
X+Y+Z	X + (Y + Z)
X-Y+Z	X - (Y - Z)
X*A/Z	X * (A / Z)
X*Y - X*Z	X * (Y - Z)
A/B/C	A / (B * C)
A / 5.0	0.2 * A

Table 7.2: FORTRAN allowable alternatives.

The example of the last line of Table 7.2 could be turned into a similar example by replacing the "5.0" by "4.0" and the "0.2" by "0.25". However, it is not possible to design a similar example by replacing the "5.0" by "3.0" because no finite sequence "0.3333 ⋯ 3" is exactly equal to 1/3: FORTRAN accepts replacements of formulas only by other formulas that are mathematically equivalent.

To summarize, FORTRAN has much more freedom when compiling floating-point expressions than C. As a consequence, the performance of a FORTRAN program is likely to be higher than that of the same program written in C.

Expression	*Forbidden* alternative	Why
I/2	0.5 * I	integer versus real operation
X*I/J	X * (I / J)	real versus integer division
I/J/A	I / (J * A)	integer versus real division
(X + Y) + Z	X + (Y + Z)	explicit parentheses
(X * Y) - (X * Z)	X * (Y - Z)	explicit parentheses
X * (Y - Z)	X*Y-X*Z	explicit parentheses

Table 7.3: FORTRAN forbidden alternatives.

7.4.2   IEEE 754 support in FORTRAN

Section 13 of the FORTRAN standard, *Intrinsic procedures and modules*, defines a machine model of the real numbers which corresponds to normalized floating-point numbers:

The model set for real x is defined by

$$x = \begin{cases} 0 & \text{or} \\ s \times b^e \times \sum_{k=1}^{p} f_k \times b^{-k} \end{cases},$$

where b and p are integers exceeding one; each f_k is a non-negative integer less than b, with f_1 non zero; s is $+1$ or -1; and e is an integer that lies between some integer maximum $e_{\max}$ and some integer minimum $e_{\min}$ inclusively. For $x = 0$, its exponent e and digits f_k are defined to be zero. (...) the integer parameters b, p, $e_{\min}$, and $e_{\max}$ determine the set of model floating-point numbers. The parameters of the integer and real models are available for each integer and real type implemented by the processor. The parameters characterize the set of available numbers in the definition of the model. The floating-point manipulation functions (13.5.10) and numeric inquiry functions (13.5.6) provide values of some parameters and other values related to the models.

Numeric inquiry functions are `DIGITS (X)`, `EPSILON (X)`, `HUGE (X)`, `MAXEXPONENT (X)`, `MINEXPONENT (X)`, `PRECISION (X)`, `RADIX (X)`, `RANGE (X)`, `TINY (X)`. Most need no further explanation, but be aware that the significand in the previous model is in $[1/b, 1)$; therefore, $e_{\min}$ and $e_{\max}$ differ from those of the IEEE 754 standard. Some of these functions have strange definitions in 13.5.10: `EPSILON (X)` is defined as *Number that is almost negligible compared to one*, which is not very precise. It becomes clearer in 13.7, *Specifications of the standard intrinsic procedures*, which defines it as b^{1-p} with the

notation of the model above. `HUGE (X)` and `TINY (X)` are defined, respectively, as *Largest number of the model* and *Smallest positive number of the model*. For these functions, remember that the model includes neither infinities nor subnormals. Indeed, `TINY (X)` is defined later in 13.7 of the FORTRAN standard as $b^{e_{\min}-1}$.

The following floating-point manipulation functions are available:

- `EXPONENT (X)` Exponent part of a model number;

- `FRACTION (X)` Fractional part of a number;

- `NEAREST (X, S)` Nearest different processor number in the direction given by the sign of S;

- `SPACING (X)` Absolute spacing of model numbers near a given number. This under-specified definition of the ulp is clarified in 3.7 as $b^{\max(e-p,e_{\min}-1)}$;

- `RRSPACING (X)` Reciprocal of the relative spacing of model numbers near a given number;

- `SCALE (X, I)` Multiply a real by its radix to an integer power;

- `SET EXPONENT (X, I)` Set exponent part of a number.

All the previous information is mostly unrelated to the IEEE 754 standard. In addition, the FORTRAN standard dedicates its Section 14 to *Exceptions and IEEE arithmetic*. This section standardizes IEEE 754 support when it is provided, but does not make it mandatory. It provides, in the intrinsic modules `IEEE EXCEPTIONS`, `IEEE ARITHMETIC` and `IEEE FEATURES`, numerous inquiry functions testing various parts of standard compliance. It also defines read and write access functions to the rounding directions, as well as read and write access functions to the underflow mode (which may be either gradual, i.e., supporting subnormals, or abrupt, i.e., without subnormals). Finally, it defines subroutines for all the functions present in the IEEE 754-1985 standard.

7.5 Java Floating Point in a Nutshell

7.5.1 Philosophy

"Write once, run anywhere (or everywhere)" is the mantra of the Java language evangelists. Reproducibility between platforms is an explicit and essential goal, and this holds for numeric computations as well. In practical terms, this is achieved through `byte-code` compilation (instead of compilation to object code) and interpretation on a `virtual machine` rather than direct execution on the native operating system/hardware combination.

In the first versions of the Java platform, this meant poor performance, but techniques such as "Just In Time" or "Ahead Of Time" compilation were later developed to bridge the gap with native execution speed.

The initial language design tried to ensure numerical reproducibility by clearly defining execution semantics, while restricting floating-point capabilities to a subset of formats and operators supported by most processors. Unfortunately, this was not enough to ensure perfect reproducibility, but enough to frustrate performance-aware programmers who could not exploit their extended precision or FMA hardware [210]. We will see how Java later evolved to try and give to the programmers the choice between reproducibility and performance.

In general terms, Java claims compliance with the IEEE 754-1985 standard, but only implements a subset of it. Let us now look at the details.

7.5.2   Types and classes

Java is an object-oriented language "with a twist": the existence of primitive types and the corresponding wrapper classes. One of the main reasons behind the existence of basic types is a performance concern. Having many small and simple variables incurs a severe access time penalty if created on the heap as objects are.

As far as floating-point numbers are concerned, there are two basic types:

- `float`: binary32 analogous, `Float` being the wrapper class;

- `double`: binary64 analogous, `Double` being the wrapper class.

As the Java Language Specification puts it, these types are "conceptually associated with the 32-bit single-precision and 64-bit double-precision format of IEEE 754 standard."

In the virtual machine

Although the virtual machine is supposed to insulate the execution from the peculiarities of the hardware, at some point, floating-point operations have to be performed on the actual processor. Of course, specialized floating-point units could be totally avoided and all instructions software emulated using integer registers, but the speed penalty would be prohibitive.

Looking at the details of the Java Virtual Machine Specification, second edition [262, Section 3.3.2], one may observe that the Java designers have been forced to acknowledge the peculiarity of the x87 hardware, which can be set to round to a 53-bit or 24-bit significand, but always with the 16-bit exponent of the double-extended format. The Java Virtual Machine Specification defines, in addition to the *float* and *double* formats, a *double-extended-exponent*

format that exactly corresponds to what is obtained when one sets an x87 FPU to rounding to 53 bits. Unfortunately, from there on, reproducibility vanishes, as the following excerpt illustrates:

> These extended-exponent value sets may, under certain circumstances, be used instead of the standard value sets to represent the values of type float or double.

The first edition of the Java Virtual Machine Specification was much cleaner, with only float and double types, and fewer "may" sentences threatening reproducibility. The only problem was that it could not be implemented efficiently on x86 hardware (or, equivalently, efficient implementations were not strictly compliant with the specification).

Neither the first edition nor the second is fully satisfactory. To be honest, this is not to be blamed on the Java designers, but on the x87 FPU. The fact that it could not round to the standard double-precision format is one of the main flaws of an otherwise brilliant design. It will be interesting to see if Java reverts to the initial specification, now that x86-compatible processors, with SSE2, are turning the x87 page and offering high-performance hardware with straightforward rounding to binary32 and binary64.

In the Java language

In Java 2 SDK 1.2, a new keyword (and the corresponding behavior) was introduced to make sure computations were realized as if binary32 and binary64 were actually used all the way (again, at some performance cost). See Program 7.1. Technically, this `strictfp` modifier is translated into a bit

```
// In this class, all operations in all methods are performed
// in binary32 or binary64 mode.
strictfp class ExampleFpStrictClass {
...
} // End class ExampleFpStrictClass

class NormalClass {
...
  // This particular method performs all its operations in binary32
  // or binary64 mode
  strictfp double aFpStrictMethod(double arg)
  {
    ...
  } // End aFpStrictMethod
} // End class NormalClass
```

Program 7.1: The use of the strictfp keyword.

associated to each method, down to the virtual machine.

The `strictfp` modifier ensures that all the computations are performed in strict binary32 or binary64 mode, which will have a performance cost on x86 hardware without SSE2. According to the specification, what non-`strictfp` allows is just an extended exponent range, "at the whim of the implementation" [153].

However, as usual, one may expect compilation flags to relax compliance to the Java specification. Here, the interested reader should look at the documentation not only of the compiler (the `javac` command or a substitute), but also of the runtime environment (the `java` command or a substitute, including just-in-time compilers). There are also Java native compilers such as JET or GCJ, which compile Java directly to machine code (bypassing the virtual machine layer). Some enable extended precision even for the significand.

7.5.3 Infinities, NaNs, and signed zeros

Java has the notion of signed infinities, NaNs, and signed zeros. Infinities and NaNs are defined as constants in their respective wrapper classes (e.g., `java.lang.Double.POSITIVE_INFINITY`, `java.lang.Float.NaN`).

A first pitfall one must be aware of is that `Double` and `double` do not compare the same, as Program 7.2 shows. Here is the output of this program:

```
NaN != NaN
NaN == NaN
aDouble and anotherDouble are different objects.
anotherDouble and aThirdDouble are the same object.
anotherDouble and aThirdDouble have the same value NaN == NaN
```

As one can see, the == operator does not behave the same for basic types and objects.

- For basic types, if the variables hold the same value, the comparison evaluates to `true`, except for the `java.lang.{Float | Double}.NaN` value, in accordance with any version of the IEEE 754 standard.

- For the object types, the == operator evaluates to `true` only if both references point to the same object. To compare the values, one must use the `equals()` method. But as you can see, `NaN` equals `NaN`, which is a bit confusing.

  We are confronted here with a tradeoff between respect for the IEEE 754 standard and the consistency with other Java language elements as maps or associative arrays. If floating-point objects are used as keys, one should be able to retrieve an element whose index is `NaN`.

```
import java.io.*;

class ObjectValue {
  public static void main(String args[])
  {
    double adouble = java.lang.Double.NaN;
    double anotherdouble = java.lang.Double.NaN;
    Double aDouble = new Double(java.lang.Double.NaN);
    Double anotherDouble = new Double(java.lang.Double.NaN);
    Double aThirdDouble = anotherDouble;

    if (adouble != anotherdouble){
      System.out.print(adouble);
      System.out.print(" != ");
      System.out.println(anotherdouble);
    }
    if (aDouble.equals(anotherDouble)){
      System.out.print(aDouble.toString());
      System.out.print(" == ");
      System.out.println(anotherDouble.toString());
    }
    if (aDouble != anotherDouble)
      System.out.println("aDouble and anotherDouble are different objects.");
    if (anotherDouble == aThirdDouble)
      System.out.println("anotherDouble and aThirdDouble are the same object.");
    if (anotherDouble.equals(aThirdDouble)){
      System.out.print("anotherDouble and aThirdDouble have the same value ");
      System.out.print(anotherDouble.toString());
      System.out.print(" == ");
      System.out.println(aThirdDouble.toString());
    }
  } // End main
} // End class ObjectValue
```

Program 7.2: Object comparison in Java.

7.5.4 Missing features

The compliance of the Java Virtual Machine [262, Section 3.8] to IEEE 754 remains partial in two main respects:

- It does not support flags or exceptions. The term *exception* must be taken here with the meaning it has in the IEEE 754 standard, not with the one it has in Java (which would more accurately translate into *trap*, in IEEE 754 parlance).

- All operations are performed with rounding to the nearest. As a consequence, some of the algorithms described in this book cannot be implemented. Worse, as this is a virtual machine limitation, there is no possibility of a machine interval arithmetic data type as first class

citizen in the Java language. This does not mean that interval arithmetic cannot be done at all in Java; several external packages have been developed for that, but they are much less efficient than operations performed using hardware directed rounding modes. This is all the more surprising because most processors with hardware floating-point support also support directed rounding modes.

7.5.5   Reproducibility

The `strictfp` keyword enables reproducibility of results computed using basic operations. Expression evaluation is strictly defined and unambiguous, with (among others) left-to-right evaluation, StrictFP compile-time constant evaluation, and widening of the operations to the largest format [153].

However, tightening basic operations is not enough. Until Java 2 SDK 1.3, when a mathematical function of the `java.lang.Math` package was called (sine, exponential, etc.), it was evaluated using the operating system's implementation, and the computed result could change from platform to platform. Java 2 SDK 1.3 was released with the new `java.lang.StrictMath` package. It tried to guarantee the same bit-for-bit result on any platform, again, at the expense of performance. Nevertheless, correct rounding to the last bit was not ensured. Eventually, in Java 2 SDK 1.4, the implementation of `java.lang.Math` functions became simple calls to their `java.lang.StrictMath` counterparts.

This enabled numerical consistency on all platforms at last, with two ill effects. Some users observed that the result changed for the same program on the same platform. Users also sometimes noticed a sharp drop in execution speed of their program, and the standard Java platform no longer offers a standard way out for users who need performance over reproducibility. Many tricks (e.g., resorting to JNI for calls to an optimized C library) were tried to gain access again to the speed and precision of the underlying platform.

To summarize this issue, the history of Java shows the difficulty of the "run anywhere with the same results" goal. At the time of writing this book, there are still some inconsistencies; for example, the fact that the default choice for elementary function evaluation is reproducibility over performance, while the default choice for expression evaluation (without `strictfp`) is performance over reproducibility.

Things will evolve favorably, however. We have already mentioned that the generalization of SSE2 extensions will render `strictfp` mostly useless. In addition, in the near future, the generalization of correctly rounded elementary functions which are recommended by IEEE 754-2008 (see Section 11.6, page 394) could reconcile performance and reproducibility for the elementary functions: the Java virtual machine could again trust the system's optimized mathematical library if it knows that it implements correct rounding. It is

unfortunate that the `java.lang.StrictMath` current implementation did not make the choice of correct rounding. It remains for Java designers to specify a way to exploit the performance and accuracy advantage of the FMA operator when available [9] without endangering numerical reproducibility.

7.5.6 The BigDecimal package

The Java designers seem to have been concerned by the need for reliable decimal floating-point, most notably for perfectly specified accounting. They provided the necessary support under the form of the `java.math.BigDecimal` package. Although this package predates the IEEE 754-2008 standard and therefore does not exactly match the decimal floating-point specification, it shares many concepts with it.

In particular, the `java.math.MathContext` class encapsulates a notion of precision and a notion of rounding mode. For instance, the preset `MathContext.DECIMAL128` defines a format matching the IEEE 754-2008 decimal128 and the "round to nearest and break ties to even" default rounding mode. Users can define their own kind of `MathContext` and have, for that purpose, a wide choice of rounding modes.

A `MathContext` can be used to control how operations are performed, but also to emulate IEEE 754 features otherwise absent from the language, such as the inexact flag. Program 7.3 illustrates this.

Current State of Java for HP

```java
import java.math.*;

class  DecimalBig {
  public static void main(String args[])
   {
     // Create a new math context with 7 digits precision, matching
     // that of MathContext.DECIMAL32 but with a different rounding
     // mode.
     MathContext mc = new MathContext(7, RoundingMode.UNNECESSARY);
     BigDecimal a = new BigDecimal(1.0, MathContext.DECIMAL32);
     BigDecimal b = new BigDecimal(3.0, MathContext.DECIMAL32);
     BigDecimal c;

     // Perform the division in the requested MathContext.
     // In this case, if the result is not exact, within the required
     // precision, an exception will be thrown.
     c = a.divide(b, mc);
     // could have been written as
     // c = a.divide(b, 7, RoundingMode.UNNECESSARY)
   } // End main
} // End class DecimalBig
```

Program 7.3: BigDecimal and MathContext.

This program will crash, since we do not catch the exception that would, in IEEE 754 parlance, raise the "inexact status flag," and will print out a message of the following type:

```
 Exception in thread "main" java.lang.ArithmeticException:
Rounding necessary
        at java.math.BigDecimal.divide(BigDecimal.java:1346)
        at java.math.BigDecimal.divide(BigDecimal.java:1413)
        at DecimalBig.main(DecimalBig.java:12)
```

This is more awkward and dangerous than the behavior proposed in IEEE 754-2008: in IEEE 754, an inexact computation silently raises a flag and does not interrupt execution. Still, when used with care, this is the closest to floating-point environment control one can find in "out-of-the-box" Java.

A completely different problem is that `BigDecimal` numbers are objects, not basic types. They incur all the performance overhead associated with objects (in addition to the performance overhead associated with software decimal operations) and require a clumsy object-oriented method syntax instead of the leaner usual infix operators.

7.6 Conclusion

We wish we convinced the reader that, from the floating-point perspective, languages and systems were not "designed equal," and that the designer of numerical programs may save on debugging time by looking carefully at the documentations of both the chosen language and the underlying system.

Obviously, considering the variety of choices made by different systems, there is no perfect solution, in particular because of the performance/reproducibility conflict (where *reproducibility* may be replaced with *portability*, *predictability*, or *numerical consistency*, depending on the programmer's concerns). The perfect solution may be a system which

- is safe by default (favoring portability) so that subtle numerical bugs, such as the infinitely looping sort, are impossible, and

- gives to the programmer the possibility of improving performance when needed, with due disclaimers with respect to numerical predictability.

Even so, the granularity of the programmer's control on this tradeoff is an issue. Compilation flags or operating-system-level behavior control are typically too coarse, while adding pragmas or `strictfp` everywhere in the code may be a lot of work, and may not be possible when external libraries are used.

We have not covered all the existing languages, of course. Some of them are well specified, and some are explicitly under-specified (C#, Perl,

Python, etc.). In the latter case, note that most recent documentations explicitly warn the user about floating-point arithmetic.

Finally, some languages, such as ECMAScript (ECMA-262 / ISO/IEC 16262), do not have integer arithmetic and rely on IEEE 754 floating-point arithmetic to emulate integer arithmetic. The only difficulty is the integer division, which is commonly implemented as a floating-point division followed by a floor, *without any justification*. Developers should be aware that some inputs can yield an incorrect result because of the rounded floating-point division, although in most cases (in particular those encountered in practice), one can prove that the result is correct [254].

Part III

Implementing Floating-Point Operators

Chapter 8

Algorithms for the Five Basic Operations

AMONG THE MANY OPERATIONS that the IEEE 754 standards specify (see Chapter 3), we will focus here and in the next two chapters on the five basic arithmetic operations: addition, subtraction, multiplication, division, and square root. We will also study the fused multiply-add (FMA) operator. We review here some of the known properties and algorithms used to implement each of those operators. Chapter 9 and Chapter 10 will detail some examples of actual implementations in, respectively, hardware and software.

Throughout this chapter, the radix β is assumed to be either 2 or 10. Following the IEEE 754-2008 standard [187], we shall further assume that extremal exponents are related by $e_{\min} = 1 - e_{\max}$ and that the formats considered are basic formats only.

8.1 Overview of Basic Operation Implementation

For the five basic operations, the IEEE 754-2008 standard requires correct rounding: the result returned must be as if the exact, infinitely precise result was computed, then rounded. The details of the cases that may occur, illustrated in Figure 8.1, are as follows.

- If the result is undefined, a Not a Number (NaN) will be returned.

- Otherwise, let us consider the real number which is the infinitely precise result of the operation.

  - If this real result is exactly representable as a floating-point number, no rounding will be needed. However, there may still be work to do: a representable result may have several possible
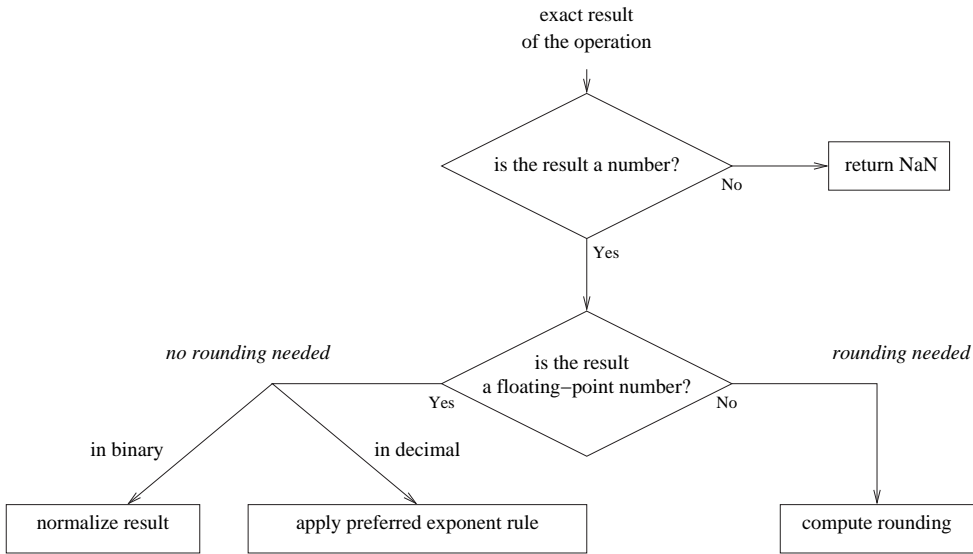
Figure 8.1: Specification of the implementation of a floating-point operation.

representations, and the implementation has to compute which one it returns out of its intermediate representation of the result:

* In binary, there is only one valid representation, which is the one with the smallest possible exponent;

* In decimal, several representations of the result may be valid (they form what is called a *cohort*). The standard precisely defines which member of such a cohort should be returned. For each operation, a *preferred exponent* is defined as a function of the operation's inputs (see Section 3.4.7, page 97). The implementation has to return the member of the cohort result whose exponent is closest to the preferred exponent;

– If the exact result is not exactly representable as a floating-point number, it has to be rounded to a floating-point number. If that floating-point number has several possible representations, the returned result, both in binary and in decimal, is the one with the smallest possible exponent.

In practice, there are two classes of operations.

• When adding, subtracting, or multiplying two floating-point numbers, or when performing a fused multiply-add (FMA) operation, the infinitely precise result is actually always finite and may be exactly computed by an implementation. Therefore, the previous discussion has a straightforward translation into an architecture or a program.[1]

---

[1]In many cases, there are better ways of implementing the operation.

- When performing a division or computing a square root or an elementary function, the exact result may have an infinite number of digits; consider, for instance, the division $1.0/3.0$. In such cases, other means are used to reduce the rounding problem to a finite computation. For division, one may compute a finite-precision quotient, then the remainder allows one to decide how to round. Similarly, for the square root, one may compute $y \approx \sqrt{x}$, then decide rounding by considering $x - y^2$. Some iterations also allow, when an FMA instruction is available, to directly get a correctly rounded quotient or square root from an accurate enough approximation (see Section 5.3). Correct rounding of the elementary functions will be the subject of Chapters 11 and 12.

Section 8.2 addresses the general issue of rounding a value (the "compute rounding" box of Figure 8.1). The subsequent sections will address specifically each of the five basic operators.

8.2 Implementing IEEE 754-2008 Rounding

8.2.1 Rounding a nonzero finite value with unbounded exponent range

Obviously every nonzero finite real number x can be written as

$$x = (-1)^s \cdot m \cdot \beta^e, \tag{8.1}$$

where β is the chosen radix, here 2 or 10, where e is an integer, and where the real m is the (possibly infinitely precise) significand of x, such that $1 \leq m < \beta$. We will call the representation (8.1) the *normalized representation* of x. (Note however that this does *not* mean that x is a normal number in the IEEE 754 sense since x may have no finite radix-β expansion.) If we denote m_i the digit of weight β^{-i} (i.e., the i-th fractional digit) in the radix-β expansion of m, we have

$$m = \sum_{i \geq 0} m_i \beta^{-i} = (m_0.m_1 m_2 \ldots m_{p-1} m_p m_{p+1} \ldots)_\beta,$$

where $m_0 \in \{1, \ldots, \beta - 1\}$ and, for $i \geq 1$, $m_i \in \{0, 1, \ldots, \beta - 1\}$. In addition, an *unbounded exponent range* is assumed from now on, so that we do not have to worry about overflow, underflow, or subnormals (they will be considered in due course).

At precision p the result of rounding x is either the floating-point number

$$x_p = (-1)^s \cdot (m_0.m_1 m_2 \ldots m_{p-1})_\beta \cdot \beta^e$$

obtained by truncating the significand m after $p - 1$ fractional digits, or

- the floating-point successor of x_p when x is positive,

- the floating-point predecessor of x_p when x is negative.

In other words, writing $\mathrm{Succ}(x)$ for the successor of a floating-point number x, the rounded value of x will always be one of the two following values:

$$(-1)^s \cdot |x_p| \qquad \text{or} \qquad (-1)^s \cdot \mathrm{Succ}(|x_p|),$$

with $|x_p| = (m_0.m_1 m_2 \ldots m_{p-1})_\beta \cdot \beta^e$.

Note that rounding essentially reduces to rounding *non-negative* values, because of the following straightforward properties of rounding operators:

$$\mathrm{RN}(-x) = -\,\mathrm{RN}(x), \qquad \mathrm{RZ}(-x) = -\,\mathrm{RZ}(x),$$
$$\mathrm{RU}(-x) = -\,\mathrm{RD}(x). \tag{8.2}$$

Computing the successor in a binary interchange format

The reader may check that the binary interchange formats (see [187] and Chapter 3) are built in such a way that *the binary encoding of the successor of a positive floating-point value is the successor of the binary encoding of this value, considered as a binary integer*. This important property (which explains the choice of a biased exponent over two's complement or sign-magnitude) is true for all positive floating-point numbers, including subnormal numbers, from $+0$ to the largest finite number (whose successor is $+\infty$). It also has the consequence that the lexicographic order on the binary representations of positive floating-point numbers matches the order on the numbers themselves.

This provides us with a very simple way of computing $\mathrm{Succ}(|x_p|)$: consider the encoding of $|x_p|$ as an integer, and increment this integer. The possible carry propagation from the significand field to the exponent field will take care of the possible exponent change.

Example 9. *Considering the binary32 format, let $x_{24} = (2 - 2^{-23}) \cdot 2^{-126}$. The bit string $X_{31} \ldots X_0$ of the 32-bit integer $X = \sum_{i=0}^{31} X_i 2^i$ that encodes x_{24} is*

$$0 \quad \underbrace{00000001}_{\text{8 exponent bits}} \quad \underbrace{11111111111111111111111}_{\text{23 fraction bits}}.$$

The successor $\mathrm{Succ}(x_{24})$ of x_{24} is encoded by the 32-bit integer $X + 1$ whose bit string is

$$0 \quad \underbrace{00000010}_{\text{8 exponent bits}} \quad \underbrace{00000000000000000000000}_{\text{23 fraction bits}}.$$

That new bit string encodes the number $1 \cdot 2^{2-127}$, which is indeed $\mathrm{Succ}(x_{24}) = x_{24} + 2^{-23} \cdot 2^{-126} = 2^{-125}$. Note how the carry in the addition $X + 1$ has propagated up to the exponent field.

For algorithms that use a few floating-point operations for computing the predecessor and successor of a floating-point number, see [351].

Choosing between $|x_p|$ and its successor $\mathrm{Succ}(|x_p|)$

As already detailed by Table 2.1, page 22 for radix 2, the choice between $|x_p|$ and $\mathrm{Succ}(|x_p|)$ depends on the sign s, the rounding mode, the value of the digit m_p of m (called the *round digit*), and a binary information telling us if there exists at least one nonzero digit among the (possibly infinitely many) remaining digits $m_{p+1}, m_{p+2}, \ldots$. In radix 2, this information may be defined as the logical OR of all the bits to the right of the round bit, and is therefore named the *sticky bit*. In radix 10, the situation is very similar. One still needs a binary information, which we still call the sticky bit. It is no longer defined as a logical OR, but as follows: its value is 0 if all the digits to the right after m_p are zero, and 1 otherwise.

Let us consider some decimal cases for illustration.

- When rounding x toward zero, the rounded number is always[2] x_p.

- When rounding a positive x toward $+\infty$, the rounded number is $\mathrm{Succ}(x_p)$, except if x was already a representable number, i.e., when both its round digit and sticky bit are equal to zero.

- When rounding to nearest with *roundTiesToEven* a positive decimal number x, if the round digit m_p belongs to $\{0, 1, 2, 3, 4\}$, then the rounded number is x_p; if m_p belongs to $\{6, 7, 8, 9\}$, then the rounded number is $\mathrm{Succ}(x_p)$. If m_p is equal to 5, then the sticky bit will decide between $\mathrm{Succ}(x_p)$ (if equal to 1) or a tie (if equal to 0). In case of a tie, the *ties to even* rule considers the last digit (of weight 10^{-p+1}) of the two candidates. The rounded result is the one whose last digit is even.

Having defined the infinitely accurate normalized representation $x = (-1)^s \cdot m \cdot \beta^e$ with $1 \leq m < \beta$ of the result allows us to manage flags and exceptional cases as well. However, note first that for some operations, overflow or underflow signaling may be decided by considering the inputs only, before any computation of the results. For example, as we will see later in this chapter, square root overflows if and only if the input is $+\infty$, never underflows, and returns NaN if and only if the input is NaN or strictly negative. The possibility of such an *early detection* of exceptional situations will be mentioned when appropriate.

8.2.2 Overflow

As stated in Section 3.4.10, page 101, the overflow exception is signaled when the absolute value of the intermediate result is strictly larger than the largest finite number $\Omega = (\beta - \beta^{1-p}) \cdot \beta^{e_{\max}}$. Here, the intermediate result is defined

---

[2]In this discussion we assume that $4.999\ldots9^\infty$ is written 5.0^∞; otherwise, this sentence is not true. This remark is academic: a computer will only deal with finite representations, which do not raise this ambiguity.

as the infinitely accurate result rounded to precision p with an unbounded exponent range.

For rounding to the nearest, this translates to: an overflow is signaled when

$$(e > e_{\max})$$

or

$$\left(e = e_{\max} \quad \text{and} \quad (m_0.m_1 m_2 \ldots m_{p-1})_\beta = \beta - \beta^{1-p} \quad \text{and} \quad m_p \geq \frac{\beta}{2} \right).$$

Note that in the case $m_p = \frac{\beta}{2}$, when $e = e_{\max}$ and $(m_0.m_1 m_2 \ldots m_{p-1})_\beta = \beta - \beta^{1-p}$, with *roundTiesToEven*, the exact result is rounded to the intermediate result $\beta^{e_{\max}+1}$; therefore, it signals overflow without having to consider the sticky bit.

When rounding a positive number to $+\infty$, an overflow is signaled when

$$(e > e_{\max})$$

or

$$\left(e = e_{\max} \quad \text{and} \quad (m_0.m_1 \ldots m_{p-1})_\beta = \beta - \beta^{1-p} \right.$$

$$\left. \text{and} \quad (m_p > 0 \quad \text{or} \quad \text{sticky} = 1) \right).$$

This reminds us that overflow signaling is dependent on the prevailing rounding direction. The other combinations of sign and rounding direction are left as an exercise to the reader.

8.2.3 Underflow and subnormal results

As stated in Section 3.4.10, page 102, the underflow exception is signaled when a nonzero result whose absolute value is strictly less than $\beta^{e_{\min}}$ is computed.[3] This translates to: an underflow is signaled if $e < e_{\min}$, where e is the exponent of the normalized infinitely precise significand.

In such cases, the previous rounding procedure has to be modified as follows: m (the normalized infinitely precise significand) is shifted right by $e_{\min} - e$ (it will no longer be normalized), and e is set to $e_{\min}$. We thus have rewritten x as

$$x = (-1)^s \cdot m' \cdot \beta^{e_{\min}},$$

with

$$m' = (m_0'.m_1' m_2' \ldots m_{p-1}' m_p' m_{p+1}' \ldots)_\beta.$$

---

[3]We remind the reader that there remains some ambiguity in the standard, since underflow can be detected before or after rounding. See Section 2.1, page 18, and Section 3.4.10, page 102, for more on this. Here, we describe underflow detection *before rounding*.

From this representation, we may define the round digit m'_p, the sticky bit (equal to 1 if there exists a nonzero m'_i for some $i > p$, and 0 otherwise), and the truncated value $|x_p| = (m'_0.m'_1 m'_2 \ldots m'_{p-1})_\beta \cdot \beta^{e_{\min}}$ as previously. As the successor function is perfectly defined on the subnormal numbers—and even easy to compute in the binary formats—the rounded value is decided among $(-1)^s \cdot |x_p|$ and $(-1)^s \cdot \mathrm{Succ}(|x_p|)$ in the same way as in the normal case.

One will typically need the implementations to build the *biased exponent* (that is, in binary, what is actually stored in the exponent field), equal to the exponent plus the bias (see Table 3.4, page 60). There is one subtlety to be aware of in binary formats: the subnormal numbers have the same exponent as the smallest normal numbers, although their biased exponent is smaller by 1. In general, we may define n_x as the "is normal" bit, which may be computed as the OR of the bits of the exponent field. Its value will be 0 for subnormal numbers and 1 for normal numbers. Then the relation between the value of the exponent e_x and the biased exponent E_x is the following:

$$e_x = E_x - \mathrm{bias} + 1 - n_x \quad . \tag{8.3}$$

This relation will allow us to write exponent-handling expressions that are valid in both the normal and subnormal cases.

In addition, n_x also defines the value of the implicit leading bit: the actual significand of a floating-point number is obtained by prepending n_x to the significand field.

8.2.4 The inexact exception

This exception is signaled when the exact result y is not exactly representable as a floating-point number ($\circ(y) \neq y$, y not a NaN). As the difference between $\circ(y)$ and y is condensed in the round digit and the sticky bit, the inexact exception will be signaled unless both the round digit and the sticky bit are equal to 0.

8.2.5 Rounding for actual operations

Actual rounding of the result of an operation involves two additional difficulties.

- Obtaining the intermediate result in normalized form may require some work, all the more as some of the inputs, or the result, may belong to the subnormal range. In addition, decimal inputs may not be normalized (see the definition of cohorts in Section 3.4.3, page 82).

- For decimal numbers, the result should not always be normalized (see the definition of preferred exponents in Section 3.4.7, page 97).

These two problems will be addressed on a per-operation basis.

Decimal rounding using the binary encoding

The entire discussion in Section 8.2 assumes that the digits of the infinitely precise significand are available in the radix in which it needs to be rounded. This is not the case for the *binary encoding* of the decimal formats (see Section 3.4.3, pages 82 and seq.). In this case, one first needs to convert the binary encoding to decimal digits, at least for the digits needed for rounding (the round digit and the digits to its right). Such radix conversion is typically done through the computation of a division by some 10^k (with $k > 0$) with remainder. Cornea et al. [85, 87] have provided several efficient algorithms for this purpose, replacing the division by 10^k with a multiplication by a pre-computed approximation to 10^{-k}. They also provide techniques to determine to which precision 10^{-k} should be precomputed.

8.3   Floating-Point Addition and Subtraction

When x or y is nonzero, the addition of $x = (-1)^{s_x} \cdot |x|$ and $y = (-1)^{s_y} \cdot |y|$ is based on the identity

$$x + y = (-1)^{s_x} \cdot \Big(|x| + (-1)^{s_z} \cdot |y| \Big), \qquad s_z = s_x \text{ XOR } s_y \in \{0, 1\}. \qquad (8.4)$$

For subtraction a similar identity obviously holds since $x - y = x + (-y)$. *Hence, in what follows we shall consider addition only.*

The IEEE 754-2008 specification for $|x| \pm |y|$ is summarized in Tables 8.2 and 8.3. Combined with (8.2) and (8.4) it specifies floating-point addition completely provided x or y is nonzero. When both x and y are zero, the standard stipulates to return $+0$ or -0, depending on the operation (addition or subtraction) and the rounding direction attribute, as shown in Table 8.1.

In Table 8.2 and Table 8.3 the sum or difference $\circ(|x| \pm |y|)$ of the two positive finite floating-point numbers

$$|x| = m_x \cdot \beta^{e_x} \qquad \text{and} \qquad |y| = m_y \cdot \beta^{e_y}$$

is given by

$$\circ \Big(|x| \pm |y| \Big) = \circ \Big(m_x \cdot \beta^{e_x} \pm m_y \cdot \beta^{e_y} \Big). \qquad (8.5)$$

The rest of this section discusses the computation of the right-hand side of the above identity.

Note that for floating-point addition/subtraction, the only possible exceptions are *invalid operation*, *overflow*, *underflow*, and *inexact* (see [126, p. 425]).

In more detail, the sequence of operations traditionally used for implementing (8.5) is as follows.

- First, the two exponents e_x and e_y are compared, and the inputs x and y are possibly swapped to ensure that $e_x \geq e_y$.

$x \text{ op } y$	$\circ(x \text{ op } y)$ for $\circ \in \{\text{RN}, \text{RZ}, \text{RU}\}$	$\text{RD}(x \text{ op } y)$
$(+0) + (+0)$	$+0$	$+0$
$(+0) + (-0)$	$+0$	-0
$(-0) + (+0)$	$+0$	-0
$(-0) + (-0)$	-0	-0
$(+0) - (+0)$	$+0$	-0
$(+0) - (-0)$	$+0$	$+0$
$(-0) - (+0)$	-0	-0
$(-0) - (-0)$	$+0$	-0

Table 8.1: *Specification of addition/subtraction when both x and y are zero. Note that floating-point addition is commutative.*

$|x| + |y|$		$|y|$			
		$+0$	(sub)normal	$+\infty$	NaN
	$+0$	$+0$	$|y|$	$+\infty$	qNaN
$|x|$	(sub)normal	$|x|$	$\circ(|x| + |y|)$	$+\infty$	qNaN
	$+\infty$	$+\infty$	$+\infty$	$+\infty$	qNaN
	NaN	qNaN	qNaN	qNaN	qNaN

Table 8.2: *Specification of addition for positive floating-point data.*

$|x| - |y|$		$|y|$			
		$+0$	(sub)normal	$+\infty$	NaN
	$+0$	± 0	$-|y|$	$-\infty$	qNaN
$|x|$	(sub)normal	$|x|$	$\circ(|x| - |y|)$	$-\infty$	qNaN
	$+\infty$	$+\infty$	$+\infty$	qNaN	qNaN
	NaN	qNaN	qNaN	qNaN	qNaN

Table 8.3: *Specification of subtraction for floating-point data of positive sign. Here ± 0 means $+0$ for "all rounding direction attributes except roundTowardNegative" ($\circ = \text{RD}$), and -0 for $\circ = \text{RD}$; see [187, §6.3].*

- A second step is to compute $m_y \cdot \beta^{-(e_x - e_y)}$ by shifting m_y right by $e_x - e_y$ digit positions (this step is sometimes called *significand alignment*). The exponent result e_r is tentatively set to e_x.

- The result significand is computed as $m_r = m_x + (-1)^{s_z} \cdot m_y \cdot \beta^{-(e_x - e_y)}$: either an addition or a subtraction is performed, depending on the signs s_x and s_y. Then if m_r is negative, it is negated. This (along with the signs s_x and s_y) determines the sign s_r of the result. At this step, we have an exact sum $(-1)^{s_r} \cdot m_r \cdot \beta^{e_r}$.

- This sum is not necessarily normalized (in the sense of Section 8.2). It may need to be normalized in two cases.

  - There was a carry out in the significand addition ($m_r \geq \beta$). Note that m_r always remains strictly smaller than 2β, so this carry is at most 1. In this case, m_r needs to be divided by β (i.e., shifted right by one digit position), and e_r is incremented, unless e_r was equal to $e_{\max}$, in which case an overflow is signaled as per Section 3.4.10, page 101.

  - There was a cancellation in the significand addition ($m_r < 1$). In general, if λ is the number of leading zeros of m_r, m_r is shifted left by λ digit positions, and e_r is set to $e_r - \lambda$. However, if $e_r - \lambda < e_{\min}$ (the cancellation has brought the intermediate result in the underflow range, see Section 3.4.10, page 102), then the exponent is set to $e_{\min}$ and m_r will be shifted left only by $e_r - e_{\min}$.

  Note that for decimals, the preferred exponent rule (mentioned in Section 3.4.7, page 97) states that *inexact* results must be normalized as just described, but not *exact* results. We will come back to this case.

- Finally, the normalized sum (which again is always finite) is rounded as per Section 8.2.

Let us now examine this algorithm more closely. We can make important remarks.

1. This algorithm never requires more than a p-digit effective addition for the significands. This is easy to see in the case of an addition: the least significant digits of the result are those of m_y, since they are added to zeros. This is also true when y is subtracted, provided the sticky bit computation is modified accordingly.

2. The alignment shift need never be by more than $p + 1$ digits. Indeed, if the exponent difference is larger than $p + 1$, y will only be used for computing the sticky bit, and it doesn't matter that it is not shifted to its proper place.

3. Leading-zero count and variable shifting will only be needed in case of a cancellation, i.e., when the significands are subtracted and the exponent difference is 0 or 1. But in this case, several things are simpler. The sticky bit is equal to zero and need not be computed. More importantly, the alignment shift is only by 0 or 1 digit.

   In other words, although two large shifts are mentioned in the previous algorithm (one for significand alignment, the other one for normalization in case of a cancellation), they are mutually exclusive. The literature defines these mutually exclusive cases as the *close* case (when the exponents are close) and the *far* case (when their difference is larger than 1).

4. In our algorithm, the normalization step has to be performed before rounding: indeed, rounding requires the knowledge of the position of the round and sticky bits, or, in the terminology of Section 8.2, it requires a *normalized* infinite significand. However, here again the distinction between the close and far cases makes things simpler. In the close case, the sticky bit is zero whatever shift the normalization entails. In the far case, normalization will entail a shift by at most one digit. Classically, the initial sticky bit is therefore computed out of the digits to the right of the $(p+2)$-nd (directly out of the lower digits of the lesser addend). The $(p+2)$-nd digit is called the *guard* digit. It will either become the round digit in case of a 1-digit shift, or it will be merged to the previous sticky bit if there was no such shift. The conclusion of this is that the bulk of the sticky bit computation can be performed in parallel with the significand addition.

Let us now detail specific cases of floating-point addition.

8.3.1 Decimal addition

We now come back to the preferred exponent rule (see Section 3.4.7, page 97), which states that *exact* results should not be normalized. As the notion of exactness is closely related to that of normalization (a result is exact if it has a normalized representation that fits in p digits), the general way to check exactness is to first normalize X, then apply the previous algorithm.

Exactness of the intermediate result is then determined combinatorially out of the carry-out and sticky bits, and the round and guard digits.

For addition, the preferred exponent is the smaller of the input exponents (in other words, e_y and not e_x). If the result is exact, we therefore need to shift m_r right and reduce e_r to e_y. Therefore, the preferred exponent rule means two large shifts.

In case of a carry out, it may happen that the result is exact, but the result's cohort does not include a member with the preferred exponent. An example is $9.999e0 + 0.0001e0$ for a $p = 4$-digit system.

Both input numbers have the same quantum exponent, yet the (exact) value of the result, 10, cannot be represented with the same quantum exponent and must be represented as $1.000e1$.

In practice, the exact case is a common one in decimal applications (think of accounting), and even hardware implementations of decimal floating-point addition distinguish it and try to make this common case fast.

The IBM POWER6 [123] distinguishes the following three cases (from the simplest to the most complex).

Case 1 Exponents are equal: This is the most common case of accounting: adding amounts of money which have the decimal point at the same place. It is also the simplest case, as no alignment shifting is necessary. Besides, the result is obtained directly with the preferred exponent. It may still require a one-digit normalization shift and one-digit rounding in case of overflow, but again such an overflow is highly rare in accounting applications using decimal64—it would correspond to astronomical amounts of money!

Case 2 Aligning to the operand with the smaller exponent: When the exponent difference is less than or equal to the number of leading zeros in the operand with the bigger exponent, the operand with the larger exponent can be shifted left to properly align it with the smaller exponent value. Again, after normalization and rounding, the preferred exponent is directly obtained.

Case 3 Shifting both operands: This is the general case that we have considered above.

The interested reader will find in [123] the detail of the operations performed in each case in the POWER6 processor. This leads to a variable number of cycles for decimal addition—9 to 17 for decimal64.

8.3.2   Decimal addition using binary encoding

A complete algorithm for the addition of two decimal floating-point numbers in the binary encoding is presented in [85, 87].

The main issue with the binary encoding is the implementation of the shifts. The number $M_1 \cdot 10^{e_1} + M_2 \cdot 10^{e_2}$, with $e_1 \geq e_2$, is computed as

$$10^{e_2} \cdot \left(M_1 \cdot 10^{e_1 - e_2} + M_2 \right).$$

Instead of a shift, the significand addition now requires a multiplication by some 10^k, with $0 \leq k \leq p$, because of remark 2 on page 248. There are few such constants, so they may be tabulated, and a multiplier or FMA will then compute the "shifted" values [85, 87]. The full algorithm takes into account the number of decimal digits required to write M_1 and M_2, which is useful to obtain the preferred exponent. This number is computed by table lookup.

For the full algorithm, the interested reader is referred to [87].

8.3.3   Subnormal inputs and outputs in binary addition

Here are the main modifications to the previous addition algorithm to ensure that it handles subnormal numbers properly in the binary case. Let us define, for the inputs x and y, the "is normal" bits n_x and n_y. One may compute n_x (resp. n_y) as the OR of the bits of the exponent field of x (resp. y).

- The implicit leading bit of the significand of x (resp. y) is now set to n_x (resp. n_y).

- If E_x and E_y are the respective biased exponents of the inputs, we now have $e_x = E_x - \text{bias} + 1 - n_x$ and $e_y = E_y - \text{bias} + 1 - n_y$. The exponent difference, used for the alignment shifting, is now computed as $E_x - n_x - E_y + n_y$. Of course, two subnormal inputs are already aligned.

- As already stated in Section 8.2.3, the normalization shift should handle subnormal outputs: the normalization shift distance will be $\min(\lambda, e_x - e_{\min})$ digit positions, where λ is the leading-zero count. The output is subnormal if $\lambda > e_x - e_{\min}$.

8.4   Floating-Point Multiplication

Floating-point multiplication is much simpler than addition. Given $x = (-1)^{s_x} \cdot |x|$ and $y = (-1)^{s_y} \cdot |y|$, the exact product $x \times y$ satisfies

$$x \times y = (-1)^{s_r} \cdot \big(|x| \times |y|\big), \qquad s_r = s_x \text{ XOR } s_y \in \{0, 1\}. \qquad (8.6)$$

The IEEE 754-2008 specification for $|x| \times |y|$ is summarized in Table 8.4. Combined with (8.2) and (8.6), it specifies floating-point multiplication completely.

$|x| \times |y|$		$|y|$			
		+0	(sub)normal	$+\infty$	NaN
	+0	+0	+0	qNaN	qNaN
	(sub)normal	+0	$\circ(|x| \times |y|)$	$+\infty$	qNaN
$|x|$	$+\infty$	qNaN	$+\infty$	$+\infty$	qNaN
	NaN	qNaN	qNaN	qNaN	qNaN

Table 8.4: Specification of multiplication for floating-point data of positive sign.

In Table 8.4 the product $\circ(|x| \times |y|)$ of the two positive finite floating-point numbers

$$|x| = m_x \cdot \beta^{e_x} \qquad \text{and} \qquad |y| = m_y \cdot \beta^{e_y}$$

is given by

$$\circ(|x| \times |y|) = \circ\bigl(m_x m_y \cdot \beta^{e_x + e_y}\bigr). \tag{8.7}$$

The rest of this section discusses the computation of the right-hand side of (8.7).

For floating-point multiplication, the only possible exceptions are *invalid operation*, *overflow*, *underflow*, and *inexact* (see [126, p. 438]).

8.4.1   Normal case

Let us first consider the case when both inputs are normal numbers such that $1 \leq m_x < \beta$ and $1 \leq m_y < \beta$ (this is notably the case for binary normal numbers). It follows that the exact product $m_x m_y$ satisfies $1 \leq m_x m_y < \beta^2$. This shows that the significand product has either one or two nonzero digits left to the point. Therefore, to obtain the normalized significand required to apply the methods given in Section 8.2, the significand product $m_x m_y$ may need to be shifted right by one position. This is exactly similar to the *far* case of addition, and will be handled similarly, with a guard and a round digit, and a partial sticky bit. Since the product of two p-digit numbers is a $2p$-digit number, this partial sticky computation has to be performed on $p - 1$ digits.

The significand multiplication itself is a fixed-point multiplication, and much literature has been dedicated to it; see for instance [126] and references therein. Hardware implementations, both for binary and decimal, are surveyed in Section 9.2.4. Chapter 10 discusses issues related to software implementations.

In binary, the exponent is equal to the biased exponent E_x minus the bias (see Section 3.1, page 56). The exponent computation is therefore $e_x + e_y = E_x - \text{bias} + E_y - \text{bias}$. One may directly compute the biased exponent of the result (before normalization) as $E_x + E_y - \text{bias}$.

8.4.2   Handling subnormal numbers in binary multiplication

We now extend the previous algorithm to accept subnormal inputs and produce subnormal outputs when needed.

Let us define again, for the inputs x and y, the "is normal" bits n_x and n_y. One may compute n_x as the OR of the bits of the exponent field E_x. This bit can be used as the implicit bit to be added to the significand, and also as the bias correction for subnormal numbers: we now have $e_x = E_x - \text{bias} + 1 - n_x$ and $e_y = E_y - \text{bias} + 1 - n_y$.

Let us first note that if both operands are subnormal ($n_x = 0$ and $n_y = 0$), the result will be zero or one of the smallest subnormals, depending on the rounding mode. This case is therefore handled straightforwardly.

Let us now assume that only one of the operands is subnormal. The simplest method is to normalize it first, which will bring us back to the normal case. For this purpose we need to count its leading zeros. Let us call l the

number of leading zeros in the significand extended by n_x. We have $l = 0$ for a normal number and $l \geq 1$ for a subnormal number. The subnormal significand is then shifted left by l bit positions, and its exponent becomes $e_{\min} - l$. Obviously, this requires a larger exponent range than what the standard format offers. In practice, the exponent data is only one bit wider.

An alternative to normalizing the subnormal input prior to a normal computation is to normalize the product after the multiplication: indeed, the same multiplication process which computes the product of two p-bit numbers will compute this product exactly if one of the inputs has l leading zeros. The product will then have l or $l+1$ leading zeros, and will need to be normalized by a left shift. However, the advantage of this approach is that counting the subnormal leading zeros can be done in parallel with the multiplication. Therefore, this alternative will be preferred in the hardware implementations presented in the next chapter.

For clarity, we now take the view that both inputs have been normalized with a 1-bit wider exponent range, and focus on producing subnormal outputs. Note that they may occur even for normal inputs. They may be handled by the standard normalization procedure of Section 8.2. It takes an arbitrary shift right of the significand: if $e_x + e_y < e_{\min}$, shift right by $e_{\min} - (e_x + e_y)$ before rounding.

To summarize, the cost of handling subnormal numbers is: a slightly larger exponent range for internal exponent computations, a leading-zero counting step, a left-shifting step of either the subnormal input or the product, and a right-shifting step before rounding. Section 9.4.4 will show how these additional steps may be scheduled in a hardware implementation in order to minimize their impact on the delay.

8.4.3   Decimal specifics

For multiplication, the preferred exponent rule (see Section 3.4.7, page 97) mentions that, for exact results, the preferred quantum exponent is $Q(x) + Q(y)$. We recall the relation $Q(x) = e_x - p + 1$, see Section 3.4.

Again, exactness may be computed by first normalizing the two inputs, then computing and normalizing the product, then observing its round digit and sticky bit.

However, exactness may also be predicted when the sum of the numbers of leading zeros of both multiplicands is larger than p, which will be a very common situation.

To understand why, think again of accounting. Multiplication is used mostly to apply a rate to an account (a tax rate, an interest rate, a currency conversion rate, etc.). After the rate has been applied, the result is rounded to the nearest cent before being further processed. Rounding to the nearest cent can be performed using the *quantize* operation specified by the IEEE 754-2008 standard.

Such rates are 3- to 6-digit numbers (the euro official conversion rates with respect to the currencies it replaces were all defined as 6-digit numbers). The product of such a rate with your bank account will be exact (when using the 16-digit format decimal64), unless your wealth exceeds 10^{10} cents (one hundred million dollars), in which case your bank will be happy to spend a few extra cycles to manage it.

In the common case when the product is exact, the quantum exponent is set to $Q(x) + Q(y)$ (it makes sense to have a zero quantum exponent for the rate, so that the product is directly expressed in cents) and the product needs no normalization. The significand to output is simply the last p digits of the product.

Counting the leading zeros of the inputs is an expensive operation, but it may be performed in parallel to the multiplication.

To summarize, an implementation may compute in parallel the $2p$-bit significand product and the two leading-zero counts of the inputs. If the sum of the counts is larger than p (common case), the result is exact, and no rounding or shift is required (fast). Otherwise, the result needs to be normalized, and then rounded as per Section 8.2. Note that the result may also be exact in this case, but then the result significand is too large to be representable with the preferred exponent. The standard requires an implementation to return the representable number closest to the result, which is indeed the normalized one.

8.5   Floating-Point Fused Multiply-Add

When computing $\circ(ab + c)$, the product ab is a $2p$-digit number, and needs to be added to the p-digit number c. Sign handling is straightforward: what actually matters is whether the operation will be an effective subtraction or an effective addition.

We base the following analysis on the actual exponent of the input numbers, denoted e_a, e_b, and e_c. Computing $ab + c$ requires first aligning the product ab and, with the summand c, using the exponent difference

$$d = e_c - (e_a + e_b).$$

In the following figures, the precision used is $p = 5$ digits.

8.5.1   Case analysis for normal inputs

If a and b are normal numbers, one has $|a| = m_a \cdot \beta^{e_a}$ and $|b| = m_b \cdot \beta^{e_b}$, and the product $|ab|$ has at most two digits in front of the point corresponding to $e_a + e_b$:

$$|ab| = \beta^{e_a + e_b} \cdot \boxed{} \qquad\qquad |c| = \beta^{e_c} \cdot \boxed{}$$
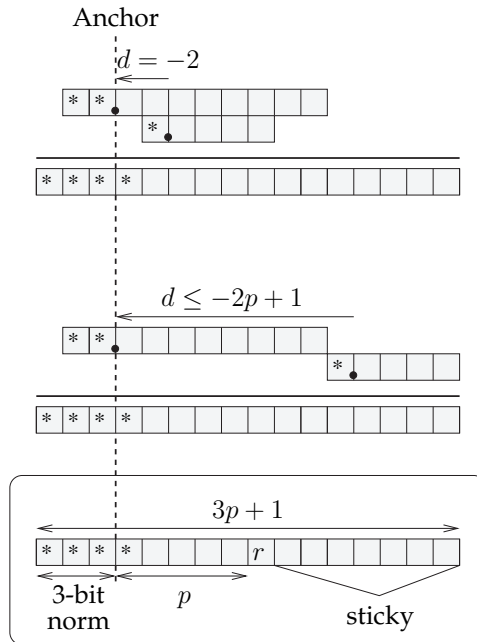
Figure 8.2: Product-anchored FMA computation for normal inputs. The stars show the possible position of the leading digit.

One may think of first performing a 1-digit normalization of this product ab, but this would add to the computation a step which can be avoided, and it only marginally reduces the number of cases to handle. Following most implementations, we therefore chose to base the discussion of the cases on the three input exponents only. We now provide an analysis of the alignment cases that may occur. These cases are mutually exclusive. After any of them, we need to perform a rounding step as per Section 8.2. This step may increment the result exponent again.

Product-anchored case

The exponent of the result is almost that of the product, and no cancellation can occur, for $d \leq -2$, as illustrated by Figure 8.2.

In this case, the leading digit may have four positions: the two possible positions of the leading digit of ab, one position to the left in case of effective addition, and one position to the right in case of effective subtraction. This defines the possible positions of the round digit. All the digits lower than the lower possible position of the round digit may be condensed in a sticky bit. An actual addition is only needed for the digit positions corresponding to the digits of ab (see Figure 8.2); therefore, all the bits shifted out of this range may be condensed into a sticky bit before addition. If $d \leq -2p + 1$, all the digits
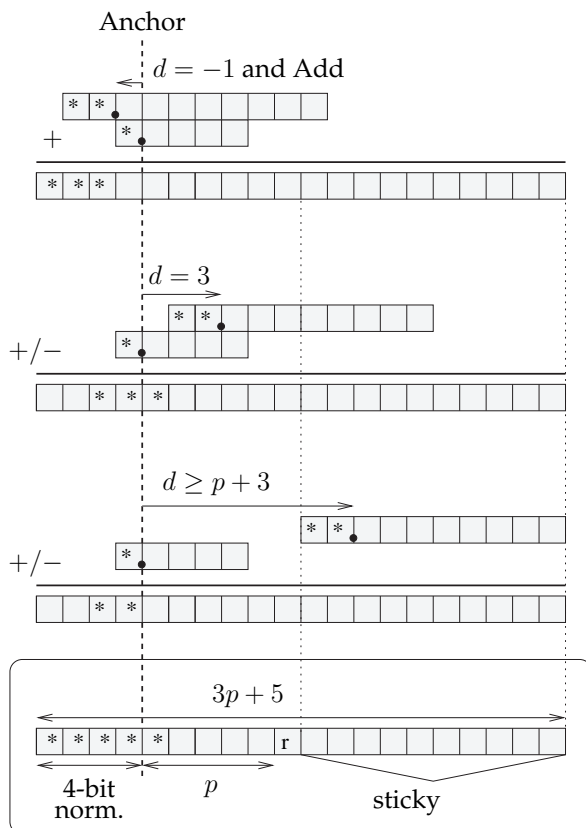
Figure 8.3: Addend-anchored FMA computation for normal inputs. The stars show the possible position of the leading digit.

of c will go to the sticky bit. This defines the largest alignment one may need to perform in this case: if $d \leq -2p + 1$, a shift distance of $2p - 1$ and a sticky computation on all the shifted p bits will provide the required information (Is there a nonzero digit to the right of the round bit?).

To summarize, this case needs to perform a shift of c by at most $2p - 1$ with a p-bit sticky computation on the lower bits of the output, a $2p$-bit addition with sticky output of the $p - 3$ lower bits, and a 3-bit normalization (updating the sticky bit). The exponent is set tentatively to $e_a + e_b$, and the 3-bit normalization will add to it a correction in $\{-1, 0, 1, 2\}$.

Addend-anchored case

The exponent of the result will be close to that of the addend (and no cancellation can occur) when ($d \geq 3$ or ($d \geq -1$ and EffectiveAdd)), as illustrated by Figure 8.3. In that case, the leading digit may be in five different positions.
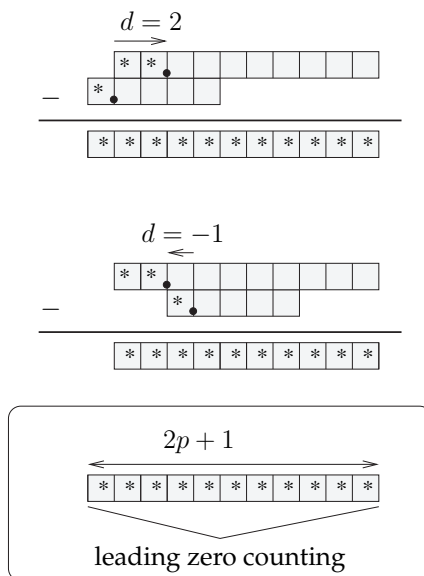
Figure 8.4: Cancellation in the FMA. The stars show the possible position of the leading digit.

The whole of ab may be condensed in a sticky bit as soon as there is a gap of at least two digits between ab and c. One gap digit is needed for the case of an effective subtraction $|c| - |ab|$, when the normalized result exponent may be one less than that of c (for instance, in decimal with $p = 3$, $1.00 - 10^{-100}$ rounded down returns 0.999). The second gap digit is the round digit for rounding to the nearest. A sufficient condition for condensing all of ab in a sticky bit is therefore $d \geq p + 3$ (see Figure 8.3).

To summarize, this case needs to perform a shift of ab by at most $p + 3$ positions, a $(p + 2)$-bit addition, a $2p$-bit sticky computation, and a 4-bit normalization (updating the sticky bit). The exponent is set tentatively to e_c, and the 4-bit normalization will add to it a correction in $\{-1, 0, 1, 2, 3\}$.

Cancellation

If $-1 \leq d \leq 2$ and the FMA performs an effective subtraction, a cancellation may occur. This happens for four values of the exponent difference d, versus only three in the case of the floating-point addition, because of the uncertainty on the leading digit of the product. Possible cancellation situations are illustrated by Figure 8.4.

In that case we need a $(2p + 1)$-digit addition, and an expensive normalization consisting of leading-zero counting and right shifting, both of size $2p + 1$.
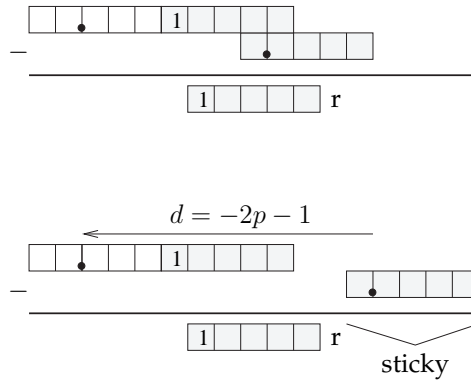
Figure 8.5: FMA $ab - c$, where a is the smallest subnormal, ab is nevertheless in the normal range, $|c| < |ab|$, and we have an effective subtraction. Again, the dot corresponds to an exponent value of $e_a + e_b$.

Example 10. *Consider in radix 2, precision p the inputs $a = b = 1 - 2^{-p}$ and $c = -(1 - 2^{-p+1})$. The FMA should return the exact result $ab + c = 2^{-2p}$. In this example $e_a = e_b = e_c = -1$, $d = 1$, and there is a $(2p - 1)$-bit cancellation.*

The result exponent is equal to $e_a + e_b + 3 - \lambda$, where λ is the leading-zero count.

8.5.2   Handling subnormal inputs

To manage subnormal inputs, we define n_a, n_b, and n_c as the "is normal" bits. In binary floating-point, these bits are inserted as leading bits of the significands, and the exponent difference that drives the aligner becomes

$$d = e_c - (e_a + e_b) = E_c - E_a - E_b + \text{bias} - 1 - n_c + n_a + n_b. \qquad (8.8)$$

If both a and b are subnormal, the whole of ab will be condensed in a sticky bit even if c is subnormal. Let us therefore focus on the case when only one of a or b is subnormal. Most of the previous discussion remains valid in this case, with the following changes.

- The significand product may now have up to p leading zeros. Indeed, as the situation where both inputs are subnormals is excluded, the smallest significand product to consider is the smallest subnormal significand, equal to β^{-p+1}, multiplied by the smallest normal significand, equal to 1. This is illustrated by Figure 8.5.

- In the product-anchored case, this requires us to extend the shift by two more digit positions, for the cases illustrated by Figure 8.5. The maximum shift distance is now $-d = 2p + 1$.

- In fact, the product-anchored case is not necessarily product anchored if one of the multiplicands is subnormal. The leading digit of the result may now come from the addend—be it normal or subnormal. Still, this requires neither a larger shift, nor a larger addition, than that shown on Figure 8.2. However, the partial sticky computation from the lower bits of the product shown on this figure is now irrelevant: one must first determine the leading digit before knowing which digits go to the sticky bit. This requires a leading-zero counting step on p digits. In this respect, the product-anchored case when either a or b is subnormal now closely resembles the cancellation case, although it requires a smaller leading-zero counting (p digits instead of $2p+1$).

- The addend c may have up to $p-1$ leading zeros, which is more than what is shown on Figure 8.3, but they need not be counted, as the exponent is stuck to $e_{\min}$ in this case.

8.5.3 Handling decimal cohorts

In decimal, the previous case analysis is valid (we have been careful to always use the word "digit"). The main difference is the handling of cohorts, which basically means that a decimal number may be subnormal for any exponent. As a consequence, there may be more stars in Figures 8.2 to 8.4. In particular, the addend-anchored case may now require up to a $(p+4)$-digit leading-zero count instead of p.

In addition, one must obey the preferred exponent rule: for inexact results, the preferred exponent is the least possible (this corresponds to normalization in the binary case, and the previous analysis applies). For exact results, the preferred quantum exponent is $\min(Q(a) + Q(b), Q(c))$. As for addition and multiplication, this corresponds to avoiding any normalization if possible.

The only available decimal FMA implementation, to our knowledge, is a software one, part of the Intel Decimal Floating-Point Math Library [85, 87]. A hardware decimal FMA architecture is evaluated in Vázquez's Ph.D. dissertation [413].

Let us now conclude this section with a complete algorithmic description of an implementation of the binary FMA.

8.5.4 Overview of a binary FMA implementation

Most early hardware FMA implementations chose to manage the three cases evoked above (product-anchored, addend-anchored, and canceling/subnormal) in a single computation path [281, 183]. In the next chapter, more recent, multiple-path implementations [373, 238, 339] will be reviewed.

Here is a summary of the basic single-path implementation handling subnormals. Figure 8.6 represents the data alignment in this case. It is a
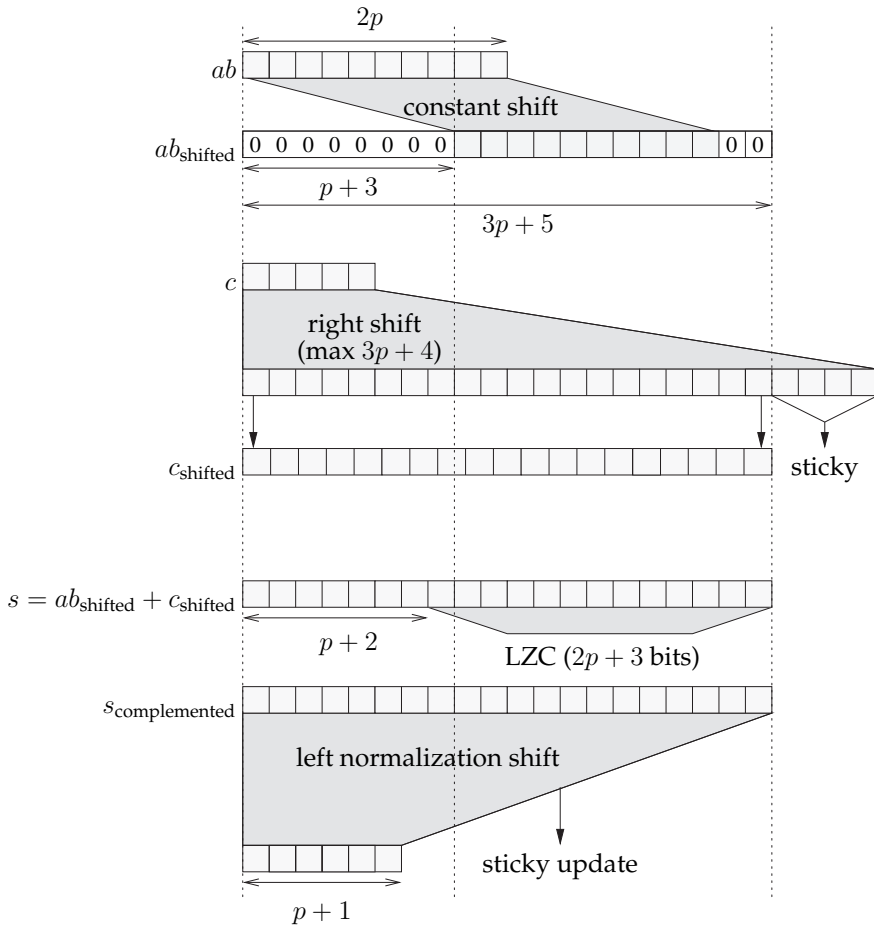
Figure 8.6: Significand alignment for the single-path algorithm.

simplified superimposition of Figures 8.2 to 8.5. The single-path approach is in essence product anchored.

- The "is normal" bits are determined, and added as implicit bits to the three input significands. The exponent difference d is computed as per (8.8).

- The $2p$-digit significand of ab is shifted right by $p+3$ digit positions (this is a constant distance). Appending two zeros to the right, we get a first $(2p + 5)$-digit number ab_{shifted}.

- The summand shift distance d' and the tentative exponent e'_r are determined as follows:

    - if $d \leq -2p + 1$ (product-anchored case with saturated shift), then $d' = 3p + 4$ and $e'_r = e_a + e_b$;
    - if $-2p + 1 < d \leq 2$ (product-anchored case or cancellation), then $d' = p + 3 - d$ and $e'_r = e_a + e_b$;
    - if $2 < d \leq p + 2$ (addend-anchored case), then $d' = p + 3 - d$ and $e'_r = e_c$;
    - if $d \geq p + 3$ (addend-anchored case with saturated shift), then $d' = 0$ and $e'_r = e_c$.

- The p-digit significand c is shifted right by d' digit positions. The maximum shift distance is $3p + 4$ digits, for instance, 163 bits for binary64.

- The lower $p - 1$ digits of the shifted c are compressed into a sticky bit. The leading $2p + 5$ digits form c_{shifted}.

- In case of effective subtraction, c_{shifted} is bitwise inverted.

- The product ab_{shifted} and the possibly inverted addend c_{shifted} are added (with a carry in in case of an effective subtraction), leading to a $(3p+4)$-digit number s. This sum may not overflow because of the gap of two zeros in the case $d = p + 3$.

- If s is negative, it is complemented. Note that the sign of the result can be predicted from the signs of the inputs and the exponents, except when the operation is an effective subtraction and either $d = 1$ or $d = 0$.

- The possibly complemented sum s now needs to be shifted left so that its leading digit is a 1. The value of the left shift distance d'' is determined as follows.

    - if $d \leq -2$ (product-anchored case or cancellation), let l be the number of leading zeros counted in the $2p + 3$ lower digits of s.

      ∗ If $e_a + e_b - l + 2 \geq e_{\min}$, the result will be normal, the left shift distance is $d'' = p + 2 + l$, and the exponent is set to $e'_r = e_a + e_b - l + 2$.

      ∗ If $e_a + e_b - l + 2 < e_{\min}$, the result will be a subnormal number, the exponent is set to $e'_r = e_{\min}$, the result significand will have $e_{\min} - (e_a + e_b - l + 2)$ leading zeros, and the shift distance is therefore only $d'' = p + 4 - e_{\min} + e_a + e_b$.

Note that this case covers the situation when either a or b is a subnormal.

  – if $d > 2$ (addend-anchored case), then $d'' = d'$: the left shift undoes the initial right shift. However, after this shift, the leading one may be one bit to the left (if effective addition) or one bit to the right (if effective subtraction), see the middle case of Figure 8.3. The large shift is therefore followed by a 1-bit normalization. The result exponent is accordingly set to one of $e'_r \in \{e_c, e_c - 1, e_c + 1\}$. If c was a subnormal, the left normalization/exponent decrement is prevented.

These shifts update the sticky bit and provide a normalized $(p+1)$-digit significand.

• Finally, this significand is rounded to p digits as per Section 8.2, using the rounding direction and the sticky bit. Overflow may also be handled at this point, provided a large enough exponent range has been used in all the exponent computations (two bits more than the exponent field width are enough).

One nice feature of this single-path algorithm is that subnormal handling comes almost for free.

8.6 Floating-Point Division

8.6.1 Overview and special cases

Given $x = (-1)^{s_x} \cdot |x|$ and $y = (-1)^{s_y} \cdot |y|$, we want to compute:

$$x/y = (-1)^{s_r} \cdot (|x|/|y|), \qquad s_r = s_x \text{ XOR } s_y \in \{0,1\}. \tag{8.9}$$

The IEEE 754-2008 specification for $|x|/|y|$ is summarized in Table 8.5 (see [187] and [198]); Combined with (8.2) and (8.9) it specifies floating-point division completely.

We now address the computation of the quotient of $|x| = m_x \cdot \beta^{e_x}$ and $|y| = m_y \cdot \beta^{e_y}$. We obtain

$$|x|/|y| = m_x/m_y \cdot \beta^{e_x - e_y}. \tag{8.10}$$

$\|x\|/\|y\|$		$\|y\|$			
		$+0$	(sub)normal	$+\infty$	NaN
$\|x\|$	$+0$	qNaN	$+0$	$+0$	qNaN
	(sub)normal	$+\infty$	$\circ(\|x\|/\|y\|)$	$+0$	qNaN
	$+\infty$	$+\infty$	$+\infty$	qNaN	qNaN
	NaN	qNaN	qNaN	qNaN	qNaN

Table 8.5: Special values for $\|x\|/\|y\|$.

In the following, we will assume that m_x and m_y are normal numbers, written $m_x = (m_{x,0}.m_{x,1} \ldots m_{x,p-1})_\beta$ and $m_y = (m_{y,0}.m_{y,1} \ldots m_{y,p-1})_\beta$, with $m_{x,0} \neq 0$ and $m_{y,0} \neq 0$. An implementation may first normalize both inputs (if they are subnormals or decimal numbers with leading zeros) using an extended exponent range. After this normalization, we have $m_x \in [1, \beta)$ and $m_y \in [1, \beta)$, therefore $m_x/m_y \in (\frac{1}{\beta}, \beta)$. This means that a 1-digit normalization may be needed before rounding.

8.6.2 Computing the significand quotient

There are three main families of division algorithms.

- Digit-recurrence algorithms, such as the family of SRT algorithms named after Sweeney, Robertson, and Tocher [344, 408], generalize the paper-and-pencil algorithm learned at school. They produce one digit of the result at each iteration. Each iteration performs three tasks (just like the pencil-and-paper method): determine the next quotient digit, multiply it by the divider, and subtract it from the current partial remainder to obtain a partial remainder for the next iteration.

  In binary, there are only two choices of quotient digits, 0 or 1; therefore, the iteration reduces to one subtraction, one test, and one shift. A binary digit-recurrence algorithm can therefore be implemented on any processor as soon as it is able to perform integer addition.

  Higher radix digit-recurrence algorithms have been designed for hardware implementation, and will be briefly reviewed in Section 9.6. Detailed descriptions of digit-recurrence division theory and implementations can be found in the books by Ercegovac and Lang [125, 126].

  One important thing about digit-recurrence algorithms is that they are exact. Starting from fixed-point numbers X and D, they compute at iteration i an i-digit quotient Q_i and a remainder R_i such that the identity $X = DQ_i + R_i$ holds. For floating-point purposes, this means that all the information needed for rounding the result is held in the pair (R_i, Q_i). In practice, to round to precision p, one needs p iterations to compute Q_p, then possibly a final addition on Q_p depending on a test on R_p.

- Functional iteration algorithms generalize Newton iteration for approximating the function $1/x$. They make sense mostly on processors having a hardware multiplier. The number of iterations is much less than in digit-recurrence algorithms ($O(\log p)$ versus $O(p)$), but each iteration involves multiplications and is therefore more expensive.

  Functional iterations are not exact; in particular, they start with an approximation of the inverse, and round their intermediate computations. Obtaining a correctly rounded result therefore requires some care. The last iteration needs to provide at least twice the target precision p, as a consequence of the exclusion lemma, see Lemma 15, Chapter 5, page 162. In Chapter 5, it has been shown that the FMA, which indeed internally computes on more than $2p$ digits, provides the required precision. However, AMD processors have used functional iteration algorithms to implement division without an FMA [307]. The iteration is implemented as a hardware algorithm that uses the full $2p$-bit result of the processor's multiplier before rounding. To accommodate double-extended precision ($p = 64$ bits) and cater to the error of the initial approximation and the rounding errors, they use a 76×76-bit multiplier [307].

- Polynomial approximation can also be used to evaluate $1/x$ to the required accuracy [430]. Note that, mathematically speaking, functional iterations evaluate a polynomial in the initial approximation error [88]. Both approaches may be combined; see [334] and [340, §9.5]. An example of the polynomial approach in a software context will be shown in Chapter 10.

Note that each division method has its specific way of obtaining the correctly rounded result.

8.6.3   Managing subnormal numbers

Subnormal inputs are best managed by first normalizing with a wider exponent range.

A subnormal result can be predicted from the exponent difference. As it will have less than p significand digits, it requires less accuracy than a standard computation. In a functional iteration, it suffices to round the high-precision intermediate result to the proper digit position.

In a digit-recurrence implementation, the simplest way to handle rounding to a subnormal number is to stop the iteration after the required number of digits has been produced, and then shift these digits right to their proper place. In this way, the rounding logic is the same as in the normal case.

8.6.4 The inexact exception

In general, the inexact exception is computed as a by-product of correct rounding. Directed rounding modes, as well as round to nearest even in the "even" case, require, respectively, exactness and half-ulp exactness detection. From another point of view, the inexact exception is straightforwardly deduced from the sticky bit. In digit-recurrence algorithms, for instance, exactness is deduced from a null remainder. Methods using polynomial approximations have to compute the remainder to round, and the inexact flag comes at no extra cost.

FMA-based functional iterations are slightly different in that they do not explicitly compute a sticky bit. However, they may be designed in such a way that the final FMA operation raises the inexact flag if and only if the quotient is inexact (see Section 5.3 or [270, page 115]).

8.6.5 Decimal specifics

For the division, the preferred exponent rule (see Section 3.4.7, page 97) mentions that for exact results, the preferred exponent is $Q(x) - Q(y)$.

8.7 Floating-Point Square Root

We end this chapter with the square root operation sqrt, which is often considered as the fifth basic arithmetic operation, after $+$, $-$, $\times$, and $\div$. Although it has similarities with division, square root is somewhat simpler to implement conformally with the IEEE 754 standards. In particular, it is univariate and, as we will recall, it never underflows, and overflows if and only if the input is $+\infty$.

8.7.1 Overview and special cases

If x is positive (sub)normal, then the correctly rounded value $\circ(\sqrt{x})$ must be returned. Otherwise, a special value must be returned conformally with Table 8.6 (see [187] and [197]).

Operand x	$+0$	$+\infty$	-0	less than zero	NaN
Result r	$+0$	$+\infty$	-0	qNaN	qNaN

Table 8.6: Special values for sqrt(x).

In the following, we will assume that m_x is a normal number, written $m_x = (m_{x,0}.m_{x,1} \ldots m_{x,p-1})_\beta$ with $m_{x,0} \neq 0$. An implementation may first normalize the input (if it is a subnormal or a decimal number having some leading zeros) using an extended exponent range.

After this normalization, we have a number of the form $m_x \cdot \beta^{e_x}$ with $m_x \in [1, \beta)$. Another normalization, by 1 digit, may be needed before taking the square root in order to make the exponent e_x even:

$$
m_x \cdot \beta^{e_x} = \begin{cases} m_x \cdot \beta^{e_x} & \text{if } e_x \text{ is even,} \\ (\beta \cdot m_x) \cdot \beta^{e_x - 1} & \text{if } e_x \text{ is odd.} \end{cases}
$$

Consequently, for $c \in \{0, 1\}$ depending on the parity of e_x, we have

$$
\sqrt{m_x \cdot \beta^{e_x}} = \sqrt{\beta^c \cdot m_x} \cdot \beta^{\frac{e_x - c}{2}},
$$

where $(e_x - c)/2 = \lfloor e_x/2 \rfloor$ is an integer. Since $\beta^c \cdot m_x \in [1, \beta^2)$, the significand square root satisfies

$$
\sqrt{\beta^c \cdot m_x} \in [1, \beta).
$$

8.7.2   Computing the significand square root

The families of algorithms most commonly used are exactly the same as for division, and a survey of these has been given by Montuschi and Mezzalama in [282].

- **Digit-recurrence algorithms.** Those techniques are extensively covered in [125] and [126, Chapter 6], with hardware implementations in mind. Here we simply note that the recurrence is typically a little more complicated than for division; see, for example, the software implementation of the restoring method that is described in Section 10.5.3, page 366.

- **Functional iteration algorithms.** Again, those methods generalize Newton iteration for approximating the positive real solution $y = \sqrt{x}$ of the equation $y^2 - x = 0$. As for division, such methods are often used when an FMA operator is available. This has been covered in Section 5.4.

- **Evaluation of polynomial approximations.** As for division, these methods consist in evaluating sufficiently accurately a "good enough" polynomial approximation of the function $\sqrt{x}$. Such techniques have been combined with functional iterations in [334] and [340, Section 11.2.3]. More recently, it has been shown in [196, 197] that, at least in some software implementation contexts, using exclusively polynomials (either univariate or bivariate) can be faster than a combination with a few steps of functional iterations. These approaches are described briefly in Section 10.5.3, page 369.

8.7.3 Managing subnormal numbers

As in division, a subnormal input is best managed by first normalizing with a wider exponent range.

Concerning output, the situation is much simpler than for division, since a subnormal result can never be produced. This useful fact is an immediate consequence of the following property.

Property 17. *For $x = m_x \cdot \beta^{e_x}$ a positive, finite floating-point number, the real $\sqrt{x}$ satisfies*

$$\sqrt{x} \in [\beta^{e_{\min}}, \beta^{\frac{1}{2}e_{\max}}).$$

Proof. The floating-point number x is positive, so $\beta^{1-p+e_{\min}} \leq x < \beta^{e_{\max}}$. Since the square root function is monotonically increasing,

$$\beta^{(1-p+e_{\min})/2} \leq \sqrt{x} < \beta^{e_{\max}/2},$$

and the upper bound follows immediately. Using $p \leq 1 - e_{\min}$ (which is a valid assumption for all the formats of [187]), we get further

$$\sqrt{x} \geq \beta^{e_{\min}}.$$

$\square$

This property also implies that

- the floating-point square root never underflows. It overflows if and only if the input is $+\infty$.

- The only exceptions to be considered are *invalid* and *inexact*.

8.7.4 The inexact exception

In digit-recurrence algorithms or polynomial-based approaches, exactness is deduced from a null remainder just as in division—the remainder here is $x - r^2$.

FMA-based functional iterations may be designed in such a way that the final FMA operation raises the inexact flag if and only if the square root is inexact [270, page 115].

8.7.5 Decimal specifics

For the square root, the preferred quantum exponent is $\lfloor Q(x)/2 \rfloor$.

Chapter 9

Hardware Implementation of Floating-Point Arithmetic

T̲HE PREVIOUS CHAPTER has shown that operations on floating-point numbers are naturally expressed in terms of integer or fixed-point operations on the significand and the exponent. For instance, to obtain the product of two floating-point numbers, one basically multiplies the significands and adds the exponents. However, obtaining the correct rounding of the result may require considerable design effort and the use of nonarithmetic primitives such as leading-zero counters and shifters. This chapter details the implementation of these algorithms in hardware, using digital logic.

Describing in full detail all the possible hardware implementations of the needed integer arithmetic primitives is beyond the scope of this book. The interested reader will find this information in the textbooks on the subject [224, 323, 126]. After an introduction to the context of hardware floating-point implementation in Section 9.1, we just review these primitives in Section 9.2, discuss their cost in terms of area and delay, and then focus on wiring them together in the rest of the chapter.

9.1 Introduction and Context

We assume in this chapter that inputs and outputs are encoded according to IEEE 754-2008, the IEEE Standard for Floating-Point Arithmetic.

9.1.1 Processor internal formats

Some systems, although compatible with the standard from a user point of view, may choose to use a different data format internally to improve performance. These choices are related to processor design issues that are out of the scope of this book. Here are a few examples.

- Many processors add *tag* bits to floating-point numbers. For instance, a bit telling if a number is subnormal saves having to detect it by checking that all the bits of the exponent field are zeros. This bit is set when an operand is loaded from memory, or by the arithmetic operator if the number is the result of a previous computation in the floating-point unit: each operator has to determine if its result is subnormal anyway, to round it properly. Other tags may indicate other special values such as zero, infinities, and NaNs. Such tags are stored in the register file of the processor along with the floating-point data, which may accordingly not be fully compliant with the standard. For instance, if there is a tag for zero, there is no need to set the data to the full string of zeros in this case.

- The fused multiply-add (FMA) of the IBM POWER6 has a short-circuit feedback path which sends results back to the input. On this path, the results are not fully normalized, which reduces the latency on dependent operations from 7 cycles to 6. They can be normalized as part of the first stage of the FMA.

- An internal data format using a redundant representation of the significands has been suggested in [134].

- Some AMD processors have a separate "denormalization unit" that formats subnormal results. This unit receives data in a nonstandard format from the other arithmetic units, which alone do not handle subnormals properly.

9.1.2   Hardware handling of subnormal numbers

In early processors, it was common to trap to software for the handling of subnormals. The cost could be several hundreds of cycles, which sometimes made the performance collapse each time subnormal numbers would appear in a computation. Conversely, most recent processors have fixed-latency operators that handle subnormals entirely in hardware. This improvement is partly due to very large-scale integration (VLSI): the overhead of managing subnormal numbers is becoming negligible with respect to the total area of a processor. In addition, several architectural improvements have also made the delay overhead acceptable.

An intermediate situation was to have the floating-point unit (FPU) take more cycles to process subnormal numbers than the standard case. The solution, already mentioned above, used in some AMD processors is a denormalizing unit that takes care of situations when the output is a subnormal number. The adder and multiplier produce a normalized result with a larger exponent. If this exponent is in the normal range, it is simply truncated to the standard exponent. Otherwise, that result is sent to the denormalizing

unit which, in a few cycles, will shift the significand to produce a subnormal number. This can be viewed as a kind of "trap," but one that is managed in hardware. An alternative approach, used in some IBM processors, saves the denormalizing unit by sending the number to be denormalized back to the shifter of the adder. The problem is then to manage conflicts with other operations that might be using this shifter.

The state of the art concerning subnormal handling in hardware is reviewed by Schwarz, Schmookler, and Trong [371]. They show that subnormal numbers can be managed with relatively little overhead, which explains why most recent FPUs in processors handle subnormal numbers in hardware. This is now even the case in graphics processing units (GPUs), the latest of which provide binary64 standard-compatible hardware. We will present, along with each operator, some of the techniques used. The interested reader is referred to [371] and references therein for more details and alternatives.

9.1.3 Full-custom VLSI versus reconfigurable circuits

Most floating-point architectures are implemented as full-custom VLSI in processors or GPUs. There has also been a lot of interest in the last decade in floating-point acceleration using reconfigurable hardware, with field-programmable gate arrays (FPGAs) replacing or complementing processors. The feasibility of floating-point arithmetic on FPGA was studied long before it became a practical possibility [378, 259, 261]. At the beginning of the century, several libraries of floating-point operators were published almost simultaneously (see [305, 247, 260, 345] among others). The increase of capacity of FPGAs soon meant that they could provide more floating-point computing power than a processor in single precision [305, 260, 345], then in double precision [441, 120, 109, 265, 178]. FPGAs also revived interest in hardware architectures for the elementary functions [119, 114, 113, 116] and other coarser or more exotic operators [442, 49, 105, 159].

We will survey floating-point implementations for both full-custom VLSI and FPGA. The performance metrics of these targets may be quite different (they will be reviewed in due course), and so will be the best implementation of a given operation.

By definition, floating-point implementation on an FPGA is application-specific. The FPGA is programmed as an "accelerator" for a given problem, and the arithmetic operators will be designed to match the requirements of the problem but no more. For instance, most FPGA implementations are parameterized by exponent and significand sizes, not being limited to those specified by the IEEE 754 standard. In addition, FPGA floating-point operators are designed with optional subnormal support, or no support at all. If tiny values appear often enough in an application to justify subnormal handling, the application can often be fixed at a much lower hardware cost by adding one bit to the exponent field. This issue is still controversial, and

subnormal handling is still needed for some applications, including those which require bit-exact results with respect to a reference software.

9.1.4   Hardware decimal arithmetic

Most of the research so far has focused on *binary* floating-point arithmetic. It is still an open question whether it is worth implementing decimal arithmetic in hardware [91, 130, 90, 413, 429], or if a software approach [85, 87], possibly with some minor hardware assistance, is more economical. This chapter covers both hardware binary and hardware decimal, but the space dedicated to hardware decimal reflects the current predominance of binary implementations.

   As exposed in detail in Section 3.4.3, the IEEE 754-2008 standard specifies two encodings of decimal numbers, corresponding to the two main competing implementations of decimal arithmetic at the time IEEE 754-2008 was designed. The *binary* encoding allows for efficient software operations, using the native binary integer operations of a processor. It is probable that processor instruction sets will be enriched to offer hardware assistance to this software approach. The *decimal* encoding, also known as *densely packed decimal* or *DPD*, was designed to make a hardware implementation of decimal floating-point arithmetic as efficient as possible.

   In the DPD format, the significand is encoded as a vector of radix-1000 digits, each encoded in 10 bits (declets). This encoding is summarized in Tables 3.18, 3.19, and 3.20, pages 88 and 89. It was designed to facilitate the conversions: all these tables have a straightforward hardware implementation, and can be implemented in three gate levels [123]. Although decimal numbers are stored in memory in the DPD format, hardware decimal arithmetic operators internally use the simpler binary coded decimal (BCD) representation, where each decimal digit is straightforwardly encoded as a 4-bit number. Of course, the main advantage of this encoding is that all the declets of a number can be processed in parallel.

   Considering that the operations themselves use the BCD encoding, should the internal registers of a processor use DPD or BCD? On one hand, having the registers in BCD format saves the time and power of converting the input operands of an operation to BCD, then converting the result back to DPD—to be converted again to BCD in a subsequent operation. On the other hand, as pointed out by Eisen et al. [123], it is unlikely that an application will intensively use binary floating-point and decimal floating-point at the same time; therefore, it makes sense to have a single register file. The latter will contain 64-bit or 128-bit registers, which is a strong case for accepting DPD numbers as inputs to a decimal FPU.

9.1.5 Pipelining

Most floating-point operator designs are pipelined. There are three characteristics of a pipelined design:

- its frequency, which is the inverse of the cycle time;

- its depth or latency, which is the number of cycles it takes to obtain the result after the inputs have been presented to the operator;

- its silicon area.

IEEE 754-compliant operations are combinatorial (memory-less) functions of their inputs. Therefore, one may in principle design a combinational (unpipelined) operator of critical path delay T, then insert n register levels to convert it into a pipelined one of latency n. If the logical depth between each register is well balanced along the critical path, the cycle time will be close to T/n. It will always be larger due to the delay incurred by the additional registers. These additional registers also add to the area. This defines a technology-dependent tradeoff between a deeply pipelined design and a shallower one [383]. For instance, recent studies suggest an optimal delay of 6 to 8 fanout-of-4 (FO4) inverter delays per stage if only performance is considered [185], but almost double if power is taken into consideration [447].

Besides, some computations will not benefit from deeper pipelines. An addition using the result of another addition must wait for this result. This is called a *data dependency*. The deeper the pipeline, the more cycles during which the processor must wait. If there are other operations that can be launched during these cycles, the pipeline will be fully utilized, but if the only possible next operation is the dependent operation, the processor will be idle. In this case, the pipeline is not used to its full capacity, and this inefficiency is worse for a deeper pipeline. Very frequently, due to data dependencies, a deeper pipeline will be less efficient, in terms of operations per cycle, than a shallower one. This was illustrated by the transition from the P6 microarchitecture (used among others in the Pentium III) to the NetBurst microarchitecture (used among others in the Pentium 4). The latter had roughly twice as deep a pipeline as the former, and its frequency could be almost twice as high, but it was not uncommon that a given piece of floating-point code would need almost twice as many cycles to execute [100], negating the performance advantage of the deeper pipeline. Considering this, recent Intel microprocessors (most notably the Core microarchitecture) have stepped back to shallower pipelines, which have additional advantages in terms of power consumption and design complexity.

In the design of a processor, the target frequency is decided early, which imposes a limit on the logic depth delay in each pipeline level. One then tries to keep the number of cycles as close as possible to the theoretical optimal (obtained by dividing the critical path delay by the cycle time).

In general-purpose processors, the area of the arithmetic operators is not as much of a problem. Thanks to Moore's law, transistors are cheap, and the floating-point units account for a few percent of the area of a processor anyway. It is not uncommon to replicate computations if this decreases the critical path. In the following pages, we will see many examples.

In GPUs or reconfigurable circuits, the context is slightly different. The applications being targeted to GPUs and FPGAs will expose enough parallelism so that the issue of operation latency will not be significant: the cycles between dependent operations will be filled with other computations. On the other hand, one wants as many operators as possible on a chip to maximally exploit this parallelism. Therefore, floating-point operators on GPUs and FPGAs will favor small area over small latency.

9.2   The Primitives and Their Cost

Let us first review the primitives which will be required for building most operations. These include integer arithmetic operations (addition/subtraction, multiplication, integer division with remainder) but also primitives used for significand alignment and normalization (shifters and leading-zero counters). Small tables of precomputed values may also be used to accelerate some computations.

These primitives may be implemented in many different ways, usually exposing a range of tradeoffs between speed, area, and power consumption. The purpose of this section is to expose these tradeoffs.

9.2.1   Integer adders

Integer addition deserves special treatment because it appears in virtually all floating-point operators, though with very different requirements. For instance, the significand addition in a floating-point adder needs to be as fast as possible, while operating on numbers of fairly large width. In contrast, exponent addition in a floating-point multiplier operates on smaller integers, and does not even need to be fast, since it can be performed in parallel with the significand product. As a third example, integer multiplication can be expressed in terms of iterated additions, and adders may be designed specifically for this context.

Fortunately, a wide range of adder architectures has been proposed and allows us to address each of these needs. We present each adder family succinctly. The interested reader will find details in the bibliographic references.

Carry-ripple adders

The simplest adder is the *carry-ripple adder*, represented in Figure 9.1 for binary inputs and in Figure 9.2 for decimal BCD inputs. Carry-ripple addition
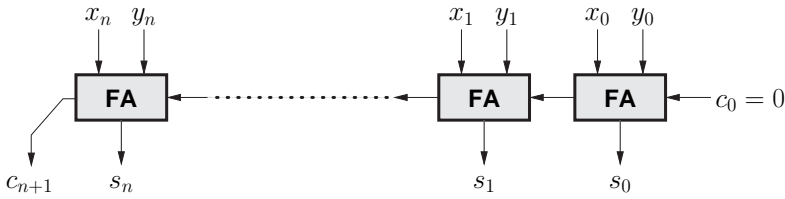
Figure 9.1: Carry-ripple adder.
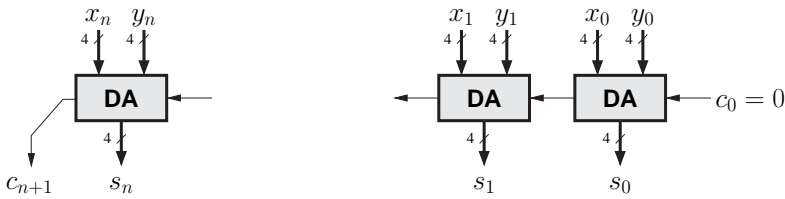
Figure 9.2: Decimal addition. Each decimal digit is coded in BCD by 4 bits.

is simply the paper-and-pencil algorithm learned at school. A carry-ripple adder has $O(n)$ area and $O(n)$ delay, where n is the size in digits of the numbers to be added.

The building block of the binary carry-ripple adder is the *full adder* (FA), which outputs the sum of three input bits x_i, y_i, and z_i (this sum is between 0 and 3) as a 2-bit binary number $c_i s_i$. Formally, it implements the equation $2c_i + s_i = x_i + y_i + z_i$. The full adder can be implemented in many ways with a two-gate delay [126]. Typically, one wants to minimize the delay on the carry propagation path (horizontal in Figure 9.1). Much research has been dedicated to implementing full adders in transistors; see, for instance, Zimmermann [443] for a review. A clean CMOS implementation requires 28 transistors, but many designs have been suggested with as few as 10 transistors (see [440, 1, 376, 61] among others). These transistor counts are given for illustration only: smaller designs have limitations, for instance they cannot be used for building carry-ripple adders of arbitrary sizes. The best choice of a full-adder implementation depends much on the context in which it is used.

Carry-ripple adders can be built in any radix β (take $\beta = 10$ for illustration). The basic block DA (for digit addition) now computes the sum of an input carry (0 or 1) and two radix-β digits (between 0 and $\beta - 1$). This sum is between 0 and $2\beta - 1$ and can therefore be written in radix β as $c_i s_i$, where c_i is an output carry (0 or 1) and s_i is a radix-β digit.

Useful radices for building hardware floating-point operators are 2, small powers of 2 (in which case the DA block is simply a binary adder as shown on Figure 9.4), 10, and small powers of 10. Figure 9.2 gives the example

Figure 9.3: An implementation of the decimal DA box.

Figure 9.4: An implementation of the radix-16 DA box.

of a radix-10 ripple-carry adder. The implementation of a typical decimal DA block is depicted by Figure 9.3. It first computes $x_i + y_i + z_i$ using a binary 4-bit adder. To detect if the sum is larger than 10, a second 5-bit adder adds 6 to this sum. The carry out of this adder is always 0 (the sum is at most $9 + 9 + 1 + 6 = 25 < 32$) and can be ignored. The bit of weight 2^4 is the decimal carry: it is equal to 1 iff $x_i + y_i + z_i + 6 \geq 16$, i.e., $x_i + y_i + z_i \geq 10$. The sum digit is selected according to this carry. Many low-level optimizations can be applied to this figure, particularly in the constant addition. However, comparing Figures 9.4 and 9.3 illustrates the intrinsic hardware overhead of decimal arithmetic over binary arithmetic.

Many optimizations can also be applied to a radix-2^p adder, such as the one depicted in Figure 9.4, particularly to speed up the carry-propagation path from c_{in} to c_{out}. In a carry-skip adder [126], each radix-2^p DA box first computes, from its x_i and y_i digit inputs (independently of its c_i input, therefore in parallel for all the digits), two signals g_i (for generate) and p_i (for propagate), such that $c_{i+1} = g_i$ OR (p_i AND c_i). Carry propagation then takes only 2 gate delays per radix-2^p digit instead of $2p$ gate delays when the DA box is implemented as per Figure 9.4. The drawback is that the DA box is

Figure 9.5: Binary carry-save addition.

now larger: this family of adders exposes a tradeoff between area and adder delay.

Parallel adders

The hardware of a carry-ripple adder can be used to perform *carry-save* addition in $O(n)$ area and $O(1)$ time, as shown in Figure 9.5. The catch is that the result is obtained in a nonstandard redundant number format: Figure 9.5 shows a carry-save adder that adds three binary numbers and returns the sum as two binary numbers: $R = \sum_{i=0}^{n+1}(c_i + s_i)2^i$.

The cost of converting this result to standard representation is a carry propagation, so this is only useful in situations when many numbers have to be added together. This is the case, e.g., for multiplication.

As shown in Figure 9.6 for radix 2^{32}, this idea works for any radix. A radix-β carry-save number is a vector of pairs (c_i, s_i), where c_i is a bit and s_i a radix-β digit, representing the value $\sum_{i=0}^{n}(c_i + s_i)\beta^i$. A radix-$\beta$ carry-save adder adds one radix-β carry-save number and one radix-β number and returns the sum as a radix-β carry-save number.

Radix-2^p carry-save representation is also called *partial carry save*. It allows for the implementation of very large adders [439] or accumulators [234, 105] working at high frequency in a pipelined way. It has also been proposed that its use as the internal format for the significands inside a processor floating-point unit would reduce the latency of most operations [134]. Conversions between the IEEE 754-2008 interchange formats presented in Section 3.4.1 and this internal format would be performed only when a number is read from or written to memory.

Fast adders

It is usual to call *fast adders* adders that take their inputs and return their results in standard binary representation, and compute addition in $O(\log n)$ time instead of $O(n)$ for carry-ripple addition. They require more area ($O(n \log n)$ instead of $O(n)$). In this case, the constants hidden behind the O notation are important.
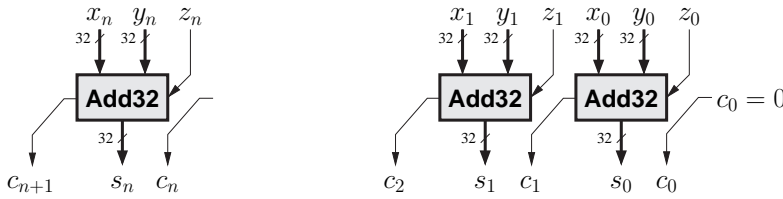
Figure 9.6: Partial carry-save addition.

Here is a primer on *prefix tree adders* [126], a very flexible family of fast adders. For any block $i : j$ of consecutive bits of the addition with $i \geq j$, one may define a *propagate* signal $P_{i:j}$ and a *generate* signal $G_{i:j}$. These signals are defined from the input bits of the range $i : j$ only, and have the following meanings: $P_{i:j} = 1$ iff the block will propagate its input carry, whatever its value; $G_{i:j} = 1$ iff the block will generate an output carry, regardless of its input carry. For a 1-bit block we have $G_{i:i} = a_i$ AND b_i and $P_{i:i} = a_i$ XOR b_i. The key observation is that generate and propagate signals can be built in an associative way: if $i \geq j \geq k$, we have

$$(G_{i:k}, P_{i:k}) \;=\; (G_{i:j} \text{ OR } (P_{i:j} \text{ AND } G_{j:k}),\ P_{i:j} \text{ AND } P_{j:k}),$$

which is usually noted $(G_{i:k}, P_{i:k}) = (G_{i:j}, P_{i:j}) \bullet (G_{j:k}, P_{j:k})$. The operator $\bullet$ is associative: for any value of the bits $a_1, b_1, a_2, b_2, a_3, b_3$, we have

$$((a_1, b_1) \bullet (a_2, b_2)) \bullet (a_3, b_3) = (a_1, b_1) \bullet ((a_2, b_2) \bullet (a_3, b_3)) .$$

Because of this associativity, it is possible to compute in parallel in logarithmic time all the $(G_{i:0}, P_{i:0})$, using a parallel-prefix tree. The sum bits are then computed in time $O(1)$ as $s_i = a_i$ XOR b_i XOR $G_{i:0}$.

The family of prefix tree adders has the advantage of exposing a wide range of tradeoffs, with delay between $O(\log n)$ and $O(n)$ and area between $O(n \log n)$ and $O(n)$. In addition, the tradeoff can be expressed in other terms such as fan-out (the number of inputs connected to the output of a gate) or in terms of wire length. With submicrometer VLSI technologies, the delay and power consumption are increasingly related to wires. See [326] for a recent survey.

Another advantage of prefix adders is that computing $A + B + 1$ on top of the computation of $A + B$ comes at the price of $O(n)$ additional hardware and $O(1)$ additional delay. Once the $(G_{i:0}, P_{i:0})$ are available, the bits of $A + B + 1$ are defined by setting the input carry to 1 as follows: $s'_i = a_i$ XOR b_i XOR $(G_{i:0}$ OR $P_{i:0})$ [411]. A fast adder designed to compute both $A + B$ and $A + B + 1$ is called a *compound adder* [375]. It is useful for floating-point operators, typically because the rounded significand is either a sum or the successor of this sum.
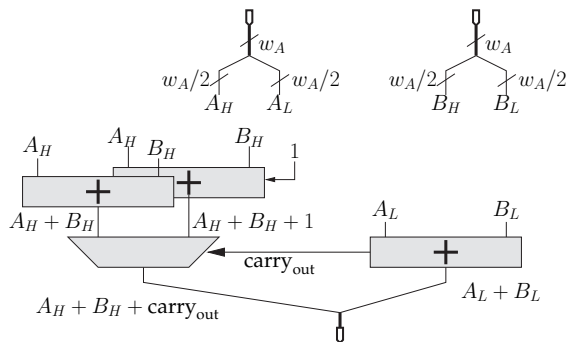
Figure 9.7: Carry-select adder.

A variation of the compound adder computes $|A - B|$ when A and B are two positive numbers (e.g., significands). In that case, using two's complement, one needs to compute either $A - B = A + \overline{B} + 1$ or $B - A = \overline{A + \overline{B}}$, depending on the most significant bit s_{n-1} of $A - B$ which is set if $A - B$ is negative (see Figure 9.12). This approach is sometimes referred to as an *end-around carry adder*.

To summarize, the design of an adder for a given context typically uses a mixture of all the techniques presented above. See [439] for a recent example.

Fast addition in FPGAs

In reconfigurable circuits, the area is measured in terms of the elementary operation, which is typically an arbitrary function of 4 to 6 inputs implemented as a look-up table (LUT). As the routing is programmable, it actually accounts for most of the delay: in a circuit, you just have a wire, but in an FPGA you have a succession of wire segments and programmable switches [99]. However, all current FPGAs also provide *fast-carry* circuitry. This is simply a direct connection of each LUT to one of its neighbors which allows carry propagation to skip the slow generic routing.

The net consequence is that fast (logarithmic) adders are irrelevant to FPGAs. For illustration, a high-end FPGA in 2009 has a typical peak frequency of 500MHz (corresponding roughly to the traversal of a few LUTs and a few programmable wires). In this cycle time, one may also perform a 30-bit carry-propagation addition using the fast-carry logic. Using that 30-bit carry propagation, the simple carry-select adder shown in Figure 9.7 provides 60-bit addition at the peak frequency, which is enough for binary64. Additions larger than 30 bits may also be implemented by adding very few pipeline levels in the fast-carry propagation.

9.2.2   Digit-by-integer multiplication in hardware

The techniques learned at school for paper-and-pencil multiplication and division require the computation of the product of a digit by a number. This operation also occurs in hardware versions of these algorithms.

Multiplying an n-bit number by a bit simply takes n AND gates operating in parallel in $O(1)$ time. However, as soon as the radix β is larger than two, the product of two digits in $\{0, ..., \beta - 1\}$ does not fit on a single digit, but on two. Computing the product of a digit by a number then takes a carry propagation, and time is now $O(n)$, still for $O(n)$ area with the simplest operators (the previously seen methods for fast addition can readily be adapted). Similarly to addition, if it is acceptable to obtain the product in nonstandard redundant format, this operation can be performed in $O(1)$ time.

For small radices, digit-by-integer multiplication in binary resumes to a few additions and constant shifts. For instance, if the radix is a power of 2, $2X = X << 1$ (X shifted left by one bit), $3X = X + (X << 1)$, etc. All the multiplications up to $10X$ can be implemented by a single addition/subtraction and constant shifts. These additions require a carry propagation. We now see a general method, *recoding*, that can avoid these carry propagations.

9.2.3   Using nonstandard representations of numbers

Note that the radix here is not necessarily 2 or 10. For instance, considering a binary number as a radix-4 number simply means considering its digits two by two. An operation expressed in radix 4 will have fewer digit operations than the same operation expressed in radix 2. However, each digit operation will be more complex. The choice of a larger radix may allow one to reduce the number of cycles for an operation while maximizing the amount of computation done within one cycle. For instance, Intel recently reduced the latency of division in their x86 processors by replacing the radix-4 division algorithm with a radix-16 algorithm [287].

The digit set to be used in radix β is not necessarily $\{0, ..., \beta - 1\}$. Any set of at least β consecutive digits including 0 allows for representation of any number. Furthermore, it is common to use a *redundant* digit set, i.e., a digit set with more than β digits. The value of a string of p digits $(d_{p-1}d_{p-2}\cdots d_0)$ in radix β is still

$$N = \sum_{i=0}^{p-1} d_i \beta^i \quad .$$

In such a system, some numbers will have more than one representation.

A digit set including *negative* digits makes it possible to represent negative numbers (these redundant number systems with negative digits are sometimes called *signed-digit* number systems).

Example 11 (Radix 10 with digit set {-5,-4,-3,-2,-1,0,1,2,3,4,5}). *Let us note the negative digits with an overline, e.g., $\bar{5} = -5$. Here are some examples of number representations in this system:*

$$
\begin{aligned}
0 &= 0 \\
9 &= 1\bar{1} \quad (= 10 - 1) \\
-8 &= \bar{1}2 \quad (= -10 + 2) \\
17 &= 2\bar{3} \\
5 &= 05 = 1\bar{5} \quad (= -10 + 2) \quad .
\end{aligned}
$$

The main point of redundant digit sets is that they provide more algorithmic freedom. For instance, carry propagation can be prevented by choosing, among two possible representations of the sum, the one that will be able to consume an incoming carry. This is the trick behind Avizienis' signed-digit addition algorithm [16]. Another example where this freedom of choice is useful is SRT division, introduced in Section 9.6.

Nonstandard digit sets also have other advantages. A trick commonly used to speed up binary multiplication is *modified Booth recoding* [37]. It consists in rewriting one of the operands of the multiplication as a radix-4 number using the digit set $\{\bar{2}, \bar{1}, 0, 1, 2\}$. The advantage is that multiplication of the other operand by any of these recoded digits still consists of a row of AND as in binary, and a possible shift. As the digits are in radix 4, they are twice as few. Compared to using standard radix 4 with digits in $\{0, 1, 2, 3\}$, we no longer have the carry propagation that was needed to compute $3X = X + 2X$.

Booth recoding can be done in $O(1)$ time using $O(n)$ area. In a first step, the initial radix-4 digits in $\{0, 1, 2, 3\}$ are rewritten in parallel as follows: 0 and 1 are untouched, 2 is rewritten $4 + \bar{2}$, 3 is rewritten $4 + \bar{1}$. Here the 4 corresponds to a carry in radix 4, sent one digit position to the left. In a second step, these carries are added, again in parallel, to the digits from the previous step, which belonged to $\{\bar{2}, \bar{1}, 0, 1\}$. The resulting digits indeed belong to $\{\bar{2}, \bar{1}, 0, 1, 2\}$. Again, thanks to redundancy, there was no carry propagation.

Recoding can also be used for decimal multiplication and will be surveyed in Section 9.2.5.

9.2.4   Binary integer multiplication

There are many ways of implementing integer multiplication with a range of area-time tradeoffs [126]. We focus here on high-performance implementations. They are typically performed in three steps, illustrated in Figure 9.8.

- First, one of the operands is Booth recoded, and each of its digits is multiplied by the other operand. This results in a *partial product array* of $(n + 1) \times n/2$ weighted bits. The result of the multiplication will be the sum of these weighted bits. This step can be performed in constant time.
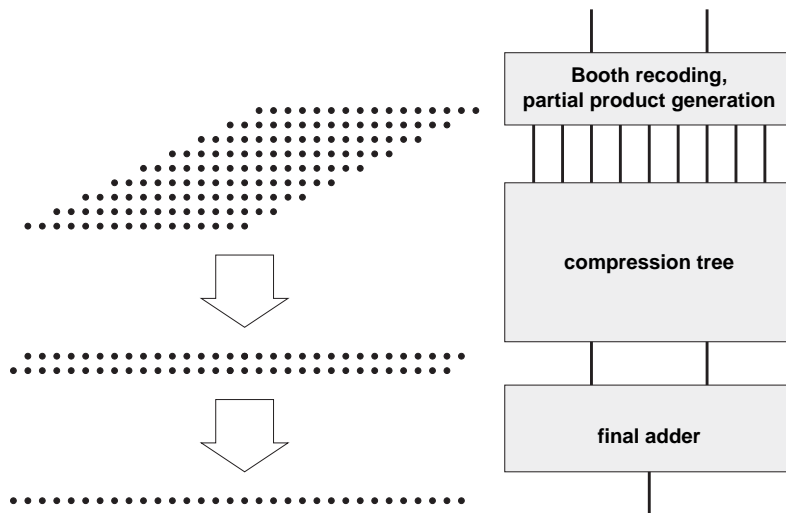
Figure 9.8: Binary integer multiplication.

- A second step reduces this array to only two lines using several carry-save adders, or more generally, *compressors*. For instance, the 3:2 compressor is the carry-save adder of Figure 9.5, a 4:2 compressor takes 4 binary numbers and writes their sum as two binary numbers (i.e., as a carry-save number). A 4:2 compressor can be implemented as two carry-save adders, but it may also be implemented more efficiently (using fewer transistors). It has been argued that a good 4:2 compressor implementation may perform the same function as Booth recoding in less time using less resources [420].

  This step can be performed in $O(\log n)$ time, using a tree of compressors.

- Finally, the carry-save result of the previous step is summed using a fast adder in $O(\log n)$ time.

This multiplier scheme is quite flexible. It easily accommodates signed integers at no extra cost. More important for floating-point, rounding can be performed at almost no cost by adding a few bits in the partial product array, as detailed in Section 9.4. Finally, computing a multiply-and-add $a \times b + c$ adds only one more line of bits (corresponding to c) to the initial partial product array of $a \times b$ depicted in Figure 9.8. In practice, the requirement of correct rounding of the FMA makes the overall data path much more complex; see Section 9.5.

```
                3 8 2
        ×       9 7 6
        ─────────────
                8 8 2
              1 4 1
              1 6 4
            2 5 1
              7 2 8
          2 7 1
        ─────────────
        3 7 2 8 3 2
```

Figure 9.9: Partial product array for decimal multiplication. One digit product, $9 \times 2 = 18$, is highlighted.

9.2.5 Decimal integer multiplication

The basic scheme presented above can be used for decimal multiplication. The design of a carry-save decimal adder is straightforward. The partial product array may be built by computing all the digit-by-digit products in parallel, as illustrated by Figure 9.9. Notice that this figure is different from the one obtained using the paper-and-pencil algorithm, which needs a carry propagation to compute the product of one number by one digit.

Here also, recoding can help in several ways. Recoding the decimal digits to $d_l \in \{-5, ..., 5\}$, for instance, and handling signs separately, reduces the number of digit-by-digit products to compute from 100 to 36, with a corresponding reduction in the complexity of the digit-by-digit multiplier [131]. Interestingly enough, Cauchy already had that idea (of course, in a different context!) in 1840 [65]. In [240], each decimal digit is recoded as the sum of two digits, $d_h + d_\ell$, with $d_h \in \{0, 5, 10\}$ and $d_\ell \in \{-2, -1, 0, 1, 2\}$. Multiplying a decimal input by such a recoded digit still leads to two lines in the partial product array, as in Figure 9.9, but the computation of these two lines requires very little hardware.

- Multiplication by 0, ± 1, and ± 10 is almost "for free."

- Multiplication by 2 resumes to a shift and a 4-bit addition. A carry may be produced to the next digit position, but, added to an even digit, it will not propagate further.

- Multiplication by 5 is performed as multiplication by 10 (digit shift) then division by 2. The latter is performed digit-wise. For odd digits, there is a fractional part equal to 0.5. It is sent as a $+5$ to the digit position to the right. Similarly to the previous case, this "anti-carry" may not propagate any further.

In [414], the standard BCD code is named BCD-8421 (indicating the weights of the 4 bits of a digit), and different codings of decimal digits are suggested: BCD-4221 or BCD-5211. The advantage of these codes is that they are slightly redundant, which in practice enables faster decimal carry-save addition. More alternatives can be found in Vásquez [413].

Other recent variations on decimal multiplication may be found in [129, 429]. Earlier decimal architectures were more sequential, more or less like the paper-and-pencil algorithm [63, 128]. In this case, a first step may be to compute multiples of the multiplicand and store them in registers.

In [298], Neto and Véstias implement decimal multiplication in FPGAs using a radix-1000 binary format. This is motivated by the availability of embedded multipliers able to perform the product of two 10-bit numbers and return the exact product. They also propose addition-based radix conversion algorithms that exploit the fast addition fabric of FPGAs.

9.2.6   Shifters

Normalizing a binary floating-point result means bringing its leading "1" to the first position of the significand. If this "1" was preceded by a string of zeros, an idea is to count these zeros first, and then feed this count to a shifter. Note that a simple way of handling subnormal inputs is to normalize them to an internal format with a few more exponent bits. This concerns practically all operations that need to manage subnormal inputs.

A hardware shifter (commonly called a *barrel shifter*) takes one input x of size n (the number to be shifted) and one input d of size $\lceil \log_2 n \rceil$ (the shift distance). It consists of $\lceil \log_2 n \rceil$ stages. The i-th stage considers the i-th bit of d, say d_i, shifts by 2^i if $d_i = 1$ and does nothing otherwise, using a 2:1 multiplexer. Ignoring fan-in and fan-out issues, such a multiplexer for a data of n bits has area $O(n)$ and delay $O(1)$. Therefore, the area of a complete barrel shifter is $O(n \log n)$, and its delay is $O(\log n)$.

Shifting to a constant distance is cheaper, as it reduces to wires and possibly a multiplexer, in $O(n)$ area and $O(1)$ time.

9.2.7   Leading-zero counters

If an intermediate result may produce an arbitrary number of leading zeros (a typical case is a cancellating subtraction), it is necessary to count these leading zeros. A leading-zero counter (LZC) provides the shift value that will allow one to normalize the result.

Of course, one may derive from an LZC an architecture that counts leading ones. A leading-zero-or-one counter (LZOC) may also be useful: if an operation may return a negative number and the latter is available in two's complement, it will have to be complemented and normalized. In this case,

one may perform the leading-zero counting in parallel with the complementing: what needs to be counted is the numbers of bits identical to the most significant bit (MSB) of the two's complement value.

Tree-based leading-zero counter

In general, counting the leading zeros in a number of size n can be performed in a binary tree manner in $O(\log n)$ time and $O(n)$ area. The basic algorithm is the following [316]. The first level considers pairs of adjacent bits and associates to each pair a 2-bit number that counts the leading zeros in this pair. The second level considers groups of 4 consecutive bits and builds 3-bit numbers that count the leading zeros in each group, and so on. Each computing node at level i considers groups of 2^i consecutive bits and outputs the leading-zero count for such a group as an $(i+1)$-bit number. A node is able to to build this count using simple multiplexers out of the counts of the previous level, because of the following observation: the MSB of such a count is 1 iff the group consists of only zeros. Algorithm 9.1 makes explicit the operation of a node.

Algorithm 9.1 One node of the i-th level of an LZC tree.

   Inputs: two i-bits numbers L and R
   $l \leftarrow$ most significant bit of L
   $r \leftarrow$ most significant bit of R
   if $l = 0$ (there is a 1 on the left group) **then**
      $D \leftarrow 0L$ (ignore the right group)
   else if $l = 1$ and $d = 1$ (both groups are all zeros) **then**
      $D \leftarrow 2^{i+1}$ (written 10...0)
   else if $l = 1$ and $d = 0$ (all zeros on the left) **then**
      $D \leftarrow 1R$ (add 2^i to the count of the right group)
   end if
   Returns: the $(i+1)$-bit number D;

    Many variations of this algorithm, e.g., using coarser levels, can be designed to match the constraints or requirements of a given operator.

Leading-zero counting by monotonic string conversion

A completely different approach to leading-zero counting consists in first converting the input into a monotonic string, i.e., a bit string of the same size, which has the same leading zeros as the input, but only ones after the leading one. This is a very simple prefix-OR computation that may be performed efficiently in hardware in several ways [364]. By ANDing this string with itself negated and shifted by one bit position, one gets a string $S = s_{n-1}...s_1 s_0$ consisting of only zeros, except at the position of the leading one. From this

last string, each bit of the count R can be computed as an $n/2$-wide OR. This approach is also well suited to *leading-zero anticipation* techniques, which will be presented in Section 9.3.3.

Combined leading-zero counting and shifting for FPGAs

Algorithm 9.2 combines leading-zero counting and shifting, which is what is needed for floating-point normalization.

Algorithm 9.2 Combined leading-zero counting and shifting.

$k \leftarrow \lceil \log_2 n \rceil$;
$x_k \leftarrow x$;
for $i = k - 1$ downto 0 **do**
    if there are 2^i leading zeros in x_{i+1} **then**
        $d_i \leftarrow 1$
        $x_i \leftarrow x_{i+1}$, shifted left by 2^i;
    else
        $d_i \leftarrow 0$
        $x_i \leftarrow x_{i+1}$;
    end if
end for
return (d, x_0);

The theoretical delay of this algorithm is worse than that of an LZC followed by a shifter (both in $O(\log n)$. Indeed, the test

$$\textbf{if } \text{there are } 2^i \text{ leading zeros in } x_{i+1}$$

is itself an expensive operation: implemented as a binary tree, it costs $O(i)$ delay. The total delay would therefore be $O(k^2) = O((\log n)^2)$.

However, in recent FPGAs, the dedicated hardware accelerating carry-propagation addition can also be used to compute the required wide OR operation. Thus, the practical delay of leading-zero counting in algorithm 9.2 is completely hidden in that of the shifter for input sizes up to roughly 30 bits. If the input is larger, pipeline levels may have to be inserted to reach the peak FPGA frequency, but this only concerns the first few stages. As this algorithm also leads to a compact implementation, it will be preferred over an LZC followed by a shifter.

9.2.8 Tables and table-based methods for fixed-point function approximation

There are several situations where an approximation to a function must be retrieved from a table. In particular, division and square root by functional

iteration require an initial approximation to the reciprocal, square root, or reciprocal square root of a significand [322, 360, 361, 362, 394, 229]. Many other applications exist, especially in the field of reconfigurable computing. For instance, table-based architectures for floating-point elementary functions are presented in [114] (exponential and logarithm), [113] (sine and cosine), and [121] (power function). Multiplication by a constant may also resort to precomputed tables, as will be detailed in Section 9.7.2. All these tables make sense in an architecture targeting FPGAs, because these circuits include a huge number of memory elements (from the LUT to larger configurable RAMs).

Plain tables

A table with an n-bit address may hold 2^n values. If each of these values is stored as m bits, the total number of bits stored in the table is $m \cdot 2^n$. Such a table is typically implemented in VLSI technology as a ROM (read-only memory) of area $O(m \cdot 2^n)$. The delay is that of the address decoding, in $O(n)$. Some tables, for instance with very small n or very redundant content, may be best implemented directly as Boolean circuits.

In an FPGA, a LUT-based table will have a LUT cost of $O(m \cdot 2^{n-k})$ (where k is the number of inputs to the elementary LUT, currently from 4 to 6). However, a table may also be implemented using one or several configurable RAM blocks—for more details on the RAM blocks available in a given FPGA circuit, refer to vendor documentation.

Table-based methods

If the function to be tabulated is continuous and derivable with well-bounded derivatives on the interval of interest (this is the case of the reciprocal on $[1, 2)$, for instance), it is possible to derive an architecture that may replace a table of the function, at a much lower hardware cost. Such an architecture is based on a piecewise linear approximation to the function, where both the constant term and the product are tabulated in two smaller tables (typically each with $2n/3$ address bit). The resulting *bipartite* architecture consists of an addition that sums the output of the two tables.

To our knowledge, this idea was first published for the sine function [398], then rediscovered independently for the reciprocal function [361]. It was later generalized to arbitrary functions and improved gradually to more than two tables [394, 291, 106]. The generalized multipartite method in [106] builds an architecture with several tables and several additions which guarantees faithful rounding of the result while optimizing a given cost function. For instance, consider a 16-bit approximation to the reciprocal on $[1, 2)$, with value in $(\frac{1}{2}, 1]$. As there is one constant input bit and one constant output bit, the function to tabulate is actually $f(x) = \frac{2}{x+1} - 1$. A 15 bits in, 15 bits
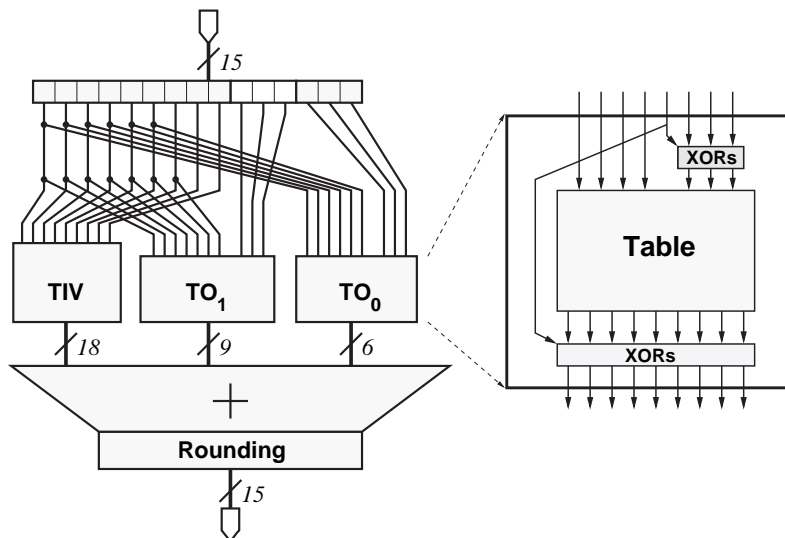
Figure 9.10: A multipartite table architecture for the initial approximation of $1/x$. The XORs implement changes of sign to exploit the symmetry of a linear approximation segment with respect to its center [394].

out faithful multipartite architecture for such a function is presented in Figure 9.10. It consists of three tables of 18.2^9, 9.2^9, and 6.2^8 bits respectively, and two additions. This represents 30 times less storage than using a plain table with 15-bit addresses. Moreover, as the three tables are accessed in parallel, the delay of a multipartite architecture is very small—in an FPGA, it is smaller than the delay of the plain table, because the smaller tables are also accessed faster, which more than compensates the delay due to the adders.

The multipartite method was then generalized to higher degree approximations with architectures involving small multipliers [112], further reducing the area at the expense of the delay.

9.3   Binary Floating-Point Addition

A binary floating-point adder basically follows the generic algorithm given in Section 8.3. We focus here on the implementation issues.

9.3.1   Overview

Let us call x and y the floating-point inputs. First, the exponent difference $e_x - e_y$ is computed, and the inputs are possibly swapped to ensure that $e_y \leq e_x$. Then comes a possibly large shift to align the significands. In case of effective subtraction, one of the significands is possibly complemented.
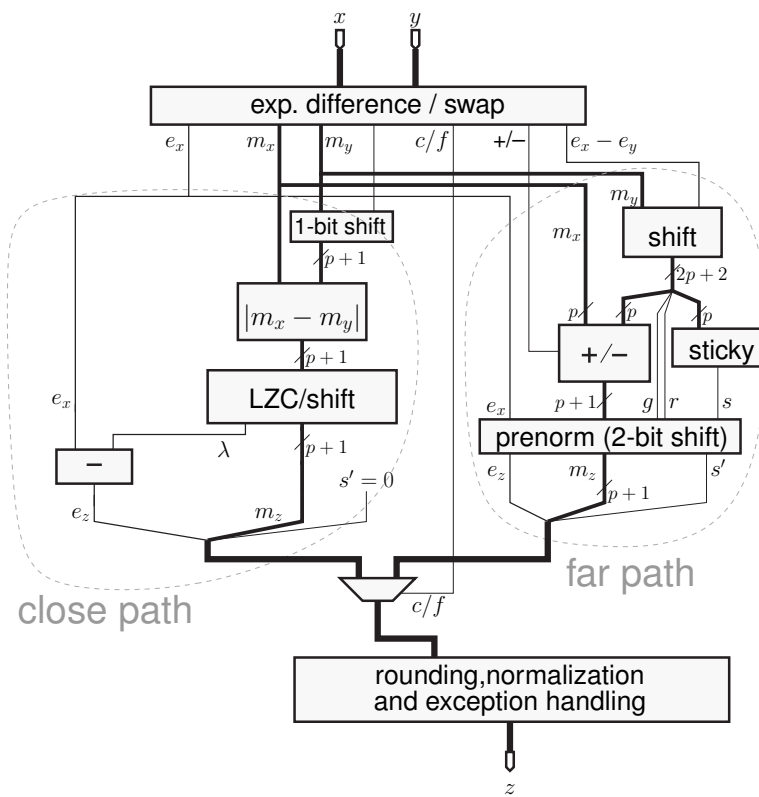
Figure 9.11: A dual-path floating-point adder.

The addition itself is then performed. Finally, the result may be normalized. In the case of a cancellation, this normalization may require an LZC and a shifter.

9.3.2   A first dual-path architecture

As remarked already in the previous chapter, the two large shifts occur in situations that are exclusive: a cancellation may only occur if the input exponents differ by at most one, but in this case there is no need for a large alignment shift. Besides, the decision of which path to choose depends only on the input exponents and can be made early, typically in the same time as the initial exponent comparison. Therefore, virtually all the recent floating-point adders are variations of the *dual-path* architecture presented in Figure 9.11. The *close* path is for situations where a massive cancellation (more than 1 bit) may occur: effective subtractions of inputs with exponents that differ by at most 1. The *far* path is for distant exponents (their difference is at least 2).

Here are a few remarks on Figure 9.11.
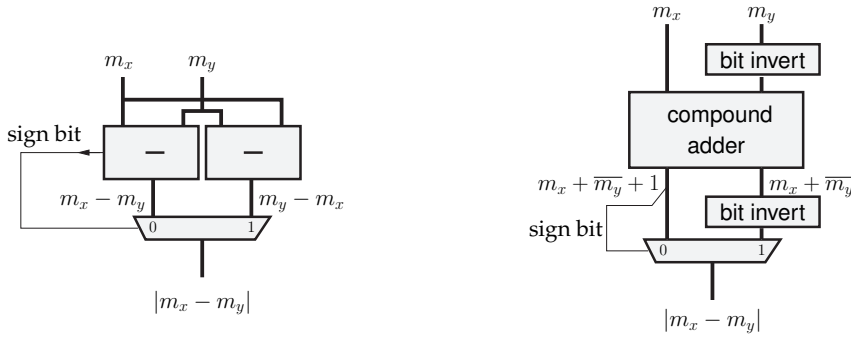
Figure 9.12: Possible implementations of significand subtraction in the close path.

- Both paths are relatively balanced in terms of critical path; however, the close path has slightly more work to do.

- The far path may perform either an addition or a subtraction. The sign of the result is known in advance; it will be the sign of the number whose exponent is larger.

- The close path always performs a significand subtraction; however, the sign of the result is unknown. If it is negative, it needs to be negated to obtain a positive significand (changing the sign bit of the result). This change of sign is represented after the subtraction in Figure 9.11. A better latency is obtained by computing in parallel $m_x - m_y$ and $m_y - m_x$ and selecting the one that is positive, as shown in Figure 9.12.

- The final normalization unit inputs an exponent e_z, a $(p + 1)$-bit significand m_z whose MSB is 1 and whose least-significant bit (LSB) is a round bit, and a sticky bit s' (the sticky bit coming from the close path is always 0). This information suffices to perform rounding according to the rules defined in Chapter 8. The integer increment trick described page 242 may be used: a $(p + w_E)$-bit integer is built by removing the leading 1 of the significand and concatenating the exponent to its left, then the rounding rules simply decide (from the round and sticky bits) if this integer should be incremented or not. This provides a normalized representation of the result even when the increment entails an overflow.

- The LZC/shift box of the close path, and the prenorm box of the far path, prepare the information for the final normalization unit.

- The prenorm box in the far path inputs the $(p + 1)$-bit significand sum (including a possible carry out), and three more bits from the shifted significand, classically called g for *guard bit*, r for *round bit*, and s a sticky bit computed from the p least significand bits of the shifted sum.

The prenorm box is essentially a multiplexer controlled by the two leading bits of significand sum (including the carry out):

- if 1x (carry out), this carry will be the leading 1 of the p-bit significand of the result. In this case the exponent is $e_z = e_x + 1$ and the sticky out is $s' = g$ OR r OR s ;

- if 01, the significand sum is still aligned with m_x. The $(p + 1)$-bit significand m_z is obtained by removing the leading zero of the sum and concatenating g as its LSB. The exponent is $e_z = e_x$ and the sticky out is $s' = r$ OR s;

- if 00 (an effective subtraction leads to a one-bit cancellation), since the shift was at least by two bit positions, it is easy to prove that the third bit is a 1. In this case the exponent is $e_x - 1$ and the sticky out is $s' = s$.

Compared to a single-path architecture, the area overhead of the dual-path architecture is limited. The only duplicated hardware are the significand additions, and the far path prenormalization, which may be considered as a 2-bit combined LZC/shifter. Besides there is also the cost of the multiplexer selecting the results from the close or far path, and in a pipelined implementation the cost of registers synchronizing both paths.

Still, the dual-path design makes sense even when area is a concern, as in FPGA implementations [247, 115]. The design of Figure 9.11 is indeed suitable for an FPGA implementation since the delay of leading-zero counting can be hidden in the shift delay.

Now in a standard full-custom VLSI technology, implementing Figure 9.11 will lead to a larger latency in the close path than in the far path. The far path still involves only two logarithmic steps in sequence (a shifter and an adder), whereas the close path has three: the subtracter (if implemented as shown in Figure 9.12), an LZC, and a shifter. The latter can not be overlapped: the LZC provides the most significant bits of the shift in its last levels, whereas the shifter needs them in its first levels.

The solution is *leading-zero anticipation*, a technique that allows us to compute an approximate of the leading-zero count in parallel with the significand subtraction.

9.3.3 Leading-zero anticipation

A leading-zero anticipator (LZA) is a block that computes or estimates the number of leading zeros from the inputs of the significand adder, instead of having to wait for its output. Figure 9.13 describes an adder architecture using an LZA. Note that some authors call the LZA a leading-one predictor or LOP.

An LZA works in two stages.
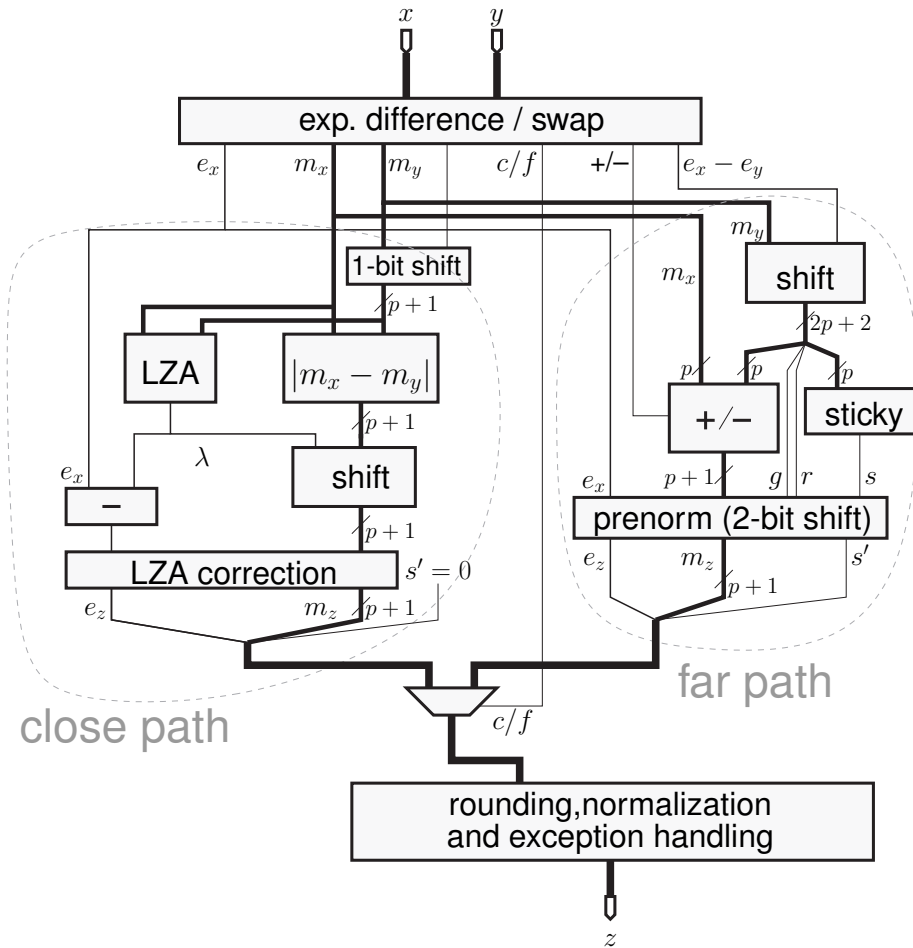
Figure 9.13: A dual-path floating-point adder with LZA.

- The first stage recodes in parallel the two p-bit inputs m_x and m_y to be subtracted into a single p-bit *indicator string* f. This string has the following property: if its leading one is in position i, then the leading one of the sum is at position i or $i + 1$. This stage has a constant delay of a few gates.

- A standard LZC on the string f then provides an estimation of the leading-zero count of the result of the subtraction, which may be off by at most 1.

The reason why the result may be off by 1 is that the LZA computes the count from the MSB to the LSB, therefore it ignores a possible carry propagation coming in the other direction. This may be corrected in two possible ways:

- either the LZA count is corrected when the addition is finished;

- or the shifter is followed by a multiplexer that implements the needed possible 1-bit shift. This is the LZA correction box of Figure 9.13. Its construction is very similar to the prenorm box, and it also has a delay of a few gates. In practice, this last stage may be efficiently fused with the last stage of the shifter: this stage normally shifts by 0 or 1 bit position, and may be modified to shift also by 2 bit positions.

Let us now give an example of computation of the indicator string (see [364] for a comprehensive survey). We first present it in the simpler case of addition. An LZA for subtraction, which is what we need for the floating-point adder, raises the additional difficulty of the sign of the result, which we will address later.

Let us denote by A and B two positive integer numbers written on n bits $a_{n-1}...a_0$ and $b_{n-1}...b_0$, respectively.

Let

$$\begin{cases} p_i &= a_i \text{ XOR } b_i \\ g_i &= a_i \text{ AND } b_i \\ k_i &= \overline{a_i} \text{ AND } \overline{b_i} \end{cases},$$

where the overline denotes the binary negation. These are the propagate, generate, and kill signals classically used in fast adders. If p_i is set, the i-th bit position propagates the incoming carry, whatever it is, to the $(i + 1)$-st bit position. If g_i is set, a carry is generated at the i-th bit position regardless of the incoming carry. If k_i is set, no carry will exit the i-th bit position, regardless of the incoming carry.

One may then check that a leading-zero string on the sum corresponds:

- either to a leftmost sequence of consecutive bit positions such that k_i is set (this is often noted k^*, using a notation inspired by regular expressions);

- or to a sequence (from left to right) of zero or more consecutive bit positions with p_i set, followed by one position with g_i set, followed by zero or more consecutive positions with k_i set (in regular expression notation, a sequence $p^* g k^*$). Note that this case corresponds to an addition of A and B with overflow (carry out). Here we ignore the carry out in the leading-zero count: this will be justified as we use such an addition for two's complement subtraction.

The trick is that these two sequences can be encoded as the indicator string defined by

$$f_i = p_i \text{ XOR } \overline{k_{i-1}}$$

(the proof is by enumeration).

The important thing here is that the computation of f_i is a Boolean function of only two bits of A and two bits of B: it may be computed with a few gates and in a very small delay.

Let us now consider an LZA for the close path of a floating-point adder. As the addition is actually a subtraction, A will be $2^p \cdot m_x$ with $n = p + 1$, and B will be the binary complement of either $2^p \cdot m_y$ or $2^{p-1} \cdot m_y$. If the result is positive, we have our estimation of the leading-zero count. However, if the result turns out to be negative, we should have counted the leading ones instead.

A conceptually simple solution is to duplicate the LZA hardware, just as the addition itself is duplicated in Figure 9.12. This approach means duplicating the LZC as well. Another solution is to derive an indicator string that works for leading zeros as well as for leading ones. This approach does not need to duplicate the LZC, but the computation of the indicator string is now more complex, as it needs to examine three bits of each input [58]. A discussion of this tradeoff is given in [364].

As a conclusion, the best implementation of leading-zero anticipation is very technology dependent. It also depends on the pipeline organization and on the implementation of the adder itself. For a comprehensive review of these issues, see the survey by Schmookler and Nowka [364]. Many patents are still filed each year on this subject.

Subnormal handling in addition

Binary floating-point addition has a nice property (Theorem 3, page 124): any addition that returns a subnormal number is exact. A subnormal either results from the addition or subtraction of two subnormals, or from a cancellation (in the close path of Figure 9.13). In the first case, no shift is needed. In the second case, the shift amount has to be limited to the one that brings the exponent to $e_{\min}$: the remaining leading zeros will be left in the subnormal significand. In both cases, we do not have to worry about rounding the result, since it is exact.

The only issue is to ensure that the detection of subnormal inputs on one side and the limitation of the shift amount in the close path on the other side do not degrade the critical path of the overall operator.

The shift at the beginning of the addition far path is now driven by the exponent difference $E_x - n_x + E_y - n_y$, where n_x and n_y are the "is normal" bits. To hide the delay of the computation of these bits (n_x is computed as the OR of the bits of E_x), one may compute in parallel n_x, n_y, and the three alternative differences $e_x - e_y$, $e_x - e_y - 1$, and $e_x - e_y + 1$, and select the correct one depending on n_x and n_y. This adds just one level of multiplexers to the critical path. Besides, n_x and n_y are the implicit bits of the significand, and are available before the significand shift, so there is no increase of the critical path delay here.

It is more difficult to limit to $e_x - e_{\min}$ the normalizing shift in the close path without increasing the close path critical path. The shift distance is now $\min(\lambda, e_x - e_{\min})$, where λ is now the tentative number of leading zeros output by the LZA. The value $e_x - e_{\min}$ may be computed early, but the delay of the comparison with λ, from the LZA output, to decide shifting is added to the critical path. A solution is to perform the two shifts (by λ bits and by $e_x - e_{\min}$ bits) tentatively while comparing these two numbers and to select the valid one when the comparison is available. This adds a lot to the area.

Another solution would be to inject a dummy leading 1 in the proper position ($e_x - e_{\min}$) before leading-zero counting. Again, $e_x - e_{\min}$ may be computed in parallel with exponent difference, but bringing a 1 in this position is a shift. It is difficult to obtain this value before the leading-zero count.

To summarize, we have to add at least the delay of a few multiplexers to both the far path and the close path, and this may cost much area. Or, a more area-efficient solution may be preferred, and it will add to the critical path the delay of an exponent addition, which remains small. The best solution in a given adder will depend on whether the critical path delay is in the close path or in the far path.

9.3.4 Probing further on floating-point adders

We have not discussed the technology-dependent decomposition of the adder in pipeline stages. The pipeline depth is typically 2 [375], 3 [308], or 4 [295] cycles for binary64 operands. In [308], the latency is variable from 1 to 3 cycles, depending on the operands.

Even and Seidel suggest a nonstandard two-path architecture in [375], and compare it to implementations in Sun and AMD processors. Their motivation is the minimization of the overall logic depth of floating-point addition [374].

Pillai et al. present a three-path approach for lower power consumption in [333].

Fahmy, Liddicoat, and Flynn suggest using an internal redundant format

for floating-point numbers within processors [134]. This enables significand addition with $O(1)$ delay. Numbers have to be converted back to the standard format only when writing back to memory.

As floating-point adders are complex designs, there have been several attempts to prove their correctness using formal methods [357, 26]. A formal approach to the design is also advocated by Even and Paul [132].

9.4   Binary Floating-Point Multiplication

9.4.1   Basic architecture

The architecture depicted by Figure 9.14 directly follows the algorithm given in Section 8.4, page 251, of Chapter 8. It does not illustrate sign or overflow handling, which is straightforward. Underflow handling is discussed in Section 9.4.4. Here are a few comments on this architecture.

- The computation in parallel of $e_x + e_y - b$ and $e_x + e_y - b + 1$ consumes little area, as it may use small and slow carry-propagate adders: the exponents are not needed before the end of the significand product, which takes much more time.

- The sticky bit computation may in principle be performed as a by-product of the multiplication, in such a way that the sticky bit is available at the same time as the result of the significand multiplexer.

- The increment in the final normalization unit may change the exponent a second time. One idea could be to use a single adder using a carry propagation from the significand to the exponent, as explained in Section 8.2.1, page 242. However, note that even rounded up, the product of two significands in $[1, 2)$ never reaches 4. Indeed, the largest possible significand is $2 - 2^{-p+1}$, whose square is $4 - 2^{-p+2} + 2^{-2p+2}$. This is the largest possible significand product, and it is rounded up to $4 - 2^{-p+2} + 2^{-p+1} < 4$. As a consequence, the exponent is never incremented twice. In other words, there will only be a carry out from the normalization incrementer if the exponent was $e_x + e_y - b$, in which case it is changed to $e_x + e_y - b + 1$.

- Not shown on Figure 9.14 are sign and exception handling, which are straightforward, except for subnormal numbers, which are managed using methods discussed in Section 9.4.4.

9.4.2   FPGA implementation

The architecture of Figure 9.14 is well suited for FPGA implementations.

Figure 9.14: Basic architecture of a floating-point multiplier without subnormal handling.

- Subnormal handling is not a strong requirement for applications using FPGA floating-point accelerators. The floating-point format used in these accelerators can be nonstandard, and in particular can have an ad hoc exponent range.

- Significand multiplication can be performed efficiently using the small integer multipliers embedded in the FPGA fabric of high-performance FPGAs. These multipliers are typically able to perform 18×18-bit products, and recent FPGAs have increased this size to 25×18-bit to facilitate the implementation of binary32 arithmetic. For larger significand sizes, several of these multipliers have to be grouped together; for instance, a 36×36-bit product can be implemented using four 18×18-bit multipliers and a few adders. In recent FPGAs, the embedded multipliers are tightly coupled to specific adders. The main purpose of these blocks is efficient multiply-and-accumulate operations for digital signal processing (DSP), but they also allow for building larger multipliers [104].

- Embedded multipliers are not able to compute the sticky bit as a by-product. However, a wide OR can be computed using the fast-carry circuitry. As soon as more than one embedded multiplier is needed, the higher part of the result comes from an addition, and the sticky computation can be overlapped with this addition.

- The increment in the final rounding unit can be performed by an area-efficient carry-propagate adder through the fast-carry circuitry.

9.4.3 VLSI implementation optimized for delay

Let us analyze the delay of the previous design when implemented in full-custom VLSI. The critical path is in the significand computation.

- One of the operands is Booth recoded (see Section 9.2.4), and the partial product array is generated. This step takes a small constant delay.

- A compression tree in $\log p$ time compresses the partial product array into a carry-save representation of the product.

- A fast adder, also in $\log p$ time, converts this carry-save product to standard representation.

- The normalization unit needs to wait for the MSB of the result of this addition to decide where to increment. This increment takes another fast adder, again with a $\log p$ delay.

It is possible to reduce the two adder latencies to only one (and a few more gate delays). Let us start from the output of the compression tree, which is a carry-save number, i.e., a pair of vectors $s_{-1}s_0s_1 \cdots s_{2p-2}$ and $c_{-1}c_0c_1 \cdots c_{2p-2}$.

An easy case is round to zero, which comes down to a truncation. In this mode, all one needs is a $2p$-bit fast adder inputting $s_{-1}s_0s_1 \cdots s_{2p-2}$ and $c_{-1}c_0c_1 \cdots c_{2p-2}$ and outputting $y_{-1}y_0y_1 \cdots y_{2p-2}$. Then, a multiplexer selects $y_{-1}y_0 \cdots y_{p-2}$ if $y_{-1} = 1$, and $y_0y \cdots y_{p-1}$ if $y_{-1} = 0$. It also selects the exponent among $e_a + e_b$ and $e_a + e_b + 1$.

The reason why we need a second large adder in the previous architecture is the possible increment of the result, which never happens in round-to-zero mode.

The trick will therefore be to reduce the other rounding modes to the round-to-zero mode. For this purpose, we add to the partial product array a few bits which depend only on the rounding mode. These *rounding injection* bits [133] may, in principle, add at most one gate delay to the delay of the reduction tree. In practice, however, the tree is designed to fit in one or two pipeline cycles, and the injection bits do not increase its latency.

The injection bits are defined as follows:

$$\text{inj} = \begin{cases} 0 & \text{if round to zero} \\ 2^{-p} & \text{if round to nearest} \\ 2^{-p+1} - 2^{-2p+2} & \text{if round to infinity.} \end{cases}$$

If the significand product is in $[1, 2)$, we indeed have $\circ(z) = RZ(z + \text{inj})$ except in one case: the round to nearest thus implemented is *round to nearest*

up, not round to nearest even. In case of a tie, it always returns the larger number. This will be easy to fix by simply considering the LSB z_{p-1} of the rounded result. If $z_{p-1} = 0$, the result is even and there is nothing to do. If $z_{p-1} = 1$, the nearest even number is obtained by simply pulling z_{p-1} to 0, which takes no more than one gate delay.

To summarize so far, the injection bits allow us to obtain the correctly rounded result in any rounding mode using only one $2p$-bit adder and a few extra gates, in the case when the significand product is in $[1, 2)$. The correctly rounded result is obtained by truncation as the bits $z_0 \cdots z_{p-1}$, with the bit z_{p-1} pulled down in case of a tie for RN mode. Detecting a tie requires a sticky bit computation as usual, which can be performed in parallel to the sum of the lower bits.

Let us now consider the case when the result will be in $[2, 4)$. In this case, the injection bits have been added one bit to the right of their proper position: the injection should have been

$$\text{inj}_{[2,4)} = \begin{cases} 0 & \text{if round to zero} \\ 2^{-p+1} & \text{if round to nearest} \\ 2^{-p+2} - 2^{-2p+2} & \text{if round to infinity.} \end{cases}$$

To correct this injection, if the result turns out to be in $[2, 4)$, we have to add a correction defined by

$$\text{corr} = \text{inj}_{[2,4)} - \text{inj} = \begin{cases} 0 & \text{if round to zero} \\ 2^{-p} & \text{if round to nearest} \\ 2^{-p+1} & \text{if round to infinity.} \end{cases}$$

The problem is that we do not know if this correction should be applied before the end of the significand adder. Of course, we don't want this final correction to involve another carry propagation delay over the full significand. The solution is to compute in parallel the two versions of the higher bits, one with the correction, and one without.

The sum of the lower bits of the carry-save product, $s_p \cdots s_{2p-2} + c_p \cdots c_{2p-2}$, is condensed into a sticky bit, a sum bit s'_p, and a carry-out bit c'_{p-1} (see Figure 9.15). This carry out will be the carry in to the sum of the higher part of the carry-save product. As the correction may add one bit at position $p - 1$, a row of half-adders is used to reduce $s_{-1} \cdots s_{p-1} + c_1 \cdots c_{p-1}$ to $s'_{-1} \cdots s'_{p-1} + c'_{-1} \cdots c'_{p-2}$. Thus, if the correction is 2^{-p+1} (round to infinity), it will be added to at most two bits at position $p - 1$ (s'_{p-1} and c'_{p-1}) and generate a carry at position $p - 2$. If the correction is 2^{-p}, it will be added to one bit of the same weight (s'_p), and again a carry may propagate to the higher bits.

Thus, the two versions of the higher sum to compute in parallel are $z_{-1} \cdots z_{p-2} = s'_{-1} \cdots s'_{p-2} + c'_{-1} \cdots c'_{p-2}$, and the same with a carry in in case of correction. These two sums can be computed by a compound adder.
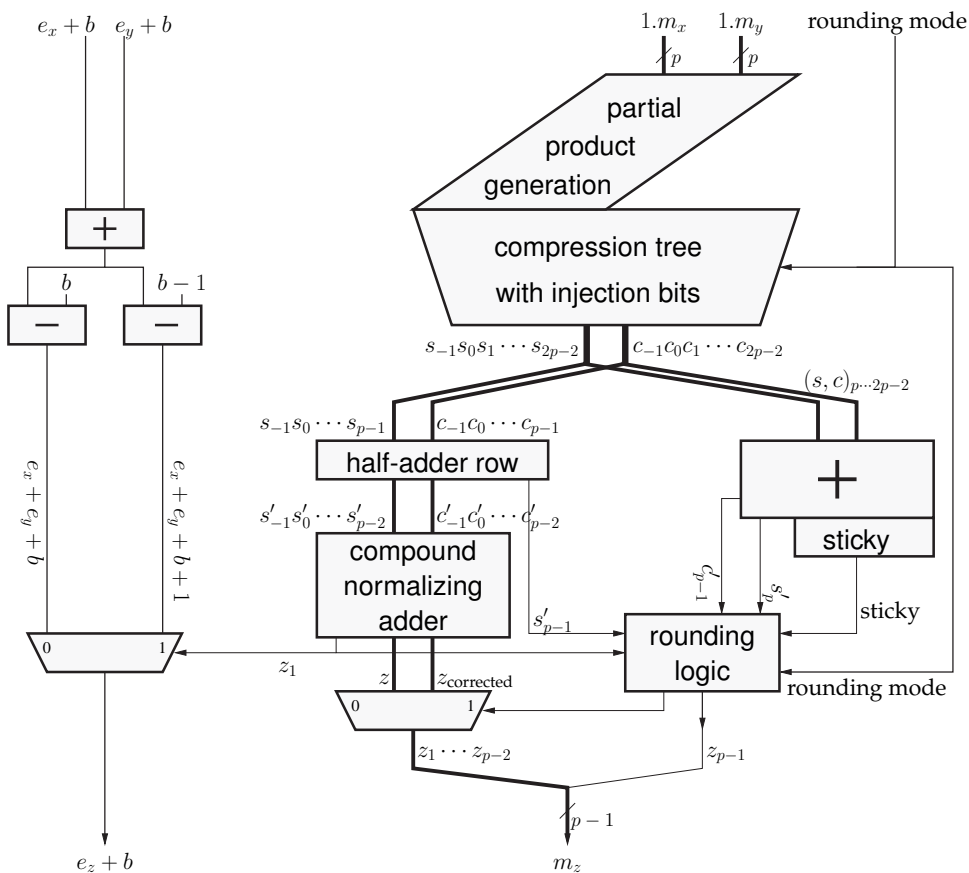
Figure 9.15: A floating-point multiplier using rounding by injection, without sub-normal handling.

The carry in—which will decide which of these results is valid—is a combinatorial function of s'_{p-1}, s'_p, c'_p, the rounding mode which defines the correction to apply, and of course the value of z_{-1} which decides normalization, and hence whether a correction should be applied. This is the z_{-1} of the sum, not of the incremented sum: there will be a carry in to the compound adder only if the sum is in $[2, 4)$, in which case we know that the incremented sum is also in $[2, 4)$.

The lower bit of the normalized result is another combinatorial function from the same information, plus the sticky bit.

The resulting architecture is shown in Figure 9.15. The compound adder on this figure outputs the normalized versions of z and $z_{\text{corrected}}$, i.e., $z_0 \cdots z_{p-3}$ if $z_{-1} = 1$ and $z_1 \cdots z_{p-2}$ if $z_{-1} = 0$.

The full detail of the Boolean equations involved can be found in [133], which also compares this rounding algorithm to two other approaches.

9.4.4 Managing subnormals

As explained in the previous chapter, handling subnormals conceptually consists in two additional steps:

- normalizing a possible subnormal input using a wider exponent range;

- detecting a possible subnormal output and shifting its significand to insert the necessary leading zeros, before rounding.

This conceptual view does not fit our previous architecture. First, the two steps added are long-latency ones, and they are sequential. Second, we perform rounding by injection in the compression tree, and if the output is subnormal, we will have performed the rounding at the wrong place. Let us analyze in more detail the issues involved in order to schedule this additional work in parallel with the existing steps, and not in sequence.

- Subnormal inputs must be detected, which means testing the exponent field for all zeros. If both inputs are subnormals, the result will be zero or one of the smallest floating-point numbers; therefore, only the situation where one input is subnormal and the other is normal needs to be managed with care. Let n_x (resp n_y) be the "is normal" bit, a flag bit of value 1 if x (resp. y) is normal, and 0 if it is subnormal. This bit can be used as the implicit bit to be added to the significand, and also the bias correction for subnormals.

- The implicit bit affects one row and one column of partial products. The simplest solution here is to delay the compression of these partial products long enough so that it begins when the implicit bit has been determined. This does not add to the critical path.

- The leading zeros of the subnormal input need to be counted. Define λ as the number of leading zeros of the significand with n_x appended as the leading bit.

- The subnormal input need not be normalized before multiplication. Indeed, the existing multiplier which computes the product of two p-bit numbers will compute this product as well if one of the inputs has leading zeros. The product will then have leading zeros, too, so it needs to be normalized, but this gives us the time to compute λ in parallel with the multiplication tree. This latency reduction comes at a hardware cost: we now have to shift two $2p$-bit strings (c and s) instead of one p-bit string.

- After this shift we have, in carry-save form (c, s), a significand in $[1, 4)$ whose exponent is $e_x + e_y - \lambda$. The result will be subnormal if $e_x + e_y + z_1 - \lambda < e_{\min}$, where z_1 is the leading bit of the (yet to be computed) sum of c and s. In this case, the result will have to be shifted right by $e_{\min} - (e_x + e_y + z_1 - \lambda)$ bit positions before rounding.

- We now need to preprocess the injected rounding bits to anticipate this shift. Ignoring z_{-1}, which will be handled by the same correction as previously, the injection has to be shifted left by $\max\{e_{\min} - (e_x + e_y - \lambda), 0\}$ bit positions. If we are able to compute this shift value and perform the shift in a delay shorter than that of the compression tree, the shifted injection will be ready to be added in a late stage of the compression tree, and this will not add to the delay.

- Finally, the two previous shifts may be combined in a single one, but this requires changing the injection again. Finally, the injection becomes:

$$
\text{inj} = \begin{cases}
0 & \text{if round to zero} \\
2^{-p-l+\max(e_{\min}-(e_x+e_y-l),0)} & \text{if round to nearest} \\
2^{-p+1-l+\max(e_{\min}-(e_x+e_y-l),0)} - 2^{-2p+2} & \text{if round to infinity.}
\end{cases}
$$

- To summarize, the delay overhead related to subnormal handling is the delay of one large significand shift. All the other computations can be hidden by the delay of the compression tree.

- Note that no correction will occur in the case of a subnormal output as z_{-1} will be 0, which means that there is nothing to change in the correction logic.

9.5   Binary Fused Multiply-Add

Let us now discuss the hardware implementation of a fused multiply-add (FMA) binary operator. Most of the algorithm is explained in Section 8.5.4,

page 259. We first present the most classical architectures, then discuss several propositions of alternative architectures.

9.5.1 Classic architecture

The first widely available FMA was that of the IBM RS/6000 [281, 183]. Its architecture closely follows the algorithm sketched in Section 8.5.4 and is depicted in Figure 9.16. The data width and alignments are identical. The main differences one may observe on this figure with respect to the algorithm are the following.

- As in the floating-point multiplier, the intermediate significand product is produced in carry-save representation. The "is normal" bits are injected in late stages of the compression tree.

- The shift of the addend is performed in parallel with the product compression tree.

- A carry-save adder inputs the carry-save product and the $2p$ lower bits of the shifted summand, and produces their sum in carry-save representation.

- This sum is completed with the higher bits of the shifted summand and a carry in that completes the bit inversion in case of effective subtraction.

- A fast adder transforms this carry-save sum in standard representation, and complements it if negative. This step may use an end-around carry adder, or two adders in parallel, with the proper result being selected depending on the sign bit.

- In parallel, an LZA determines λ, the number of leading zeros needed in case of cancellation or of an input subnormal. In some implementations, in cases when the result will be subnormal, a 1 is injected before the LZA to limit the count to the proper value [371].

- The normalization box is mostly a large shifter, but it also performs the case analysis described in Section 8.5.4 and determines the exponent before rounding.

- The rounding box performs the rounding and the possible subsequent 1-bit normalization, and handles overflow and sign.

This architecture is typically pipelined in 3 to 7 cycles. In the POWER6 FMA implementation, the latency is reduced from 7 to 6 cycles for dependent operations, as the final rounding is performed in the beginning of the next operation [410].
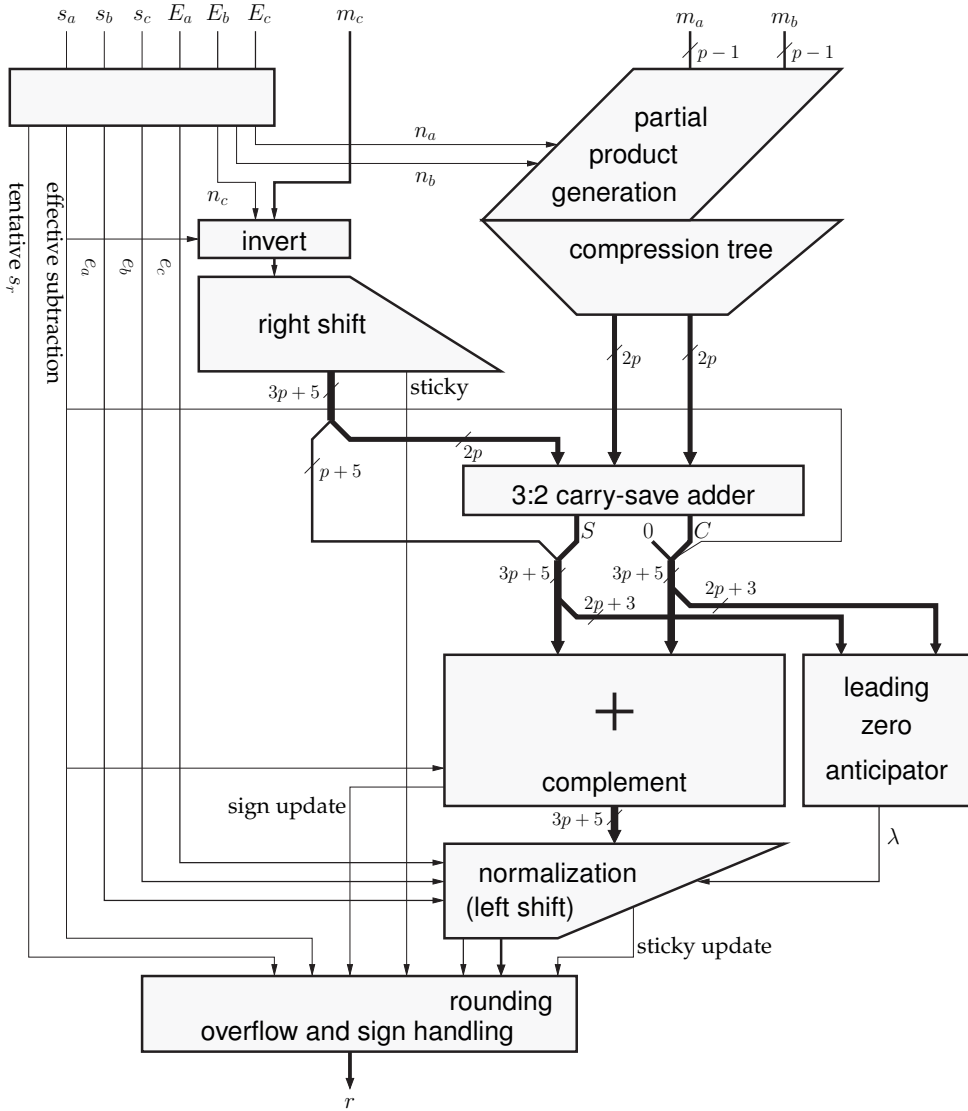
Figure 9.16: The classic single-path FMA architecture.

9.5.2   To probe further

An overview of the issues related to subnormal handling is given in [371].

Several authors [373, 339] have investigated multiple-path FMA implementations.

Lang and Bruguera described an FMA architecture that anticipates the normalization step before the addition [238]. This enables a shorter-latency rounding-by-injection approach. They also suggested an FMA architecture that reduces the latency in case of an addition by bypassing the multiplier stages [59].

Quinnell et al. [373, 339] propose a bridge FMA architecture that adds FMA functionality to an existing floating-point unit by reusing parts of the adder and of the multiplier.

An example of a recent deep submicrometer, high-frequency implementation is the POWER6 FMA described in [410, 439]. To minimize wiring, the pipeline of this FMA is laid out as a U with the floating-point registers on the top.

The single-precision FMAs in the synergistic processing elements (SPEs) of the Cell/B.E. processors are non IEEE-compliant [315].

9.6   Division

There are three main families of division algorithms: digit recurrence, Newton–Raphson based, and polynomial based. Newton–Raphson-based algorithms have been reviewed in Chapter 5, and an example of polynomial based algorithm will be presented in Chapter 10. These two approaches rely on multiplication, and make sense mostly in a software or microcode context, when a multiplier is available. In this section, we focus on division by digit recurrence, which is the algorithm of choice for a stand-alone hardware implementation because it reduces to simpler primitives: addition and digit-by-integer multiplication. In addition, we will see that it exposes a wide range of tradeoffs, which means that it may be adapted to a wide range of contexts.

In microprocessors, the current trend is not to include a division operator and compute divisions using the FMA as shown in Chapter 5. However, this design decision is still open. For instance, in [147] there is a discussion of the pros and cons of both approaches for the IBM z990, and the digit-recurrence approach is preferred.

In FPGA-accelerated applications, a stand-alone hardware divider often makes sense. Current implementations usually employ digit recurrences, but this choice could be reassessed considering the availability of embedded multipliers. Some multiplication-based dividers have been published for FPGAs [334, 430], but so far they do not provide correct rounding.

Let us now focus on a hardware floating-point divider using a digit-recurrence algorithm. More details on digit-recurrence division theory and
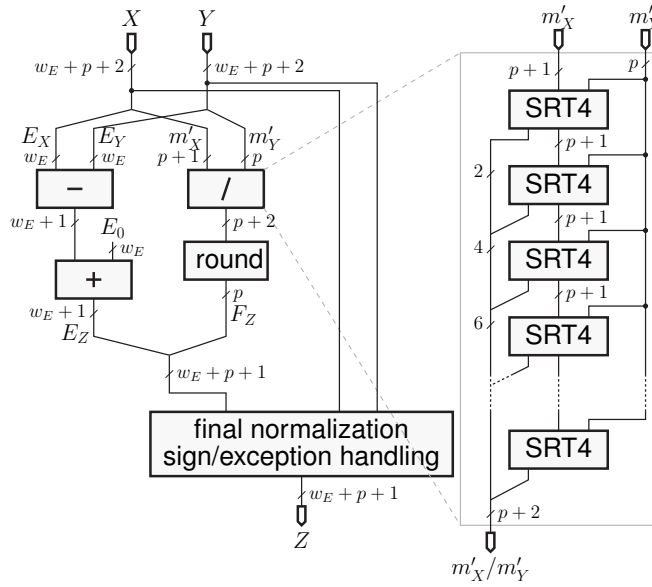
Figure 9.17: A pipelined SRT4 floating-point divider without subnormal handling. Note that an alternative, often used in processors, is to reuse $p/2$ times the same SRT4 stage. This means smaller area, but divisions can no longer be pipelined.

implementations can be found in textbooks by Ercegovac and Lang [125, 126].

9.6.1   Digit-recurrence division

We assume the sign and special cases have been handled as per Section 8.6, page 262, and that the input numbers have been normalized. We refer to Section 8.6 for sign and exponent handling (see Figure 9.17 for an illustration), and focus on dividing the significand

$$X = x_0.x_1 \cdots x_{p-1}$$

by the significand

$$D = d_0.d_1 \cdots d_{p-1},$$

with $x_0 \neq 0$ and $d_0 \neq 0$. We have $X \in [1, \beta)$ and $D \in [1, \beta)$; therefore, $X/D \in (1/\beta, \beta)$. For clarity, in the rest of this section, upper-case letters denote multi-digit numbers, and lower-case letters denote digits.

In digit-recurrence approaches, the division is expressed using the Euclidean equation

$$X \;=\; QD + R \quad \text{with} \quad 0 \leq R < D\,\mathrm{ulp}(Q). \tag{9.1}$$

This equation corresponds to rounding the quotient down. For the other rounding directions, a final correction will be needed.

Algorithm 9.3 is a generic radix-β digit recurrence that computes two numbers $Q = q_0.q_1 \cdots q_{p-1}$ and R satisfying

$$X = QD + R.$$

Note that this algorithm depicts the paper-and-pencil division. In this algorithm, β may indeed be 10, or 2, or small powers of 2. We will come back to the choice of number representation.

Algorithm 9.3 Generic digit-recurrence algorithm.

$R^{(0)} \leftarrow X$
$Q^{(0)} \leftarrow 0$
for $j = 0$ to $p - 1$ **do**
   $q_j \leftarrow \text{Sel}(R^{(j)}, D)$
   $R^{(j+1)} \leftarrow \beta R^{(j)} - q_j D$
end for
$R \leftarrow R^{(p)}$
$Q \leftarrow q_0.q_1 \cdots q_{p-1}$

The reader may check that, if we call $Q^{(j)} = q_0.q_1 \cdots q_j$ the quotient computed at iteration j, this algorithm maintains the invariant

$$X = Q^{(j)}D + R^{(j+1)}.$$

Inside the loop, the second line is composed of a subtraction and a digit-by-significand multiplication. Both operations are much simpler than significand-by-significand multiplication, having an $O(p)$ digit-level operation cost, and possibilities of $O(1)$ computation time when using redundant number systems. Also, note that it is possible to precompute, before starting the iteration, all the $q_j D$ for all the possible digit values. They may even be computed in parallel. Then, the iteration no longer involves any product.

The first line of the loop is the selection of the next quotient digit. The choice has to be made in order to ensure the convergence of the algorithm towards a final residual such that $0 \leq R < D\,\text{ulp}(Q)$. This is obtained by ensuring this condition at each iteration; thus, the invariant of the algorithm becomes

$$\left\{ \begin{array}{l} X = Q^{(j)}D + R^{(j+1)}, \\ 0 \leq R^{(j+1)} < D\,\text{ulp}(Q^{(j)}). \end{array} \right.$$

In the paper-and-pencil method, the Sel function is implemented by intuition. The human operator performs an approximation of the computation of the second line, using leading digits only. It fails very rarely, when βR^j is very close to a nontrivial multiple of D such as $7D$. In this case, the human computes the next residual $R^{(j+1)}$, observes that it is too large or negative, and backtracks with a neighboring value of q_j.

In a hardware implementation, a form of backtracking can be used in the binary case ($\beta = 2$) as follows. The subtraction is performed as if $q_j = 1$:

$$R_1^{(j+1)} = 2R^{(j)} - D.$$

Then, if $R_1^{(j+1)} < 0$, it means that the correct choice was $q_j = 0$; therefore, the next partial remainder is set to the current one, shifted: $R^{(j+1)} = 2R^{(j)}$. Otherwise $R^{(j+1)} = R_1^{(j+1)}$. Here, backtracking means simply restoring the previous partial remainder and does not involve a new computation. This algorithm is called the *restoring division algorithm*. A classical variant, called the nonrestoring algorithm, uses one register less [125].

In higher radices, backtracking is not a desirable option as it would need a recomputation of $R^{(j+1)}$ and thus lead to a variable-latency operator. A better alternative is the use of a redundant digit set for the quotient Q. Redundancy will give some freedom in the choice of q_j. Thus, in the difficult cases (again, when βR^j is close to a nontrivial multiple of D), there will be two valid choices of q_j. For both choices, the iteration will converge. This in turn means that the selection function $\text{Sel}(R^{(j)}, D)$ need not consider the full $2p$-digit information $(R^{(j)}, D)$ to make its choice. As the human operator in the paper-and-pencil approach, it may consider the leading digits of $R^{(j)}$ and D only, but it will do so in a way that never requires backtracking.

The family of division algorithms obtained this way is called *SRT division*, after the initials of the names of Sweeney, Robertson, and Tocher, who invented it independently [344, 408] . Its theory and a comprehensive survey of implementations are available in the book by Ercegovac and Lang [126]. We now briefly overview performance and cost issues.

The choice of the radix β has an obvious impact on performance. In binary, we have to choose $\beta = 2^k$. Each iteration then produces k bits of the quotient, so the total number of iterations is roughly p/k. Notice, thus, that here β is not necessarily the radix of the floating-point system: it is a power of that radix. Intel recently reduced the latency of division in their x86 processors by replacing a radix-4 division algorithm with a radix-16 algorithm [287]. However, each iteration is also more complex for larger k, in particular for the product $q_j D$. In a processor, the choice of radix is limited by the target cycle time.

A second factor that has an impact on the cost is the choice of the digit set used for the quotient. Most implementations use a symmetrical digit set:

$$q_j \in \{-\alpha, \dots, \alpha\} \quad \text{with} \quad \lceil \beta/2 \rceil \leq \alpha < \beta.$$

The choice of α is again a tradeoff. A larger α brings in more redundancy and therefore eases the implementation of the selection function. However it also means that more digit-significand products must be computed.

With a quotient using a redundant digit set, the selection function may be implemented as a table indexed by the leading digits of R^j and D. Other choices are possible; see [126] for a review.

Another variation on digit-recurrence algorithms is *prescaling*. The idea is that if the divider D is closer to 1, the selection function is easier to implement. To understand it, think again of the paper-and-pencil algorithm: when dividing for instance by 1.0017, one gets a very quick intuition of the next quotient digit by simply looking at the first digit of the partial remainder. In SRT algorithms, it is possible to formally express the benefit of prescaling. In practice, prescaling consists in multiplying both X and D by an approximation to the inverse of D. This multiplication must remain cheap, typically equivalent to a few additions. Prescaling has also been used to implement complex division [127].

9.6.2 Decimal division

An implementation of a decimal Newton–Raphson iteration was proposed by Wang and Schulte [428].

The SRT scheme is well suited to decimal implementations. The implementation of a digit recurrence decimal divider in the POWER6 processor is described by Eisen et al. in [123]. They use a radix-10 implementation with digit set $\{-5, \ldots, 5\}$ and prescaling to get the scaled divisor in $[1, 1.11)$. This is probably the only decimal division architecture actually in operation at the time of writing this book.

A complete architecture for decimal SRT division with the same digit set, but with D recoded in the BCD-5421 code (see Section 9.2.5), is also presented in Vásquez's thesis [413].

9.7 Conclusion: Beyond the FPU

The current trend in the processor world is to converge toward a single unifying operator, the FMA. The FMA replaces addition, subtraction, and multiplication, delivers to most applications better throughput and better accuracy for a reduced register usage, and allows for efficient and flexible implementations of division, square root, and elementary functions. It is about as bulky as an adder and a multiplier together, but this complexity brings subnormal number handling almost for free. Its architecture is not yet as mature as that of the legacy operators, but it has been receiving much attention recently. Legacy FPUs with their four arithmetic operators for $\pm$, $\times$, $/$, and $\sqrt{\ }$, will be around at least as long as x86-compatible processors, but both AMD and Intel announced an FMA even for such processors in the coming years.

Therefore, one may argue that the processor floating-point landscape is getting simpler. Not only do we have a consensus, but the consensus is on a single operator, the FMA, to replace four different ones. Processors need "one size fits all" operators.

The situation is exactly opposite in the emerging domain of FPGA-based floating-point accelerators. People initially ported the four operations and used them to accelerate computing cores. It was then realized that these operations could be optimized differently in different contexts. For instance, when programming an FPGA, you don't have to make a dramatic choice between a large and fast divider or a small and slow one. One will suit one application, and the other will suit another application. You could even use both for different tasks at the same time in one application. Similarly, you don't have to make a dramatic choice between small but inaccurate binary32 or slow but accurate binary64. You may use binary42 if it represents the soft spot in performance/accuracy for your application. Most floating-point libraries targeted for FPGAs are parameterized in precision.

This section is a survey of the architectural opportunities offered by the "one size need not fit all" feature of the FPGA target with respect to floating-point. The FPGA is the near-term target of the operators prospected here, and indeed some of these operators are so context specific that they make no sense at all in a processor. However, some others have found their way into the FPUs of application-specific circuits. For instance, GPUs include hardware for the evaluation of common elementary functions in binary32 arithmetic. Finally, we also review some operators which could be valuable additions to the FPUs of future general-purpose processors.

9.7.1   Optimization in context of standard operators

Optimization in context can be more radical than just a change of precision. Consider a common example, the computation of the Euclidean norm of a three-dimensional floating-point vector, $\sqrt{x^2 + y^2 + z^2}$.

- The multipliers can be optimized in this context. On one side, computing the significand square m_x^2 requires in principle half the work of an arbitrary significand multiplication $m_x m_y$, due to symmetries in the partial product array. On the other side, the exponent (before normalization) of x^2 is $2e_x$, which saves one addition.

- The adders also can be optimized in this context. The squares are positive, so these additions are effective additions, which means that the close path of Figure 9.11 can be removed altogether [115].

These optimizations preserve a bit-exact result when compared to unoptimized operators in sequence. If one lifts this requirement, one may go further.

- Overflows which occur in the intermediate computation of $x^2 + y^2 + z^2$, although the result $\sqrt{x^2 + y^2 + z^2}$ is in the floating-point range, may be prevented by using two more exponent bits in the

intermediate results, or by dividing in such cases the three inputs by a power of two, the result being then multiplied by the same value. When implemented in hardware, both solutions have a very small area and latency overhead. The scaling idea can be used in software, but its relative overhead (several tests and additional operations) is much higher.

- The alignment of the three significand squares can be computed in parallel before adding them, reducing the critical path.

- Then, a 3-input significand adder is more efficient than two 2-input ones in sequence.

- Intermediate normalizations and roundings can be saved or relaxed. For instance, it is possible to build a Euclidean norm operator that returns a faithful result, and is thus more accurate than the combination of correctly rounded operators, at a much lower hardware cost than this combination. Such a fused operator may have other advantages, for instance, symmetry in its three inputs.

Going even further, Takagi and Kuwahara [403] have described an optimized architecture for the Euclidean norm that fuses the computation of the square root with that of the sum of squares. This approach could even be extended to inverse square root [402], and other useful algebraic combinations such as

$$\frac{x}{\sqrt{x^2 + y^2}}.$$

The Euclidean norm is one example of a coarser operator that is of general use and in the context of which the basic operators can be optimized. Other coarser operators will be considered in Section 9.7.5. Let us first focus on an important case of operator optimization in context: the case of a constant operand.

9.7.2 Operation with a constant operand

A typical floating-point program involves many constants. Addition and subtraction with a constant operand do not allow for much optimization of the operator, but multiplication and division by a constant do.

By definition, a constant has a fixed exponent, therefore the floating point is of little significance here: all the research that has been done on integer constant multiplication [67, 248, 101, 437, 160, 118] can be used straightforwardly. Let C be an integer constant and X an input integer. Chapman's approach [67, 437] is FPGA specific: it exploits the structure of FPGAs, based on k-input LUTs, by writing the input X in radix 2^k: $X = \sum_{i=0}^n (2^k)^i x_i$. Then

$$CX = \sum_{i=0}^n (2^k)^i C x_i,$$

where the Cx_i may be efficiently tabulated in k-input LUTs. Most other recent approaches consider the binary expression of the constant. Let X be a p-bit integer. The product is written $CX = \sum_{i=0}^{k} 2^i c_i X$, and by only considering the nonzero c_i, it is expressed as a sum of $2^i X$; for instance, $17X = X + 2^4 X$. In the following, we will note this using the shift operator $\ll$, which has higher priority than $+$ and $-$; for instance, $17X = X + X \ll 4$. As in standard multipliers, recoding the constant using signed bits allows us to reduce the number of nonzero bits, replacing some of the additions with subtractions; for instance, $15X = X \ll 4 - X$. Finally, one may reuse sub-constants that have already been computed; for instance, $4369X = 17X + (17X) \ll 8$. One needs to resort to heuristics to find the best (or a nearly best) decomposition of a large constant into shifts and additions [248, 41, 160, 423]. When applied to hardware, the cost function of a decomposition must take into account not only the number of additions, but also the sizes of these additions [111, 200, 6, 49]. Other approaches are possible; for instance, the complexity of multiplication by a constant has recently been shown to be sublinear in the size of the constant [118], using an algorithm that is not based on the binary decomposition of the input, but is inefficient in practice.

To summarize, it is possible to derive an architecture for a multiplier by an integer constant that is smaller than that of a standard multiplier. How much smaller depends on the constant. A full performance comparison with operators using embedded multipliers or Digital Signal Processing (DSP) blocks remains to be done, but when these DSP blocks are a scarce resource, the multiplications to be implemented in logic should be the constant ones.

The architecture of a multiplier by a floating-point constant of arbitrary size is a straightforward specialization of the usual floating-point multiplier [49]. There are extreme cases, such as the multiplication by the constant 2.0, which reduces to one addition on the exponent.

We may also define constant multipliers that are much more accurate than what can be obtained with a standard floating-point multiplier. For instance, consider the irrational constant π. It cannot be stored on a finite number of bits, but it is nevertheless possible to design an operator that provably always returns the correctly rounded result of the (infinitely accurate) product πx [49], using techniques similar to those presented in Section 5.5, page 171. The number of bits of the constant that is needed depends on the constant, and may be computed using continued fraction arguments [51]. Although one needs to use typically $2p$ bits of the constant to ensure correct rounding, the resulting constant multiplier may still be smaller than a generic one.

A practical application is division by a floating-point constant [52]. It is implemented as a multiplication by the inverse, but to obtain a bit-exact result (the same result as using a divider), one needs to consider the inverse as an infinitely accurate constant. The correctly rounded product of X by this infinitely accurate constant is then equal to the correctly rounded division.
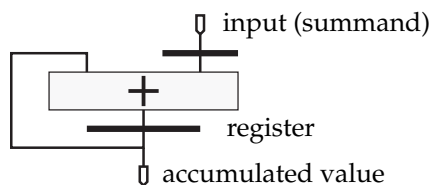
Figure 9.18: Iterative accumulator.

9.7.3 Block floating point

When an input floating-point vector is to be multiplied by another constant one (as happens in filters, Fourier transforms, etc.), one may use block floating point, a technique first used in the 1950s, when floating point arithmetic was implemented in software, and more recently applied to FPGAs [8]. The technique consists in an initial alignment of all the input significands to the largest one, which brings them all to the same exponent (hence the phrase "block floating point"). After this alignment, all the computations (multiplications by constants and accumulation) can be performed in fixed point, with a single normalization at the end. Compared with the same computation using standard operators, this approach saves the renormalization shifts of the intermediate results. The argument is that the information lost in the initial shifts would have been lost in later shifts anyway. As seen in Section 6.3, this argument may be disputable in some cases. In practice, however, a typical block floating-point implementation will accumulate the dot product in a fixed-point register slightly larger than the input significands, thus ensuring a better accuracy than that achieved using standard operators.

9.7.4 Specific architectures for accumulation

As already explained in Chapter 6, summing many independent terms is a very common operation. Scalar product, matrix-vector, and matrix-matrix products are defined as sums of products. Another common pattern is integration: when a value is defined by some integral, the computation of this value will consist in adding many elementary contributions. Monte Carlo simulations also typically involve sums of many independent terms.

For a few simple terms, one may build trees of adders, but when one has to add an arbitrary number of terms, one needs the iterative accumulator depicted by Figure 9.18.

In this case, the latency l of floating-point addition (typically 3 cycles in a processor, 6 to 12 cycles for FPGA high-frequency implementations) becomes a problem. Either the pipeline will actually work one cycle out of l, or one needs to compute l independent sub-sums and add them together at the end.

It is a common situation that the error due to the computation of one summand is independent of the other summands and of the sum, while the error due to the summation grows with the number of terms to sum. This happens in integration and sum of products, for instance. In this case, it makes sense to have more accuracy in the accumulation than in the summands.

A first idea, to accumulate more accurately, is to use a standard floating-point adder with a larger significand. However, this leads to several inefficiencies. In particular, this large significand will have to be shifted, sometimes twice (first to align both operands and then to normalize the result). These shifts are in the critical path loop of the sum (see Figure 9.11).

The large accumulator concept

A better solution may be to perform the accumulation in a large fixed-point register, typically much larger than a significand (see Figure 9.20). This removes all the shifts from the critical path of the loop, as illustrated by Figure 9.19. The loop is now a fixed-point accumulation for which current FPGAs are highly efficient. Fast-carry logic enables high frequencies for medium-sized accumulators, and larger ones may use partial carry save (see Figure 9.6).

The shifters now only concern the summand (see Figure 9.19) and can be pipelined as deeply as required by the target frequency.

The normalization of the result may be performed at each cycle, also in a pipelined manner. However, most applications won't need all the intermediate sums: they will output the contents of the fixed-point accumulator (or only some of its MSBs), and the final normalization may be performed offline in software, once the sum is complete, or in a single normalizer shared by several accumulators (case of matrix operations). Therefore, it makes sense to provide this final normalizer as a separate component, as shown by Figure 9.19.

For clarity, implementation details are missing from these figures—the interested reader will find them in [105]. For example, the accumulator stores a two's complement number, so the shifted summand has to be sign-extended. The normalization unit also has to convert back from two's complement to sign-magnitude, and perform a carry propagation in case the accumulator holds a partial carry-save value. None of this is on the critical path of the loop.

Designing an application-specific accurate accumulator

In addition to being simpler, the proposed accumulator has another decisive advantage over the one using the standard floating-point adder: it may also be designed to be much more accurate. Indeed, it will even be exact (entailing no roundoff error whatsoever) if the accumulator size is large enough so that
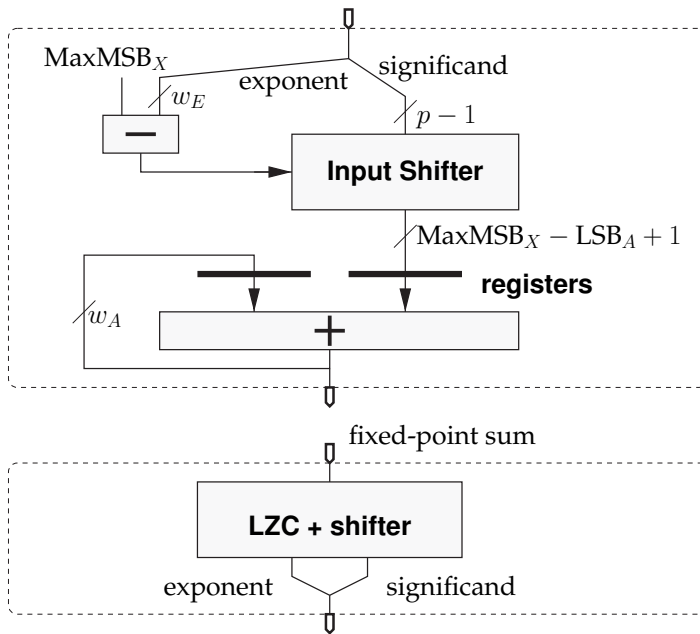
Figure 9.19: The proposed accumulator (top) and post-normalization unit (bottom). Only the registers on the accumulator itself are shown. The rest of the design is combinatorial and can be pipelined arbitrarily.
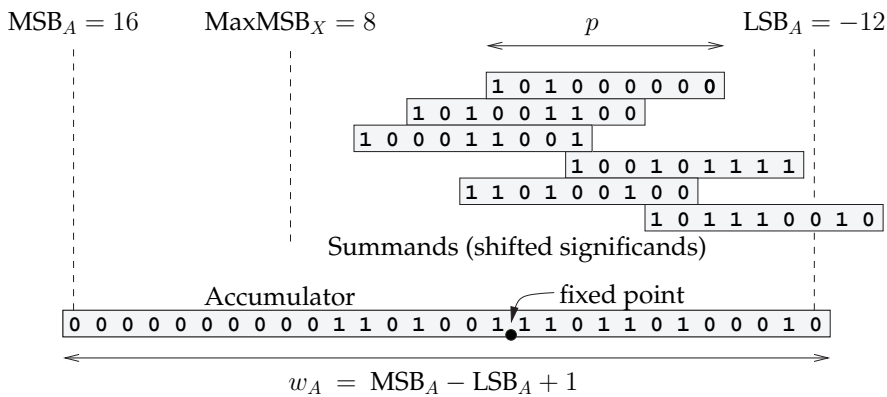
Figure 9.20: Accumulation of floating-point numbers into a large fixed-point accumulator.

its LSB is smaller than that of all the inputs, and its MSB is large enough to ensure that no overflow may occur. Figure 9.20 illustrates this idea, showing the significands of the summands, and the accumulator itself.

This idea was advocated by Kulisch [233, 234, 235] for implementation in microprocessors. He made the point that an accumulator of 640 bits for single precision (and 4288 bits for double precision) would allow for arbitrary dot products to be computed exactly, except when the final result is out of the floating-point range. Processor manufacturers always considered this idea too costly to implement.[1]

When accelerating a specific application using an FPGA, things are different: instead of a huge generic accumulator, one may choose the size that matches the requirements of the application. There are five parameters on Figures 9.19 and 9.20: w_E and p are the exponent and significand size of the summands; MSB_A and LSB_A are the respective weights of the most and least significant bits of the accumulator (the size in bits of the accumulator is $w_A = MSB_A - LSB_A + 1$), and $MaxMSB_X$ is the maximum expected weight of the MSB of a summand. By default $MaxMSB_X$ will be equal to MSB_A, but sometimes the designer is able to tell that each summand is much smaller in magnitude than the final sum. For example, when integrating a function that is known positive, the size of a summand could be bounded by the product of the integration step and the max of the function. In this case, providing $MaxMSB_X < MSB_A$ will save hardware in the input shifter.

Defining these parameters requires some trial-and-error, or (better) some error analysis which will involve one more hidden parameter, the number N of summands to accumulate. In most cases, the application dictates an *a priori* bound either on $MaxMSB_X$ or on MSB_A. LSB_A may be viewed as controlling the absolute accuracy, which still depends on N. The error analysis can be loose. For instance, adding to the maximum expected value of the result a margin of three orders of magnitude means adding only 10 bits to the accumulator.

This issue is surveyed in more detail in [105], where it is also shown that such an accumulator is much better in terms of area and latency than one using standard floating-point operators.

Example 12. *In [92], FPGAs are used to accelerate the computation of the inductance of a set of coils. This inductance is computed by the integration of inductances from elementary wire segments. Physical expertise tells us that the sum will be less than 10^5 (using arbitrary units), while profiling of a software version of the computation showed that the absolute value of an elementary inductance was always between 10^{-2} and 2.*

Converting to bit positions, and adding two orders of magnitude (or 7 bits) for safety in all directions, this defines $MSB_A = \lceil \log_2(10^2 \times 10^5) \rceil = 24,$

---

[1]Note that many earlier (fixed-point) desk calculators offered such a wide accumulator, and they are still commonly used in fixed-point DSP processors.

MaxMSB$_X$ = 8 and LSB$_A$ = −p − 14 where p is the significand width of the sum-mands. For p = 24 (binary32), we conclude that an accumulator stretching from LSB$_A$ = −24 − 14 = −38 (least significant bit) to MSB$_A$ = 24 (most significant bit) will be able to absorb all the additions without any rounding error: according to our profiling, no summand will ever add bits lower than 2^{-38}, and the accumulator is large enough to ensure it never overflows. The accumulator size should therefore be w$_A$ = 24 + 38 + 1 = 63 bits.

Of course this is an optimistic view: profiling does not guarantee that no sum-mand will ever be smaller than 10^{-4}, so in practice this accumulator will not be exact. However, profiling does show that such very small summands are extremely rare, and the accumulation of errors due to them is very likely to have no measurable effect.

Remark that in this application, only LSB$_A$ depends on p, since the other parameters (MSB$_A$ and MaxMSB$_X$) are related to physical quantities, regardless of the precision used to simulate them. This illustrates that LSB$_A$ is the parameter that allows one to manage the accuracy/area tradeoff for an accumulator.

Exact dot products and matrix operations

Kulisch extends the previously presented accurate accumulator to accurate dot products, which are exactly computed in his proposal [234, 235]. The idea is simply to accumulate the exact results of all the multiplications. To this purpose, instead of standard multipliers, we use *exact* multipliers that return all the bits of the exact product: for p-bit input significands, these multipliers return a $2p$-bit significand floating-point number. The exponent range is also doubled, which means adding one bit to w_E. Such multipliers incur no round-ing error, and are actually *cheaper* to build than the standard (w_E, p) ones. Indeed, the latter also have to compute $2p$ bits of the result, and in addition have to round it. In the exact floating-point multiplier, we save all the round-ing logic altogether. Results do not even need to be normalized, as they will be immediately sent to the fixed-point accumulator. The only additional cost is in the accumulator, which requires a larger input shifter (see Figure 9.19).

With these exact multipliers, if we are able to bound the exponents of the inputs so as to obtain a reasonably small accumulator, it becomes easy to prove that the whole dot-product process is exact, whatever the dimension of the input vectors. Otherwise, it is just as easy to provide accuracy bounds which are better than standard, and arbitrarily small. As matrix-vector and matrix-matrix products are parallel instances of dot products, these advan-tages extend to such operations.

9.7.5 Coarser-grain operators

If a sequence of floating-point operations is central to a given computation, it is often possible to design a specific operator for this sequence.

The successful recipe for such designs will be to perform as much as possible of the computation in fixed point. If the compound operator is proven to always compute more accurately than the succession of elementary operators, it is likely to be accepted. Another possible requirement may be to provide results guaranteed to be faithful (1 ulp accurate) with respect to the exact result.

The real question is: When do we stop? Which of these optimized operators are sufficiently general and offer sufficient optimization potential to justify that they are included in a library? There is a very pragmatic answer to this question: as soon as an operator is designed for a given application, it may be placed in a library. From there on, other applications will be able to use it. Again, this approach is very specific to the FPGA paradigm. In the CPU world, adding a hardware operator to an existing processor line must be backed by a lot of benchmarking showing that the cost does not outweigh the benefit. Simply take the example of division: Is a hardware divider wasted silicon in a CPU? Roughly at the same time, Flynn et al. did such benchmarking to advocate hardware dividers [309], while the industry designed new instruction sets around the FMA and without hardware dividers.

We now list some of these compound operators in more or less detail, depending on the state of the art. We do not pretend to exhaustiveness. Many applications will bring in some compound operation worth investigating.

Algebraic operators

We have already mentioned Euclidean norm. Other useful classes of algebraic operators are operations on complex numbers and polynomial or rational evaluators.

2Sum and 2Mul for compensated algorithms

Let us discuss the hardware acceleration of the 2Sum Algorithm (Algorithm 4.4, page 130) and the Dekker product (Algorithm 4.7, page 135). These algorithms are used in the compensated summation techniques reviewed in Chapter 6. In an FPGA, the application-specific approach presented in Section 9.7.4 should make more sense if it is applicable. What we discuss here is a prospective processor enhancement. Still, even for FPGA applications, compensated algorithms have several advantages. They require less error analysis, they scale better to problem sizes unknown *a priori*, and they are software compatible.

These approaches rely on two basic blocks called 2Sum and 2Mul that respectively compute the exact sum and the exact product of two floating-point numbers (see Figure 9.21). In both cases, as Chapter 4 has shown, the result fits in the unevaluated sum of two floating-point numbers. In a processor, these 2Sum and 2Mul operators require long sequences of standard
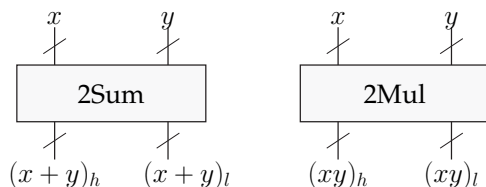
Figure 9.21: The 2Sum and 2Mul operators.

floating-point additions and multiplications: 3 to 6 floating-point additions for an exact sum, depending on the context (see the Fast2Sum and 2Sum algorithms in Section 4.3), and between 2 and 17 operations for the exact multiplication, depending on the availability of an FMA (see Section 4.4). Besides, most of these operations are data dependent, preventing an efficient use of the pipeline.

This is very inefficient: indeed, the information needed to build the lower part of the result is computed during the initial sum or product, but it is then discarded in the rounding process and has to be recomputed. Therefore, Dieter et al. [117] suggested the following modification to floating-point units. First, an additional p-bit register receives the residual bits that, in classical implementations of addition and multiplication, are used only to compute the sticky bits. Second, an additional instruction converts this residual register into a normal floating-point register. Filling the residual register involves no additional work with respect to a classical floating-point operator. Specifically, a floating-point multiplier has to compute the full $2p$-bit significand product to determine rounding. A floating-point adder does not perform the full addition, but the lower bits are untouched by the addition, so no additional computation is needed to recover them. Then the residual register holds an unnormalized significand. It is normalized, only when needed, by the copy instruction, which uses the LZC and shifter available in the adder. Actually, the residual register also has to store a bit of information saying whether the rounded significand was the truncated significand or its successor, as this information propagates to the residual and may change its sign. Consider for example (in decimal for clarity), the product of the 4-digit numbers 6.789 and 5.678. The product $3.8547942 \cdot 10^1$ may be represented by the unevaluated sum $3.854 \cdot 10^1 + 7.942 \cdot 10^{-3}$, but 2Sum and 2Mul return in their higher part the correct rounding of the exact result. Therefore, the product returned should be $3.855 \cdot 10^1 - 2.058 \cdot 10^{-3}$ (the reader can check that the sum is the same). The details may be found in [117].

With this approach, 2Sum and 2Mul now only cost two instructions each, for an area overhead of less than 10%. The authors of [117] were concerned with the binary32 units of GPUs, but the same approach could be applied to processor FPUs.

In an FPGA implementation of the 2Sum and 2Mul hardware operators, one probably wants a fully pipelined operator suitable for fully pipelined compensated algorithm implementations. Therefore, reusing the normalization hardware of the adder is not an option. The area and delay of these operators will still be less than a factor 2 of the standard ones. In terms of raw performance, comparing with the cost of software implementations of 2Mul and 2Sum on a processor, we conclude that, on algorithms based on these operators such as those presented in Chapter 6, the FPGA would recover the performance gap due to its 5 to 10 times slower frequency.

Similar improvements could probably be brought to the basic blocks used in the architecture of DeHon and Kapre [107] that computes a parallel sum of floating-point numbers which is bit-compatible with the sum obtained in a sequential order.

Elementary and compound functions

Some recent work has been dedicated to floating-point elementary function computation by FPGAs (exp and log in single precision [119, 114] and double precision [116], trigonometric functions in single precision [319, 113], and the power function in single precision [121]). The goal of such works is to design specific, combinatorial architectures based on fixed-point computations for such functions. These architectures can then be pipelined arbitrarily to run at the typical frequency of the target FPGA, with latencies that can be quite long (up to several tens of cycles for double precision), but with a throughput of one elementary function per cycle. The area of the state-of-the-art architecture is comparable to that of a few multipliers. As these pipelined architectures compute one result per cycle, the performance here is one order of magnitude better than that of a processor.

9.8   Probing Further

The reader wishing to examine actual designs will find several open source implementations of hardware FPUs in the OpenCores project.[2] For FPGAs, the FloPoCo project[3] from ENS-Lyon provides standard operators and explores nonstandard ones such as those listed in Section 9.7. Other open source floating-point implementations exist, in particular in the VFLOAT project from Northeastern University.[4]

---

[2]`http://www.opencores.org/`

[3]`http://www.ens-lyon.fr/LIP/Arenaire/Ware/FloPoCo/`

[4]`http://www.ece.neu.edu/groups/rpl/projects/floatingpoint/`

Chapter 10

Software Implementation of Floating-Point Arithmetic

THE PREVIOUS CHAPTER has presented the basic paradigms used for implementing floating-point arithmetic in hardware. However, some processors may not have such dedicated hardware, mainly for cost reasons. When it is necessary to handle floating-point numbers on such processors, one solution is to implement floating-point arithmetic in software. The goal of this chapter is to describe a set of techniques for writing an efficient implementation of the five floating-point operators $+$, $-$, $\times$, $\div$, and $\sqrt{\cdot}$ by working exclusively on integer numbers.

We will focus on algorithms for radix-2 floating-point arithmetic and provide some C99 codes for the binary32 format, for which the usual floating-point parameters β (radix), k (width), p (precision), and $e_{\max}$ (maximum exponent), are

$$\beta = 2, \qquad k = 32, \qquad p = 24, \qquad e_{\max} = 127.$$

Details will be given on how to implement some key specifications of the IEEE 754 standard for each arithmetic operator. How should we implement special values? How should we implement correct rounding? For simplicity, we restrict ourselves here to "rounding to nearest even" (called roundTiesTo-Even in IEEE 754-2008 [187, §4.3.1]), but other rounding direction attributes can be implemented using very similar techniques. Note also that our codes will not handle exceptions.

Except for Listings 10.24 and 10.29, the codes we give in this chapter are taken from the FLIP software library.[1] For other software implementations for radix 2, we refer especially to Hauser's SoftFloat library [175].[2] More recently,

---

[1]FLIP (Floating-point library for integer processors) can be downloaded at `http://flip.gforge.inria.fr/` and is designed by the Arénaire project of CNRS, Université de Lyon, and Inria.

[2]SoftFloat is available at `http://www.jhauser.us/arithmetic/SoftFloat.html`.

some software for radix-10 floating-point arithmetic has also been released. It will not be covered here, but the interested reader may refer to [85, 87].

10.1   Implementation Context

10.1.1   Standard encoding of binary floating-point data

Assume that the width of the floating-point format is k, and that the binary floating-point numbers are represented according to one of the basic formats specified by the IEEE 754 standard. As explained in Chapter 3, the encoding chosen for the standard allows one to compare floating-point numbers as if they were integers. We will make use of that property, and frequently manipulate the floating-point representations as if they were representations of integers. This is the "standard encoding" into k-bit unsigned integers [187, §3.4].

For a nonzero finite binary floating-point number

$$x = (-1)^{s_x} \cdot m_x \cdot 2^{e_x},$$

the standard encoding is the integer X represented by the bit string $X_{k-1} \ldots X_0$ such that

$$X = \sum_{i=0}^{k-1} X_i 2^i. \tag{10.1}$$

Here the X_i are defined from the following splitting of the bit string into three fields:

- **Sign bit:** $X_{k-1} = s_x$;

- **Biased exponent:**

$$\sum_{i=0}^{k-p-1} X_{i+p-1} 2^i \;\; = \;\; E_x \tag{10.2}$$
$$= \;\; e_x - e_{\min} + n_x,$$

  where

$$e_{\min} = 1 - e_{\max}$$

  and where n_x is the "is normal" bit[3] of x:

$$n_x = \begin{cases} 1 & \text{if } x \text{ is normal,} \\ 0 & \text{if } x \text{ is subnormal;} \end{cases} \tag{10.3}$$

- **Fraction bits:** $X_i = m_{x,p-i-1}$ for $i = 0, \ldots, p-2$.

---

[3]That bit is not actually stored in the floating-point representation. It is easily obtained from the exponent field, and we will use it many times.

Notice that E_x is indeed nonnegative, for $e_{\min} \leq e_x \leq e_{\max}$ and $n_x \geq 0$. Also, this encoding allows us to represent zeros, infinities, and Not a Number data (NaNs). For the binary32 format, this is made clear in Table 10.1. The table displays the integer value (or range) of the encoding as well as its bit string. In this chapter, both will be used heavily, leading to fast code sequences; for similar implementation tricks in a different context (using SSE2 instructions on the Intel IA-32 architecture) see [12].

Of course, the same encoding will be used for the other operand y (if any) and for the result r of an operation involving x and/or y. Consequently, implementing, say, a multiplication operator for $\circ = \mathrm{RN}$ and the binary32 format means writing a C function of the form

```
uint32_t RN_binary32_mul(uint32_t X, uint32_t Y) { ... }
```

which returns an unsigned 32-bit integer R encoding the multiplication result r. The goal of Section 10.3 will be precisely to show how the core { ... } of this function can be implemented in C using exclusively 32-bit integer arithmetic.

10.1.2 Available integer operators

Concerning the operations on input or intermediate variables, we assume available the basic arithmetic and logical operators

```
+,  -,  <<,  >>,  &,  |,  ^,
```

etc. We assume further that we have a fast way of computing the following functions:

- **Maximum or minimum of two unsigned integers:** $\max(A, B)$ and $\min(A, B)$ for $A, B \in \{0, \ldots, 2^k - 1\}$. These functions will be written

  ```
  maxu,  minu
  ```

  in all our C codes.

- **Maximum or minimum of two signed integers:** $\max(A, B)$ and $\min(A, B)$ for $A, B \in \{-2^{k-1}, \ldots, 2^{k-1} - 1\}$. These functions will be written

  ```
  max,  min
  ```

  in all our C codes.

- **Number of leading zeros of an unsigned integer:** This function counts the number of leftmost zeros of the bit string of $A \in \{0, \ldots, 2^k - 1\}$.

Value or range of integer X	Floating-point datum x	Bit string $X_{31}\ldots X_0$
0	$+0$	$00000000000000000000000000000000$
$(0, 2^{23})$	positive subnormal number	$000000000\,X_{22}\ldots X_0$ with some $X_i = 1$
$[2^{23}, 2^{31} - 2^{23})$	positive normal number	$0\,\underbrace{X_{30}X_{29}X_{28}X_{27}X_{26}X_{25}X_{24}X_{23}}_{\text{not all ones, not all zeros}}\,X_{22}\ldots X_0$
$2^{31} - 2^{23}$	$+\infty$	$01111111100000000000000000000000$
$(2^{31} - 2^{23}, 2^{31} - 2^{22})$	sNaN	$011111110\,X_{21}\ldots X_0$ with some $X_i = 1$
$[2^{31} - 2^{22}, 2^{31})$	qNaN	$0111111111\,X_{21}\ldots X_0$
2^{31}	-0	$10000000000000000000000000000000$
$(2^{31}, 2^{31} + 2^{23})$	negative subnormal number	$100000000\,X_{22}\ldots X_0$ with some $X_i = 1$
$[2^{31} + 2^{23}, 2^{32} - 2^{23})$	negative normal number	$1\,\underbrace{X_{30}X_{29}X_{28}X_{27}X_{26}X_{25}X_{24}X_{23}}_{\text{not all ones, not all zeros}}\,X_{22}\ldots X_0$
$2^{32} - 2^{23}$	$-\infty$	$11111111100000000000000000000000$
$(2^{32} - 2^{23}, 2^{32} - 2^{22})$	sNaN	$111111110\,X_{21}\ldots X_0$ with some $X_i = 1$
$[2^{32} - 2^{22}, 2^{32})$	qNaN	$1111111111\,X_{21}\ldots X_0$

Table 10.1: Binary32 datum x and its encoding into integer $X = \sum_{i=0}^{31} X_i 2^i$ (see [197]).

- **Lower half of the product of two unsigned integers:** This function computes $AB \bmod 2^k$, for $A, B \in \{0, \ldots, 2^k - 1\}$.

- **Upper half of the product of two unsigned integers:** This function computes $\lfloor AB/2^k \rfloor$, where $\lfloor \cdot \rfloor$ denotes the usual floor function.

The last three functions will be written, respectively,

$$\texttt{nlz}, \quad *, \quad \texttt{mul}$$

in all our C codes.

Some typical C99 implementations of the `maxu`, `minu`, `max`, `min`, `mul`, and `nlz` operators are given in Listings 10.1 through 10.4 for $k = 32$.

C listing 10.1 Implementation of the `maxu` and `minu` operators for $k = 32$.

```
uint32_t maxu(uint32_t A, uint32_t B)
{ return A > B ? A : B; }

uint32_t minu(uint32_t A, uint32_t B)
{ return A < B ? A : B; }
```

C listing 10.2 Implementation of the `max` and `min` operators for $k = 32$.

```
int32_t max(int32_t A, int32_t B)
{ return A > B ? A : B; }

int32_t min(int32_t A, int32_t B)
{ return A < B ? A : B; }
```

C listing 10.3 Implementation of the `mul` operator for $k = 32$.

```
uint32_t mul(uint32_t A, uint32_t B)
{
  uint64_t t0, t1, t2;

  t0 = A;
  t1 = B;
  t2 = (t0 * t1) >> 32;
  return t2;
}
```

C listing 10.4 Implementation of the `nlz` operator for $k = 32$.

```c
uint32_t nlz(uint32_t X)
{
 uint32_t Z = 0;

 if (X == 0) return(32);
 if (X <= 0x0000FFFF) {Z = Z +16; X = X <<16;}
 if (X <= 0x00FFFFFF) {Z = Z + 8; X = X << 8;}
 if (X <= 0x0FFFFFFF) {Z = Z + 4; X = X << 4;}
 if (X <= 0x3FFFFFFF) {Z = Z + 2; X = X << 2;}
 if (X <= 0x7FFFFFFF) {Z = Z + 1;}
 return Z;
}
```

Therefore, in principle all that is needed to run our implementation examples is a C99 compiler that supports 32-bit integer arithmetic. However, optimizations (to achieve low latency) depend on the features of the targeted architectures and compilers. Some of these features will be described in Section 10.1.4.

Before that, we shall give in the next section a few examples that show how one can implement some basic building blocks (needed in several places later in this chapter) by means of some of the integer operators we have given above.

10.1.3   First examples

For $k = 32$, given an integer X as in (10.1), we consider here three sub-tasks. How does one deduce E_x as in (10.2)? How shall we deduce n_x as in (10.3)? Assuming that x is (sub)normal and defining λ_x as the number of leading zeros of the binary expansion of the significand m_x, that is,

$$m_x = [\underbrace{0.0 \ldots 0}_{\lambda_x \text{ zeros}} 1 m_{x,\lambda_x+1} \ldots m_{x,23}], \tag{10.4}$$

how does one compute that number λ_x?

Extracting the exponent field

Since by (10.2) the bit string of E_x consists of the bits $X_{30}, \ldots, X_{23}$, a left shift by one position (to remove the sign bit X_{31}) followed by a right shift by 24 positions (to remove in particular the fraction bits $X_{22}, \ldots, X_0$) will give us what we want. This can be implemented as in Listing 10.5.

C listing 10.5 Computation of the biased exponent E_x of a binary32 datum x.

```
uint32_t Ex;

Ex = (X << 1) >> 24;
```

Of course, an alternative to the left shift would be to mask the sign bit by taking the bitwise AND of X and

$$2^{31} - 1 = (0\underbrace{1111111111111111111111111111111}_{31 \text{ ones}})_2 = (7FFFFFFF)_{16}.$$

This alternative is implemented in Listing 10.6. In what follows we will often interpret the fact of masking the sign bit as taking the "absolute value" of X (even if X encodes a NaN), and we will write $|X|$ for the corresponding integer (which is in fact simply $X \bmod 2^{31}$) and `absX` for the corresponding variable. This variable `absX` will turn out to be extremely useful for implementing addition, multiplication, and division (see Sections 10.2, 10.3, and 10.4).

C listing 10.6 Another way of computing the biased exponent E_x of a binary32 datum x.

```
uint32_t absX, Ex;

absX = X & 0x7FFFFFFF;
Ex = absX >> 23;
```

Computing the "is normal" bit

Assume that X encodes a (sub)normal binary floating-point number x and recall that the "is normal" bit of x is defined by (10.3). Once `absX` has been computed (see Listing 10.6), we deduce from Table 10.1 that x is normal if and only if `absX` is at least $2^{23} = (800000)_{16}$. Hence the code in Listing 10.7.

C listing 10.7 For a (sub)normal binary32 number x, computation of its "is normal" bit n_x.

```
uint32_t absX, nx;

absX = X & 0x7FFFFFFF;
nx = absX >= 0x800000;
```

Computing the number of leading zeros of a significand

Assume again that X encodes a (sub)normal binary floating-point number $x = (-1)^{s_x} \cdot m_x \cdot 2^{e_x}$ and recall that the number of leading zeros of m_x is $\lambda_x \geq 0$ as in (10.4).

Let us define L_X as the number of leading zeros of $|X|$. Clearly, since the leading bit of $|X|$ is zero, one has $L_X \geq 1$. On the other hand, since x is nonzero, $|X|$ is nonzero as well, and one has $L_X \leq 31$. More interestingly, one can show (see [197]) that λ_x is related to L_X as follows:

$$\lambda_x = M_X - w, \qquad M_X = \max(L_X, w),$$

where $w = k - p$ is the bit width of the biased exponent field. For the binary32 format, it follows from $k = 32$ and $p = 24$ that $w = 8$.

Consequently, one can implement the computation of λ_x by using the integer operators of Section 10.1.2. This is shown in Listing 10.8.

C listing 10.8 For a (sub)normal binary32 number x, computation of the number λ_x of leading zeros of the significand of x.

```
uint32_t absX, MX, lambdax;

absX = X & 0x7FFFFFFF;
MX = maxu(nlz(absX),8);
lambdax = MX - 8;
```

The number λ_x appears in some formulas used for computing the exponent of a product or a quotient in Sections 10.3 and 10.4. However, we will see there that these formulas also involve a constant term which can be updated in order to carry the -8 of identity $\lambda_x = M_X - 8$. Consequently, in such cases, λ_x itself is not needed and computing M_X will be enough.

10.1.4   Design choices and optimizations

We have already seen with the examples of Listings 10.5 and 10.6 that even a simple expression can be implemented in a variety of ways. Which one is best depends on the goal to achieve (low latency or high throughput, for example), as well as on some features of the targeted architecture and compiler.

The C codes given in the following sections have been written so as to expose some instruction-level parallelism (ILP). In fact, we will see that the algorithms used for implementing addition, multiplication, division, and square root very often lead in a fairly natural way to some code with relatively high ILP. For example, it is clear that multiplying two floating-point numbers can be done by, roughly, multiplying the significands and *simultaneously* adding the exponents. However, we shall show further that some ILP can already be exposed for performing the exponent addition itself.

Our codes have also been written with some particular architectural and compiler features in mind. These features are those of the ST231 VLIW[4] processor and compiler (see [340, §2] and [196, §II]). Concerning the processor, these features include the following.

---

[4]VLIW is an acronym for *very long instruction word*.

- Four instructions can be launched simultaneously at every cycle, with some restrictions (for example, at most two of them can be $*$ or `mul`).

- The latency of 32-bit integer arithmetic and logic is 1, except for $*$ and `mul`, whose latency is 3.

- Registers consist in 64 general-purpose registers and 8 condition registers.

- It is possible to encode immediate operands up to 32 bits.

- An efficient branch architecture is available, with multiple condition registers.

- Execution is predicated through `select` instructions.

The compiler features include:

- if-conversion optimization [57]: it generates mostly straight-line code by emitting efficient sequences of `select` instructions instead of costly control flow;

- the linear assembly optimizer (LAO) [98]: it generates a schedule for the instructions that is often very close to the optimal.

In practice, the codes we propose in the following sections are well suited to these features in the sense that when compiling for the ST231 architecture, the obtained latencies are indeed low (from 20 to 30 cycles, depending on the operator). For more performance results in this context, we refer to [196, 197, 198] and to the FLIP software library. [5]

10.2 Binary Floating-Point Addition

An implementation of the method of Section 8.3 for computing $\circ(x + y)$ will be described here for $\circ = \mathrm{RN}$ and the binary32 format:

$$\beta = 2, \qquad k = 32, \qquad p = 24, \qquad e_{\max} = 127.$$

The case where either x or y is a special datum (like ± 0, $\pm\infty$, or NaN) is discussed in Section 10.2.1, while the case where both x and y are (sub)normal numbers is described in Sections 10.2.2 through 10.2.4.

---

[5]See `http://flip.gforge.inria.fr/`.

10.2.1   Handling special values

In the case of addition, the input (x, y) is considered a *special input* when x or y is ± 0, $\pm\infty$, or NaN. For each possible case the IEEE 754-2008 standard requires that a special value be returned. When both x and y are zero, these special values are given by Table 8.1; when x or y is nonzero, they follow from Tables 8.2 and 8.3, page 247, by adjoining the correct sign, using

$$x + y = (-1)^{s_x} \cdot \Big(|x| + (-1)^{s_z} \cdot |y| \Big), \qquad s_z = s_x \text{ XOR } s_y. \qquad (10.5)$$

(Notice that the standard does not specify the sign of a NaN result; see [187, §6.3].)

   As said above, considering $|x|$ means setting the sign bit of X to zero, that is, considering $X \bmod 2^{k-1}$ instead of X. Recall that we use the notation

$$|X| := X \bmod 2^{k-1}. \qquad (10.6)$$

Detecting that a special value must be returned

Special inputs can be filtered out very easily using Table 10.2, which is a direct consequence of the standard binary encoding [187, §3.4]. Indeed, we deduce

Value or range of integer X	Floating-point datum x
0	$+0$
$(0, 2^{k-1} - 2^{p-1})$	positive (sub)normal number
$2^{k-1} - 2^{p-1}$	$+\infty$
$(2^{k-1} - 2^{p-1}, 2^{k-1} - 2^{p-2})$	sNaN
$[2^{k-1} - 2^{p-2}, 2^{k-1})$	qNaN

Table 10.2: Some floating-point data encoded by X.

from this table that (x, y) is a special input if and only if

$$|X| \text{ or } |Y| \text{ is in } \{0\} \cup [2^{k-1} - 2^{p-1}, 2^{k-1}). \qquad (10.7)$$

An equivalent formulation of (10.7) that allows us to use the max operator is

$$\max\Big((|X| - 1) \bmod 2^k, (|Y| - 1) \bmod 2^k \Big) \geq 2^{k-1} - 2^{p-1} - 1. \qquad (10.8)$$

For example, when $k = 32$ and $p = 24$, an implementation of (10.8) is given by lines 3, 4, 5 of Listing 10.9.

C listing 10.9 Special value handling in a binary32 addition operator.

```
 1  uint32_t absX, absY, absXm1, absYm1, Max, Sx, Sy;
 2
 3  absX = X & 0x7FFFFFFF;  absY = Y & 0x7FFFFFFF;
 4  absXm1 = absX - 1;       absYm1 = absY - 1;
 5  if (maxu(absXm1, absYm1) >= 0x7F7FFFFF)
 6  {
 7     Max = maxu(absX, absY); Sx = X & 0x80000000; Sy = Y & 0x80000000;
 8     if (Max > 0x7F800000 ||
 9        (Sx != Sy && absX == 0x7F800000 && absY == 0x7F800000))
10        return 0x7FC00000 | Max;          // qNaN with payload equal to
11                                          // the last 22 bits of X or Y
12     if (absX > absY) return X;
13     else if (absX < absY) return Y;
14     else return X & Y;
15  }
```

Returning special values as recommended or required by IEEE 754-2008

Once our input (x, y) is known to be special, one must return the corresponding result as specified in Tables 8.2 and 8.3, page 247. First, a quiet NaN (qNaN) must be returned as soon as one of the following two situations occurs:

- If $|X|$ or $|Y|$ encodes a NaN, that is, according to Table 10.2, if

$$\max\left(|X|, |Y|\right) > 2^{k-1} - 2^{p-1}; \tag{10.9}$$

- If $s_z = 1$ and if both $|X|$ and $|Y|$ encode $+\infty$, that is, if

$$X_{k-1} \text{ XOR } Y_{k-1} = 1 \quad \text{and} \quad |X| = |Y| = 2^{k-1} - 2^{p-1}. \tag{10.10}$$

When $k = 32$ and $p = 24$, one has $2^{k-1} - 2^{p-1} = 2^{31} - 2^{23} = (7F800000)_{16}$. Therefore, the conditions in (10.9) and (10.10) can be implemented as in lines 7, 8, 9 of Listing 10.9.

Let us raise here a few remarks about the qNaN we return at line 10 of Listing 10.9. Since $(7FC00000)_{16} = 2^{31} - 2^{22}$, the bit string of this qNaN has the form

$$0\underbrace{111111111}_{9 \text{ ones}}\underbrace{Z_{21}\ldots Z_0}_{\text{payload}},$$

where the string $Z_{21}\ldots Z_0$ is either $X_{21}\ldots X_0$ or $Y_{21}\ldots Y_0$ (we feel free to set the sign bit to zero because, as stated above, the standard does not specify the sign of a NaN result). That particular qNaN carries the last 22 bits of one of the operand encodings. So, in a sense, this follows the IEEE 754-2008 recommendation that, in order "to facilitate propagation of diagnostic information

contained in NaNs, as much of that information as possible should be pre-
served in NaN results of operations" (see [187, §6.2]). More precisely, if only
one of the inputs is NaN, then its payload is propagated (by means of the
bitwise OR involving the variable Max at line 10), as recommended in [187,
§6.2.3]); if both inputs x and y are NaN, then the payload of one of them is
propagated, again as recommended in [187, §6.2.3]).

Assume now that the case of a qNaN result has been handled, and let us
focus on the remaining special values by inspecting three cases:

- If $|X| > |Y|$ then, using Tables 8.2 and 8.3, page 247, together with (10.5),
  we must return $(-1)^{s_x} \cdot |x| = x$.

- If $|X| < |Y|$ then we return $(-1)^{s_x} \cdot |y|$ if $s_z = 0$, and $(-1)^{s_x} \cdot (-|y|)$ if
  $s_z = 1$, that is, y in both cases.

- If $|X| = |Y|$ then both x and y are either zero or infinity. However,
  since at this stage qNaN results have already been handled, the only
  possible inputs to addition are $(\pm 0, \pm 0)$, $(+\infty, +\infty)$, and $(-\infty, -\infty)$.
  Using Table 8.1, page 247, for $\mathrm{RN}(x+y)$ shows that for all these inputs
  the result is given by the bitwise AND of X and Y.

An example of implementation of these three cases is given for the binary32
format at lines 12, 13, 14 of Listing 10.9.

10.2.2   Computing the sign of the result

We assume from now on that the input (x, y) is not *special; that is, both x and y are
finite nonzero (sub)normal numbers.*

Recalling that $s_z = s_x$ XOR s_y, one has

$$x + y = (-1)^{s_x} \cdot \left(|x| + (-1)^{s_z} \cdot |y| \right) = (-1)^{s_y} \cdot \left((-1)^{s_z} \cdot |x| + |y| \right). \quad (10.11)$$

Hence, the sign of the exact result $x + y$ is s_x if $|x| > |y|$, and s_y if $|x| < |y|$.
What about the particular case where $|x| = |y|$ and $s_x \neq s_y$?[6] In that case
$x + y = \pm 0$, and the standard prescribes that $\circ(x+y) = +0$ "in all rounding-
direction attributes except roundTowardNegative; under that attribute, the
sign of an exact zero sum (or difference) shall be -0." (See [187, §6.3].)

Recalling that $\mathrm{RN}(x) \geq 0$ if and only if $x \geq 0$, we conclude that the sign
s_r to be returned is given by

$$s_r = \begin{cases} s_x & \text{if } |x| > |y|, \\ s_y & \text{if } |x| < |y|, \\ s_x = s_y & \text{if } |x| = |y| \text{ and } s_z = 0, \\ 0 & \text{if } |x| = |y| \text{ and } s_z = 1. \end{cases}$$

---

[6]We remind the reader that we have assumed at the beginning of this section that x and y
are nonzero.

Now, because of the standard encoding of binary floating-point data, the condition $|x| > |y|$ is equivalent to $|X| > |Y|$. Consequently, the computation of s_r for the binary32 format can be implemented as shown in Listing 10.10, assuming s_x, s_y, $|X|$, and $|Y|$ are available (all of them have been computed, e.g., in Listing 10.9 when handling special values due to special inputs).

C listing 10.10 Sign computation in a binary32 addition operator, assuming rounding to nearest ($\circ = \mathrm{RN}$) and that s_x, s_y, $|X|$, and $|Y|$ are available.

```
1  uint32_t Sr;
2
3  if (absX > absY)
4      Sr = Sx;
5  else if (absX < absY)
6      Sr = Sy;
7  else
8      Sr = minu(Sx,Sy);
```

In fact, Listing 10.10 can be used even if the input (x, y) is special. Hence, if special values are handled *after* the computation of s_r, one can reuse the variables Sr and Max, and replace lines 12, 13, 14 of Listing 10.9 with

```
return Sr | Max;
```

10.2.3 Swapping the operands and computing the alignment shift

As explained in Section 8.3, page 246, one may classically compare the exponents e_x and e_y and then, if necessary, swap x and y in order to ensure $e_x \geq e_y$. It turns out that in radix 2 comparing $|x|$ and $|y|$ is enough (and is cheaper in our particular context of a software implementation), as we will see now.

Recalling that $\mathrm{RN}(-x) = -\mathrm{RN}(x)$ (symmetry of rounding to nearest) and that $s_z = s_x$ XOR s_y, we deduce from the identities in (10.11) that the correctly rounded sum satisfies

$$
\begin{aligned}
\mathrm{RN}(x + y) &= (-1)^{s_x} \cdot \mathrm{RN}\Big(|x| + (-1)^{s_z} \cdot |y|\Big) \\
&= (-1)^{s_y} \cdot \mathrm{RN}\Big((-1)^{s_z} \cdot |x| + |y|\Big).
\end{aligned}
$$

Thus, it is natural to first ensure that

$$
|x| \geq |y|,
$$

and then compute only one of the two correctly rounded values above, namely,

$$
\mathrm{RN}\Big(|x| + (-1)^{s_z} \cdot |y|\Big).
$$

Since $|x| = m_x \cdot 2^{e_x}$ and $|y| = m_y \cdot 2^{e_y}$, this value is in fact given by

$$r := \text{RN}\Big(|x| + (-1)^{s_z} \cdot |y|\Big) = \text{RN}(m_r \cdot 2^{e_x}),$$

where

$$m_r = m_x + (-1)^{s_z} \cdot m_y \cdot 2^{-\delta} \qquad\qquad (10.12)$$

and

$$\delta = e_x - e_y. \qquad\qquad (10.13)$$

Interestingly enough, the condition $|x| \geq |y|$ implies in particular that $e_x \geq e_y$. More precisely, we have the following property.

Property 18. *If $|x| \geq |y|$ then $m_r \geq 0$ and $\delta \geq 0$.*

Proof. Let us show first that $e_x < e_y$ implies $|x| < |y|$. If $e_x < e_y$ then $e_y \geq e_x + 1 > e_{\min}$. Thus, y must be a normal number, which implies that its significand satisfies $m_y \geq 1$. Since $m_x < 2$, one obtains $m_y \cdot 2^{e_y} \geq 2^{e_x+1} > m_x \cdot 2^{e_x}$.

Let $\delta = e_x - e_y$. We can show that $m_x - m_y \cdot 2^{-\delta}$ is non-negative by considering two cases:

- If $e_x = e_y$ then $\delta = 0$, and $|x| \geq |y|$ is equivalent to $m_x \geq m_y$.

- If $e_x > e_y$ then $\delta \geq 1$ and, reasoning as before, x must be normal. On the one hand, $\delta \geq 1$ and $m_y \in (0, 2)$ give $m_y \cdot 2^{-\delta} \in (0, 1)$. On the other hand, x being normal, one has $m_x \geq 1$. Therefore, $m_x - m_y \cdot 2^{-\delta} \geq 0$, which concludes the proof.

$\square$

Operand swap

To ensure that $|x| \geq |y|$, it suffices to replace the pair $(|x|, |y|)$ by the pair $(\max(|x|, |y|), \min(|x|, |y|))$. Using the `maxu` and `minu` operators and assuming that $|X|$ and $|Y|$ are available, an implementation for the binary32 format is straightforward, as shown at line 3 of Listing 10.11.

Alignment shift

For $|x| \geq |y|$, let us now compute the non-negative integer δ in (10.13) that is needed for shifting the significand m_y right by δ positions (in order to align it with the significand m_x as in (10.12)). Recall that n_x and n_y are the "is normal" bits of x and y (so that $n_x = 1$ if x is normal, and 0 if x is subnormal). Recall also that the biased exponents E_x and E_y of x and y satisfy

$$E_x = e_x - e_{\min} + n_x \qquad \text{and} \qquad E_y = e_y - e_{\min} + n_y.$$

Therefore, the shift δ in (10.13) is given by

$$\delta = (E_x - n_x) - (E_y - n_y).$$

For the binary32 format, this expression for δ can be implemented as shown at lines 4, 5, 6 of Listing 10.11. Note that E_x, n_x, E_y, and n_y can be computed in parallel, and that the differences $E_x - n_x$ and $E_y - n_y$ can then be computed in parallel too.

C listing 10.11 Operand swap and alignment shift computation in a binary32 addition operator, assuming $|X|$ and $|Y|$ are available.

```
1  uint32_t Max, Min, Ex, Ey, nx, ny, delta;
2
3  Max = maxu(absX, absY);   Min = minu(absX, absY);
4  Ex = Max >> 23;           Ey = Min >> 23;
5  nx = Max >= 0x800000 ;    ny = Min >= 0x800000;
6  delta = (Ex - nx) - (Ey - ny);
```

10.2.4   Getting the correctly rounded result

It remains to compute the correctly rounded value $r = \mathrm{RN}(m_r \cdot 2^{e_x})$, where m_r is defined by (10.12). Recalling that n_x and n_y are "is normal" bits, the binary expansions of m_x and $m_y \cdot 2^{-\delta}$ are, respectively,

$$
\begin{aligned}
m_x &= (n_x.m_{x,1}\ldots\ldots\ldots\ldots m_{x,p-1} \quad 0 \ldots\ldots\ldots 0)_2 \\
m_y \cdot 2^{-\delta} &= (\underbrace{0.0\ldots0}_{\delta \text{ zeros}} n_y m_{y,1}\ldots m_{y,p-1-\delta} m_{y,p-\delta}\ldots m_{y,p-1})_2.
\end{aligned}
$$

Therefore, the binary expansion of m_r must have the form

$$m_r = (cs_0.s_1\ldots s_{p-1}s_p\ldots s_{p+\delta-1})_2, \quad c \in \{0,1\}. \tag{10.14}$$

We see that m_r is defined by at most $p + \delta + 1$ bits. Note that $c = 1$ is due to a possible carry propagation when adding $m_y \cdot 2^{-\delta}$ to m_x. If no carry propagates during that addition, or in the case of subtraction, then $c = 0$.

A first easy case: $x = -y \neq 0$

This case, for which it suffices to return $+0$ when $\circ = \mathrm{RN}$, will not be covered by the general implementation described later (the exponent of the result would not always be set to zero). Yet, it can be handled straightforwardly once $|X|$, $|Y|$, and s_z are available.

C listing 10.12 Addition for the binary32 format in the case where $x = -y \neq 0$ and assuming rounding to nearest ($\circ = \mathrm{RN}$).

```
if (absX == absY && Sz == 1) return 0;
```

A second easy case: both x and y are subnormal numbers

When $|x| \geq |y|$, this case occurs when $n_x = 0$. Indeed, if $n_x = 0$ then x is subnormal and therefore $|x| < 2^{e_{\min}}$. Now, the assumption $|x| \geq |y|$ clearly implies $|y| < 2^{e_{\min}}$ as well, which means that y is subnormal too.

In this case, x and y are such that

$$|x| = (0.m_{x,1} \ldots m_{x,p-1})_2 \cdot 2^{e_{\min}} \qquad \text{and} \qquad |y| = (0.m_{y,1} \ldots m_{y,p-1})_2 \cdot 2^{e_{\min}},$$

so that

$$\delta = 0$$

and

$$m_r = (0.m_{x,1} \ldots m_{x,p-1})_2 \pm (0.m_{y,1} \ldots m_{y,p-1})_2.$$

This is a fixed-point addition/subtraction and m_r can thus be computed exactly. Furthermore, one has $m_r \in [0, 2)$ and $e_x = e_{\min}$, so that $r = \mathrm{RN}(m_r \cdot 2^{e_x})$ is in fact given by

$$r = m_r \cdot 2^{e_{\min}},$$

with

$$m_r = (m_{r,0}.m_{r,1} \ldots m_{r,p-1})_2.$$

Note that the result r can be either normal ($m_{r,0} = 1$) or subnormal ($m_{r,0} = 0$).

The code in Listing 10.13 shows how this can be implemented for the binary32 format. Here we assume that $|x| \geq |y|$ and that the variables Max and Min encode, respectively, the integers $|X|$ and $|Y|$. We also assume that n_x (the "is normal" bit of x) and the operand and result signs s_x, s_y, s_r are available. (These quantities have been computed in Listings 10.9, 10.10, and 10.11.)

C listing 10.13 Addition for the binary32 format in the case where both operands are *subnormal* numbers.

```
1  uint32_t Sz, Mx, My, compMy, Mr;
2  // Assume x and y are non-special and such that |x| >= |y|.
3  if (nx == 0)
4  {
5      Sz = (Sx != Sy);
6      compMy = (My ^ (0 - Sz)) + Sz;
7      Mr = Mx + compMy;
8      return Sr | Mr;
9  }
```

Since both x and y are subnormal, the bit strings of Mx and My in Listing 10.13 are

$$[0\;\underbrace{00000000}_{8\ \text{zeros}}m_{x,1} \ldots m_{x,23}] \qquad \text{and} \qquad [0\;\underbrace{00000000}_{8\ \text{zeros}}m_{y,1} \ldots m_{y,23}].$$

Recalling that $\delta = 0$, we see that it suffices to encode $(-1)^{s_z} \cdot m_y$. This is achieved by computing `compMy`, using two's complement in the case of effective subtraction: `compMy` is equal to `My` if $s_z = 0$, and to `~My + 1` if $s_z = 1$, where "~" refers to the Boolean bitwise NOT function.

Notice also that the bit string of the returned integer is

$$[s_r \underbrace{0000000}_{7 \text{ zeros}} m_{r,0} m_{r,1} \ldots m_{r,23}].$$

Therefore, the correct biased exponent has been obtained automatically as a side effect of concatenating the result sign with the result significand. Indeed,

- if $m_{r,0} = 1$, then r is normal and has exponent $e_{\min}$, whose biased value is $e_{\min} + e_{\max} = 1$;

- if $m_{r,0} = 0$, then r is subnormal, and its biased exponent will be zero.

Implementation of the general case

Now that we have seen two preliminary easy cases, we can describe how to implement, for the binary32 format, the general algorithm recalled in Section 8.3, page 246 (note that our general implementation presented here will in fact cover the previous case where both x and y are subnormal numbers).

In the general case, because of significand alignment, the exact sum m_r in (10.14) has $p + \delta - 1$ fraction bits (which can be over 200 for the binary32 format). However, it is not necessary to compute m_r exactly in order to round correctly, and using $p - 1$ fraction bits together with three additional bits (called guard bits) is known to be enough (see, for example, [126, §8.4.3] as well as [340, §5]).

Thus, for the binary32 format, we can store m_x and m_y into the following bit strings of length 32 (where the last three bits will be used to update the guard bits):

$$[0\,0\,0\,0\,0\,n_x m_{x,1} \ldots m_{x,23}\,0\,0\,0]$$

and

$$[0\,0\,0\,0\,0\,n_y m_{y,1} \ldots m_{y,23}\,0\,0\,0].$$

This corresponds to the computation of integers `mx` and `my` at lines 3 and 4 of Listing 10.14.

C listing 10.14 Addition/subtraction of (aligned) significands in a binary32 addition operator.

```
 1  uint32_t mx, my, highY, lowY, highR;
 2
 3  mx = (nx << 26) | ((Max << 9) >> 6);
 4  my = (ny << 26) | ((Min << 9) >> 6);
 5
 6  highY = my >> minu(27, delta);
 7
 8  if (delta <= 3)
 9      lowY = 0;
10  else if (delta >= 27)
11      lowY = 1;
12  else
13      lowY = (my << (32 - delta)) != 0;
14
15  if (Sz == 0)
16      highR = mx + (highY | lowY);
17  else
18      highR = mx - (highY | lowY);
```

The leading bits of $m_y \cdot 2^{-\delta}$ are then stored into the 32-bit integer `highY` obtained by shifting `my` right by δ positions:

$$\texttt{highY} = [\underbrace{000\ldots000}_{5 + \delta \text{ zeros}} n_y m_{y,1} \ldots m_{y,26-\delta}].$$

We see that if $\delta \geq 27$, then all the bits of `highY` are zero: hence, the use of the `minu` instruction at line 6 of Listing 10.14. This prevents us from shifting by a value greater than 31, for which the behavior of the bitwise shift operators `>>` and `<<` is undefined.

The bits $m_{y,27-\delta}, \ldots, m_{y,23}$ of m_y that have been discarded by shifting are used to compute the sticky bit T. In fact, all we need is to know whether all of them are zero or not. This information about the lower part of $m_y \cdot 2^{-\delta}$ can thus be collected into an integer `lowY` as follows.

- If $\delta \leq 3$ then, because of the last three zeros of the bit string of `my`, none of the bits of m_y has been discarded and thus `lowY` is equal to 0.

- If $\delta \geq 27$ then all the bits of m_y have been discarded and, since at least one of them is nonzero because $m_y \neq 0$, we must have `lowY` equal to 1.

- If $\delta \in \{4, \ldots, 26\}$ then the last $\delta - 3$ bits of m_y can be extracted from the integer `my` by shifting it left by $32 - \delta$ positions.

A possible implementation of `lowY` is thus as in lines 8–13 of Listing 10.14.

Once the value 0 or 1 of `lowY` has been obtained, it is used to update the last bit (sticky bit position) of `highY`. Only then can the effective operation

(addition if $s_z = 0$, subtraction if $s_z = 1$) be performed. Since the bit string of lowY consists of 31 zeros followed by either a zero or a one, a possible implementation is as in lines 15–18 of Listing 10.14.

The bit string of the result highR is

$$\text{highR} = [0\,0\,0\,0\,c\,r_0r_1\ldots r_{23}r_{24}r_{25}r_{26}].$$

As stated in Section 8.3, page 246, normalization may now be necessary, either because of *carry propagation* ($c = 1$) or because of *cancellation* ($c = 0$ and $r_0 = \cdots = r_i = 0$ for some $i \geq 0$). Such situations, which further require that the tentative exponent e_x be adjusted, can be detected easily using the nlz instruction to count the number n of leading zeros of the bit string of highR shown above:

- If $n = 4$ then $c = 1$. In this case, the guard bit and the sticky bit are, respectively,

$$G = r_{23} \qquad \text{and} \qquad T = \text{OR}(r_{24}, r_{25}, r_{26}).$$

Consequently, $r = \text{RN}(m_r \cdot 2^{e_x})$ is obtained as

$$r = \left((1.r_0\ldots r_{22})_2 + B \cdot 2^{-23}\right) \cdot 2^{e_x+1}, \qquad (10.15)$$

where, for rounding to nearest, the bit B is defined as

$$B = G \text{ AND } \left(r_{22} \text{ OR } T\right),$$

(see for example [126, page 425]). An implementation of the computation of B from the integer highR is detailed in Listing 10.15. Notice that G and T can be computed in parallel (line 7). Note also that since the & operator is a *bitwise* operator and since G is either 0 or 1, the bit r_{22} need not be extracted explicitly. Instead, the whole significand

$$M = (1.r_0\ldots r_{22})_2 \cdot 2^{23} \qquad (10.16)$$

can be used to produce $B \in \{0, 1\}$ (line 8). The value of the integer M can also be computed simultaneously with the values of G and T.

C listing 10.15 Computation of the rounding bit in a binary32 addition operator, in the case of carry propagation ($c = 1$) and rounding to nearest ($\circ = \text{RN}$).

```
1  uint32_t n, M, G, T, B;
2
3  n = nlz(highR);
4
5  if (n == 4)
6  {
7     M = highR >> 4;      G = (highR >> 3) & 1;      T = (highR << 29) != 0;
8     B = G & (M | T);
9  }
```

- If $n = 5$ then $c = 0$ and $r_0 = 1$. In this case, the guard bit and the sticky bit are, respectively,

$$G = r_{24} \qquad \text{and} \qquad T = \text{OR}(r_{25}, r_{26}).$$

Consequently, $r = \text{RN}(m_r \cdot 2^{e_x})$ is obtained as

$$r = \left((1.r_1 \ldots r_{23})_2 + B \cdot 2^{-23} \right) \cdot 2^{e_x}, \qquad (10.17)$$

where, for rounding to nearest, the bit B is now defined as

$$B = G \text{ AND } \left(r_{23} \text{ OR } T \right).$$

An implementation can be obtained in the same way as the one of Listing 10.15. Note here that, unlike the previous case ($n = 4$), no normalization is necessary. Indeed, the tentative exponent e_x had to be adjusted to $e_x + 1$ in (10.15), while it is kept unchanged in (10.17) since neither carry propagation nor cancellation has occurred.

- If $n \geq 6$ then $c = r_0 = \cdots = r_i = 0$ for some $i \geq 0$. Hence, normalization is required, by shifting left the bits of highR until either the leading 1 reaches the position of r_0, or the exponent obtained by decrementing e_x reaches $e_{\min}$. Again, this can be implemented in the same fashion as in Listing 10.15. However, when $n \geq 7$, a simplification occurs since, as stated in Section 8.3, page 246, δ must be either 0 or 1 in this case, implying $G = T = 0$.

At this stage, the truncated normalized significand M and the rounding bit B have been computed, and only the result exponent is missing. It turns out that it can be deduced easily from e_x for each of the preceding cases: $n = 4$, $n = 5$, and $n \geq 6$.

For example, an implementation for the case $n = 4$ is given at line 5 of Listing 10.16. When $n = 4$, we can see in (10.15) that the result exponent before rounding is

$$d = e_x + 1.$$

Note that $d > e_{\min}$, so that the result r is either infinity or a normal number, for which the bias is $e_{\max}$. Since the integer M in (10.16) already contains the implicit 1, we will in fact not compute $D = d + e_{\max}$ but rather

$$D - 1 = e_x + e_{\max},$$

and then add M to $(D - 1) \cdot 2^{23}$. Recalling that $E_x = e_x - e_{\min} + n_x$ and that $e_{\min} + e_{\max} = 1$, the value $D - 1$ can be obtained as shown at line 5 and stored in the unsigned integer Dm1. Notice the parenthesizing, which allows us to reuse the value of $E_x - n_x$, already needed when computing the shift δ in Listing 10.11. Then two situations may occur.

- If `Dm1` is at least $2e_{\max} = 254 = (FE)_{16}$, then $e_x + 1 > e_{\max}$ because $e_{\min} = 1 - e_{\max}$. Consequently, $\pm\infty$ must be returned in this case (line 8).

- Otherwise, $e_x + 1$ is at most $e_{\max}$ and one can round and pack the result. The line 10 follows from (10.15), (10.16), and the standard encoding of binary32 data, and the considerations about rounding/computing a successor in Section 8.2, page 241. More precisely, if the addition of B to M propagates a carry up to the exponent field, then, necessarily, $B = 1$ and $M = 2^{24} - 1$. In this case, $(D - 1) \cdot 2^{23} + M + B$ equals $(D + 1) \cdot 2^{23}$ and the result exponent is not $e_x + 1$ but $e_x + 2$. (In particular, overflow will occur if $e_x = 126$.) Else, $M + B$ fits in 24 bits and then $(D - 1) \cdot 2^{23} + M + B$ encodes the floating-point number whose normalized representation is given in (10.15). One may check that in both cases the bit string of $(D - 1) \cdot 2^{23} + M + B$ has the form $[0 * \cdots *]$ and thus has no overlap with the bit string $[* 0 \cdots 0]$ of `Sr`.

C listing 10.16 Computation of the correctly rounded result in a binary32 addition operator, in the case of carry propagation ($c = 1$) and rounding to nearest ($\circ = \mathrm{RN}$).

```
 1   uint32_t Dm1;
 2
 3   if (n == 4)
 4   {
 5       Dm1 = (Ex - nx) + 1;
 6
 7       if ( Dm1 >= 0xFE ) // overflow
 8           return Sr | 0x7F800000;
 9       else
10           return ((Sr | (Dm1 << 23)) + M) + B;
11   }
```

10.3 Binary Floating-Point Multiplication

We now turn to the implementation of the method of Section 8.4, page 251, for computing $\circ(x \times y)$, again for $\circ = \mathrm{RN}$ and the binary32 format:

$$\beta = 2, \qquad k = 32, \qquad p = 24, \qquad e_{\max} = 127.$$

The case where either x or y is a special datum (like ± 0, $\pm\infty$, or NaN) is described in Section 10.3.1, while the case where both x and y are (sub)normal numbers is discussed in Sections 10.3.2 through 10.3.4.

10.3.1 Handling special values

In the case of multiplication, the input (x, y) is considered a *special input* when x or y is ± 0, $\pm\infty$, or NaN. For each possible case the IEEE 754-2008 standard

requires that a special value be returned. These special values follow from those given in Table 8.4, page 251, by adjoining the correct sign, using

$$x \times y = (-1)^{s_r} \cdot \Big(|x| \times |y| \Big), \qquad s_r = s_x \text{ XOR } s_y. \tag{10.18}$$

We remind the reader that the standard does not specify the sign of a NaN result (see [187, §6.3]).

Detecting that a special value must be returned

Special inputs can be filtered out exactly in the same way as for addition (see Section 10.2.1); hence, when $k = 32$ and $p = 24$, lines 3, 4, 5 of Listing 10.17. Here and hereafter $|X|$ will have the same meaning as in (10.6).

C listing 10.17 Special value handling in a binary32 multiplication operator.

```
 1  uint32_t absX, absY, Sr, absXm1, absYm1, Min, Max, Inf;
 2
 3  absX = X & 0x7FFFFFFF; absY = Y & 0x7FFFFFFF; Sr = (X ^ Y) & 0x80000000;
 4  absXm1 = absX - 1;      absYm1 = absY - 1;
 5  if (maxu(absXm1, absYm1) >= 0x7F7FFFFF)
 6  {
 7    Min = minu(absX, absY); Max = maxu(absX, absY); Inf = Sr | 0x7F800000;
 8    if (Max > 0x7F800000 || (Min == 0 && Max == 0x7F800000))
 9      return Inf | 0x00400000 | Max;    // qNaN with payload equal to
10                                        // the last 22 bits of X or Y
11    if (Max != 0x7F800000) return Sr;
12    return Inf;
13  }
```

Returning special values as recommended or required by IEEE 754-2008

Once the input (x, y) is known to be special, one must return the corresponding result as specified in Table 8.4, page 251. First, that table shows that a qNaN must be returned as soon as one of the following two situations occurs:

- if $|X|$ or $|Y|$ encodes a NaN, that is, according to Table 10.2, if

$$\max (|X|, |Y|) > 2^{k-1} - 2^{p-1}; \tag{10.19}$$

- if $(|X|, |Y|)$ encodes either $(+0, +\infty)$ or $(+\infty, +0)$, that is, if

$$\min(|X|, |Y|) = 0 \quad \text{and} \quad \max(|X|, |Y|) = 2^{k-1} - 2^{p-1}. \tag{10.20}$$

When $k = 32$ and $p = 24$, the conditions in (10.19) and (10.20) can be implemented as in lines 7 and 8 of Listing 10.17. Here, we must raise two remarks.

- The condition in (10.19) is the same as the one used in (10.9) for handling NaN results in binary floating-point addition. However, the condition in (10.20) is specific to multiplication.

- The qNaN returned at line 9 of Listing 10.17 enjoys the same nice properties as for binary floating-point addition. Roughly speaking, just as for addition, our code for multiplication returns a qNaN that keeps as much information on the input as possible, as recommended by IEEE 754-2008 (see [187, §6.2]).

Once the case of a qNaN output has been handled, it follows from Table 8.4, page 251, that a special output must be ± 0 if and only if neither x nor y is $\pm\infty$, that is, according to Table 10.2, if and only if

$$\max(|X|, |Y|) \neq 2^{k-1} - 2^{p-1}.$$

For $k = 32$ and $p = 24$, the latter condition is implemented at line 11 of Listing 10.17. Finally, the remaining case, for which one must return $(-1)^{s_r}\infty$, is handled by line 12.

10.3.2 Sign and exponent computation

We assume from now on that the input (x, y) is not special; that is, both x and y are finite nonzero (sub)normal numbers.

The sign s_r of the result is straightforwardly obtained by taking the XOR of the sign bits of X and Y. It has already been used in the previous section for handling special values (for an example, see variable Sr at line 3 of Listing 10.17).

Concerning the exponent of the result, let us first recall that using (10.18) together with the symmetry of rounding to nearest gives

$$\mathrm{RN}(x \times y) = (-1)^{s_r} \cdot \mathrm{RN}\Big(|x| \times |y|\Big).$$

Second, defining λ_x and λ_y as the numbers of leading zeros of the significands m_x and m_y, and defining further

$$m'_x = m_x \cdot 2^{\lambda_x} \qquad \text{and} \qquad m'_y = m_y \cdot 2^{\lambda_y}, \tag{10.21}$$

and

$$e'_x = e_x - \lambda_x \qquad \text{and} \qquad e'_y = e_y - \lambda_y, \tag{10.22}$$

we obtain a product expressed in terms of *normalized* significands:

$$\mathrm{RN}\Big(|x| \times |y|\Big) = \mathrm{RN}\Big(m'_x m'_y \cdot 2^{e'_x + e'_y}\Big), \qquad m'_x, m'_y \in [1, 2).$$

Third, taking

$$c = \begin{cases} 0 & \text{if } m'_x m'_y \in [1, 2), \\ 1 & \text{if } m'_x m'_y \in [2, 4), \end{cases} \tag{10.23}$$

one has

$$\mathrm{RN}\Big(|x| \times |y|\Big) = \mathrm{RN}(\ell \cdot 2^d),$$

with

$$\ell = m'_x m'_y \cdot 2^{-c} \qquad \text{and} \qquad d = e'_x + e'_y + c. \tag{10.24}$$

Here c allows one to ensure that ℓ lies in the range $[1, 2)$. Thus, computing the exponent should, in principle, mean computing the value of d as above. There are in fact two possible situations:

- if $d \geq e_{\min}$ then the real number $\ell \cdot 2^d$ lies in the normal range or in the overflow range, and therefore $\mathrm{RN}(\ell \cdot 2^d) = \mathrm{RN}(\ell) \cdot 2^d$;

- if $d < e_{\min}$, which may happen since both e'_x and e'_y can be as low as $e_{\min} - p + 1$, the real $\ell \cdot 2^d$ falls in the subnormal range. As explained in Section 8.4.2, in this case d should be increased up to $e_{\min}$ by shifting ℓ right by $e_{\min} - d$ positions.

For simplicity, here we will detail an implementation of the first case only:

$$d \geq e_{\min}. \tag{10.25}$$

The reader may refer to the FLIP software library for a complete implementation that handles both cases.

To compute the exponent d of the result, we shall as usual manipulate its biased value, which is the integer D such that

$$D = d + e_{\max}. \tag{10.26}$$

And yet, similarly to the implementation of binary floating-point addition (see for example Listing 10.16), we will in fact compute $D - 1$ and then let this tentative (biased) exponent value be adjusted automatically when rounding and packing. Recalling that $e_{\min} = 1 - e_{\max}$ and using (10.25), one has

$$D - 1 = d - e_{\min} \geq 0.$$

Computing the non-negative integer $D - 1$

Using (10.22) and (10.24) together with the fact that

$$E_x = e_x - e_{\min} + n_x,$$

one may check that

$$D - 1 = (E_x - n_x) + (E_y - n_y) - (\lambda_x + \lambda_y - e_{\min}) + c.$$

The identity

$$\lambda_x + \lambda_y - e_{\min} = M_X + M_Y - (2w + e_{\min}),$$

where $2w + e_{\min}$ is a constant defined by the floating-point format, shows further that computing λ_x and λ_y is in fact not needed.

In practice, among all the values involved in the preceding expression of $D - 1$, the value of condition $c = [m'_x m'_y \geq 2]$ may be the most expensive to determine. On the other hand, n_x, n_y, E_x, and E_y require 2 instructions, while M_X and M_Y require 3 instructions. Hence, recalling that $2w + e_{\min}$ is known *a priori*, a scheduling for the computation of $D - 1$ that exposes some ILP is given by the following parenthesizing:

$$D - 1 = \Big(\big[(E_x - n_x) + (E_y - n_y) \big] - \big[(M_X + M_Y) - (2w + e_{\min}) \big] \Big) + c. \quad (10.27)$$

For example, $w = 8$ and $e_{\min} = -126$ for the binary32 format. Then $2w + e_{\min} = -110$, and an implementation of the computation of $D - 1$ that uses (10.27) is thus as shown in Listing 10.18.

C listing 10.18 Computing $D - 1$ in a binary32 multiplication operator, assuming $d \geq e_{\min}$ and that $|X|$, $|Y|$, and c are available.

```
1   uint32_t Ex, Ey, nx, ny, MX, MY, Dm1;
2
3   Ex = absX >> 23;          Ey = absY >> 23;
4   nx = absX >= 0x800000;    ny = absY >= 0x800000;
5   MX = maxu(nlz(absX), 8);  MY = maxu(nlz(absY), 8);
6
7   Dm1 = (((Ex - nx) + (Ey - ny)) - ((MX + MY) + 110)) + c;
```

Notice that in Listing 10.18 n_x, n_y, E_x, E_y, M_X, and M_Y are independent of each other. Consequently, with unbounded parallelism and a latency of 1 for all the instructions involved in the code, the value of $D - 1 - c$ can be deduced from X and Y in 6 cycles. Then, once c has been computed, it can be added to that value in 1 cycle.

10.3.3 Overflow detection

As stated in Chapter 8, overflow can occur for multiplication only when $d \geq e_{\min}$. In this case one has $|x| \times |y| = \ell \cdot 2^d$, with $\ell \in [1, 2)$ given by (10.24).

Overflow before rounding

A first case of overflow is when $d \geq e_{\max} + 1$, since then the exact product $|x| \times |y|$ is larger than the largest finite number $\Omega = (2 - 2^{1-p}) \cdot 2^{e_{\max}}$, and so will be its rounded value. This first case can be detected easily from the value of $D - 1$, applying (10.26):

$$d \geq e_{\max} + 1 \quad \text{if and only if} \quad D - 1 \geq 2e_{\max}.$$

For the binary32 format, $2e_{\max} = 254 = (FE)_{16}$ and this first case of overflow can be implemented as shown in Listing 10.19.

C listing 10.19 Returning $\pm\infty$ when $d \geq e_{\max} + 1$ in the binary32 format.

```
    if (Dm1 >= 0xFE) return Inf;                    // Inf = Sr | 0x7F800000;
```

Notice that `Inf` has already been used for handling special values (see line 7 of Listing 10.17). Notice also the similarity with binary floating-point addition (see lines 7 and 8 of Listing 10.16).

Overflow after rounding

A second case of overflow is when $d = e_{\max}$ and $\mathrm{RN}(\ell) = 2$. This case is handled automatically when rounding the significand and packing the result via the integer addition

$$(D - 1) \cdot 2^{p-1} + \mathrm{RN}(\ell) \cdot 2^{p-1}$$

(see the end of Section 10.3.4). For example, for the binary32 format, when $D - 1 = 2e_{\max} - 1 = 253 = (11111101)_2$ and $\mathrm{RN}(\ell) = 2$, the bit string of $(D - 1) \cdot 2^{p-1}$ is

$$[0\,11111101\,\underbrace{00000000000000000000000}_{p-1\,=\,23\text{ bits}}],$$

and the bit string of $\mathrm{RN}(\ell) \cdot 2^{p-1} = 2 \cdot 2^{p-1}$ is

$$[0\,00000010\,\underbrace{00000000000000000000000}_{p-1\,=\,23\text{ bits}}].$$

Summing them gives the integer that encodes $+\infty$, and it remains to concatenate its bit string with the correct sign. The result will then be a correctly signed infinity, as required.

10.3.4   Getting the correctly rounded result

It remains to compute the correctly rounded value $r = \mathrm{RN}(\ell \cdot 2^d)$. As in the previous section, assume for simplicity that $d \geq e_{\min}$. Then one has

$$r = \mathrm{RN}(\ell) \cdot 2^d$$

and, since a biased value $D - 1$ of d has already been computed (see Listing 10.18), we are left with the computation of $\mathrm{RN}(\ell)$, with

$$\ell = m'_x m'_y \cdot 2^{-c}$$

as in (10.24). The following paragraphs explain how to implement this for the binary32 format.

Computing the normalized significands m'_x and m'_y

Recall that the integer λ_x is defined as the number of leading zeros of the binary expansion of the significand m_x:

$$m_x = [\underbrace{0.0\ldots 0}_{\lambda_x \text{ zeros}} 1 m_{x,\lambda_x+1}\ldots m_{x,23}].$$

Consequently, we start by storing the bits of m'_x into the following string of length 32:

$$[1\, m_{x,\lambda_x+1}\ldots m_{x,23}\,\underbrace{000\ldots 000}_{\lambda_x + 8 \text{ zeros}}]. \qquad (10.28)$$

If $\lambda_x = 0$ then this bit string is deduced from the bit string of $|X|$ by shifting it left by 8 positions and simultaneously introducing the implicit 1. If $\lambda_x > 0$ then it suffices to shift $|X|$ left by $8 + \lambda_x$ positions. In both cases, the amount of the shift is

$$M_X = \max(\texttt{nlz}(|X|), 8).$$

An implementation can be found at line 3 of Listing 10.20, where the bit string of the variable mpX is as in (10.28). The same is done for storing m'_y by means of variable mpY.

Computing the product $m'_x m'_y$ exactly

Since m'_x and m'_y can each be represented using at most 24 bits, their exact product can be represented using at most 48 bits. Those bits fit into two 32-bit integers, which are called highS and lowS, and are defined as follows:

$$m'_x m'_y = (\underbrace{cs_0.s_1\ldots s_{30}}_{=: \text{ highS}}\underbrace{s_{31}\ldots s_{46}\overbrace{000\ldots 000}^{16 \text{ zeros}}}_{=: \text{ lowS}})_2. \qquad (10.29)$$

Here $c = 0$ if $m'_x m'_y < 2$, and $c = 1$ if $m'_x m'_y \geq 2$. In the latter case, a carry has been propagated; thus, normalization will be necessary to obtain ℓ in $[1, 2)$.

The computation of the integers highS, lowS, and c is done at lines 5 and 6 of Listing 10.20, using our basic multiply instructions mul and * as well as the fact that $c = 1$ if and only if the integer highS is at least $2^{31} = (80000000)_{16}$.

Computing the guard and sticky bits needed for rounding correctly

Once the product $m'_x m'_y$ (and thus ℓ as well) is known exactly, one can deduce $\text{RN}(\ell)$ in the classical way, by means of a guard bit G and a sticky bit T. We now discuss how to compute those bits efficiently. We start by considering the cases $c = 0$ and $c = 1$ separately, and then propose a single code for handling both of them.

- If $c = 0$ then, since $m'_x m'_y \geq 1$, the bit s_0 in (10.29) must be equal to 1. Thus, combining (10.24) and (10.29) gives

$$\ell = (1.s_1 \ldots s_{46})_2,$$

from which it follows that the guard bit and the sticky bit are, respectively,

$$G = s_{24} \quad \text{and} \quad T = \text{OR}(s_{25}, \ldots, s_{46}).$$

Finally, the correctly rounded value $\text{RN}(\ell)$ is obtained as

$$\text{RN}(\ell) = (1.s_1 \ldots s_{23})_2 + B \cdot 2^{-23},$$

where, for rounding to nearest, the bit B is defined as

$$B = G \text{ AND } \left(s_{23} \text{ OR } T \right)$$

(see, for example, [126, page 425]). Using (10.29) we see that G can be obtained by shifting $\texttt{highS} = [01s_1 \ldots s_{24}s_{25} \ldots s_{30}]$ right by 6 positions and then masking with 1:

```
G = (highS >> 6) & 1;
```

On the other hand, $T = \text{OR}(s_{25}, \ldots, s_{30}, \texttt{lowS})$ by definition of $\texttt{lowS}$. Therefore, the value of T can be obtained as follows:

```
T = ((highS << 26) != 0) | (lowS != 0);
```

Clearly, the two comparisons to zero that appear in the computation of T can be done in parallel. Besides, given $\texttt{highS}$ and $\texttt{lowS}$, the computation of T itself can be performed in parallel with that of G.

- If $c = 1$ then (10.24) and (10.29) give

$$\ell = (1.s_0 s_1 \ldots s_{46})_2.$$

It follows that $\text{RN}(\ell) = (1.s_0 \ldots s_{22})_2 + B \cdot 2^{-23}$, where, for rounding to nearest, the bits B, G, and T are now given by

$$B = G \text{ AND } \left(s_{22} \text{ OR } T \right), \qquad G = s_{23}, \qquad T = \text{OR}(s_{24}, \ldots, s_{46}).$$

The computation of G and T can be implemented as before, the only difference being the values by which we shift:

```
G = (highS >> 7) & 1;        T = ((highS << 25) != 0) | (lowS != 0);
```

It is now clear that the two cases $c = 0$ and $c = 1$ that we have just analyzed can be handled by a single code fragment, which corresponds to lines 8, 10, 12 of Listing 10.20. As in Listing 10.15 for binary floating-point addition, computing B does not require us to extract the last bit (s_{23} if $c = 0$, s_{22} if $c = 1$) of the significand (stored in the variable M). Instead, we work directly on M.

Also, since highT will generally be more expensive to obtain than lowT and M, one can expose more ILP by ORing lowT and M during the computation of highT. This is done at line 10 of Listing 10.20, and is just one example of the many ways of optimizing this kind of code. The FLIP software library implements some other tricks that allow one to expose even more ILP and thus, in some contexts, to save a few cycles.

Rounding the significand and packing the result

Once the sign, the (biased) exponent, the significand, and the rounding bit have been computed (and are available in the integers Sr, Dm1, M, and B, respectively), one can round the significand and pack the result. This is done in the same way as in Listing 10.16 for binary floating-point addition; see line 14 of Listing 10.20.

C listing 10.20 Computation of the rounding bit and of the correctly rounded result in a binary32 multiplication operator, in the case of rounding to nearest ($\circ = \mathrm{RN}$) and assuming $|X|$, $|Y|$, M_X, and M_Y are available.

```
1  uint32_t mpX, mpY, highS, lowS, c, G, M, highT, lowT, M_OR_lowT, B;
2
3  mpX = (X << MX) | 0x80000000;        mpY = (Y << MY) | 0x80000000;
4
5  highS = mul(mpX, mpY);               lowS = mpX * mpY;
6  c = highS >= 0x80000000;             lowT = (lowS != 0);
7
8  G = (highS >> (6 + c)) & 1;          M = highS >> (7 + c);
9
10 highT = (highS << (26 - c)) != 0;    M_OR_lowT = M | lowT;
11
12 B = G & (M_OR_lowT | highT);
13
14 return ((Sr | (Dm1 << 23)) + M) + B;
```

10.4   Binary Floating-Point Division

This section describes how to implement the floating-point division of binary32 data. The main steps of the algorithm have been recalled in Section 8.6. When subnormal numbers are not supported, a complete description of a possible implementation can be found in [198].

The case where either x or y is a special datum (like ± 0, $\pm \infty$, or NaN) is described in Section 10.4.1, while the case where both x and y are (sub)normal numbers is discussed in Sections 10.4.2 through 10.4.4.

10.4.1   Handling special values

Similarly to addition and multiplication, the input (x, y) to division is considered a *special input* when x or y is ± 0, $\pm \infty$, or NaN. For each possible case the IEEE 754-2008 standard requires that a special value be returned by the division operator. These special values are obtained from those in Table 8.5, page 263, by adjoining the correct sign, using

$$x/y = (-1)^{s_r} \cdot \Big(|x|/|y| \Big), \qquad s_r = s_x \text{ XOR } s_y. \tag{10.30}$$

We remind the reader that the standard does not specify the sign of a NaN result; see [187, §6.3].

Detecting that a special value must be returned

For division, a special input (x, y) can be filtered out in the same way as for addition and multiplication. When $k = 32$ and $p = 24$, such a filter can be implemented as shown at lines 3, 4, 5 of Listing 10.21.

C listing 10.21 Special value handling in a binary32 division operator.

```
 1  uint32_t absX, absY, Sr, absXm1, absYm1, Max, Inf;
 2
 3  absX = X & 0x7FFFFFFF; absY = Y & 0x7FFFFFFF; Sr = (X ^ Y) & 0x80000000;
 4  absXm1 = absX - 1;      absYm1 = absY - 1;
 5  if (maxu(absXm1, absYm1) >= 0x7F7FFFFF)
 6  {
 7     Max = maxu(absX, absY); Inf = Sr | 0x7F800000;
 8     if (Max > 0x7F800000 || absX == absY)
 9        return Inf | 0x00400000 | Max;    // qNaN with payload equal to
10                                          // the last 22 bits of X or Y
11     if (absX < absY) return Sr;
12     return Inf;
13  }
```

Returning special values as recommended or required by IEEE 754-2008

Here and hereafter $|X|$ will have the same meaning as in (10.6). Once our input (x, y) is known to be special, one must return the corresponding result as specified in Table 8.5, page 263. From that table we see that a qNaN must be returned as soon as one of the following two situations occurs:

- if $|X|$ or $|Y|$ encodes a NaN, that is, according to Table 10.2, if

$$\max\left(|X|,|Y|\right) > 2^{k-1} - 2^{p-1}; \tag{10.31}$$

- if $(|X|,|Y|)$ encodes either $(+0,+0)$ or $(+\infty,+\infty)$, that is, since (x,y) is known to be special and assuming the case where x or y is NaN is already handled by (10.31), if

$$|X| = |Y|. \tag{10.32}$$

When $k = 32$ and $p = 24$, the conditions in (10.31) and (10.32) can be implemented as in lines 7 and 8 of Listing 10.21. One can raise the following remarks:

- The condition in (10.31) is the same as the one used for addition and multiplication (see (10.9) and (10.19)). On the contrary, the condition in (10.32) is specific to division.

- The qNaN returned at line 9 of Listing 10.21 enjoys the same nice properties as for binary floating-point addition and binary floating-point multiplication: in a sense, it keeps as much information on the input as possible, as recommended by IEEE 754-2008 (see [187, §6.2]).

Once the case of a qNaN output has been handled, it follows from Table 8.5, page 263, that a special output must be ± 0 if and only if $|x| < |y|$, that is, according to Table 10.2, if and only if

$$|X| < |Y|.$$

For $k = 32$ and $p = 24$, the latter condition is implemented at line 11 of Listing 10.21. Finally, the remaining case, for which one must return $(-1)^{s_r}\infty$, is handled by line 12.

10.4.2 Sign and exponent computation

We assume from now on that the input (x,y) is not *special; that is, that both x and y are finite nonzero (sub)normal numbers.*

Exactly as for multiplication, the sign s_r of the result is straightforwardly obtained by taking the XOR of the sign bits of X and Y. It has already been used in the previous section for handling special values (for an example, see variable Sr at line 3 of Listing 10.21).

Concerning the exponent of the result, the approach is also very similar to what we have done for multiplication in Section 10.3.2. First, using (10.30) together with the symmetry of rounding to nearest, we obtain

$$\mathrm{RN}(x/y) = (-1)^{s_r} \cdot \mathrm{RN}\Big(|x|/|y|\Big).$$

Then, using the same notation as in Section 10.3.2, we can express the fraction in terms of *normalized* significands:

$$\mathrm{RN}\Big(|x|/|y|\Big) = \mathrm{RN}\Big(m'_x/m'_y \cdot 2^{e'_x - e'_y}\Big), \qquad m'_x, m'_y \in [1, 2).$$

Finally, as in multiplication, we introduce a parameter c that will be used for normalizing the fraction m'_x/m'_y. Indeed, $1 \le m'_x, m'_y < 2$ implies that $m'_x/m'_y \in (1/2, 2)$. Hence, $2m'_x/m'_y \in (1, 4) \subset [1, 4)$ and so $2m'_x/m'_y \cdot 2^{-c} \in [1, 2)$, provided c satisfies

$$c = \begin{cases} 0 & \text{if } m'_x < m'_y, \\ 1 & \text{if } m'_x \ge m'_y. \end{cases} \tag{10.33}$$

It follows that the binary floating-point number $\mathrm{RN}(|x|/|y|)$ is given by

$$\mathrm{RN}\Big(|x|/|y|\Big) = \mathrm{RN}(\ell \cdot 2^d),$$

with

$$\ell = 2m'_x/m'_y \cdot 2^{-c} \qquad \text{and} \qquad d = e'_x - e'_y - 1 + c. \tag{10.34}$$

Since by definition $\ell \in [1, 2)$, the result exponent will thus be set to d.

As in multiplication, the following two situations may occur:

- if $d \ge e_{\min}$ then the real number $\ell \cdot 2^d$ lies in the normal range or the overflow range, and therefore

$$\mathrm{RN}(\ell \cdot 2^d) = \mathrm{RN}(\ell) \cdot 2^d; \tag{10.35}$$

- if $d < e_{\min}$, which may happen since e'_x can be as low as $e_{\min} - p + 1$ and e'_y can be as large as $e_{\max}$, the real number $\ell \cdot 2^d$ falls in the subnormal range. As explained in Section 8.6.3, several strategies are possible in this case, depending on the method chosen for the approximate computation of the rational number ℓ.

For simplicity, here we detail an implementation of the first case only:

$$d \ge e_{\min}. \tag{10.36}$$

The reader may refer to the FLIP software library for a complete implementation that handles both cases by means of polynomial approximation.

Note, however, that unlike multiplication the binary floating-point number $\mathrm{RN}(\ell)$ is always strictly less than 2. This fact is a direct consequence of the following property, shown in [198]:

Property 19. *If $m'_x \ge m'_y$ then $\ell \in [1, 2 - 2^{1-p}]$, else $\ell \in (1, 2 - 2^{1-p})$.*

When $d \geq e_{\min}$ it follows from this property that $\mathrm{RN}(\ell) \cdot 2^d$ is the normalized representation of the correctly rounded result sought.[7] Consequently, d is not simply a tentative exponent that would require updating after rounding, but is already the exponent of the result. We will come back to this point in Section 10.4.3 when considering the overflow exception.

Let us now compute d using (10.34). As usual, what we actually compute is the integer $D-1$ such that $D = d + e_{\max}$ (biased value of d). Since $e_{\min} = 1 - e_{\max}$, the assumption (10.36) gives, as in binary floating-point multiplication,

$$D - 1 = d - e_{\min} \geq 0. \tag{10.37}$$

Computing the non-negative integer $D - 1$

The approach is essentially the same as for multiplication, with d now defined by (10.34) instead of (10.24). Recalling that $E_x = e_x - e_{\min} + n_x$ and that $\lambda_x = M_X - w$, one obtains

$$D - 1 = (E_x - n_x) - (E_y - n_y) - (M_X - M_Y) + c - e_{\min} - 1. \tag{10.38}$$

Before parenthesizing this expression for $D - 1$, let us consider the computation of c. Interestingly enough, the value of c is in fact easier to obtain for division than for multiplication. Indeed, unlike (10.23), its definition in (10.33) does not involve the product $m'_x m'_y$, and it suffices to compute the normalized significands m'_x and m'_y, and to compare them to deduce the value of c. For the binary32 format, a possible implementation is as shown in Listing 10.22.

C listing 10.22 Computation of the normalized significands m'_x and m'_y and of the shift value c in a binary32 division operator.

```
1  uint32_t absX, absY, MX, MY, mpX, mpY, c;
2
3  absX = X & 0x7FFFFFFF;          absY = Y & 0x7FFFFFFF;
4  MX = maxu(nlz(absX), 8);        MY = maxu(nlz(absY), 8);
5  mpX = (X << MX) | 0x80000000;   mpY = (Y << MY) | 0x80000000;
6
7  c = mpX >= mpY;
```

With unbounded parallelism and a latency of 1 for each of the instructions in Listing 10.22, we see that c can be obtained in 6 cycles. In comparison, the shift value c for binary floating-point multiplication (obtained at line 6 of Listing 10.20) costs 6 cycles plus the latency of mul. For example, on the STMicroelectronics ST200 processor, which has four issues and where mul has

---

[7]Since ℓ is upper bounded by the largest binary floating-point number strictly less than 2, this is *not* specific to rounding to nearest even.

a latency of 3 cycles, computing c costs 9 cycles in the case of binary floating-point multiplication, but only 6 cycles in the case of binary floating-point division.

Computing c as fast as possible is important, for c is used in the definition of the fraction ℓ in (10.34), whose approximate computation usually lies on the critical path. Therefore, every cycle saved for the computation of c should, in principle, allow one to reduce the latency of the whole division operator.

Let us now turn to the computation of $D - 1$ as given by (10.38). Still assuming unbounded parallelism and with a latency of 6 cycles for c, one can keep the parenthesizing proposed at line 7 of Listing 10.18 since there the expression to be added to c can also be computed in 6 cycles. The only modification consists in replacing the constant $-(2w + e_{\min})$ with $-e_{\min} - 1$. For the binary32 format, $-e_{\min} - 1 = 125$, and the corresponding code is given by Listing 10.23.

C listing 10.23 Computing $D - 1$ in a binary32 division operator, assuming $d \geq e_{\min}$ and that $|X|$, $|Y|$, and c are available.

```
1   uint32_t Ex, Ey, nx, ny, MX, MY, Dm1;
2
3   Ex = absX >> 23;            Ey = absY >> 23;
4   nx = absX >= 0x800000;      ny = absY >= 0x800000;
5   MX = maxu(nlz(absX), 8);    MY = maxu(nlz(absY), 8);
6
7   Dm1 = (((Ex - nx) + (Ey - ny)) - ((MX + MY) - 125)) + c;
```

With unbounded parallelism, the code given above thus produces $D - 1$ in 7 cycles. Of course, this is only one of the many possible ways of parenthesizing the expression of $D - 1$. Other schemes may be better suited, depending on the algorithm chosen for approximating ℓ and on the actual degree of parallelism of the target processor. Examples of other ways of computing $D - 1$ have been implemented in FLIP, with the STMicroelectronics ST200 processor as the main target.

10.4.3   Overflow detection

Because of Property 19, binary floating-point division will never overflow because of rounding, but only when

$$d \geq e_{\max} + 1.$$

This is different from the case of binary floating-point multiplication (see Section 10.3.3).

The situation where $d \geq e_{\max} + 1$ clearly requires that our assumption $d \geq e_{\min}$ be true and thus, using (10.37), it is characterized by the condition

$$D - 1 \geq 2e_{\max}.$$

This characterization has already been used for multiplication in Section 10.3.3 and one can reuse the implementation provided by Listing 10.19.

10.4.4 Getting the correctly rounded result

As in the previous sections, assume for simplicity that $d \geq e_{\min}$ (the general case is handled in the FLIP library). Recall from (10.35) that in this case the correctly rounded result to be returned is $\mathrm{RN}(\ell) \cdot 2^d$. Since the values of c and $D - 1 = d - e_{\min}$ have already been computed in Listing 10.22 and Listing 10.23, we are left with the computation of

$$\mathrm{RN}(\ell) = (1.r_1 \ldots r_{p-1})_2,$$

where ℓ is defined by (10.34) and thus *a priori* has an infinite binary expansion of the form

$$\ell = (1.\ell_1 \ldots \ell_{p-1}\ell_p \ldots)_2. \tag{10.39}$$

As explained in Section 8.6, many algorithms exist for such a task, and they belong to one of the following three families: digit-recurrence algorithms (SRT, etc.), functional iteration algorithms (Newton iteration, etc.), and polynomial approximation-based algorithms.

We will now show how to implement in software two examples of such algorithms. The first one is the *restoring division* algorithm, which is the simplest digit-recurrence algorithm (see, for example, the Sections 1.6.1 and 8.6.2 of [126] for a detailed presentation of this approach). The second one consists in computing approximate values of a function underlying ℓ by means of polynomial evaluation [198]. Note that this second approach has already been used for implementing division (and square root) in software in a different context, namely when a floating-point FMA instruction is available; see, for example, [2].

As we will see, our first example leads to a highly sequential algorithm, while our second example allows us to expose much instruction-level parallelism (ILP). Hence those examples can be seen as two extremes of a wide range of possible implementations.

First example: restoring division

Restoring division is an iterative process, where iteration i produces the value of the bit ℓ_i in (10.39).

In order to see when to stop the iterations, recall first that since we have assumed that $d \geq e_{\min}$, the rational number ℓ lies in the normal range or the

overflow range. Then in this case it is known that ℓ cannot lie exactly halfway between two consecutive normal binary floating-point numbers (see for example [126, page 452] or [88, page 229]).[8] Consequently, the binary floating-point number $\mathrm{RN}(\ell)$ can be obtained with only one guard bit (the sticky bit is not needed, in contrast to addition and multiplication):

$$\mathrm{RN}(\ell) = (1.\ell_1 \ldots \ell_{p-1})_2 + \ell_p \cdot 2^{1-p}. \tag{10.40}$$

We conclude from this formula that p iterations suffice to obtain $\mathrm{RN}(\ell)$.

   Let us now examine the details of one iteration and how to implement it in software using integer arithmetic. To do so, let us first write the binary expansions of the numerator and denominator of $\ell = 2m'_x \cdot 2^{-c}/m'_y$ as follows:

$$2m'_x \cdot 2^{-c} = (N_{-1}N_0.N_1 \ldots N_{p-1})_2$$

and

$$m'_y = (01.M_1 \ldots M_{p-1})_2.$$

Notice that $N_{-1} = 1 - c$, where c is given by (10.33). It follows from the two identities above that the positive integers

$$N = \sum_{i=0}^{p} N_{p-1-i} 2^i$$

and

$$M = 2^{p-1} + \sum_{i=0}^{p-2} M_{p-1-i} 2^i$$

satisfy

$$\ell = N/M.$$

   Next, define for $i \geq 0$ the pair (q_i, r_i) by $q_i = (1.\ell_1 \ldots \ell_i)_2$ and $N = q_i \cdot M + r_i$. Clearly, $r_i \geq 0$ for all $i \geq 0$, and (q_i, r_i) goes to $(\ell, 0)$ when i goes to infinity. Defining further $Q_i = q_i \cdot 2^i$ and $R_i = r_i \cdot 2^i$, we arrive at

$$2^i N = Q_i \cdot M + R_i, \qquad i \geq 0. \tag{10.41}$$

Since $q_i > 0$ has at most i fractional bits, Q_i is indeed a positive integer. Besides, since the identity $R_i = 2^i N - Q_i \cdot M$ involves only integers, R_i is an integer as well. Note also that since $\ell - q_i$ is, on the one hand, equal to $2^{-i}R_i/M$ and, on the other hand, equal to $(0.0 \ldots 0\ell_{i+1}\ell_{i+2}\ldots)_2 \in [0, 2^{-i})$, we have

$$0 \leq R_i < M. \tag{10.42}$$

---

[8]This is not the case when $d < e_{\min}$ since in that case ℓ lies in the subnormal range; see the FLIP library on how to handle such cases.

This tells us that Q_i and R_i are, respectively, the quotient and the remainder in the Euclidean division of integer $2^i N$ by integer M. Note the scaling of N by 2^i.

The preceding analysis shows that, given

$$Q_0 = 1 \quad \text{and} \quad R_0 = N - M,$$

computing the first p fractional bits of ℓ iteratively simply reduces to deducing (Q_i, R_i) from (Q_{i-1}, R_{i-1}), for $1 \leq i \leq p$. Using $Q_i = q_i \cdot 2^i$ and $q_i = q_{i-1} + \ell_i \cdot 2^{-i}$, it is not hard to verify that

$$Q_i = 2Q_{i-1} + \ell_i. \tag{10.43}$$

Then, the identity in (10.41) gives

$$
\begin{aligned}
2^i N &= (2Q_{i-1} + \ell_i) \cdot M + R_i \quad \text{using (10.43)} \\
&= 2Q_{i-1}M + \ell_i \cdot M + R_i \\
&= 2(2^{i-1}N - R_{i-1}) + \ell_i \cdot M + R_i \quad \text{using (10.41) with } i - 1.
\end{aligned}
$$

Hence, after simplification,

$$R_i = 2R_{i-1} - \ell_i \cdot M. \tag{10.44}$$

Equations (10.43) and (10.44) lead to Algorithm 10.1. For $i \geq 1$, one computes a tentative remainder $T = 2R_{i-1} - M$; that is, we do as if $\ell_i = 1$. Then two situations may occur:

- if T is negative, then ℓ_i is in fact equal to zero. In this case $Q_i = 2Q_{i-1}$ and we restore the correct remainder $R_i = 2R_{i-1}$ by adding M to T;

- if $T \geq 0$, then $R_i = T$ satisfies (10.42), so that $\ell_i = 1$ and $Q_i = 2Q_{i-1} + 1$.

Algorithm 10.1 The first p iterations of the restoring method for binary floating-point division.

$(Q_0, R_0) \leftarrow (1, N - M)$
for $i \in \{1, \ldots, p\}$ **do**
    $T \leftarrow 2R_{i-1} - M$
    if $T < 0$ **then**
        $(Q_i, R_i) \leftarrow (2Q_{i-1}, T + M)$
    else
        $(Q_i, R_i) \leftarrow (2Q_{i-1} + 1, T)$
    end if
end for

An implementation of Algorithm 10.1 for the binary32 format, for which $p = 24$, is detailed in Listing 10.24.

Recalling that the bit strings of m'_x and m'_y are stored in the variables mpX and mpY (see (10.28) as well as line 5 of Listing 10.22), one can deduce the integers N and M simply by shifting to the right. The initial values $Q_0 = 1$ and $R_0 = N - M \in [0, \ldots, 2^{p+1})$ can then be stored exactly into the 32-bit unsigned integers Q and R. These integers will be updated during the algorithm so as to contain, respectively, Q_i and R_i at the end of the i-th iteration. Obviously, the multiplications by two are implemented with shifts by one position to the left.

C listing 10.24 First 24 iterations of the restoring method, rounding and packing for a binary32 division operator. Here we assume that $d \geq e_{\min}$ and that Sr, Dm1, mpX, mpY, and c are available.

```
 1   uint32_t N, M, Q, R, i;
 2   int32_t T;
 3
 4   N = mpX >> (7 + c);      M = mpY >> 8;
 5   Q = 1;                   R = N - M;
 6
 7   for (i = 1; i < 25; i++)
 8   {
 9      T = (R << 1) - M;
10
11      if (T < 0)
12      {
13         Q = Q << 1;        R = T + M;
14      }
15      else
16      {
17         Q = (Q << 1) + 1; R = T;
18      }
19   }
20
21   return (Sr | (Dm1 << 23)) + ((Q >> 1) + (Q & 1));
```

At the end of iteration 24, the bit string of Q contains the first 24 fraction bits of ℓ:

$$\text{Q} = [\underbrace{0000000}_{7 \text{ zeros}} 1\ell_1 \ldots \ell_{23}\ell_{24}].$$

Applying (10.40) with $p = 24$, we have $\text{RN}(\ell) = (1.\ell_1 \ldots \ell_{23})_2 + \ell_{24} \cdot 2^{-23}$. Consequently, the 32-bit unsigned integer

$$S_r \cdot 2^{31} + (D - 1) \cdot 2^{23} + \text{RN}(\ell) \cdot 2^{23}$$

that encodes the result is thus obtained as in line 21 of Listing 10.24.

From this code it is clear that the restoring method is sequential in nature. Although at iteration i both variables Q and R can be updated independently from each other, 24 iterations are required. A rough (but reasonable)

count of at least 3 cycles per iteration thus yields a total latency of more than 70 cycles, and this only for the computation of $\ell_1, \ldots, \ell_{24}$. Adding to this the cost of computing c, handling special values, computing the sign and exponent, rounding and packing, will result in an even higher latency.

To reduce latency, one may use higher radix digit recurrence algorithms. For example, some implementations of radix-4 and radix-512 SRT algorithms have been studied in [340, §9.4], which indeed are faster than the restoring method on some processors of the ST200 family. In the next paragraph, we present a third approach that allows one to obtain further speed-ups by expressing even more ILP. In the ST200 processor family, the latency is then typically reduced to less than 30 cycles for the *complete* binary floating-point division operator (see [198] and the division code in FLIP).

Second example: division by polynomial evaluation

This example summarizes an approach first introduced in [197] for square root, and then adapted to division in [198]. For ℓ as in (10.34), this approach uses two main steps to produce $\mathrm{RN}(\ell)$. First, for each input (x, y) we compute a value v which is representable with at most k bits and which approximates ℓ from above as follows:

$$- 2^{-p} < \ell - v \le 0. \tag{10.45}$$

Then, we can deduce from v the correctly rounded value $\mathrm{RN}(\ell)$ by implementing a method essentially similar to the method outlined in [126, page 460].

Computing the one-sided approximation v

Although a functional iteration approach (Newton iteration or one of its variants; see Section 5.3, page 155, and Section 8.6.2, page 263) could be used to compute v, more ILP can be exposed by evaluating a suitable polynomial that approximates the exact quotient ℓ. Instead of (10.45) we shall in fact ensure the slightly stronger condition

$$\left|(\ell + 2^{-p-1}) - v\right| < 2^{-p-1}. \tag{10.46}$$

This form is more symmetric and more natural to attain than (10.45). First, the rational number $\ell + 2^{-p-1}$ can be considered as the exact value at some particular point (σ, τ) of the function

$$F(s, t) = 2^{-p-1} + \frac{s}{1 + t}.$$

Recalling that $\ell = 2m'_x/m'_y \cdot 2^{-c}$, a typical choice is

$$(\sigma, \tau) = (2m'_x \cdot 2^{-c}, m'_y - 1).$$

Then F is approximated over a suitable domain $\mathcal{S} \times \mathcal{T}$ (that contains all the possible values for (σ, τ) when x and y vary) by a polynomial P of the form

$$P(s,t) = 2^{-p-1} + s \cdot a(t), \quad \text{with} \quad a(t) = \sum_{i=0}^{\delta} a_i t^i. \qquad (10.47)$$

Finally, the polynomial P is evaluated at (σ, τ) and the obtained value is assigned to v.

This process introduces two kinds of errors: an *approximation error* which quantifies the fact that $P \approx F$ over $\mathcal{S} \times \mathcal{T}$ and, since the evaluation of P at (σ, τ) is done by a finite-precision program, an *evaluation error* which quantifies the fact that $v \approx P(\sigma, \tau)$.

In [198] some sufficient conditions on these two errors have been given in order to ensure that (10.46) holds. Then, a suitable polynomial $a(t)$ has been obtained under such conditions. By "suitable" we mean a polynomial of smallest degree δ and for which each coefficient absolute value $|a_i|$ fits in a 32-bit unsigned integer. Such a polynomial was obtained as a "truncated minimax" polynomial approximation, [9] using the software environment *Sollya*.[10] More precisely, the polynomial $a(t)$ has the form

$$a(t) = \sum_{i=0}^{10} (-1)^i \cdot A_i \cdot 2^{-32} \cdot t^i, \qquad (10.48)$$

where $A_i \in \{0, \ldots, 2^{32} - 1\}$ for $0 \le i \le 10$. *Sollya* has further been used to guarantee, using interval arithmetic, that the approximation error induced by this choice of polynomial is small enough for our purposes.

A minimal degree δ has been sought for obvious speed reasons (intuitively, the smaller the polynomial a is, the faster our division code will be). However, once δ has been fixed, there are still many ways of evaluating the arithmetic expression $P(s,t)$ for P as in (10.47) and a as in (10.48). A classical way is Horner's rule:

$$P(s,t) = 2^{-p-1} + s \cdot \Big(a_0 + t \cdot \big(\cdots + t \cdot (a_9 + a_{10} \cdot t) \ldots \big) \Big).$$

This parenthesizing of $P(s,t)$ involves 11 additions and 11 multiplications. Assuming a latency of 1 for an addition and 3 for multiplication (of type `mul`; see Section 10.1.2), one can, in principle, deduce from s and t an approximate value for $P(s,t)$ in 44 cycles. This already seems faster than the rough estimate of 70 cycles used by the 24 iterations in the restoring method (see Listing 10.24).

---

[9] We have used a minimax-like approximation and not, for example, a simpler Taylor-like polynomial $1 - t + t^2 - t^3 + \cdots$ because, as detailed in [198], the domain $\mathcal{T}$ on which $a(t)$ should approximate $1/(1+t)$ is $[0, 1 - 2^{1-p}]$ and thus contains values that are close to 1.

[10] See `http://sollya.gforge.inria.fr/` and Lauter's Ph.D. dissertation [245].

And yet, Horner's rule is fully sequential (just like the restoring method), and other parenthesizings of $P(s,t)$ exist that make it possible to expose much more ILP. In such cases, provided the degree of parallelism is high enough, much lower latencies can be expected. An example of such a parenthesizing, generated automatically, has been given in [198]. It reduces the 44-cycle latency of Horner's rule shown above to only 14 cycles, provided one can launch at least 3 operations + or `mul` simultaneously, at most 2 of them being `mul`. Note that this latency of 14 cycles comes from viewing $P(s,t)$ as a bivariate polynomial, and could not be obtained by first evaluating the univariate polynomial $a(t)$ as fast as possible, and then applying the final Horner step $2^{-p-1} + s \cdot a(t)$.

Once a fast/highly parallel evaluation scheme has been found, it remains to implement it using 32-bit integer arithmetic and to check the numerical accuracy of the resulting program (evaluation error). An example of a C program that implements the 14-cycle evaluation scheme just mentioned is given in Listing 10.25. The variables S, T, V used in Listing 10.25 are integer representations of σ, τ, v, respectively:

$$S = \sigma \cdot 2^{30}, \qquad T = \tau \cdot 2^{32}, \qquad V = v \cdot 2^{30}.$$

Also, each hexadecimal constant corresponds to a particular A_i as in (10.48). It turns out that the variables `r1`, ..., `r25`, `V` are indeed in the range $\{0, \ldots, 2^{32}-1\}$ of 32-bit unsigned integers, and that $v = V \cdot 2^{-30}$ satisfies (10.45), as desired. Checking such properties by paper-and-pencil calculations, if possible, would surely be long and error-prone. In [198] they have thus been verified mechanically using the *Gappa* software.[11]

Rounding to nearest even

For some details on how to effectively implement the move from v to $\mathrm{RN}(\ell)$, we refer to the second example of a square root operator in Section 10.5.3.[12] A complete implementation of division, including subnormal numbers as well as all the IEEE 754-2008 rounding direction attributes, can be found in the FLIP library.

10.5   Binary Floating-Point Square Root

Now, let us deal with the software implementation, for the binary32 format and $\circ = \mathrm{RN}$, of a square root operator. Our exposition follows [196, 197].

The case where x is a special datum (like ± 0, $\pm \infty$, or NaN) is described in Section 10.5.1, while the case where x is a (sub)normal is discussed in Sections 10.5.2 and 10.5.3.

---

[11]`http://lipforge.ens-lyon.fr/www/gappa/` and Chapter 13.

[12]For the square root, the situation will be a little simpler since the output cannot be a subnormal number.

C listing 10.25 Polynomial evaluation program used in a binary32 division operator (Listing 1 in [198]).

```
1    uint32_t S, T, V,
2              r0, r1, r2, r3, r4, r5, r6, r7, r8, r9, r10, r11, r12, r13,
3              r14, r15, r16, r17, r18, r19, r20, r21, r22, r23, r24, r25;
4
5    r0  = mul( T , 0xffffe7d7 );
6    r1  = 0xfffffff8 - r0;
7    r2  = mul( S , r1 );
8    r3  = 0x00000020 + r2;
9    r4  = mul( T , T );
10   r5  = mul( S , r4 );
11   r6  = mul( T , 0xffbad86f );
12   r7  = 0xfffbece7 - r6;
13   r8  = mul( r5 , r7 );
14   r9  = r3 + r8;
15   r10 = mul( r4 , r5 );
16   r11 = mul( T , 0xf3672b51 );
17   r12 = 0xfd9d3a3e - r11;
18   r13 = mul( T , 0x9a3c4390 );
19   r14 = 0xd4d2ce9b - r13;
20   r15 = mul( r4 , r14 );
21   r16 = r12 + r15;
22   r17 = mul( r10 , r16 );
23   r18 = r9 + r17;
24   r19 = mul( r4 , r4 );
25   r20 = mul( T , 0x1bba92b3 );
26   r21 = 0x525a1a8b - r20;
27   r22 = mul( r4 , 0x0452b1bf );
28   r23 = r21 + r22;
29   r24 = mul( r19 , r23 );
30   r25 = mul( r10 , r24 );
31   V   = r18 + r25;
```

As we will see, our analysis and code have many similarities with those for division in the previous section. However, things are simpler for at least two obvious reasons: first, $\sqrt{x}$ is univariate while x/y was bivariate; and second, $\sqrt{x} \geq 0$ over the reals, so that the sign is known in advance.[13] A third reason is that binary floating-point square roots never underflow or overflow, and an efficient implementation should take advantage of this.

10.5.1   Handling special values

For the square root, the input x is considered a *special input* when it is either ± 0, $\pm \infty$, NaN, or a binary floating-point number that is finite, nonzero,

---

[13]Note however that the IEEE 754 standards require that $\sqrt{-0} = -0$, a special case which is covered in Section 10.5.1.

and negative. The special output values that the IEEE 754-2008 standard mandates in such situations have been listed in Table 8.6, page 265.

Detecting that a special value must be returned

Using Table 10.2 we see that x is a special input if and only if $X \notin (0, 2^{k-1} - 2^{p-1})$; that is, if and only if

$$(X - 1) \bmod 2^k \geq 2^{k-1} - 2^{p-1} - 1. \tag{10.49}$$

For example, when $k = 32$ and $p = 24$, an implementation of (10.49) is given by lines 3 and 4 of Listing 10.26.

C listing 10.26 Special value handling in a binary32 square root operator.

```
 1  uint32_t Xm1;
 2
 3  Xm1 = X - 1;
 4  if (Xm1 >= 0x7F7FFFFF)
 5  {
 6      if (X <= 0x7F800000 || X == 0x80000000)
 7          return X;
 8      else
 9          return (0x7FC00000 | X);          // qNaN with payload equal to
10                                            // the last 22 bits of X
11  }
```

Returning special values as recommended or required by IEEE 754-2008

Once x is known to be special, we deduce from Table 8.6, page 265, that there are only two cases to be considered: the result must be either x itself, or a qNaN. Since here x is known to be *not* a positive (sub)normal number, this first case, which is $x \in \{+0, +\infty, -0\}$, is characterized by

$$X \leq 2^{k-1} - 2^{p-1} \qquad \text{or} \qquad X = 2^{k-1}.$$

When $k = 32$ and $p = 24$, an implementation of this condition is given at line 6 of Listing 10.26, using

$$2^{31} - 2^{23} = (7F800000)_{16} \qquad \text{and} \qquad 2^{31} = (80000000)_{16}.$$

Returning a qNaN is done in the same way as for addition, multiplication, or division, by masking with the constant $(7FC00000)_{16}$, whose bit string is

$$[0 \underbrace{111111111}_{9 \text{ ones}} \underbrace{0000000000000000000000}_{22 \text{ zeros}}].$$

Again, this kind of mask keeps as much of the information of the input as possible, and, in particular, the payload of x, as recommended by the IEEE 754-2008 standard (see [187, §6.2]).

10.5.2   Exponent computation

We assume from now on that the input x is not special; that is, x is a positive finite nonzero (sub)normal number.

Formula for the exponent of the result

In order to see how to compute the exponent of our correctly rounded square root, let us first rewrite $x = m_x \cdot 2^{e_x}$ as

$$x = m'_x \cdot 2^{e'_x},$$

with m'_x as in (10.21) and e'_x as in (10.22). Let us also introduce the shift value c defined as

$$c = \begin{cases} 0 & \text{if } e'_x \text{ is even,} \\ 1 & \text{if } e'_x \text{ is odd.} \end{cases} \tag{10.50}$$

Rewriting x as $x = (2^c \cdot m'_x) \cdot 2^{e'_x - c}$, we obtain $\sqrt{x} = \ell \cdot 2^d$, so that, after rounding,

$$\mathrm{RN}(\sqrt{x}) = \mathrm{RN}(\ell \cdot 2^d),$$

where

$$\ell = \sqrt{2^c \cdot m'_x} \qquad \text{and} \qquad d = \frac{e'_x - c}{2}. \tag{10.51}$$

Notice that, by definition of c, the real ℓ lies in $[1, 2)$. Moreover, as recalled in Section 8.7, floating-point square roots never underflow or overflow, and thus

$$e_{\min} \le d \le e_{\max}. \tag{10.52}$$

Hence, the correctly rounded value of the square root of x is given by the following product:

$$\mathrm{RN}(\sqrt{x}) = \mathrm{RN}(\ell) \cdot 2^d. \tag{10.53}$$

In general, applying a rounding operator $\circ(\cdot)$ to a real number $\ell \in [1, 2)$ yields the weaker enclosure $\circ(\ell) \in [1, 2]$. However, in the case of square root with rounding to nearest, one has the following nice property (see [197] for a proof).

Property 20. $\mathrm{RN}(\ell) \in [1, 2)$.

From Property 20 and the inequalities in (10.52) we conclude that

$$\mathrm{RN}(\ell) \cdot 2^d$$

in (10.53) is already the *normalized representation* of $\mathrm{RN}(\sqrt{x})$. In particular, neither overflow detection nor exponent update is needed.[14]

---

[14]The latter is of course still true for $\circ \in \{\mathrm{RZ}, \mathrm{RD}\}$, but not for $\circ = \mathrm{RU}$, as shown in [197].

In conclusion, the result exponent is indeed d as in (10.51). Since c is either zero or one, and since $e'_x = e_x - \lambda_x$, we arrive at the following formula:

$$d = \left\lfloor \frac{e_x - \lambda_x}{2} \right\rfloor, \tag{10.54}$$

where $\lfloor \cdot \rfloor$ denotes the usual floor function. Note that the shift value c in (10.50) does *not* appear explicitly in the above formula for d. In practice, this means that c and d can be computed independently of each other.

Implementation for the binary32 format

Let us now implement (10.54). As for other basic operations, we shall in fact compute the biased value

$$D - 1 = d + e_{\max} - 1.$$

Combining (10.54) with $E_x = e_x - e_{\min} + n_x$ and $\lambda_x = M_X - w$ and $e_{\min} = 1 - e_{\max}$, one can deduce that

$$D - 1 = \left\lfloor \frac{E_x - n_x - M_X + w - e_{\min}}{2} \right\rfloor.$$

For the binary32 format, $w - e_{\min} = 8 + 126 = 134$, and thus $D - 1$ can be implemented as shown in Listing 10.27.

C listing 10.27 Computing $D - 1$ in a binary32 square root operator.

```
1  uint32_t Ex, nx, MX, Dm1;
2
3  Ex = X >> 23;     nx = X >= 0x800000;    MX = max(nlz(X), 8);
4
5  Dm1 = ((Ex - nx) + (134 - MX)) >> 1;
```

Notice that in the above code x is assumed to be nonspecial, which in the case of square root implies $X = |X|$. Hence, computing $|X|$ is not needed, and the biased exponent value E_x can be deduced simply by shifting X.

10.5.3 Getting the correctly rounded result

In (10.53), once d has been obtained through the computation of $D - 1$, it remains to compute the binary floating-point number

$$\mathrm{RN}(\ell) = (1.r_1 \ldots r_{p-1})_2,$$

where

$$\ell = \sqrt{2^c \cdot m'_x} = (1.\ell_1 \ldots \ell_{p-1}\ell_p \ldots)_2. \tag{10.55}$$

Exactly as for division in Section 10.4.4,

- in general, the binary expansion of ℓ is infinite;

- we will give two examples of how to implement the computation of $RN(\ell)$: the first uses the classical *restoring* method for square rooting (see for example [126, §6.1] for a description of the basic recurrence). The second one, which comes from [197], is based on polynomial approximation and evaluation.

Before that, we shall see how to implement the computation of the shift value c defined in (10.50). Since the value of c was not explicitly needed for obtaining the square root exponent d, it has not been computed yet. This is in contrast to binary floating-point multiplication and division, where the corresponding c value was already used for deducing $D - 1$ (see line 7 in Listing 10.18 and line 7 in Listing 10.23).

Computation of the shift value c

From (10.50) we see that c is 1 if and only if e'_x is odd. Again, combining the facts that $E_x = e_x - e_{\min} + n_x$ and $\lambda_x = M_X - w$ with the definition $e'_x = e_x - \lambda_x$, we have for the binary32 format (where $e_{\min} = -126$ and $w = 8$)

$$e'_x = E_x - n_x - M_X - 118.$$

Therefore, e'_x is odd if and only if $E_x - n_x - M_X$ is odd.

For the binary32 format, we have seen in Listing 10.27 how to deduce E_x, n_x, and M_X from X. Hence, we use the code in Listing 10.28 for deducing the value of c. Since M_X is typically more expensive to obtain than E_x and n_x, one can compute M_X in parallel with $E_x - n_x$. Of course, other implementations are possible (for example, using logic exclusively), but this one reuses the value $E_x - n_x$ that appears in Listing 10.27.

C listing 10.28 Computing the shift value c in a binary32 square root operator. Here we assume that `Ex`, `nx`, and `MX` are available.

```
1  uint32_t c;
2
3  c = ((Ex - nx) - MX) & 1;
```

First example: restoring square root

Similarly to restoring division (which we have recalled in detail in Section 10.4.4), restoring square root is an iterative process whose iteration number $i \geq 1$ produces the bit ℓ_i of the binary expansion of ℓ in (10.55).

Exactly as for division, it is known (see, for example, [88, page 242] and [126, page 463]) that the square root of a binary floating-point number cannot be the exact middle of two consecutive (normal) binary floating-point numbers. Therefore, the correctly rounded result $RN(\ell)$ can be obtained

using the same formula as the one used for division in (10.40), and p iterations are again enough. Compared to division, the only specificity of the restoring square root method is the definition of the iteration itself, which we will recall now.

Let $q_0 = 1$. The goal of iteration $1 \le i \le p$ is to produce ℓ_i, or, equivalently, to deduce $q_i = (1.\ell_1 \ldots \ell_i)_2$ from $q_{i-1} = (1.\ell_1 \ldots \ell_{i-1})_2$. For $0 \le i \le p$, one way of encoding the rational number q_i into a k-bit unsigned integer ($k > p$) is through the integer

$$Q_i = q_i \cdot 2^p, \tag{10.56}$$

whose bit string is

$$[\underbrace{000 \ldots 000}_{k-p-1 \text{ zeros}} 1\, q_1 \ldots q_i \underbrace{000 \ldots 000}_{p-i \text{ zeros}}].$$

One has $0 \le \ell - q_i < 2^{-i}$, and thus $q_i \ge 0$ approximates ℓ from below. Hence, q_i^2 approximates ℓ^2 from below as well, and one can define the remainder $r_i \ge 0$ by

$$\ell^2 = q_i^2 + r_i. \tag{10.57}$$

Now, $\ell^2 = 2^c \cdot m_x'$ has at most $p - 1$ fraction bits, and

$$q_i^2 = \left((1.q_1 \ldots q_i)_2 \right)^2$$

has at most $2i$ fraction bits. Since $0 \le i \le p$, both ℓ^2 and q_i^2 have at most $p + i$ fraction bits and we conclude that the following scaled remainder is a *non-negative integer*:

$$R_i = r_i \cdot 2^{p+i}. \tag{10.58}$$

The following property shows further that the integer R_i can be encoded using at most $p+2$ bits. For example, for the binary32 format, $p+2 = 26 \le 32$, and so R_i will fit into a 32-bit unsigned integer.

Property 21. *For $0 \le i \le p$, the integer R_i satisfies $0 \le R_i < 2^{p+2}$.*

Proof. We have already seen that R_i is an integer such that $0 \le R_i$. Let us now show the upper bound. From $0 \le \ell - q_i < 2^{-i}$ it follows that

$$\ell^2 - q_i^2 = (\ell - q_i)(2q_i + (\ell - q_i)) \le 2^{-i}(2q_i + 2^{-i}).$$

Now, $q_i = (1.\ell_1 \ldots \ell_i)_2 \le 2 - 2^{-i}$, so that $2q_i + 2^{-i} \le 4 - 2^{-i} < 4$. From this we conclude that

$$R_i = (\ell^2 - q_i^2) \cdot 2^{p+i} < 4 \cdot 2^{-i} \cdot 2^{p+i} = 2^{p+2}.$$

$\square$

Combining the definition of Q_i in (10.56) with the facts that $q_0 = 1$ and, for $1 \leq i \leq p$, that $q_i = q_{i-1} + \ell_i \cdot 2^{-i}$, we get

$$Q_0 = 2^p \tag{10.59}$$

and, for $1 \leq i \leq p$,

$$Q_i = Q_{i-1} + \ell_i \cdot 2^{p-i}. \tag{10.60}$$

Then, using (10.59) and (10.60) together with the invariant in (10.57) and the definition of R_i in (10.58), one may check that

$$R_0 = m'_x \cdot 2^{p+c} - 2^p \tag{10.61}$$

and, for $1 \leq i \leq p$,

$$R_i = 2R_{i-1} - \ell_i \cdot (2Q_{i-1} + \ell_i \cdot 2^{p-i}). \tag{10.62}$$

Note that the recurrence relation (10.62) is, up to a factor of 2^p, analogous to Equation (6.12) in [126, page 333].

As for division, we deduce from the above recurrence relations for Q_i and R_i a *restoring* algorithm (see Algorithm 10.2), which works as follows. A tentative remainder T is computed at each iteration, assuming $\ell_i = 1$ in the recurrence (10.62) that gives R_i. Then two situations may occur:

- if $T < 0$ then the true value of ℓ_i was in fact 0. Hence, by (10.60) and (10.62), $Q_i = Q_{i-1}$ while R_i is equal to $2R_{i-1}$;

- if $T \geq 0$ then $\ell_i = 1$ was the right choice. In this case, (10.60) and (10.62) lead to $Q_i = Q_{i-1} + 2^{p-i}$ and $R_i = T$.

Algorithm 10.2 The first p iterations of the restoring method for binary floating-point square root.

$(Q_0, R_0) \leftarrow (2^p, m'_x \cdot 2^{p+c} - 2^p)$
for $i \in \{1, \ldots, p\}$ **do**
  $T \leftarrow 2R_{i-1} - (2Q_i + 2^{p-i})$
  if $T < 0$ **then**
    $(Q_i, R_i) \leftarrow (Q_{i-1}, 2R_{i-1})$
  else
    $(Q_i, R_i) \leftarrow (Q_{i-1} + 2^{p-i}, T)$
  end if
end for

An implementation of Algorithm 10.2 for the binary32 format, for which $p = 24$, is detailed in Listing 10.29. Here are a few remarks about lines 4 to 20:

- The variable Q is initialized with

$$Q_0 = 2^{24} = (1000000)_{16}$$

and, at the end of iteration $1 \le i \le 24$, contains the integer Q_i.

- Similarly, the variable R is initialized with R_0 and then contains the successive values of R_i. Note that the initialization of R requires the knowledge of both m'_x and c. These are assumed available in variables mpX and c, respectively. Concerning mpX, we may compute it exactly as for division (see for example Listing 10.22). Concerning c, we have seen how to compute it in Listing 10.28. The right shift of $7 - c$ positions at line 4 of Listing 10.29 comes from the fact that the bit string of mpX in Listing 10.22 has the form

$$[1 \underbrace{* \cdots *}_{23 \text{ bits}} \underbrace{00000000}_{8 \text{ bits}}],$$

and from the fact that the bit string of $m'_x \cdot 2^{p+c}$ has the form

$$[\underbrace{0000000}_{7 \text{ bits}} 1 \underbrace{* \cdots *}_{23 \text{ bits}} 0] \qquad \text{if } c = 0,$$

and

$$[\underbrace{000000}_{6 \text{ bits}} 1 \underbrace{* \cdots *}_{23 \text{ bits}} 00] \qquad \text{if } c = 1.$$

- The tentative remainder T is stored in variable T, while S is initialized with 2^{24} and contains the integer 2^{24-i} at the end of the i-th iteration.

Final rounding of the significand and packing with the result exponent is done at line 22 of Listing 10.29. The variable Dm1 which encodes the (biased) result exponent has been computed in Listing 10.23.

The only difference with division (see line 21 of Listing 10.24) is that the sign bit of $\text{RN}(\sqrt{x})$ is known to be zero when x is not special.

Second example: square root by polynomial evaluation

For $\ell = \sqrt{2^c \cdot m'_x}$, a way of computing $\text{RN}(\ell)$ that is much faster on some parallel architectures has been recently proposed in [196, 197]. Its generalization to division has already been outlined in the second example of Section 10.4.4, following [198], and extensions to other functions are currently being investigated.

Computing the one-sided approximation v

As for division, $\text{RN}(\ell)$ is obtained by first computing a one-sided approximation v to ℓ that satisfies (10.46) and is representable using at most k bits.

C listing 10.29 First 24 iterations of the restoring method, rounding and packing for a binary32 square root operator. Here we assume that `Dm1`, `mpX`, and `c` are available.

```
 1  uint32_t Q, R, S, i;
 2  int32_t T;
 3
 4  Q = 0x1000000;                R = (mpX >> (7 - c)) - Q;
 5  S = 0x1000000;
 6
 7  for (i = 1; i < 25; i++)
 8  {
 9     S = S >> 1;
10     T = (R << 1) - ((Q << 1) + S);
11
12     if (T < 0)
13     {
14        R = R << 1;
15     }
16     else
17     {
18        Q = Q + S; R = T;
19     }
20  }
21
22  return (Dm1 << 23) + ((Q >> 1) + (Q & 1));
```

The main differences concern the choice of function F to be approximated, the domain $S \times T$ on which it is approximated,[15] the fact that the evaluation point (σ, τ) is not exactly representable in finite precision, and the sufficient conditions on the approximation and evaluation errors. In particular, for the binary32 format those conditions are simpler and, above all, much easier to verify, than for division (see [197] and [198] for the details).

To summarize, this first step is aimed at producing a code with high ILP, in the same spirit as Listing 10.25 and whose output is a 32-bit unsigned integer V such that $v = V \cdot 2^{-30}$ satisfies

$$\left|(\ell + 2^{-25}) - v\right| < 2^{-25}. \tag{10.63}$$

Rounding to nearest even

Let us now see how to deduce $\mathrm{RN}(\ell)$ from v. The binary expansion of v has the form

$$v = (1.v_1 \dots v_{30})_2.$$

---

[15]Although the square root is a univariate function, the real $\ell = \sqrt{2^c \cdot m_x'}$ depends on c and m_x'. Hence, $\ell + 2^{-25}$ may be considered as the exact value of a suitable function $F : S \times T \to \mathbb{R}$.

Therefore, truncating v after 24 fraction bits and using (10.63) gives a value

$$w = (1.v_1 \ldots v_{23}v_{24})_2$$

such that

$$0 \leq v - w < 2^{-24}. \tag{10.64}$$

By combining (10.63) and (10.64), we conclude that

$$|\ell - w| < 2^{-24}. \tag{10.65}$$

Given this "truncated approximation" w, one can now deduce the correctly rounded value $\mathrm{RN}(\ell)$ fairly easily by means of Algorithm 10.3.

Algorithm 10.3 Deducing $\mathrm{RN}(\ell)$ from the truncated approximation w.

---
if $w \geq \ell$ **then**
    $\mathrm{RN}(\ell) \leftarrow$ truncate w after 23 fraction bits
else
    $\mathrm{RN}(\ell) \leftarrow$ truncate $w + 2^{-24}$ after 23 fraction bits
end if

It was shown in [197] that Algorithm 10.3 indeed returns $\mathrm{RN}(\ell)$. In order to implement it, all we need is a way of evaluating the condition $w \geq \ell$. This can be done because of the following property [197, Property 4.1].

Property 22. *Let W, P, and Q be the 32-bit unsigned integers such that*

$$w = W \cdot 2^{-30},$$

$$P = \mathtt{mul}(W, W),$$

and

$$\ell^2 = Q \cdot 2^{-28}.$$

One has $w \geq \ell$ if and only if $P \geq Q$.

As an immediate consequence of this property, we obtain the code in Listing 10.30 that implements the computation of the correctly rounded significand $\mathrm{RN}(\ell)$ and its packing with the biased exponent $D - 1$.

C listing 10.30 Truncating a one-sided approximation, rounding, and packing for a binary32 square root operator. Here we assume that V, mpX, c, and Dm1 are available.

```
1  uint32_t W, P, Q;
2
3  W = V & 0xFFFFFFC0;
4
5  P = mul(W, W);         Q = mpX >> (3 - c);
6
7  if (P >= Q)
8      return (Dm1 << 23) + (W >> 7);
9  else
10     return (Dm1 << 23) + ((W + 0x40) >> 7);
```

Part IV

Elementary Functions

Chapter 11

Evaluating Floating-Point Elementary Functions

T HE ELEMENTARY FUNCTIONS are the most common mathematical functions: sine, cosine, tangent and their inverses, exponentials and logarithms of radices e, 2 or 10, etc. They appear everywhere in scientific computing; thus being able to evaluate them quickly and accurately is important for many applications. Various very different methods have been used for evaluating them: polynomial or rational approximations, shift-and-add algorithms, table-based methods, etc. The choice of the method greatly depends on whether the function will be implemented on hardware or software, on the target precision (for instance, table-based methods are very good for low precision, but unrealistic for very high precision), and on the required performance (in terms of speed, accuracy, memory consumption, size of code, etc.). With regard to performance, one will also resort to different methods depending on whether one wishes to optimize average performance or worst-case performance.

The goal of this chapter is to present a short survey of the existing methods and to introduce the basic techniques. More details will be given in books dedicated to elementary function algorithms, such as [88, 270, 293].

Approximation algorithms

The first algorithms used for approximating elementary functions were based on their power series expansions. Shift-and-add algorithms (i.e., algorithms that only use additions, multiplications by a power of the radix—that is, shifts—and small tables) are quite old too, since the first one goes back to the 1600s. Henry Briggs designed the first algorithm of that kind to build the 14-digit accurate tables of logarithms published in his *Arithmetica Logarithmica* (1624). Another shift-and-add algorithm, the COordinate Rotation DIgital Computer (CORDIC) algorithm, was introduced in 1959 by Volder [421, 422],

and generalized in 1971 by Walther [426, 427]. That algorithm allowed one to evaluate square roots, sines, cosines, arctangents, exponentials, and logarithms. Its versatility made it attractive for hardware implementations. CORDIC (or variants of CORDIC) has been implemented in pocket calculators (for instance, Hewlett-Packard's HP 35 [74]) and in arithmetic coprocessors such as the Intel 8087 [296]. There are still, each year, a few new papers presenting new variants of that algorithm. The *Journal of VLSI Signal Processing* devoted a special issue to CORDIC (June 2000). Although the original CORDIC algorithm was designed for radix-2 arithmetic, there exist radix-10 variants [363, 359].

Power series are still of interest for multiple-precision evaluation of functions, and shift-and-add methods may still be attractive for hardware-oriented implementations.[1] And yet, most current implementations are software implementations and use other polynomial approximations to functions.

The theory of the polynomial approximation to functions was developed mainly in the nineteenth and the early twentieth centuries. It makes it possible to build polynomials that better approximate a given function, in a given interval, than truncated power series.

The easiest approximations to obtain are *least squares approximations* (also called L^2 approximations). Assuming that w is a continuous, non-negative function, the L^2 approximation to some function f on the interval $[a, b]$ with weight function w is the polynomial p of degree less than or equal to n that minimizes

$$||f - p||_{L^2, [a,b], w} = \int_a^b w(x) \left(f(x) - p(x) \right)^2 dx. \tag{11.1}$$

The *minimax* degree-n polynomial approximation (also called the L^∞ approximation) to some function f on the interval $[a, b]$ is the polynomial p of degree less than or equal to n that minimizes

$$||f - p||_{\infty, [a,b]} = \sup_{x \in [a,b]} |f(x) - p(x)|.$$

The Remez algorithm [174, 342, 69], introduced in 1934, allows one to iteratively compute minimax polynomial approximations to functions. A generalization of that algorithm also gives minimax rational approximations.

Figures 11.1 and 11.2 show the difference between the natural logarithm ln function in $[1, 2]$ and its degree-5 Taylor (at point 1.5) and minimax approximations. One immediately sees that the minimax approximation is much better than the truncated Taylor series.

---

[1]As we write this book, for general-purpose systems, software implementations that use polynomial approximations are used on all platforms of commercial significance [270, 88]. This does not mean that CORDIC-like algorithms are not interesting for special-purpose circuits.
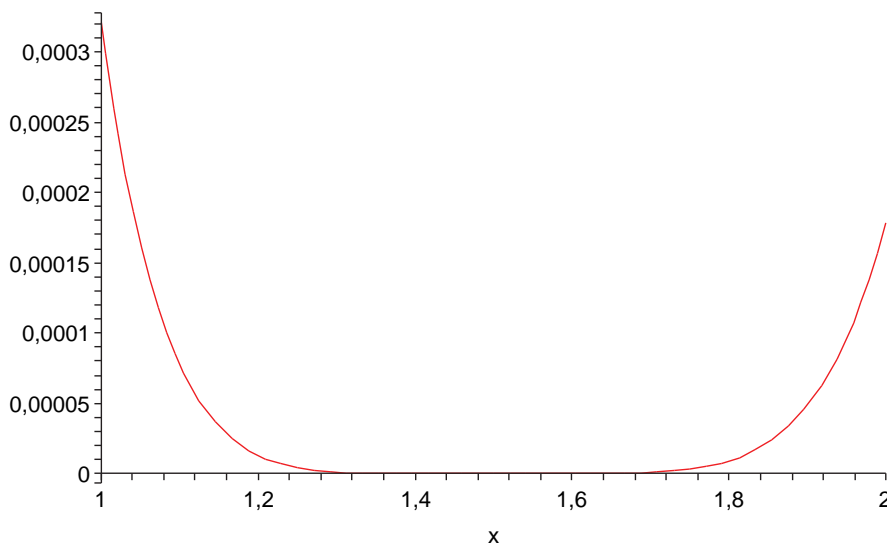
Figure 11.1: The difference between the logarithm function and its degree-5 Taylor approximation (taken at point 1.5) in the interval $[1, 2]$.
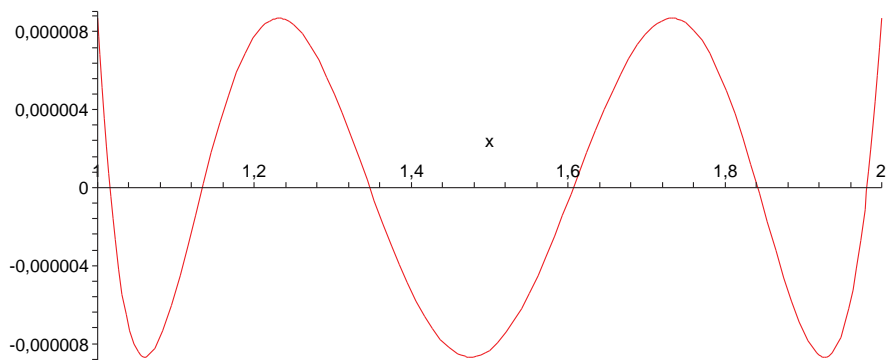
Figure 11.2: The difference between the logarithm function and its degree-5 minimax approximation in the interval $[1, 2]$. *By comparing with Figure 11.1, we see that the minimax approximation is much better than the truncated Taylor series.*

Range reduction

The elementary function algorithms allow us to approximate a function f in a given bounded domain D, which is generally rather small. Hence, the first step when evaluating f at a given input argument x consists in finding an intermediate argument $y \in D$, such that $f(x)$ can be deduced from $f(y)$ (or, sometimes, from $g(y)$, where g is a related function). A typical example is function $\sin(x)$: assume we have algorithms (e.g., polynomial approximations) that evaluate $\sin(y)$ and $\cos(y)$ for $y \in [0, \pi/2]$. The computation of $\sin(x)$ can be decomposed in three steps:

- compute y and k such that $y \in [0, \pi/2]$ and $y = x - k\pi/2$;

- compute

$$g(y, k) = \begin{cases} \sin(y) & \text{if} \quad k \bmod 4 = 0 \\ \cos(y) & \text{if} \quad k \bmod 4 = 1 \\ -\sin(y) & \text{if} \quad k \bmod 4 = 2 \\ -\cos(y) & \text{if} \quad k \bmod 4 = 3; \end{cases} \tag{11.2}$$

- obtain $\sin(x) = g(y, k)$.

When high accuracy is at stake, y is represented by two (or more) floating-point numbers.

The first step (computation of y and k from x) is called *range reduction* (or, sometimes, *argument reduction*). In this case, it is an *additive* range reduction: y is equal to x plus or minus a multiple of some constant (here, $\pi/2$). Range reduction can also be *multiplicative*; for instance, when we use

$$\cos(x) = 2\cos^2\left(\frac{x}{2}\right) - 1$$

to iteratively reduce the argument. Let us deal with the previous example of additive range reduction. When x is large (or very close to an integer multiple of $\pi/2$), computing y by the naive method, i.e., actually subtracting the result of the floating-point operation $k \times \mathrm{RN}(\pi/2)$ from x leads to a *catastrophic cancellation*: the obtained result may be very poor. Consider the following example in the binary64 format of the IEEE 754-2008 standard, assuming round to nearest even. We wish to evaluate the sine of $x = 5419351$. The value of k is 3450066. The double-precision number that is nearest to $\pi/2$ is

$$\begin{aligned} P &= \frac{884279719003555}{562949953421312} \\ &= 1.5707963267948965579898173427209258079528807952880859375. \end{aligned}$$

Therefore, the value of kP computed in the binary64 format will be 5818983827636183/1073741824. This will lead to a computed value of $x - kP$

(namely, $\mathrm{RN}(x - \mathrm{RN}(kP))$) equal to

$$\frac{41}{1073741824} = 0.0000000381842255592346191406255,$$

whereas the correct value of the reduced argument is

$$0.00000003820047507089661120930116470427948776308032614\ldots.$$

This means that if we use that naive range reduction for evaluating $\sin(x)$, our final result will only have two significant digits. If a fused multiply-add (FMA) instruction is available, the computed reduced argument (namely, $\mathrm{RN}(x - kP)$) will be

$$\frac{10811941}{281474976710656} = 0.00000003841173068508396681863814592361450195312255,$$

which, surprisingly enough,[2] is even worse. And yet, x is not a huge number: one can build worse cases. The following sections of this chapter will be devoted to the problem of range reduction. After, we will discuss the classical methods used to approximate a given function in an interval.

11.1 Basic Range Reduction Algorithms

In the following, we deal with *additive* range reduction. Let us denote by x the initial floating-point argument, and C the constant of the range reduction. We are looking for an integer k and a real number $y \in [0, C]$ or $[-C/2, +C/2]$ such that

$$y = x - kC.$$

11.1.1 Cody and Waite's reduction algorithm

Cody and Waite [75, 77] tried to improve the accuracy of the naive range reduction by representing C more accurately, as the sum of two floating-point numbers. More precisely,

$$C = C_1 + C_2,$$

where the significand of C_1 has sufficiently many zeros on its right part, so that when multiplying C_1 by an integer k whose absolute value is not too large (say, less than k_{max}, where $k_{max}C$ is the order of magnitude of the largest input values for which we want the reduction to be accurate), the result is exactly representable as a floating-point number. The reduction operation consists in computing

$$y = \mathrm{RN}(\mathrm{RN}(x - \mathrm{RN}(kC_1)) - \mathrm{RN}(kC_2)), \tag{11.3}$$

---

[2]In general, the result of the naive reduction will be *slightly* better with an FMA.

which corresponds, in the C programming language, to the line:

```
y = (x - k*C1) - k*C2
```

In (11.3), the product kC_1 is exact (provided that $|k| \leq k_{max}$). Also, the subtraction $x - kC_1$ is also exact, by Sterbenz's lemma (Lemma 2 in Chapter 4). Hence, the only sources or error are the computation of the product kC_2 and the subtraction of that product from $x - kC_1$: we have somehow simulated a larger precision. Still consider our previous example (sine of $x = 5419351$ in binary64 arithmetic, with $C = \pi/2$, which gives $k = 3450066$). Consider the two binary64 floating-point numbers:

$$C_1 = \frac{1686629713}{1073741824}$$

$$= 1.1001001000011111101101010101000 1 \quad \text{(binary)}$$

$$= 1.5707963267341256141662597656 25 \quad \text{(decimal)}$$

and

$$C_2 = \frac{4701928774853425}{77371252455336267181195264}$$

$$= 1.00001011010001100001000110100110001$$

$$00110001100110001 \times 2^{-34} \quad \text{(binary)}$$

$$= 6.07710050650619224931915769153843113$$

$$331570080504206998739391565322875976562 5 \times 10^{-11} \quad \text{(decimal)}.$$

C_1 is exactly representable with 31 bits only in the significand. The product of C_1 by any integer less than or equal to 5340353 is exactly representable (hence, exactly computed) in binary64 arithmetic. In the considered case, the computed value of kC_1 will be exactly equal to kC_1; that is,

$$\frac{2909491913705529}{536870912}$$

$$= 1010010101100010101010110.1111111111111100100100000111001 \quad \text{(binary)}$$

$$= 5419350.999790297821164131164550781 25 \quad \text{(decimal)},$$

and the computed value of $(x - kC_1)$ will be exactly equal to $x - kC1$,

$$\frac{112583}{536870912}$$

$$= 1.1011011111000111 \times 2^{-13} \quad \text{(binary)}$$

$$= 0.000209702178835868835449218 75 \quad \text{(decimal)}.$$

Therefore, the computed value of $y = (x - kC_1) - kC_2$ will be

$$\frac{704674387127}{18446744073709551616}$$

$$= 1.0100100000100011101110111101010010110111 \times 2^{-25} \quad \text{(binary)}$$

$$= 3.8200475070899238966628214095017\cdots \times 10^{-8} \quad \text{(decimal)},$$

which is much more accurate than what we got using the naive method (we recall that the exact result is $3.820047507089866112093011647\cdots \times 10^{-8}$).

Cody and Waite's idea of splitting constant C into two floating-point numbers can be generalized. For instance, in the library CRlibm[3] [95], it is used with a constant $C = \pi/256$ for small arguments to trigonometric functions ($|x| < 6433$, this threshold value being obtained by error analysis of the algorithm). However, the reduced argument must be known with an accuracy better than 2^{-53}. To this purpose the final subtraction is performed using the Fast2Sum procedure (introduced in Section 4.3.1, page 126). For larger arguments ($6433 \leq |x| < 13176794$), the constant is split into three floating-point numbers, again with a Fast2Sum at the end.

11.1.2 Payne and Hanek's algorithm

Payne and Hanek [327] designed a range reduction algorithm for the trigonometric functions that is very interesting when the input arguments are large. Assume we wish to compute

$$y = x - kC, \tag{11.4}$$

where $C = 2^{-t}\pi$ is the constant of the range reduction. Typical values of C that appear in current programs are between $\pi/256$ and $\pi/4$. Assume x is a binary floating-point number of precision p, and define e_x as its exponent and X as its integral significand, so that $x = X \times 2^{e_x - p + 1}$. Equation (11.4) can be rewritten as

$$y = 2^{-t}\pi \left(\frac{2^{t + e_x - p + 1}}{\pi} X - k \right). \tag{11.5}$$

Let us denote

$$0.v_1 v_2 v_3 v_4 \cdots$$

the infinite binary expansion of $1/\pi$. The main idea behind Payne and Hanek's algorithm is to note that when computing y using (11.5), we do not need to use too many bits of $1/\pi$ for evaluating the product $\frac{2^{t + e_x - p + 1}}{\pi} X$:

---

[3]CRlibm is developed by the Arénaire team of CNRS, INRIA, and ENS Lyon, France. It is available at `http://lipforge.ens-lyon.fr/www/crlibm/`.

- the bit v_i is of weight 2^{-i}. Once multiplied by $2^{t+e_x-p+1}X$, $v_i 2^{-i}$ will become an integral multiple of $2^{t+e_x-p+1-i}$. Hence, once multiplied, later on, by $2^{-t}\pi$, it will become an integral multiple of $2^{e_x-p+1-i}\pi$. Therefore, as soon as $i \leq e_x - p$, the contribution of bit v_i in Equation (11.5) will result in an integral multiple of 2π: it will have no influence on the trigonometric functions. So, in the product

$$\frac{2^{t+e_x-p+1}}{\pi}X,$$

  we can replace the bits of $1/\pi$ of rank i less than or equal to $e_x - p$ by zeros;

- since $|X| \leq 2^p - 1$, the contribution of bits v_i, v_{i+1}, v_{i+2}, v_{i+3}, ... in the reduced argument is less than

$$2^{-t}\pi \times 2^{t+e_x-p+1} \times 2^{-i+1} \times 2^p < 2^{e_x-i+4};$$

  therefore, if we want the reduced argument with an absolute error less than $2^{-\ell}$, we can replace the bits of $1/\pi$ of rank i larger than or equal to $4 + \ell + e_x$ by zeros.

11.2   Bounding the Relative Error of Range Reduction

Payne and Hanek's algorithm (as well as Cody and Waite's algorithm or similar methods) allows one to easily bound the absolute error on the reduced argument. And yet, we are more interested in bounding *relative errors* (or errors in ulps). To do that, we need to know the smallest possible value of the reduced argument. That is, we need to know what is the smallest possible value of

$$x - k \cdot C,$$

where x is a floating-point number of absolute value larger than C.

If X is the integral significand of x and e_x is its exponent, the problem becomes that of finding very good rational approximations to C, of the form

$$\frac{X \cdot 2^{e_x-p+1}}{k},$$

which is a typical continued fraction problem (see Section 16.1). This problem is solved using a continued fraction method due to Kahan [203] (a very similar method is also presented in [380]). See [293, 302] for more details.

Program 11.1, written in Maple, computes the smallest possible value of the reduced argument and the input value for which it is attained.

```
        worstcaseRR := proc(Beta,p,efirst,efinal,C,ndigits)
          local epsilonmin,powerofBoverC,e,a,Plast,r,Qlast,
                Q,P,NewQ,NewP,epsilon,
              numbermin,expmin,ell;
              epsilonmin := 12345.0 ;
              Digits := ndigits;
              powerofBoverC := Beta^(efirst-p)/C;
              for e from efirst-p+1 to efinal-p+1 do
                  powerofBoverC := Beta*powerofBoverC;
                  a := floor(powerofBoverC);
                  Plast := a;
                  r := 1/(powerofBoverC-a);
                  a := floor(r);
                  Qlast := 1;
                  Q := a;
                  P := Plast*a+1;
                  while Q < Beta^p-1 do
                      r := 1/(r-a);
                      a := floor(r);
                      NewQ := Q*a+Qlast;
                      NewP := P*a+Plast;
                      Qlast := Q;
                      Plast := P;
                      Q := NewQ;
                      P := NewP
                  od;
                  epsilon :=
                      evalf(C*abs(Plast-Qlast*powerofBoverC));
                  if epsilon < epsilonmin then
                      epsilonmin := epsilon; numbermin := Qlast;
                      expmin := e
                  fi
              od;
              printf("The worst case occurs\n for x = %a * %a ^ %a,\n",
                  numbermin,Beta,expmin);
              printf("The corresponding reduced argument is:\n %a\n",
                  epsilonmin);
              ell := evalf(log(epsilonmin)/log(Beta),10);
              printf("whose radix %a logarithm is %a",Beta,ell)
          end:
```

Program 11.1: This Maple program, extracted from [293], gives the worst cases for range reduction, for constant C, assuming the floating-point systems has radix Beta, *precision* p, *and that we consider input arguments of exponents between* efirst *and* efinal. *It is based on Kahan's algorithm [203]. Variable* ndigits *indicates the radix-10 precision with which the Maple calculations must be carried out. A good rule of thumb is to choose a value slightly larger than* $(\text{efinal} + 1 + 2p) \log(\text{Beta})/\log(10) + \log(\text{efinal} - \text{efirst} + 1)/\log(10)$.

For instance, by entering

```
worstcaseRR(2,53,0,1023,Pi/2,400);
```

we get

```
The worst case occurs
 for x = 6381956970095103 * 2 ^ 797,
The corresponding reduced argument is:
 .46871659242546276111225828019638843989495... e-18
whose radix 2 logarithm is -60.88791794
```

which means that, for double-precision/binary64 inputs and $C = \pi/2$, a reduced argument will never be of absolute value less than $2^{-60.89}$. This has interesting consequences, among them:

- to make sure that the relative error on the reduced argument will be less than η, it suffices to make sure that the absolute error is less than $\eta \times 2^{-60.89}$;

- since the sine and tangent of $2^{-60.89}$ are much larger than $2^{e_{\min}} = 2^{-1022}$, the sine, cosine, and tangent of a double-precision/binary64 floating-point number larger than $2^{e_{\min}}$ cannot underflow;

- since the tangent of $\pi/2 - 2^{-60.89}$ is much less than $\Omega \approx 1.798 \times 10^{308}$, the tangent of a double-precision/binary64 floating-point number cannot overflow.

Table 11.1, extracted from [293], gives the input values for which the smallest reduced argument is reached, for various values of C and various formats. These values are obtained using Program 11.1.

11.3   More Sophisticated Range Reduction Algorithms

The most recent elementary function libraries use techniques that are more sophisticated than the Cody and Waite or the Payne and Hanek methods presented above. First, several reduction steps may be used. Many current implementations derive from P. Tang's *table-based methods* [404, 405, 406, 407]. Second, the range reduction and polynomial evaluation steps are not really separated, but somewhat interleaved. Also, very frequently, the reduced argument y is not just represented by one floating-point number: it is generally represented by several floating-point values, whose sum approximates y (this is called an *expansion*, see Chapter 14). Another important point is the following: although in the first implementations, the reduced arguments would lie in fairly large intervals (typically, $[-\pi/4, +\pi/4]$ for trigonometric functions, $[0, 1]$ or $[0, \ln(2)]$ for the exponential function), current implementations tend to use reduced arguments that lie in much smaller intervals. This

β	p	C	efinal	Worst case	$-\log_r(\epsilon)$
2	24	$\pi/2$	127	$16367173 \times 2^{+72}$	29.2
2	24	$\pi/4$	127	$16367173 \times 2^{+71}$	30.2
2	24	$\ln(2)$	127	8885060×2^{-11}	31.6
2	24	$\ln(10)$	127	9054133×2^{-18}	28.4
10	10	$\pi/2$	99	$8248251512 \times 10^{-6}$	11.7
10	10	$\pi/4$	99	$4124125756 \times 10^{-6}$	11.9
10	10	$\ln(10)$	99	$7908257897 \times 10^{+30}$	11.7
2	53	$\pi/2$	1023	$6381956670095103 \times 2^{+797}$	60.9
2	53	$\pi/4$	1023	$6381956670095103 \times 2^{+796}$	61.9
2	53	$\ln(2)$	1023	$5261692873635770 \times 2^{+499}$	66.8

Table 11.1: Worst cases for range reduction for various floating-point systems and reduction constants C [293]. The corresponding reduced argument is ϵ.

is due to the fact that the degree of the smallest polynomial that approximates a function f in an interval I within some given error decreases with the width of I (the speed at which it decreases depends on f, but it is very significant). This is illustrated by Table 11.2, extracted from [293].

a	arctan	exp	$\ln(1+x)$
10	19	16	15
1	6	5	5
0.1	3	2	3
0.01	1	1	1

Table 11.2: Degrees of the minimax polynomial approximations that are required to approximate some functions with error less than 10^{-5} on the interval $[0, a]$. When a becomes small, a very low degree suffices [293].

We now give two examples of reduction methods used in the CRlibm library of correctly rounded elementary functions [95, 245]. We assume double-precision/binary64 input values.

11.3.1   An example of range reduction for the exponential function

The natural way to deal with the exponential function is to perform an additive range reduction, and later on to perform a multiplicative "reconstruction." First, from the input argument x, we find z, $|z| < \ln(2)$, so that

$$e^x = e^{E \cdot \ln(2) + z} = \left(e^{\ln(2)}\right)^E \cdot e^z = 2^E \cdot e^z, \tag{11.6}$$

where E is a signed integer.

One immediately notes that the use of such an argument reduction implies a multiplication by the transcendental constant $\ln(2)$. This means that the reduced argument will not be exact, and the reduction error has to be taken into account when computing a bound on the overall error of the implemented function.

A reduced argument obtained by the reduction shown above is generally still too large to be suitable for a polynomial approximation of reasonable degree. A *second reduction step* is therefore necessary. By the use of table lookups, the following method can be employed to implement that second step. It yields smaller reduced arguments. Let ℓ be a small positive integer (a typical value is around 12). Let w_1 and w_2 be positive integers such that $w_1 + w_2 = \ell$. Let

$$k = \left\lfloor z \cdot \frac{2^\ell}{\ln(2)} \right\rceil,$$

where $\lfloor u \rceil$ is the integer closest to u. We compute the final reduced argument y, and integers M, $i_1 < 2^{w_1}$, and $i_2 < 2^{w_2}$ such that

$$
\begin{aligned}
y &= z - k \cdot \frac{\ln(2)}{2^\ell} \\
k &= 2^\ell \cdot M + 2^{w_1} \cdot i_2 + i_1,
\end{aligned}
\tag{11.7}
$$

which gives

$$
\begin{aligned}
e^z &= e^y \cdot 2^{k/2^\ell}, \\
&= e^y \cdot 2^{M + i_2/2^{w_2} + i_1/2^\ell}.
\end{aligned}
\tag{11.8}
$$

The two integers w_1 and w_2 are the widths of the indices to two tables T_1 and T_2 that store

$$t_1 = 2^{i_2/2^{w_2}}$$

and

$$t_2 = 2^{i_1/2^\ell}.$$

A polynomial approximation will be used for evaluating e^y. From (11.6) and (11.8), we deduce the following "reconstruction" step:

$$e^x = 2^E \cdot e^z = 2^{M+E} \cdot t_1 \cdot t_2 \cdot e^y.$$

This argument reduction ensures that

$$|y| \leq \ln(2)/2^\ell < 2^{-\ell}.$$

This magnitude is small enough to allow for efficient polynomial approximation.

Typical values currently used are $\ell = 12$ and $w_1 = w_2 = 6$.

The subtraction in $y = z - k \cdot \text{RN}\left(\ln(2)/2^\ell\right)$ can be implemented exactly, but it leads to a catastrophic cancellation that amplifies the absolute error of the potentially exact multiplication of k by the approximated $\ln(2)/2^\ell$. A careful implementation must take that into account.

11.3.2 An example of range reduction for the logarithm

The range reduction algorithm shown below is derived from one due to Wong and Goto [438] and discussed further in [293]. The input argument x is initially reduced in a straightforward way, using integer arithmetic, in order to get an integer E' and a double-precision/binary64 floating-point number m so that:

$$x = 2^{E'} \cdot m$$

with $1 \leq m < 2$.

In the general case (i.e., when x is a normal number), E' is the exponent of x and m is its significand. And yet, this first decomposition is performed so that in any case (i.e., even if x is subnormal) $1 \leq m < 2$: in the subnormal case, the exponent of x is adjusted accordingly.

This first argument reduction corresponds to the equation

$$\ln(x) = E' \cdot \ln(2) + \ln(m). \tag{11.9}$$

Using (11.9) directly would lead to a catastrophic cancellation in the case $E' = -1$ and $m \approx 2$. To overcome this problem, an adjustment is necessary. It is made by defining an integer E and a double-precision/binary64 floating-point number y as follows:

$$E = \begin{cases} E' & \text{if} \quad m \leq \sqrt{2} \\ E' + 1 & \text{if} \quad m > \sqrt{2} \end{cases}, \tag{11.10}$$

and

$$y = \begin{cases} m & \text{if} \quad m \leq \sqrt{2} \\ m/2 & \text{if} \quad m > \sqrt{2} \end{cases}. \tag{11.11}$$

The test $m \leq \sqrt{2}$ need not be performed exactly—of course, the very same test must be performed for (11.10) and (11.11). Here, the bound $\sqrt{2}$ is somewhat arbitrary. Indeed, software implementations perform this test using integer arithmetic on the high order bits of m, whereas hardware implementations

have used 1.5 instead of $\sqrt{2}$ to reduce this comparison to the examination of two bits [116].

The previous reduction step can be implemented exactly, as it mainly consists in extracting the exponent and significand fields of a floating-point number, and multiplying by powers of 2. We have:

$$\ln(x) = E \cdot \ln(2) + \ln(y), \tag{11.12}$$

where

$$-\frac{\ln(2)}{2} \leq \ln(y) \leq +\frac{\ln(2)}{2}.$$

The magnitude of this first reduced argument y is too large to allow one to approximate $\ln(y)$ by a polynomial of reasonably small degree with very good accuracy: a second range reduction step will be necessary. That second reduction step is performed using a table with 2^ℓ entries as follows:

- using the high order ℓ bits of y as an index i, one looks up a tabulated value r_i that approximates $\frac{1}{y}$ very well;

- setting $z = y \cdot r_i - 1$, one obtains

$$\ln(y) = \ln(1 + z) - \ln(r_i). \tag{11.13}$$

Since y and $1/r_i$ are very close, the magnitude of the final reduced argument z is now small enough (roughly speaking, $|z| < 2^{-\ell}$) to allow good approximation of $\ln(1 + z)$ by a minimax polynomial $p(z)$ of reasonably small degree. The method requires tabulation of the values $\ln(r_i)$. Current implementations use values of ℓ between 6 and 8, leading to tables of 64 to 256 entries.

It is important to notice that the final reduction step

$$z = y \cdot r_i - 1$$

can be implemented exactly. However, this requires the reduced argument z to be represented either in a format wider than the input format (a few more bits are enough), or as an unevaluated sum of two floating-point numbers.

From (11.12) and (11.13), we easily deduce the "reconstruction" step:

$$\ln(x) \approx E \cdot \ln(2) + p(z) - \ln(r_i).$$

11.4 Polynomial or Rational Approximations

After the range reduction step, our problem is reduced to the problem of approximating a given function f by a polynomial p, in a rather small interval $[a, b]$. We will consider two cases, depending on whether we wish to minimize

$$||f - p||_{L^2, [a,b], w} = \int_a^b w(x) \left(f(x) - p(x) \right)^2 dx,$$

(this is the L^2 *case*), where w is a positive *weight* function, or

$$||f - p||_{\infty,[a,b]} = \sup_{x \in [a,b]} |f(x) - p(x)|$$

(this is the L^∞, *or minimax case*).

We will first recall the classical methods for getting polynomial approximations. Then, we will deal with the much more challenging problem of getting approximations "with constraints." Examples of constraints that we wish to use are requiring all coefficients to be exactly representable in floating-point arithmetic, or requiring some of them to be zero.

11.4.1 L^2 case

Finding an L^2 approximation is done by means of a *projection*. More precisely,

$$< f, g > = \int_a^b w(x) f(x) g(x) dx$$

defines an inner product in the vector space of the continuous functions from $[a, b]$ to $\mathbb{R}$. We get the degree-n L^2 approximation p^* to some function f by projecting f (using the projection associated with the above-defined inner product) on the subspace $\mathcal{P}_n$ constituted by the polynomials of degree less than or equal to n. Computing p^* is easily done once we have precomputed an orthogonal basis of $\mathcal{P}_n$. The basis we will use is a family $(T_k), 0 \leq k \leq n$, of polynomials, called *orthogonal polynomials*, such that:

- T_k is of degree k;

- $< T_i, T_j > = 0$ if $i \neq j$.

  Once the T_k are computed,[4] p^* is obtained as follows:

- compute, for $0 \leq k \leq n$,

$$a_k = \frac{< f, T_k >}{< T_k, T_k >},$$

- then get

$$p^* = a_0 T_0 + a_1 T_1 + \cdots + a_n T_n.$$

This is illustrated by Figure 11.3.

A very useful example of a family of orthogonal polynomials is the *Chebyshev polynomials*. The Chebyshev polynomials can be defined either by the recurrence relation

---

[4]For the usual weight functions $w(x)$ and the interval $[-1, 1]$, the orthogonal polynomials have been known for decades, so there is no need to recompute them.
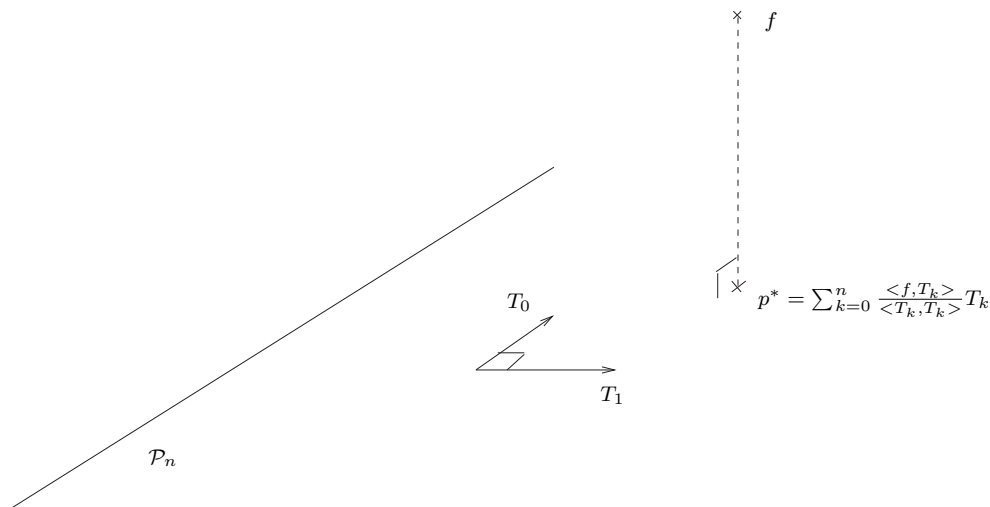
Figure 11.3: The L^2 approximation p^* is obtained by projecting f on the subspace generated by T_0, T_1, ..., T_n.

$$\left\{ \begin{array}{rcl} T_0(x) & = & 1, \\ T_1(x) & = & x, \\ T_n(x) & = & 2xT_{n-1}(x) - T_{n-2}(x); \end{array} \right.$$

or by

$$T_n(x) = \left\{ \begin{array}{l} \cos\left(n\cos^{-1}x\right) \ \text{for}\ |x| \le 1, \\ \cosh\left(n\cosh^{-1}x\right)\ \text{for}\ x > 1. \end{array} \right.$$

They are orthogonal polynomials for the inner product associated with the weight function

$$w(x) = \frac{1}{\sqrt{1-x^2}},$$

and the interval $[a, b] = [-1, 1]$. A detailed presentation of Chebyshev polynomials can be found in [39] and [343]. These polynomials play a central role in approximation theory.

11.4.2 L^∞, or minimax case

As previously, define $\mathcal{P}_n$ as the subspace constituted by the polynomials of degree less than or equal to n. The central result in minimax approximation theory is the following theorem, due to Chebyshev.
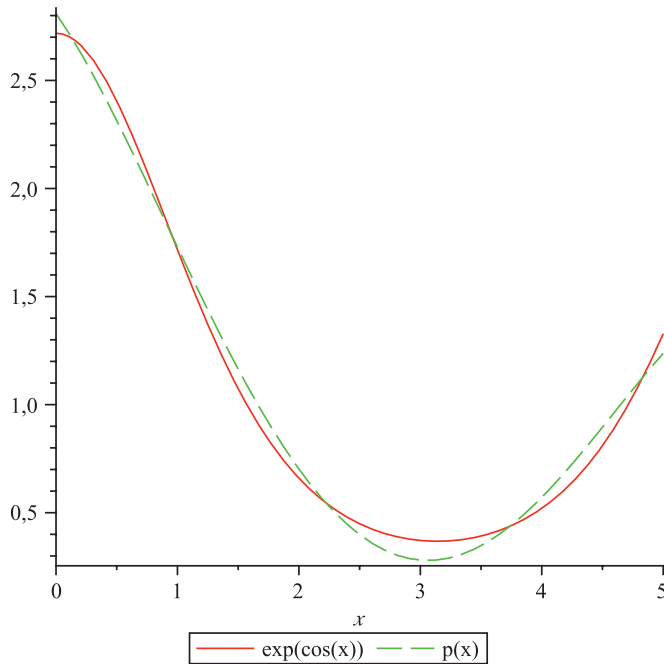
Figure 11.4: The $\exp(\cos(x))$ *function and its degree-4 minimax approximation on* $[0, 5]$, $p(x)$. *There are six values where the maximum approximation error is reached with alternate signs.*

Theorem 34 (Chebyshev). p^* *is the minimax degree-n approximation to f on $[a, b]$ if and only if there exist at least $n + 2$ values*

$$a \leq x_0 < x_1 < x_2 < \cdots < x_{n+1} \leq b$$

such that:

$$p^*(x_i) - f(x_i) = (-1)^i \left[p^*(x_0) - f(x_0) \right] = \pm ||f - p^*||_\infty.$$

Chebyshev's theorem shows that if p^* is the minimax degree-n approximation to f, then the largest approximation error is reached at least $n + 2$ times, and that the sign of the error *alternates*. This is illustrated by Figure 11.4. That property is used by an algorithm, due to Remez [174, 342], that computes the minimax degree-n approximation to a continuous function iteratively.

There is a similar result concerning minimax rational approximations to functions [293].

A good implementation of the Remez algorithm that works even in slightly degenerate cases is fairly complex. Here, we just give a rough sketch of that algorithm. It consists in iteratively building the set of points $x_0, x_1, \ldots,$ x_{n+1} of Theorem 34. This is done as follows.

1. First, we start from an initial set of points $x_0, x_1, \ldots, x_{n+1}$ in $[a, b]$. There are some heuristics for choosing a "good" starting set of points.

2. We consider the linear system (whose unknowns are $p_0, p_1, \ldots, p_n$ and ϵ):

$$\begin{cases} p_0 + p_1 x_0 + p_2 x_0^2 + \cdots + p_n x_0^n - f(x_0) & = & +\epsilon \\ p_0 + p_1 x_1 + p_2 x_1^2 + \cdots + p_n x_1^n - f(x_1) & = & -\epsilon \\ p_0 + p_1 x_2 + p_2 x_2^2 + \cdots + p_n x_2^n - f(x_2) & = & +\epsilon \\ \cdots & & \cdots \\ p_0 + p_1 x_{n+1} + p_2 x_{n+1}^2 + \cdots + p_n x_{n+1}^n - f(x_{n+1}) & = & (-1)^{n+1}\epsilon. \end{cases}$$

   In all nondegenerated cases, it will have a unique solution $(p_0, p_1, \ldots, p_n, \epsilon)$. Solving this system therefore gives us a polynomial $P(x) = p_0 + p_1 x + \cdots + p_n x^n$.

3. We compute the set of points y_i in $[a, b]$ where $P - f$ has its extremes, and we start again (step 2), replacing the $x_i's$ by the y_i's.

It can be shown [139] that this is a convergent process and that, under some conditions, the speed of convergence is quadratic [415].

What we have done here (approximating a continuous function by a degree-n polynomial, i.e., a linear combination of the monomials $1, x, x^2, \ldots,$ x^n) can be generalized to the approximation of a continuous function by a linear combination of continuous functions $g_0, g_1, \ldots, g_n$ that satisfy the *Haar condition*: for any subset of distinct points $x_0, x_1, \ldots, x_n$, the determinant

$$\begin{vmatrix} g_0(x_0) & g_1(x_0) & g_2(x_0) & \cdots & g_n(x_0) \\ g_0(x_1) & g_1(x_1) & g_2(x_1) & \cdots & g_n(x_1) \\ g_0(x_2) & g_1(x_2) & g_2(x_2) & \cdots & g_n(x_2) \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ g_0(x_n) & g_1(x_n) & g_2(x_n) & \cdots & g_n(x_n) \end{vmatrix}$$

is nonzero.

11.4.3 "Truncated" approximations

One frequently wishes to have polynomial approximations with a particular form:

- approximations whose coefficients are exactly representable in floating-point arithmetic (or, for some coefficients that have much influence on the final accuracy, that are equal to the sum of two or three floating-point numbers);

- approximations that have some coefficients equal to zero (for instance, of the form $x + x^3 p(x^2)$ for the sine function: this preserves symmetry);

- approximations whose first coefficients coincide with those of the Taylor series (in general, to improve the behavior of the approximation near zero).

Of course, one can first compute a conventional approximation (e.g., using the Remez algorithm), and then constraint (by rounding) its coefficients to satisfy the desired requirements. In most cases, the obtained approximation will be significantly worse than the best approximation that satisfies the requirements.

Consider the following example, drawn from CRlibm. To evaluate function arcsin near 1 with correct rounding in the double-precision/binary64 format, after a change of variables we actually have to compute

$$g(z) = \frac{\arcsin(1 - (z + m)) - \frac{\pi}{2}}{\sqrt{2 \cdot (z + m)}},$$

where 0xBFBC28F800009107 $\leq z \leq$ 0x3FBC28F7FFFF6EF1 (roughly speaking $-0.110 \leq z \leq 0.110$) and $m =$ 0x3FBC28F80000910F $\simeq 0.110$. We want to generate a degree-21 polynomial approximation. If we round to nearest the coefficients of the Remez polynomial, we get a maximum error of around 8×10^{-32}. If we use tools designed in the Arénaire team to find "nearly best" approximations whose coefficients are floating-point numbers, we get a maximum error of around 8×10^{-37}: the generated polynomial is almost $10,000$ times more accurate than the naive rounded Remez polynomial.

The tools we use are based on two techniques: linear programming and the Lenstra–Lenstra–Lovász (LLL) algorithm. See [53, 54, 48] for more details.

11.5 Evaluating Polynomials

Once we have found a convenient polynomial approximation, we are reduced to the problem of evaluating $p(x)$, where x lies in some small domain.

The polynomial $p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$, obtained using one of the methods presented in the previous section, has coefficients that are either floating-point numbers or sums of a few (generally, two) floating-point numbers. Also, the variable x, obtained after range reduction, may be represented as the unevaluated sum of two or three floating-point numbers.

Hence, evaluating $p(x)$ quickly and accurately on a modern pipelined floating-point unit is a nontrivial task.

Of course, one should never evaluate $p(x)$ using the straightforward sequence of operations

$$a_0 + (a_1 \times x) + (a_2 \times x \times x) + (a_3 \times x \times x \times x) + \cdots + (a_n \times \underbrace{x \times x \times \cdots \times x}_{n-1 \text{ multiplications}}).$$

Evaluating a degree-n polynomial using that method would require $n(n + 3)/2$ floating-point operations: $n(n + 1)/2$ multiplications and n additions.

All recent evaluation schemes are mix-ups of the well-known *Horner's rule* (Algorithm 6.3, page 185), and *Estrin's* method. Choosing a good evaluation scheme requires considering the target architecture (availability of an FMA instruction, depth of the pipelines, etc.) and performing some error analysis. A good rule of thumb is that Estrin's method wins for latency, and Horner's rule for throughput.

An example is given in Section 11.7.2. A group at Intel who designed elementary functions for the IA-64 processor [173, 157] determined by (almost) exhaustive enumeration the optimal evaluation method (in terms of latency) of polynomials up to moderate degrees.

The accuracy of Horner's rule is given by Equation (6.6), page 186. A similar bound can be derived for Estrin's method.

Horner's rule computes $p(x) = \sum_{i=0}^{n} a_i x^i$ by nested multiplications as

$$p(x) = \big(\dots ((a_n x + a_{n-1})x + a_{n-2})x + \cdots \big)x + a_0,$$

while Estrin's method uses a binary evaluation tree implied by splitting $p(x)$ as

$$\big(\sum_{0 \le i < h} a_{h+i} x^i \big) x^h + \big(\sum_{0 \le i < h} a_i x^i \big),$$

where $h = (n + 1)/2$ is a power of 2.

11.6   Correct Rounding of Elementary Functions to binary64

11.6.1   The Table Maker's Dilemma and Ziv's onion peeling strategy

With a few exceptions, the image y of a floating-point number x by a transcendental function f is a transcendental number (see Section 12.3.1, page 420, for a definition of these notions), and can therefore not be represented exactly in standard number systems. As a consequence, we need to round it. We would like to round it *correctly*; that is, to always return $\circ(f(x))$, where $\circ$ is the active rounding mode (or rounding direction attribute), chosen among the four presented in Section 2.2, page 20.

A computer may evaluate an approximation $\hat{y}$ to the real number y with relative accuracy $\bar{\epsilon}$. This means that the real value y belongs to the interval $[\hat{y}(1 - \bar{\epsilon}), \hat{y}(1 + \bar{\epsilon})]$. Sometimes, however, this information is not enough to decide correct rounding. For example, if $[\hat{y}(1 - \bar{\epsilon}), \hat{y}(1 + \bar{\epsilon})]$ contains the midpoint between two consecutive floating-point numbers, it is impossible to decide which of these two numbers is the floating-point number nearest to y.

This problem is known as the *Table Maker's Dilemma (TMD)*. It will be discussed in detail in Chapter 12.

Ziv's technique [444] was implemented in the `libultim` library.[5] It consists in progressively improving the accuracy $\bar{\epsilon}$ of the approximation until the correctly rounded value can be decided. Given a function f and an argument x, the value of $f(x)$ is first quickly approximated, using a "simple and fast" approximation of accuracy $\bar{\epsilon}_1$. Knowing $\bar{\epsilon}_1$ and the approximation, it is possible to decide if correct rounding is possible, or if more accuracy is required, in which case the computation is restarted using a slower yet more accurate approximation (of accuracy $\bar{\epsilon}_2$), and so on. If the $\bar{\epsilon}_i$ are adequately chosen, this approach leads to good average performance, as the slower steps are rarely taken.

However, there was, until recently, no practical bound on the termination time of Ziv's iteration: it may be proven to terminate for some transcendental functions (because of Lindemann's theorem; see Section 12.4.1, page 429), but the actual maximal accuracy required in the worst case was unknown. In `libultim`, the measured worst-case execution time is indeed three orders of magnitude larger than that of the usual `libm`s. A related problem is memory requirement, which is, for the same reason, unbounded in theory and much higher than the usual `libm`s in practice.

11.6.2   When the TMD is solved

Much effort has been devoted during the last 10 years by Lefèvre and Muller, and then by Stehlé, Hanrot, and Zimmermann, to compute the actual worst-case accuracies needed to guarantee correct rounding of the main elementary functions in double-precision/binary64 arithmetic.[6]

The techniques used will be the subject of Chapter 12. It was determined, for example, that evaluating a logarithm to an accuracy of 2^{-118} was enough to be able to decide its correct rounding to a binary64 floating-point number for any binary64 input.

Because of such results, it is now possible to obtain correct rounding in two Ziv steps only, which we may then optimize separately:

- the first *quick* step is as fast as a current `libm`, and provides an accuracy

---

[5]`libultim` was released by IBM. An updated version is now part of the GNU `glibc` and available under the GNU General Public License.

[6]Some results have recently been obtained in decimal64 arithmetic: for example, the hardest-to-round case for the exponential function in that format is known. Getting hardest-to-round cases for the binary32 or decimal32 formats only requires a few hours of computation: we need to examine 2^{32} cases for each function, which is easily done with current computers. Getting hardest-to-round cases in formats significantly larger than binary64 or decimal64 seems a difficult challenge: with the smallest possible extended precisions associated to binary64 or decimal64, as defined in Table 3.24, page 94, it will be feasible within a few years, but larger precisions such as binary128 or decimal128 seem out of reach with current techniques.

between 2^{-60} and 2^{-80} (depending on the function), which is sufficient to round correctly to the 53 bits of double precision in most cases;

- the second *accurate* step is dedicated to challenging cases. It is slower but has a reasonably bounded execution time, being tightly targeted at the hardest-to-round cases given in Chapter 12. In particular, there is no need anymore for arbitrary multiple precision.

This was the approach used in CRlibm. Let us now detail some tricks used in that library.

11.6.3 Rounding test

Let $\hat{y}_1$ be the approximation to $y = f(x)$ obtained at the end of the fast step. The test on $\hat{y}_1$, which either returns a correctly rounded value or launches the second step, is called a *rounding test*. The property that a rounding test must ensure is the following: a value will be returned only if it can be proven to be the correctly rounded value of y; otherwise (in doubt), the second step will be launched. To get good performance, it is essential that the rounding test be fast.

A rounding test depends on a bound $\overline{\epsilon}_1$ on the overall relative error of the first step. This bound is usually computed statically, although in some cases it can be refined at runtime. Techniques for computing $\overline{\epsilon}_1$ will be surveyed in Section 11.7.

The implementation of a rounding test depends on the rounding mode and the nature of $\hat{y}_1$, which may be a double-extended or a "double-double" number.[7] Besides, in each case, there are several sequences which are acceptable as rounding tests.

Some rounding tests are conceptually simple. If $\hat{y}_1$ is a double-extended number, it suffices to extract the significand of $\hat{y}_1$, then perform bit mask operations on the bits after bit 53, looking for a string of zeros in the case of directed rounding mode, or for a string of the form 10^k or 01^k for round to nearest. Here, k is deduced from $\overline{\epsilon}_1$. Examples can be found in Itanium-specific code inside the CRlibm distribution, because this architecture offers both native 64-bit arithmetic and efficient instructions for extracting the significand of a floating-point number.

If $\hat{y}_1$ is computed as a double-double, a better option is to use only double-precision floating-point operations. Listing 11.1 describes a rounding test for round to nearest.

---

[7]A "double-double" number is the unevaluated sum of two binary64 floating-point numbers. Techniques for manipulating such "double-word" numbers are presented in Chapter 14.

C listing 11.1 Floating-point-based rounding test.

```
if( yh == yh + yl*e)) )
    return yh;
else
    /* more accuracy is needed, launch accurate phase */
```

The constant e is slightly larger than 1, and its relationship to $\bar{\epsilon}_1$ is given by Theorem 35, taken from the CRlibm documentation [95] (this test is already present in Ziv's `libultim`, but neither the test itself nor the way the constants e have been obtained is documented).

Theorem 35 (Correct rounding of a double-double to the nearest double, avoiding subnormals). *Let y be a real number, and $\bar{\epsilon}_1$, e, y_h, and y_l be double-precision floating-point numbers such that*

- $y_h = \circ(y_h + y_l)$

- *neither y_h nor y_l is a NaN or $\pm\infty$,*

- $|y_h| \geq 2^{-1022+54}$ *(i.e., $\frac{1}{4}$ ulp(y_h) is not subnormal),*

- $|y_h + y_l - y| < \bar{\epsilon}_1.|y|$ *(i.e., the total relative error of $y_h + y_l$ with respect to y is bounded by $\bar{\epsilon}_1$),*

- $0 < \bar{\epsilon}_1 \leq 2^{-53-k}$ *with $k \geq 3$ integer,*

- $e \geq (1 - 2^{-53})^{-1} \left(1 + \dfrac{2^{54}\bar{\epsilon}_1}{1 - \bar{\epsilon}_1 - 2^{-k+1}}\right)$ *and* $e \leq 2.$

The code sequence 11.1, with the two operations performed in round-to-nearest mode, determines whether y_h is the correctly rounded value of y in round-to-nearest mode.

One may note that the condition $|y_h| \geq 2^{-1022+54}$ implies that y_h is a normal number. This test is not proven for subnormal numbers, but this is not a problem in practice for elementary functions.

- For the trigonometric functions (sine, cosine, tangent, and arctangent), one may deduce from the worst cases of argument reduction obtained by Program 11.1 that the value of the function never comes close to a subnormal.

- All the other elementary functions (as well as sine and tangent in the neighborhood of zero) have asymptotic properties that may be used to avoid the computation and the rounding test altogether, when their result is subnormal. These properties will be provided for each function in the next chapter, see Tables 12.4 (page 417) and 12.5 (page 418).

Proof. The implication we need to prove is: if the test is true, then $y_h = \circ(y)$ (failure of the test does not necessarily mean that $y_h \neq \circ(y)$).

Let us note $u = \mathrm{ulp}(y_h)$ and consider only the case when y_h is positive (as the other case is symmetrical).

We have to consider separately the following two cases.

- **If y_h is not a power of 2 or $y_l \geq 0$**

  In this case we will always assume that $y_l \geq 0$, as the case $y_l \leq 0$ is symmetrical when y_h is not a power of 2. To prove that $y_h = \circ(y)$, it is enough to prove that $|y_h - y| \leq u/2$. As $|y_h + y_l - y| < \bar{\epsilon}_1.|y|$ (fourth hypothesis) it is enough to prove that $u/2 - y_l > \bar{\epsilon}_1 y$.

  By definition of the ulp of a positive normal number, we have $y_h \in [2^{52}u, (2^{53} - 1)u]$.

  From the first hypothesis we have

  $$y_l \leq \frac{1}{2}u. \tag{11.14}$$

  Therefore, $y_h + y_l \leq (2^{53} - 1)u + \frac{1}{2}u$, and

  $$y < (y_h + y_l)/(1 - \bar{\epsilon}_1).$$

  Hence,

  $$y < \frac{2^{53} - \frac{1}{2}}{1 - \bar{\epsilon}_1}u.$$

  As a consequence, since $\bar{\epsilon}_1 \leq 2^{-56}$,

  $$y < 2^{53}u. \tag{11.15}$$

  The easy case is when we have $y_h = \circ(y)$ regardless of the result of the test. This is true as soon as y_l is sufficiently distant from $u/2$. More specifically, if $0 \leq y_l < \left(\frac{1}{2} - 2^{-k}\right)u$, we combine (11.15) with the fifth hypothesis to get $\bar{\epsilon}_1 y < 2^{-k}u$. From $y_l < \left(\frac{1}{2} - 2^{-k}\right)u$ we deduce $u/2 - y_l > 2^{-k}u > \bar{\epsilon}_1 y$, which proves that $y_h = \circ(y)$. Now consider the case when $y_l \geq \left(\frac{1}{2} - 2^{-k}\right)u$. The condition $|y_h| \geq 2^{-1022+54}$ ensures that $u/4$ is a normal number, and now $y_l > u/4$, so in this case y_l is a normal number. As $1 < e \leq 2$, the result is also normal. Therefore,

  $$y_l \times e(1 - 2^{-53}) \leq \circ(y_l \times e) \leq y_l \times e(1 + 2^{-53}).$$

  Suppose that the test is true ($\circ(y_h + \circ(y_l \times e)) = y_h$). With rounding to nearest, this implies $|\circ(y_l \times e)| \leq \frac{u}{2}$, which in turn implies

$y_l \times e(1 - 2^{-53}) \leq \dfrac{u}{2}$ (as y_l is a normal number and $1 < e \leq 2$). This is rewritten as:

$$\frac{u}{2} - y_l \geq y_l \left(e\left(1 - 2^{-53}\right) - 1\right).$$

Using $y_l \geq (\frac{1}{2} - 2^{-k})u$, we get

$$\frac{u}{2} - y_l \geq \left(\frac{1}{2} - 2^{-k}\right) u \left(e\left(1 - 2^{-53}\right) - 1\right).$$

We want to ensure that $\dfrac{u}{2} - y_l \geq \bar{\epsilon}_1 y$; we will again use (11.15) and ensure that $\dfrac{u}{2} - y_l \geq 2^{53}\bar{\epsilon}_1 u$. This provides the condition that must be fulfilled by e for the theorem to hold in this case: we need

$$\left(\frac{1}{2} - 2^{-k}\right) u \left(e(1 - 2^{-53}) - 1\right) \geq 2^{53}\bar{\epsilon}_1 u$$

rewritten as:

$$e \geq (1 - 2^{-53})^{-1} \left(1 + \frac{2^{54}\bar{\epsilon}_1}{1 - 2^{-k+1}}\right).$$

- **If y_h is a power of 2 and $y_l < 0$**

  To prove that $y_h = \circ(y)$, it is enough to prove that $|y_h - y| \leq u/4$. As $y_l \leq 0$, we have $y \leq y_h$ and $|y_h - y| = y - y_h$.

  From the fourth hypothesis, it is enough to prove that $u/4 + y_l > \bar{\epsilon}_1 y$.

  By our definition of the ulp of a normal number, we have $y_h = 2^{52}u$ in this case.

  We have $y_h = 2^{52}u$ and $y_l \leq 0$; therefore, $y_h + y_l \leq 2^{52}u$, and

  $$y < \frac{2^{52}u}{1 - \bar{\epsilon}_1}. \tag{11.16}$$

  The easy case is when we have $y_h = \circ(y)$ regardless of the result of the test. This is true as soon as y_l is sufficiently distant from $-u/4$. More specifically, if $-\left(\frac{1}{4} - \frac{2^{-k-1}}{1-\bar{\epsilon}_1}\right) u < y_l \leq 0$, after combining (11.16) with the fifth hypothesis to get $\bar{\epsilon}_1 y < \frac{2^{-k-1}u}{1-\bar{\epsilon}_1}$, we deduce $y_l + \frac{u}{4} > \frac{2^{-k-1}}{1-\bar{\epsilon}_1}u > \bar{\epsilon}_1 y$, which proves that $y_h = \circ(y)$.

  Now consider the case when $-y_l \geq \left(\frac{1}{4} - \frac{2^{-k-1}}{1-\bar{\epsilon}_1}\right) u$. The condition $|y_h| \geq 2^{-1022+54}$ ensures that $u/8$ is a normal number, and now $y_l > u/8$, so in this case y_l is a normal number. As $1 < e \leq 2$, the result is also normal; therefore,

  $$-y_l \times e(1 - 2^{-53}) \leq -\circ(y_l \times e) \leq -y_l \times e(1 + 2^{-53}).$$

Suppose that the test is true $(\circ(y_h + \circ(y_l \times e)) = y_h)$. For this value of y_h and this sign of y_l, this implies $|\circ (y_l \times e)| \leq \dfrac{u}{4}$, which in turn implies

$$-y_l \times e(1 - 2^{-53}) \leq \frac{u}{4}.$$

This is rewritten as:

$$\frac{u}{4} + y_l \geq -y_l \left(e \left(1 - 2^{-53} \right) - 1 \right).$$

Using $-y_l \geq \left(\frac{1}{4} - \frac{2^{-k-1}}{1 - \bar{\epsilon}_1} \right) u$, we get

$$\frac{u}{4} + y_l \geq \left(\frac{1}{4} - \frac{2^{-k-1}}{1 - \bar{\epsilon}_1} \right) u \left(e \left(1 - 2^{-53} \right) - 1 \right).$$

To ensure that $\dfrac{u}{4} + y_l \geq \bar{\epsilon}_1 y$, we again use (11.16) and ensure that $\dfrac{u}{4} + y_l \geq \dfrac{2^{52} u}{1 - \bar{\epsilon}_1} \bar{\epsilon}_1$. This provides the condition that must be fulfilled by e for the theorem to hold in this case: we need

$$\left(\frac{1}{4} - \frac{2^{-k-1}}{1 - \bar{\epsilon}_1} \right) u \left(e(1 - 2^{-53}) - 1 \right) \geq \frac{2^{52} u}{1 - \bar{\epsilon}_1} \bar{\epsilon}_1$$

rewritten as:

$$e \geq (1 - 2^{-53})^{-1} \left(1 + \frac{2^{54} \bar{\epsilon}_1}{1 - \bar{\epsilon}_1 - 2^{-k+1}} \right).$$

Taking for constraint on e the max of these values completes the proof of the theorem. □

11.6.4 Accurate second step

For the second step, correct rounding requires an accuracy of 2^{-120} to 2^{-150}, depending on the function. Several approaches are possible, and they will be reviewed in Chapter 14. The important point here is that this accuracy is known statically, so the overhead due to arbitrary multiple precision is avoided.

The result of the accurate step must finally be rounded to a double-precision number in the selected rounding mode. For some multiple-precision representations, this is a nontrivial task. For instance, in the CRlibm library, the result of the accurate step is represented by a triple-double number (see Section 14.1, pages 494 through 503). Converting this result to a double-precision/binary64 number is equivalent to computing the correctly rounded-to-nearest sum of three double-precision/binary64 numbers. This can be done using techniques presented in Section 6.3.4, page 199.

11.6.5 Error analysis and the accuracy/performance tradeoff

The probability p_2 of launching the accurate step is the probability that the interval $[\hat{y}_1(1 - \bar{\epsilon}_1), \hat{y}_1(1 + \bar{\epsilon}_1)]$ contains the midpoint between two consecutive floating-point numbers (or a floating-point number in directed rounding modes)—see Figure 12.2, page 408. Therefore, it is expected to be proportional to the error bound $\bar{\epsilon}_1$ computed for the first step.

This defines the main performance tradeoff one has to manage when designing a correctly rounded function: the average evaluation time will be

$$T_{\text{avg}} = T_1 + p_2 T_2, \tag{11.17}$$

where T_1 and T_2 are the execution time of the first and second phase respectively, and p_2 is the probability of launching the second phase.

For illustration, $T_2 \approx 100 T_1$ in CRlibm using SCSlib,[8] and $T_2 \approx 10 T_1$ in CRlibm using double-extended numbers or triple-double numbers.

Typically, we aim at choosing (T_1, p_2, T_2) such that the average cost of the second step is negligible. Indeed, in this case the performance price to pay for correct rounding will be almost limited to the overhead of the rounding test, which is a few cycles only.

The second step is built to minimize T_2; there is no tradeoff there. Then, as p_2 is almost proportional to $\bar{\epsilon}_1$, to minimize the average time, we have to

- balance T_1 and p_2: this is a performance/precision tradeoff (the more accurate the first step, the slower),

- and compute a tight bound on the overall error $\bar{\epsilon}_1$.

Computing this tight bound is the most time-consuming part in the design of a correctly rounded elementary function. The proof of the correct rounding property only needs a proven bound, but a loose bound will mean a larger p_2 than strictly required, which directly impacts average performance. Compare $p_2 = 1/1000$ and $p_2 = 1/500$ for $T_2 = 100 T_1$, for instance. As a consequence, when there are multiple computation paths in the algorithm, it may make sense to precompute different values of $\bar{\epsilon}_1$ on these different paths [102].

With the two-step approach, the proof that an implementation always returns the correctly rounded result resumes to two tasks:

- computing a bound on the overall error of the second step, and checking that this bound is less than the bound deduced from the hardest-to-round cases (e.g., 2^{-118} for natural logarithms);

---

[8]SCSlib, the Software Carry-Save Library, is a reasonably fast and lightweight multiple-precision library developed in the Arénaire project at ENS Lyon (http://www.ens-lyon.fr/LIP/Arenaire/Ware/SCSLib/). It was used in the first versions of CRlibm.

- proving that the first step returns a value only if this value is correctly rounded, which also requires a proven (and tight) bound on the evaluation error of the first step.

Let us now survey how such error bounds can be obtained.

11.7   Computing Error Bounds

The evaluation of any mathematical function entails two main sources of errors:

- *approximation errors* (also called methodical errors), such as the error that comes from approximating a function by a polynomial. One may have a mathematical bound for them (given by a Taylor formula, for instance), or one may have to compute such a bound using numerics; for example, if the polynomial has been computed using the Remez algorithm. If this is the case, one must be certain that the numerical method will never underestimate the approximation errors [70];

- *rounding errors*, produced by most (but not all!) floating-point operations of the code.

The distinction between both types of errors is sometimes arbitrary. For example, the error due to rounding the polynomial coefficients to floating-point numbers is usually included in the approximation error of the polynomial. The same holds for the rounding of table values, which is accounted far more accurately as an approximation error than as a rounding error. This point is mentioned here because a lack of accuracy in the definition of the various errors involved in a given code may lead to one of them being forgotten.

11.7.1   The point with efficient code

Efficient code is especially difficult to analyze and prove because of all the techniques and tricks used by expert programmers.

For instance, many floating-point operations are exact, and the experienced developer of floating-point code will try to use them. Examples include multiplication by a power of the radix of the floating-point system, subtraction of numbers of similar magnitude due to Sterbenz's lemma (Lemma 2, Chapter 4, page 122), exact addition and exact multiplication algorithms such as Fast2Sum (Algorithm 4.3, page 126) and 2MultFMA (Algorithm 5.1, page 152), multiplication of a small integer by a floating-point number whose significand ends with enough zeros, etc.

The expert programmer will also do his or her best to avoid computing more accurately than strictly needed. He or she will remove from the computation some operations that are not expected to improve the accuracy of

the result by much. This can be expressed as an additional approximation. However, it soon becomes difficult to know what is an approximation to what, especially as the computations are re-parenthesized to maximize floating-point accuracy.

The resulting code obfuscation is best illustrated by an example.

11.7.2 Example: a "double-double" polynomial evaluation

Listing 11.2 is an extract of the code of a sine function in CRlibm. The "target" arithmetic is double precision/binary64, and we sometimes need to represent big precision numbers as the unevaluated sum of two binary64/double-precision numbers (as stated above, in the somehow clumsy computer arithmetic jargon, such numbers are sometimes called "double-double" numbers).

These three lines compute the value of an odd polynomial,

$$p(y) = y + s_3 \times y^3 + s_5 \times y^5 + s_7 \times y^7,$$

close to the Taylor approximation of the sine function (its degree-1 coefficient is equal to 1). In our algorithm, the reduced argument y is ideally obtained by subtracting from the floating-point input x an integer multiple of $\pi/256$. As a consequence, $y \in [-\pi/512, \pi/512] \subset [-2^{-7}, 2^{-7}]$.

However, as y is an irrational number, the implementation of this range reduction has to return a number more accurate than a double-precision number; otherwise, there is no hope of achieving an accuracy of the sine that allows for correct rounding in double precision. In our implementation, the range reduction step therefore returns a double-double number yh + yl.

To minimize the number of operations, Horner's rule is used for the polynomial evaluation:

$$p(y) = y + y^3 \times (s_3 + y^2 \times (s_5 + y^2 \times s_7)).$$

For a double-double input $y = $ yh+yl, the expression to compute is thus

$$(\text{yh} + \text{yl}) + (\text{yh} + \text{yl})^3 \times (s_3 + (\text{yh} + \text{yl})^2 \times (s_5 + (\text{yh} + \text{yl})^2 \times s_7)).$$

The actual code uses an approximation to this expression: the computation is accurate enough if all the Horner steps except the last one are computed in double-precision arithmetic. Thus, y_l will be neglected for these iterations, and coefficients s_3 to s_7 will be stored as double-precision numbers noted s3, s5, and s7. The previous expression becomes:

$$(\text{yh} + \text{yl}) + \text{yh}^3 \times (\text{s3} + \text{yh}^2 \times (\text{s5} + \text{yh}^2 \times \text{s7})).$$

However, if this expression is computed as parenthesized above, it has a poor accuracy. Specifically, the floating-point addition yh + yl (by definition of a double-double number) returns yh, so the information held by yl is completely lost. Fortunately, the other part of the Horner evaluation also has a much smaller magnitude than yh—this is deduced from $|y| \leq 2^{-7}$, which gives $|y^3| \leq 2^{-21}$. The following parenthesizing leads therefore to a much more accurate algorithm:

$$\text{yh} + \left(\text{yl} + \text{yh} \times \text{yh}^2 \times \left(\text{s3} + \text{yh}^2 \times \left(\text{s5} + \text{yh}^2 \times \text{s7}\right)\right)\right).$$

In this last version of the expression, only the leftmost addition has to be accurate. So we will use a Fast2Sum (Algorithm 4.3, page 126), which as we saw in Chapter 4 provides an exact addition of two double-precision numbers, returning a double-double number. The other operations use the native (and therefore fast) double-precision arithmetic. We obtain the code of Listing 11.2.

C listing 11.2 Three lines of C.

```
yh2 = yh * yh; ts = yh2 * (s3 + yh2 * (s5 + yh2 * s7));
Fast2Sum(sh, sl, yh, yl + yh * ts);
```

To summarize, this code implements the evaluation of a polynomial with many layers of approximation. For instance, variable yh2 approximates y^2 through the following layers:

- y was approximated by yh + yl with the relative accuracy ϵ_{argred};

- yh + yl is approximated by yh in most of the computation;

- yh^2 is approximated by yh2, with a floating-point rounding error.

In addition, the polynomial is an approximation to the sine function, with a relative error bound of ϵ_{approx} which is supposed known.

Thus, the difficulty of evaluating a tight bound on an elementary function implementation is to combine all these errors without forgetting any of them, and without using overly pessimistic bounds when combining several sources of errors. The typical tradeoff here will be that a tight bound requires considerably more work than a loose bound (and its proof, since it is much longer and more complex, might inspire considerably less confidence). Some readers may get an idea of this tradeoff by relating each intermediate value with its error to confidence intervals, and propagating these errors using interval arithmetic. In many cases, a tighter error will be obtained by splitting confidence intervals into several cases, and treating them separately, at the expense of an explosion of the number of cases. This is one of the tasks that a tool such as *Gappa* (see Section 13.3, page 474) was designed to automate.

Chapter 12

Solving the Table Maker's Dilemma

12.1   Introduction

A s we have seen in previous chapters (especially in Chapters 2 and 4), requiring correctly rounded arithmetic operations has a number of advantages. Among them:

- it greatly improves the portability of software;

- it allows one to design algorithms that use this requirement;

- this requirement can be used for designing formal proofs of software (see Chapter 13);

- one can easily implement interval arithmetic, or more generally one can get certain lower or upper bounds on the exact result of a sequence of arithmetic operations.

The IEEE 754-1985 and 854-1987 standards required correctly rounded arithmetic operations. It seems natural to try to enforce the same requirement for the most common mathematical functions (simple algebraic functions such as $1/\sqrt{x}$ and also a few transcendental functions such as sine, cosine, exponentials, and logarithms of radices e, 2, and 10, etc.).

This was not done in the 754-1985 and 854-1987 standards, mainly because of a difficulty known as the *Table Maker's Dilemma*. The name *Table Maker's Dilemma* (TMD) was coined by Kahan. Let us quote him [209]:

> *Why can't Y^W be rounded within half an ulp like SQRT? Because nobody knows how much computation it would cost to resolve what I long ago christened "The Table-Maker's Dilemma" $(\cdots)$. No general way exists to predict how many extra digits will have to be carried*

> to compute a transcendental expression and round it correctly to some
> preassigned number of digits. Even the fact (if true) that a finite number
> of extra digits will ultimately suffice may be a deep theorem.

Indeed, there are solutions for *algebraic* functions (see Definition 14),
even if they are not fully satisfactory (the precision with which the compu-
tations must be performed is, in general, coarsely overestimated). Although
no solution seems to exist for a general *transcendental* expression (see Defi-
nition 15), we will show in this chapter that for the most common functions
and a given, not too large, floating-point format, one can find satisfactory
solutions to that problem. This is why within the IEEE 754-2008 standard,
correct rounding of some functions becomes recommended (yet not manda-
tory). These functions are:

$$e^x, e^x - 1, 2^x, 2^x - 1, 10^x, 10^x - 1,$$
$$\ln(x), \log_2(x), \log_{10}(x), \ln(1+x), \log_2(1+x), \log_{10}(1+x),$$
$$\sqrt{x^2 + y^2}, 1/\sqrt{x}, (1+x)^n, x^n, x^{1/n} (n \text{ is an integer}), x^y,$$
$$\sin(\pi x), \cos(\pi x), \arctan(x)/\pi, \arctan(y/x)/\pi,$$
$$\sin(x), \cos(x), \tan(x), \arcsin(x), \arccos(x), \arctan(x), \arctan(y/x),$$
$$\sinh(x), \cosh(x), \tanh(x), \sinh^{-1}(x), \cosh^{-1}(x), \tanh^{-1}(x).$$

Before going further, let us now define the TMD more precisely.

12.1.1 The Table Maker's Dilemma

Assume we wish to implement a mathematical function f, and that we use
a radix-β floating-point format of precision p. Let us denote $\circ$ the active
rounding mode, and call *rounding breakpoints* the values where the value of $\circ$
changes:

- for "directed" rounding (i.e., toward $+\infty$, $-\infty$ or 0), the breakpoints are
  the (finite) floating-point numbers;

- for rounding to nearest, they are the exact middle of two consecutive
  floating-point numbers.

The real number $f(x)$ cannot, in general, be represented with a finite
number of digits. Furthermore, in many cases (e.g., trigonometric functions,
logarithms, ...), the function f cannot be exactly reduced to a finite number
of arithmetic operations.

Hence, we must *approximate* it by some other function that is easier to
evaluate (e.g., a piecewise polynomial or rational function), say $\hat{f}$. More pre-
cisely, what we will call $\hat{f}$ here is the *computed* approximation, not a real-
valued *theoretical* approximation. If x is a floating-point number, then $\hat{f}(x)$
is a floating-point number, of possibly much higher precision than the "tar-
get precision" p. All various rounding errors are included in the definition

of $\hat{f}$, including the roundings of the coefficients of the polynomial or rational approximation and the roundings of the arithmetic operations.

We assume that the very accurate but finitely precise significand of $\hat{f}(x)$ approximates the infinitely precise significand of $f(x)$ within an error less than β^{-m}, where m is significantly larger than p. In the case of several $\hat{f}$'s with different m's, we will write $\hat{f}_m$. The only information that we have on $f(x)$ is that it is located in some interval I_x, centered[1] on $\hat{f}_m(x)$, of width $2\beta^{-m}$. We would like to always return $\circ(f(x))$, and yet the only thing we can easily return is $\circ(\hat{f}_m(x))$: Can we guarantee that if m is large enough, then these two values will always be equal?

The TMD occurs, for a given x, when the interval I_x contains a breakpoint, i.e., when $\hat{f}_m(x)$ is so close to a breakpoint that, taking into account the error β^{-m} of the approximation, it is impossible to decide whether $f(x)$ is above or below the breakpoint. In such a case, one does not know which result should be returned: due to the inaccuracy, it could be the precision-p floating-point number right above the breakpoint or the one right below it. This is exemplified by Figures 12.1 and 12.2.
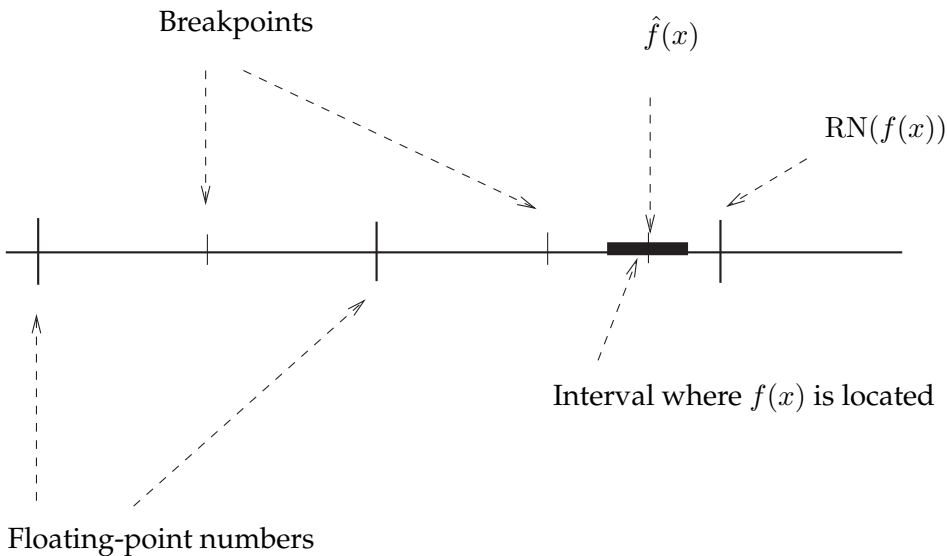
Figure 12.1: In this example (assuming rounding to nearest), the interval around $\hat{f}(x)$ where $f(x)$ is known to be located contains no breakpoint. Hence, $\mathrm{RN}(f(x)) = \mathrm{RN}(\hat{f}(x))$: we can safely return a correctly rounded result.

When the TMD occurs, the only possible solution consists in increasing m, i.e., in performing the computation again, with an approximation more accurate than the one provided by $\hat{f}_m$. We then compute $\hat{f}_{m'}(x)$ with $m' > m$. Ziv suggests to progressively increase the precision m of the computation,

---

[1]Sometimes, we know the sign of some of the errors, so that the interval is not exactly centered on $\hat{f}(x)$. This does not change the reasoning much.
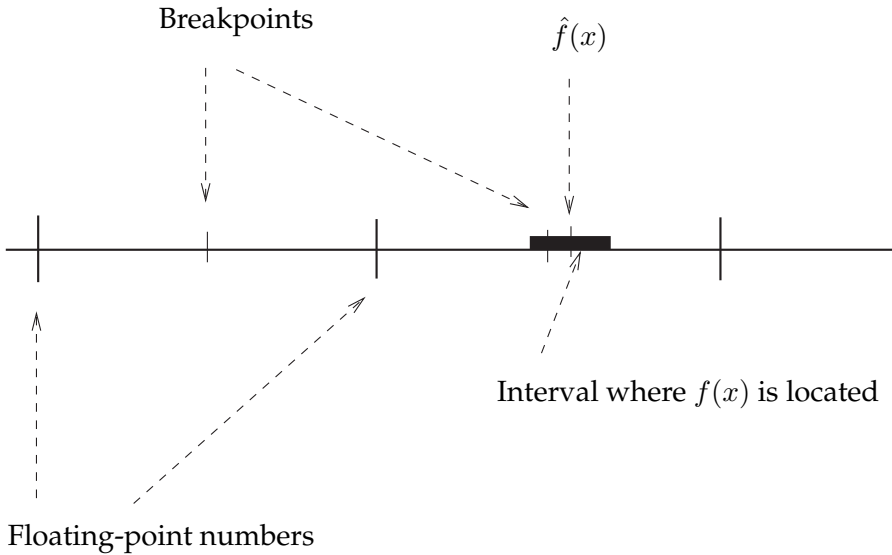
Figure 12.2: In this example (assuming rounding to nearest), the interval around $\hat{f}(x)$ where $f(x)$ is known to be located contains a breakpoint. We do not have enough information to provide a correctly rounded result.

until we are able to provide a correctly rounded result [444]. A problem with such a strategy is that we may not know when the computation stops (for many functions, we do not even know if it actually always stops).

To summarize, we wish to always return $\circ(f(x))$ as the result of the computation, but in practice, the best we can do is to return $\circ(\hat{f}(x))$. Hence, our problem can be reworded as follows:

Can we make sure, if m is large enough, that $\circ(\hat{f}(x))$ will always be equal to $\circ(f(x))$?

A sufficient condition to ensure that $\circ(\hat{f}(x)) = \circ(f(x))$ for all floating-point numbers x is that the infinitely precise significand of $f(x)$ should never be within a distance β^{-m} from a breakpoint, so that there cannot be a breakpoint between $\hat{f}(x)$ and $f(x)$. But if $f(x)$ is a breakpoint, this sufficient condition cannot be satisfied for any value of m. Depending on the function f, several cases may occur.

- For the most common mathematical functions, either $f(x)$ is never equal to a breakpoint (when x is a floating-point number), or there are only a few straightforward cases for which this happens (such as $\cos(0) = 1$ or $\log_2(2^k) = k$ for integers k).

- For some functions, there are many floating-point numbers x for which $f(x)$ is a breakpoint, but these values of x are known and can be easily detected. A typical example is the power function [246].

- Finally, there are some functions (such as the number theoretic Gamma function Γ, that generalizes the factorial function to real numbers) for which we currently know (close to) nothing.

Considering the possibility of $f(x)$ being a breakpoint, our problem becomes:

> Find m as small as possible such that for *all* floating-point numbers x in the domain of implementation of f, either $f(x)$ is a breakpoint or the infinitely precise significand of $f(x)$ is not within a distance β^{-m} from a breakpoint.

The precision-p values x such that the infinitely precise significand of $f(x)$ is closest to a breakpoint will be called *worst cases* for the TMD. We will (informally) call *bad cases* the floating-point numbers x for which $f(x)$ is somehow close to a breakpoint. The best (i.e., lowest) bound m will be called the *hardness to round*. We give a more formal definition.

Definition 10 (Hardness to round). *For a given floating-point format of radix β and a given rounding mode, the* hardness to round *for function f in interval $[a, b]$ is the smallest integer m such that for all floating-point numbers $x \in [a, b]$, either $f(x)$ is a breakpoint or the infinitely precise significand of $f(x)$ is not within β^{-m} from a breakpoint.*

The fact that m must be as small as possible immediately follows from the requirement that the very accurate but finitely precise significand of $\hat{f}$ should approximate the infinitely precise significand of f with accuracy β^{-m}. With a large value of m, big and thus expensive precisions could be necessary in the worst case. Obtaining the lowest m allows one to provide the best efficiency guarantees for implementing f.

For example, in the single-precision/binary32 of IEEE 754, assuming round-to-nearest mode, the hardness to round the sine function in $[1/2, 1)$ is $m = 45$ bits. There are five floating-point numbers $x \in [1/2, 1)$ such that the infinitely precise significand of $\sin(x)$ is within 2^{-45} from a breakpoint. One of them is

$$x = 0.58756285905838012695312 5_{10} = 0.10010110011010101000010 1_2,$$

for which

$$2\sin(x) = \overbrace{\underbrace{1.00011011110100011011001}_{24 \text{ bits}} 011111111111111111111111}^{46 \text{ bits}} 00010\cdots_2.$$

The bad rounding cases derived from the study of the TMD can be used to test if implementations of mathematical functions comply with the correct rounding requirements. By means of approximations, it is not very hard

to devise implementations that are almost always correctly rounded. Testing random values is unlikely to disclose any error, since bad rounding cases are extremely infrequent (we will give an estimate of their probability in Section 12.2.1). A list of bad rounding cases seems much more suited to investigate the reliability of such implementations.

Note that depending on function f, finding a valid m that possibly overestimates the hardness to round can already be a difficult problem. In that case, the TMD is only partially solved, but this suffices for providing guaranteed implementations of the function f. These implementations of the function f may be reasonably efficient as long as m is not too large.

Some methods described in this chapter only provide upper bounds to the hardness to round. This includes for example some of those related to Liouville's theorem (see Section 12.3.3) and variants of the Stehlé–Lefèvre–Zimmermann (SLZ) algorithm (see Section 12.4.6). On the opposite side, some methods only provide lower bounds to the hardness to round: they build bad cases, but may miss the worst ones. That includes Kahan's technique relying on Hensel lifting, described in Section 12.3.4 (note that the original goal of that technique was not to find worst cases: it aimed at building bad cases, for function testing purposes). The TMD is fully solved when the lowest possible m has been determined, along with worst cases. Overall, the TMD Grail is:

> For any standardized radix, rounding mode, precision p, and function f (that is possibly multivariate), determine the hardness to round f as well as the corresponding worst cases.

Note that we defined the TMD only for univariate functions f, but the definition easily generalizes to several variables. At the time we are writing this book, the TMD Grail remains far from being reached, although much progress has been made in the last 15 years.

12.1.2   Brief history of the TMD

We should mention the pioneering work of Schulte and Swartzlander [369], who found worst cases for some functions in the (binary) IEEE 754-1985 single-precision format, by exhaustive searching, and suggested ways of designing a correctly rounded hardware implementation of these functions in single precision. For general functions, the first improvement over the exhaustive search was proposed by Lefèvre in his Ph.D. thesis. Along with Muller, Tisserand, Stehlé, and Zimmermann, he progressively obtained worst cases for many functions in double precision [252, 251, 389, 390]. For instance, the worst case for the natural logarithm in the full IEEE 754 double-precision/binary64 range [251] is attained for

$$x = 1.011000101010100010000110000100110110001010$$
$$0110110110 \times 2^{678}$$

whose logarithm is

$$\ln x = \overbrace{111010110.0100011110011110101\cdots110001}^{53\text{ bits}}$$
$$\underbrace{00000000000000000\cdots000000000000000}_{65\text{ zeros}}1110\cdots.$$

This means that the hardness to round logarithms in double precision (for any rounding mode) is $52 + 65 = 117$. The example given above is a "difficult case" in the directed rounding modes since it is very near a floating-point number. One of the two worst cases for radix-2 exponentials in the full double-precision range [251] is

$$1.1110010001011001011001010010011010111111$$
$$100101001101 \times 2^{-10}$$

whose radix-2 exponential is

$$\overbrace{1.00000000010100111111111000010111\cdots0011}^{53\text{ bits}}$$
$$0\underbrace{11111111111111111\cdots111111111111111111}_{59\text{ ones}}0100\cdots.$$

It is a difficult case for rounding to nearest, since it is very close to the middle of two consecutive floating-point numbers.

Now, in decimal, the worst case (for $|x| \geq 3 \times 10^{-11}$) for the exponential function in the decimal64 format of the IEEE 754-2008 standard [253] is attained for

$$9.407822313572878 \times 10^{-2},$$

whose exponential is

$$\underbrace{1.098645682066338}_{16\text{ digits}}5\underbrace{0000000000000000}_{16\text{ zeros}}2780\cdots.$$

This is a difficult case for rounding to nearest.

Independently of the algorithmic improvements for the search of worst cases for general functions, Iordache, Matula, Lang, Muller, and Brisebarre [195, 239, 50] devised techniques that are specific to algebraic functions (see Definition 14). These specific techniques provide upper bounds to the hardness to round, for essentially no cost at all (the costs of the general algorithms grow exponentially with the precision p). Unfortunately, these upper bounds are most often quite far from being tight.

12.1.3 Organization of the chapter

In this chapter, we first provide some simple preliminary remarks on the TMD. In particular, we explain what should be expected from the worst cases

and how to derive information on the TMD for a given function from information on the TMD for another one. We then describe a few methods that (most often only partially) solve the TMD in the case of algebraic functions. We then present two algorithms that are more expensive but allow us to exhaustively compute worst cases, for general mathematical functions. We finally give the worst cases that have been obtained so far in double/binary64 precisions, for the most common functions and domains.

12.2 Preliminary Remarks on the Table Maker's Dilemma

12.2.1 Statistical arguments: what can be expected in practice

Consider that we wish to implement function f with correct rounding. For simplification, in this section, we assume radix 2 and rounding to nearest, although what we will state can be generalized rather easily.

Let x be a floating-point number. The infinite significand y of $f(x)$ has the form

$$y = y_0.y_1y_2\cdots y_{n-1}\overbrace{01111111\cdots 11}^{k \text{ bits}}xxxxx\cdots$$
$$\text{or}$$
$$y = y_0.y_1y_2\cdots y_{n-1}\overbrace{10000000\cdots 00}^{k \text{ bits}}xxxxx\cdots$$

with $k \geq 1$. The hardness to round function f in a given interval will be $p - 1 + k_{\max}$, where $k_{\max}$ is the largest value of k attained for all floating-point numbers x in that interval.

If the precision p is small enough, we can exhaustively check all floating-point numbers in an interval of reasonable size. We have done that for the sine function in the interval $[0.5, 1]$, and values of p ranking from 8 to 24 (the results are given in Table 12.3). Looking at that table, we immediately see that $k_{\max}$ always seems to be quite close to p. There is a statistical explanation for that phenomenon. It does not prove anything, but it allows one to understand why we obtain such figures.

A probabilistic model

Let us assume that when x is a precision-p binary floating-point number, the bits of $f(x)$ after the p-th position can be viewed as if they were *sequences of independent random zeros and ones, with probability* $1/2$ *for* 0 *as well as for* 1.

This assumption is not realistic when the function is too simple: for instance, the digits of the quotient of two floating-point numbers form an eventually periodic sequence, which clearly cannot be modeled by a random sequence. Also, even with a much more "complex function," this probabilistic

k	Actual number of occurrences	Expected number of occurrences
1	16397	16384
2	8151	8192
3	4191	4096
4	2043	2048
5	1010	1024
6	463	512
7	255	256
8	131	128
9	62	64
10	35	32
11	16	16
12	7	8
13	6	4
14	0	2
15	1	1

Table 12.1: Actual and expected numbers of digit chains of length k of the form $1000\cdots0$ or $0111\cdots1$ just after the p-th bit of the infinitely precise significand of sines of floating-point numbers of precision $p = 16$ between $1/2$ and 1.

model may not hold in some domains. For instance, the exponential of a tiny number ϵ is so close to $1 + \epsilon$ that the first bits after the p-th position cannot be viewed as "random" (indeed, that property makes it possible to round exponentials correctly around zero, without having to actually compute worst cases, see Section 12.2.2). And yet, in most cases, this probabilistic model will allow us to predict, quite accurately, the order of magnitude of the hardness to round functions (alas, these predictions are not proofs).

With our assumption, the "probability" of having $k = k_0$ exactly is 2^{-k_0}. Hence, if we consider N possible input floating-point numbers x, the number of input values for which $k \geq k_0$ should be around

$$N \times 2^{-k_0}. \tag{12.1}$$

We have checked this by exhaustively counting the number of occurrences of the various values of k, for the sine function and numbers between $1/2$ and 1 (which gives $N = 2^{p-1}$), for $p = 16$ and 24. The results (along with the predictions of the probabilistic model) are given in Tables 12.1 and 12.2. One can readily see that the accordance with the prediction of the probabilistic model is excellent.

According to (12.1), as soon as k_0 is significantly larger than $\log_2(N)$, there should no longer be any input value for which $k \geq k_0$. Table 12.3 gives

k	actual number of occurrences	expected number of occurrences
1	4193834	4194304
2	2098253	2097152
3	1048232	1048576
4	522560	524288
5	263414	262144
6	131231	131072
7	65498	65536
8	32593	32768
9	16527	16384
10	8194	8192
11	4093	4096
12	2066	2048
13	1063	1024
14	498	512
15	272	256
16	141	128
17	57	64
18	32	32
19	25	16
20	14	8
21	6	4
22	5	2
23	0	1

Table 12.2: Actual and expected numbers of digit chains of length k of the form $1000\cdots0$ *or* $0111\cdots1$ *just after the p-th bit of the infinitely precise significand of sines of floating-point numbers of precision $p = 24$ between $1/2$ and 1.*

p	$k_{\max}$ (sine function)	$k_{\max}$ (exp function)
8	11	8
9	10	9
10	11	8
11	11	9
12	10	16
13	12	14
14	18	13
15	14	19
16	15	14
17	20	15
18	21	16
19	22	22
20	20	22
21	23	20
22	22	22
23	26	22
24	22	24

Table 12.3: Length $k_{\max}$ of the largest digit chain of the form $1000\cdots0$ or $0111\cdots1$ just after the p-th bit of the infinitely precise significands of sines and exponentials of floating-point numbers of precision p between $1/2$ and 1, for various p.

the largest attained value of k for sines and exponentials of floating-point numbers between $1/2$ and 1 (which gives $N = 2^{p-1}$), and various precisions p. One can see that the largest value of k is always around p (see also, in the case of function $1/\sqrt{x}$, Table 12.7).

Hence, a consequence of the probabilistic model is that the expected hardness to round function f in interval $[a, b]$ should be around $p + \log_2(N)$, where N is the number of floating-point numbers in $[a, b]$. For instance, if $[a, b]$ is one binade,[2] then $N = 2^{p-1}$ and the predicted hardness to round is around $2p$.

Of course, we have not proved anything: the probabilistic model just gives us a hint on the order of magnitude of the value of the hardness to round functions. We now have to actually compute (or sometimes just get an upper bound on) that hardness to round, for the most usual functions.

---

[2]A binade is the interval between two consecutive integer powers of 2.

And yet, this probabilistic model is very useful: it will help to tune the algorithms that actually find the worst cases. Also, the various results presented in Section 12.5 show that the predictions of the probabilistic model are quite accurate.

12.2.2   In some domains, there is no need to find worst cases

For many functions, if the input argument is small enough, reasoning based on the Taylor expansion of the function being considered allows one to return correctly rounded results without having to actually find worst cases or compute the hardness to round.

For instance, in a radix-β, precision-p system, if one wishes to evaluate the exponential of ϵ with $0 \leq \epsilon < \beta^{-p}$ (which implies $\epsilon \leq (\beta^{-p} - \beta^{-2p})$), then

$$e^\epsilon \leq 1 + (\beta^{-p} - \beta^{-2p}) + \frac{\beta^{-2p}}{2} + \frac{\beta^{-3p}}{6} + \frac{\beta^{-4p}}{24} + \cdots,$$

which implies

$$e^\epsilon < 1 + \beta^{-p} = 1 + \frac{1}{\beta} \operatorname{ulp}(1) \leq 1 + \frac{1}{2} \operatorname{ulp}(1).$$

Therefore, in such a case, one can safely return 1 as the correctly rounded to the nearest even value of e^ϵ: there is no need to compute worst cases for input values of exponent less than $-p$.

Similarly, in round-to-nearest mode and double precision/binary64 format,

- if $|\epsilon| \leq \operatorname{RN}\left(3^{1/3}\right) \times 2^{-26} = 1.4422 \cdots \times 2^{-26}$, then $\sin(\epsilon)$ can be replaced by ϵ;

- if $\operatorname{RN}\left(3^{1/3}\right) \times 2^{-26} < \epsilon \leq 2^{-25}$, then $\sin(\epsilon)$ can be replaced by $\epsilon^- = \epsilon - 2^{-78}$ (the case $-2^{-25} \leq \epsilon < -\operatorname{RN}\left(3^{1/3}\right) \times 2^{-26}$ is obviously symmetrical).

Tables 12.4 and 12.5 give results derived from similar reasoning for some functions, assuming the double-precision/binary64 format of the IEEE 754 standard.

Similarly, one does not need either to search for worst cases of some functions on large (positive and/or negative) values. For instance, the expm1 function, defined as $\operatorname{expm1}(x) = \exp(x) - 1$, gives -1 on values less than $-54 \ln(2)$, in binary64, rounding to nearest.

This function	can be replaced by	when
$\exp(\epsilon), \epsilon \geq 0$	1	$\epsilon < 2^{-53}$
$\exp(\epsilon), \epsilon \leq 0$	1	$\lvert\epsilon\rvert \leq 2^{-54}$
$\exp(\epsilon) - 1$	ϵ	$\lvert\epsilon\rvert < \mathrm{RN}(\sqrt{2}) \times 2^{-53}$
$\exp(\epsilon) - 1, \epsilon \geq 0$	ϵ^+	$\mathrm{RN}(\sqrt{2}) \times 2^{-53}$ $\leq \epsilon < \mathrm{RN}(\sqrt{3}) \times 2^{-52}$
$\mathrm{log1p}(\epsilon) = \ln(1 + \epsilon)$	ϵ	$\lvert\epsilon\rvert < \mathrm{RN}(\sqrt{2}) \times 2^{-53}$
$2^\epsilon, \epsilon \geq 0$	1	$\epsilon < 1.4426 \cdots \times 2^{-53}$
$2^\epsilon, \epsilon \leq 0$	1	$\lvert\epsilon\rvert < 1.4426 \cdots \times 2^{-54}$
$10^\epsilon, \epsilon \geq 0$	1	$\epsilon < 1.7368 \times 2^{-55}$
$10^\epsilon, \epsilon \leq 0$	1	$\lvert\epsilon\rvert < 1.7368 \times 2^{-56}$
$\sin(\epsilon), \sinh(\epsilon), \sinh^{-1}(\epsilon)$	ϵ	$\lvert\epsilon\rvert \leq \alpha = \mathrm{RN}(3^{1/3}) \times 2^{-26}$
$\arcsin(\epsilon)$	ϵ	$\lvert\epsilon\rvert < \alpha = \mathrm{RN}(3^{1/3}) \times 2^{-26}$
$\sin(\epsilon), \sinh^{-1}(\epsilon)$	$\epsilon^- = \epsilon - 2^{-78}$	$\alpha < \epsilon \leq 2^{-25}$
$\sinh(\epsilon)$	$\epsilon^+ = \epsilon + 2^{-78}$	$\alpha < \epsilon < 2^{-25}$
$\arcsin(\epsilon)$	$\epsilon^+ = \epsilon + 2^{-78}$	$\alpha \leq \epsilon < 2^{-25}$
$\cos(\epsilon)$	1	$\lvert\epsilon\rvert < \gamma = \mathrm{RN}(\sqrt{2}) \times 2^{-27}$
$\cos(\epsilon)$	$1^- = 1 - 2^{-53}$	$\gamma \leq \lvert\epsilon\rvert \leq 1.2247 \times 2^{-26}$
$\cosh(\epsilon)$	1	$\lvert\epsilon\rvert < 2^{-26}$
$\cosh(\epsilon)$	$1^+ = 1 + 2^{-52}$	$2^{-26} \leq \lvert\epsilon\rvert \leq \mathrm{RN}(\sqrt{3}) \times 2^{-26}$
$\cosh(\epsilon)$	$1^{++} = 1 + 2^{-51}$	$\mathrm{RN}(\sqrt{3}) \times 2^{-26}$ $< \lvert\epsilon\rvert \leq 1.118 \times 2^{-25}$
$\tan(\epsilon), \tanh^{-1}(\epsilon)$	ϵ	$\lvert\epsilon\rvert < \eta = \mathrm{RN}(12^{1/3}) \times 2^{-27}$
$\tanh(\epsilon), \arctan(\epsilon)$	ϵ	$\lvert\epsilon\rvert \leq \eta$
$\tan(\epsilon), \tanh^{-1}(\epsilon)$	$\epsilon^+ = \epsilon + 2^{-78}$	$\eta \leq \epsilon \leq 1.650 \times 2^{-26}$
$\arctan(\epsilon), \tanh(\epsilon)$	$\epsilon^- = \epsilon - 2^{-78}$	$\eta < \epsilon \leq 1.650 \times 2^{-26}$

*Table 12.4: Some results for small values in the double-precision/binary64 format, assuming **rounding to nearest** (some of these results are extracted from [293]). These results make finding worst cases useless for numbers of tiny absolute value. The number $\alpha = \mathrm{RN}(3^{1/3}) \times 2^{-26}$ is approximately equal to $1.4422 \cdots \times 2^{-26}$, and $\eta \approx 1.1447 \times 2^{-26}$. If x is a real number, we let x^- denote the largest floating-point number strictly less than x.*

This function	can be replaced by	when
$\exp(\epsilon), \epsilon \geq 0$	1	$\epsilon < 2^{-52}$
$\exp(\epsilon), \epsilon < 0$	$1^- = 1 - 2^{-53}$	$|\epsilon| \leq 2^{-53}$
$\exp(\epsilon) - 1$	ϵ	$|\epsilon| < \mathrm{RN}(\sqrt{2}) \times 2^{-52}$
$\ln(1 + \epsilon), \epsilon \neq 0$	ϵ^-	$-2^{-52} < \epsilon \leq \mathrm{RN}(\sqrt{2}) \times 2^{-52}$
$2^\epsilon, \epsilon \geq 0$	1	$\epsilon < 1.4426 \cdots \times 2^{-52}$
$2^\epsilon, \epsilon < 0$	$1^- = 1 - 2^{-53}$	$|\epsilon| < 1.4426 \cdots \times 2^{-53}$
$10^\epsilon, \epsilon \geq 0$	1	$\epsilon < 1.7368 \times 2^{-54}$
$10^\epsilon, \epsilon < 0$	$1^- = 1 - 2^{-53}$	$|\epsilon| < 1.7368 \times 2^{-55}$
$\sin(\epsilon), \sinh^{-1}(\epsilon), \epsilon > 0$	ϵ^-	$\epsilon \leq \tau = \mathrm{RN}(6^{1/3}) \times 2^{-26}$
$\sin(\epsilon), \sinh^{-1}(\epsilon), \epsilon \leq 0$	ϵ	$|\epsilon| \leq \tau$
$\sin(\epsilon), \sinh^{-1}(\epsilon), \epsilon > 0$	ϵ^{--}	$\tau < \epsilon \leq 2^{-25}$
$\sin(\epsilon), \sinh^{-1}(\epsilon), \epsilon < 0$	ϵ^+	$\tau < |\epsilon| \leq 2^{-25}$
$\arcsin(\epsilon), \sinh(\epsilon), \epsilon \geq 0$	ϵ	$\epsilon < \tau$
$\arcsin(\epsilon), \sinh(\epsilon), \epsilon < 0$	ϵ^-	$|\epsilon| < \tau$
$\arcsin(\epsilon), \sinh(\epsilon), \epsilon \geq 0$	$\epsilon^+ = \epsilon + 2^{-78}$	$\tau \leq \epsilon < 2^{-25}$
$\arcsin(\epsilon), \sinh(\epsilon), \epsilon < 0$	$\epsilon^{--} = \epsilon - 2^{-77}$	$\tau \leq |\epsilon| < 2^{-25}$
$\cos(\epsilon), \epsilon \neq 0$	$1^- = 1 - 2^{-53}$	$|\epsilon| < 2^{-26}$
$\cosh(\epsilon)$	1	$|\epsilon| < \mathrm{RN}(\sqrt{2}) \times 2^{-26}$
$\cosh(\epsilon)$	$1^+ = 1 + 2^{-52}$	$\mathrm{RN}(\sqrt{2}) \times 2^{-26} \leq |\epsilon| < 2^{-25}$
$\tan(\epsilon), \tanh^{-1}(\epsilon), \epsilon \geq 0$	ϵ	$\epsilon \leq 1.4422 \cdots \times 2^{-26}$
$\tan(\epsilon), \tanh^{-1}(\epsilon), \epsilon < 0$	ϵ^-	$|\epsilon| \leq 1.4422 \cdots \times 2^{-26}$
$\tanh(\epsilon), \arctan(\epsilon), \epsilon > 0$	ϵ^-	$\epsilon \leq 1.4422 \cdots \times 2^{-26}$
$\tanh(\epsilon), \arctan(\epsilon), \epsilon \leq 0$	ϵ	$|\epsilon| \leq 1.4422 \cdots \times 2^{-26}$

*Table 12.5: Some results for small values in the double-precision/binary64 format, assuming **rounding toward** $-\infty$ (some of these results are extracted from [293]). These results make finding worst cases useless for numbers of tiny absolute value. If x is a real number, we let x^- denote the largest floating-point number strictly less than x. The number $\tau = \mathrm{RN}(6^{1/3}) \times 2^{-26}$ is approximately equal to $1.817 \cdots \times 2^{-26}$.*

12.2.3 Deducing the worst cases from other functions or domains

The worst cases of some functions in some domains can be deduced from those of other functions. For instance, for sufficiently large inputs, the worst cases of $\sinh$ and $\cosh$ can be obtained from those of $\exp$. Indeed, for $x \geq x_0$:

$$\sinh(x) = \frac{1}{2} \exp(x) \left(1 + \varepsilon_s\right) \quad \text{and} \quad \cosh(x) = \frac{1}{2} \exp(x) \left(1 + \varepsilon_c\right),$$

where the relative error can be bounded: $|\varepsilon_s|, |\varepsilon_c| \leq \exp(-2x_0)$. For $x_0 = 2^6$, since $-2x_0 / \log(2) < -184$, one has $|\varepsilon_s|, |\varepsilon_c| < 2^{-184}$, which gives an error of at most 2^{-183} on the significand. Thus, a bad case with a 2^{-m} bound for $\sinh$ or $\cosh$ will correspond to a bad case with a bound $2^{-\min(m-1,182)}$ for $\exp$. Only a small domain can be avoided in the search for the worst cases (since $\cosh(x)$ yields an overflow for $x \geq 711$), but in practice, this domain will be by far the most difficult to deal with for these functions.[3] Also, note that due to the factor $1/2$ in the result, the tests of $\exp$ must be carried out a little further than the overflow threshold (of $\exp$) in order to make sure that one obtains *all* the worst cases of $\sinh$ and $\cosh$.

Other bad-case deductions can be obtained from the fact that the bad-case condition depends only on the significand of the result, not on its exponent (assuming an unbounded exponent range, i.e., ignoring overflows and underflows). Reasoning on overflows and underflows will be done specifically for each considered function, in order to reduce the tested domain (if possible). Let us mention two examples.

- For the integer power functions $f(x) = x^n$, where $n \in \mathbb{Z}$, one has $f(\beta x) = (\beta x)^n = \beta^n x^n = \beta^n f(x)$. Thus, $f(\beta x)$ and $f(x)$ have the same significand. As a consequence, all the worst cases can be obtained from the results in $[1, \beta)$, using an extended exponent range, so that if $f(x)$ overflows or underflows for some $x \in [1, \beta)$ but $f(\beta^{-1}x)$ fits in the normal exponent range, one still gets the potential bad cases. However, if for some $x \in [1, \beta)$, $f(x)$ overflows and $f(\beta^{-1}x)$ underflows, then one does not need to take this argument into account. In practice, as $e_{\min} \approx -e_{\max}$, this means that for n larger than some bound, one will be able to remove some interval (that depends on n) around $\beta^{1/2}$ from the tested domain. Alternatively, one can test $f(x)$ for $x \in [x_{\min}(n), x_{\max}(n)]$, where $x_{\min}(n) \geq \beta^{-1/2}$ and $x_{\max}(n) \leq \beta^{1/2}$ are chosen from the underflow/overflow thresholds of $f(x) = x^n$.

- For $f(x) = 2^x$ in binary, one has $f(x + n) = 2^{x+n} = 2^x 2^n = 2^n f(x)$, so that if $n \in \mathbb{Z}$, then $f(x + n)$ and $f(x)$ have the same significand. Moreover, for $|x| \leq \frac{1}{2}$, if $x + n$ is a machine number, then x is also a

---

[3]They will be approximated by polynomials, but the error of the approximation quickly increases with the input values; practical tests on such large values were up to 40 times slower than for small input values.

machine number. As a consequence, all the worst cases of 2^x can be obtained from the results in $[-\frac{1}{2}, \frac{1}{2}]$.

This property can also be used to obtain *all* the worst cases of the inverse function $\log_2$. However, we will see in Section 12.4.4 that only one function needs to be tested among f and its inverse f^{-1}, and it happens that 2^x is the most efficient to test for $x > x_0$, where x_0 is of the order of 1 (the exact bound depends on the implementation). For instance, using the discussion in Section 12.4.4, we test $\log_2(x)$ for $x \in [1/2, 2)$ and 2^x for $x \in [1, 2)^4$, so that one obtains all the worst cases of 2^x. This also covers the full domain for $\log_2$, but due to the large-factor problem mentioned in Section 12.4.4, we also need to test 2^x with a smaller bound on k (38 instead of 44), on larger values of x from an interval of length 1, namely $[32, 33)$, to be able to get all the wanted worst cases of $\log_2$ (see [249, §3.4.1]).

12.3   The Table Maker's Dilemma for Algebraic Functions

For some functions that are simple enough, called the *algebraic functions* (see Definition 14), one can rather easily find reasonable bounds on the hardness to round. Before going further, we must define some classes of numbers and functions that will be needed in the remainder of this chapter.

12.3.1   Algebraic and transcendental numbers and functions

A rational number p/q is a root of the degree-1 polynomial equation $qx - p = 0$. This provides an algebraic definition of rational numbers. Namely, a number is rational if it is the root of a degree-1 polynomial equation with integer coefficients. The *algebraic numbers* are the generalization of that definition where polynomials of higher degree are also allowed.

Definition 11 (Algebraic number). *A complex number z is algebraic if there exists a nonzero polynomial P with integer coefficients such that*

$$P(z) = 0.$$

For a given algebraic number z, there exist infinitely many polynomials P such that $P(z) = 0$: if P is such a polynomial, all its multiples satisfy the same property. Among these polynomials, we distinguish the *minimal polynomial* of z.

---

[4]This is just an example. We do not claim that these are the best intervals, and for ease of implementation and bug detection, there is some redundancy.

Definition 12 (Minimal polynomial and degree of an algebraic number). *If α is an algebraic number, the minimal polynomial of α (over the integers) is the nonzero polynomial P, of relatively prime integer coefficients and positive leading coefficient, of smallest degree, such that*

$$P(\alpha) = 0.$$

If d is the degree of P, we say that α is an algebraic number of degree d.

Definition 13 (Transcendental number). *A complex number z is transcendental if it is not algebraic.*

Examples of algebraic numbers are 1, $5/7$ (and all the rational numbers, including the floating-point numbers in any radix), $\sqrt{3}$, $\sqrt{1 + \sqrt{7}}$, i. Examples of transcendental numbers are e, π, $\sin(1)$.

Definition 14 (Algebraic function). *A complex-valued function f is an algebraic function if there exists a nonzero bivariate polynomial P with integer coefficients such that for all x in the domain of f we have*

$$P(x, f(x)) = 0.$$

Examples of algebraic functions are $x + 1$, $\sqrt{x}$, $1/\sqrt{2 + \sqrt{x}}$, $x^{4/9}$. When x is a floating-point number and f is an algebraic function, then $f(x)$ is an algebraic number.

Definition 15 (Transcendental function). *A function f is a transcendental function if it is not algebraic.*

Examples of transcendental functions are x^x, $\cos(x)$, $\exp(x)$,

$$\Gamma(x) = \int_0^\infty t^{x-1}e^{-t}dt.$$

To show that a function is not algebraic, it is sufficient to find an algebraic number x such that $f(x)$ is not an algebraic number.

Definition 16 (Elementary functions [431]). *An elementary function is a function built from a finite number of (complex) exponentials, logarithms, constants, one variable, and roots of equations through composition and combinations using the four elementary operations $(+, -, \times, \div)$.*

Examples of elementary functions are $1/x$, $\sin(x)$, $(1+x)^{3/4}$, $\cos(1+e^{-x^2})$. An example of a function that is not elementary is $\Gamma(x)$.

12.3.2 The elementary case of quotients

Quotients are functions that are sufficiently simple so that we do not need the analyses described in the rest of this chapter to handle them. The following elementary properties allow us to understand their hardness to round.

Lemma 36 (Exclusion lemma for directed rounding modes). *The infinitely precise significand of the quotient of two radix-β, precision-p, floating-point numbers satisfies the following property:*

- *either it is exactly representable on p digits;*

- *or its radix-β expansion cannot contain more than $p - 1$ consecutive zeros or digits $(\beta - 1)$ after the p-th digit.*

Lemma 36 shows that in directed rounding modes, the hardness to round function $(x, y) \to x/y$ is at most $2p - 1$.

Proof. Let x and y be radix-β, precision-p, floating-point numbers, and let X and Y be their integral significands. Without loss of generality, we can assume that x and y are normal. We have

$$\beta^{p-1} \le X, Y < \beta^p.$$

The radix-β expansion of X/Y has the form

$$\frac{X}{Y} = \delta \times \left(0.q_1 q_2 q_3 \cdots q_p q_{p+1} q_{p+2} \cdots \right),$$

where δ is 1 or β. Thus, we can write

$$\frac{X}{Y} = \delta \times \left(0.q_1 q_2 q_3 \cdots q_p + \rho \right),$$

with $\rho < \beta^{-p}$. This gives

$$X = \delta \times \left((Y \times 0.q_1 q_2 q_3 \cdots q_p) + Y\rho \right).$$

The term $Y \times 0.q_1 q_2 q_3 \cdots q_p$ is an integer multiple of β^{-p}. That term, added to $Y\rho$, is a multiple of $1/\delta$, which is either 1 or $1/\beta$. Therefore, $Y\rho$ is a multiple of β^{-p}, which is less than $Y\beta^{-p}$. This implies that either $\beta^{-p} \le Y\rho \le Y\beta^{-p} - \beta^{-p}$ or $Y\rho = 0$. Therefore, either

$$\beta^{-2p} < \beta^{-p}/Y \le \rho \le \beta^{-p} - \beta^{-p}/Y < \beta^{-p} - \beta^{-2p}$$

or $\rho = 0$. We conclude that either the result is exactly representable on p digits or

$$0.q_1 q_2 \cdots q_p \underbrace{00\cdots0}_{p-1\text{ zeros}} 1 < 0.q_1 q_2 \cdots q_p + \rho < 0.q_1 q_2 \cdots q_p \underbrace{(\beta-1)\cdots(\beta-1)}_{p-1\text{ digits }(\beta-1)}(\beta-1).$$

□

Lemma 37 (Exclusion lemma for round-to-nearest mode). *The infinitely precise significand of the quotient of two radix-β, precision-p, floating-point numbers satisfies the following property:*

- *either it is exactly equal to a breakpoint (i.e., the exact middle of two consecutive floating-point numbers);*

- *or it is equal to a floating-point number;*

- *or its radix-β expansion cannot contain more than $p - 1$ consecutive 0's or $(\beta - 1)$'s after the $p + 1$-th digit.*

The proof is similar to the proof of Lemma 36. In radix 2 and round-to-nearest mode, the situation is even simpler.

Lemma 38 (Additional property in radix 2). *If the radix is 2, when the quotient of two floating-point numbers is of absolute value larger than the underflow threshold $2^{e_{\min}}$, it cannot be a breakpoint of the round-to-nearest mode (i.e., the exact middle of two consecutive floating-point numbers).*

Proof. Let x and y be radix-2, precision-p, floating-point numbers, and let X and Y be their integral significands. Without loss of generality, we can assume that x and y are normal. We have

$$2^{p-1} \leq X, Y < 2^p.$$

Assume that the radix-2 expansion of X/Y has the form

$$\frac{X}{Y} = \delta \times \left(0.q_1 q_2 q_3 \cdots q_p + 2^{-p-1}\right),$$

with $\delta \in \{1, 2\}$ (since $|x/y|$ is larger than $2^{e_{\min}}$, if it is equal to a breakpoint, its infinitely precise significand is necessarily of the form $2 \times (0.q_1 q_2 q_3 \cdots q_p + 2^{-p-1})$). This would give

$$2^{p+1} X = \delta Y (2Q + 1), \tag{12.2}$$

where Q is the integer whose binary representation is $q_1 q_2 q_3 \cdots q_p$. Since $2Q + 1$ is odd, Equation (12.2) implies that δY must be a multiple of 2^{p+1}, which implies that Y must be a multiple of 2^p. And we cannot have $Y = 2^p$, since we assumed $Y < 2^p$. $\qquad\square$

We must raise some important remarks with respect to Lemma 38.

- That property is not true in radix 10. For instance, in precision-2, rounded to nearest, decimal arithmetic, the quotient $0.25/2.0$ is exactly equal to the breakpoint 0.125.

- In radix 2, for that property to hold, it is essential that the quotient is of absolute value above the underflow threshold. For instance, if $p = 3$, then $1.11_2 \times 2^{e_{\min}}$ divided by $1.00_2 \times 2^1$ is $0.111_2 \times 2^{e_{\min}}$, which is the exact middle of the two consecutive floating-point numbers $0.11_2 \times 2^{e_{\min}}$ and $2^{e_{\min}}$.

- It is frequently believed that Lemma 38 holds whenever the radix β is a prime number. This is not true. For instance, in radix 3, with $p = 2$, 2.1_3 divided by 2.0_3 is the exact middle of the two consecutive floating-point numbers 1.0_3 and 1.1_3.

12.3.3   Around Liouville's theorem

A bound on the hardness to round m can always be obtained when the studied function f is algebraic (see Definition 14). Iordache and Matula [195] gave some bounds for the division, the square root, and the square root reciprocal functions. Lang and Muller [239] provided bounds for some other functions. Some of the results given by Lang and Muller in [239] are summarized in Table 12.6. The approaches of [195, 239] are related to the following Diophantine approximation theorem, due to Liouville [264].

Theorem 39 (Liouville [264]). *Let α be an algebraic number of degree $d \geq 2$. There exists a constant C_α such that for all integers u, v, with $v \geq 1$,*

$$\left| \alpha - \frac{u}{v} \right| > \frac{C_\alpha}{v^d}.$$

The effective constant C_α is given by

$$C_\alpha = \frac{1}{\max_{|t-\alpha| \leq 1/2} |P'(t)|},$$

where P is the minimal polynomial of α over $\mathbb{Z}$.

In [50], Brisebarre and Muller have investigated the implications of Liouville's theorem for the TMD for the algebraic functions. Their approach allows us to obtain rather easily certain upper bounds on the hardness to round. For example, let us consider function $x^{a/b}$ with $\gcd(a, b) = 1$ and $a > 0$ for a floating-point number $x \in [1, \beta^b)$ and round-to-nearest mode. We define $k \in \{0, \ldots, a-1\}$ such that $x \in [\beta^{kb/a}, \beta^{(k+1)b/a})$, l as the integer such that $x \in [\beta^l, \beta^{l+1})$, and the two quantities $u = k \cdot b + (p-1) \cdot \max\{a-b, 0\}$ and $v = l \cdot a + (p-1) \cdot \max\{b-a, 0\}$. Then if $x^{a/b}$ is not the middle of two consecutive floating-point numbers, the distance between $x^{a/b}$ and the middle of two consecutive floating-point numbers is always larger than $\beta^{-\mu}$, where

$$\begin{aligned}
\mu &= (p-1) \cdot \max\{a, b\} + (b-1) \cdot (k+1) + \log_\beta(b) + b \cdot \log_\beta(2) \\
&\quad - \min\{u, v\} - \log_\beta \left[\gcd \left(2^b, \beta^{\max\{u-v, 0\}} \right) \right].
\end{aligned}$$

Notice that when a, b, and β are fixed, the quantity μ above grows essentially like $p \cdot \max\{a, b\}$ when the precision p increases. Except for a very few values of a and b, this is much larger than what one can expect from the probabilistic model described in Section 12.2.1.

In general, the bounds derived from Liouville's theorem are crude overestimations of the hardnesses to round. To see this, it suffices to look at the actual hardness to round for function $1/\sqrt{x}$ (which is not in the class of functions considered just above, but for which similar results are also obtained in [50]) for various values of the precision p, given in Table 12.7, and to compare these values to the bound given in Table 12.6.

There have been some improvements to Liouville's theorem, such as the following, due to Roth.

Theorem 40 (Roth [348]). *Let α be an algebraic number of degree $d \geq 2$. For all $\epsilon > 0$, there exists $C_{\epsilon,\alpha} > 0$ such that for all integers u, v, with $v \geq 1$,*

$$\left| \alpha - \frac{u}{v} \right| > \frac{C_{\epsilon,\alpha}}{v^{2+\epsilon}}.$$

Unfortunately, the constant $C_{\epsilon,\alpha} > 0$ in Theorem 40 is not effectively computable.

12.3.4 Generating bad rounding cases for the square root using Hensel 2-adic lifting

In [206], Kahan described a novel way to test the correct rounding of the implementations of the square root function $\sqrt{x}$, in radix 2 and for any precision p and any rounding mode. His method, later strengthened by Parks in [324, 325], explicitly constructs bad rounding cases. It was implemented in the U.C. Berkeley test suite (see Section 3.8.3, page 115), which aims at discovering non-compliances to the IEEE 754 standard. In an independent work, Cornea [84] showed that the hardness to round the square root function can be obtained by solving Diophantine equations that are very similar to the ones below. Unfortunately, that reference does not provide details on how to solve them.

Below, we give a simplified exposition of the method, and refer the interested reader to [325]. In the rest of this section, we suppose that the radix is 2. For simplicity, we will restrict ourselves to input values in the binade $[1, 2)$. Let us first study the rounding breakpoints of the square root function. If the precision-p floating-point number $x \in [1, 2)$ is a bad rounding case, then there exist a p-bit integer y and a real ϵ of tiny absolute value such that:

$$2^{p-1}\sqrt{x} = y + \epsilon \qquad \text{in the case of a directed rounding mode;}$$
$$2^p \sqrt{x} = 2y + 1 + \epsilon \qquad \text{in the case of the round-to-nearest mode.}$$

Function	Size of the largest chain $01111\cdots1$	Size of the largest chain $10000\cdots0$
Reciprocal	$= p$ (p odd) $\le p$ (p even)	$= p$
Division	$= p$	$= p$
Square root	$= p + 2$	$= p$
Inverse square root	$\le 2p + 2$	$\le 2p + 2$
Norm $\sqrt{x^2 + y^2}$ with $\frac{1}{2} \le x, y < 1$	$= p + 2$	$= p + 2$
2D normalization $\dfrac{x}{\sqrt{x^2+y^2}}$ with $\frac{1}{2} \le x, y < 1$	$\le 3p + 3$	$\le 3p + 3$

Table 12.6: Some bounds given by Lang and Muller in [239] on the size $k_{\max}$ of the largest digit chain of the form $1000\cdots0$ or $0111\cdots1$ just after the p-th bit of the infinitely precise significand of $f(x)$ (or $f(x,y)$), for some simple algebraic functions.

p	x (binary)	$1/\sqrt{x}$	$k_{\max}$
4	1.101	$0.110010001101\cdots$	4
5	11.110	$0.10000100001100\cdots$	5
6	11.0100	$0.10001110000000001101\cdots$	10
7	11.11110	$0.100000010000001100\cdots$	7
8	10.011011	$0.1010010001111111110100\cdots$	10
9	11.1111110	$0.100000001000000001100\cdots$	9
10	11.11111110	$0.10000000001000000001100\cdots$	10
11	11.111111110	$0.1000000000001000000000001100\cdots$	11
12	11.1111111110	$0.100000000000010000000000001100\cdots$	12
13	11.11111111110	$0.10000000000000100000000000$ $01100\cdots$	13
14	1.0010001110011	$0.1110111111011101^{14}0101\cdots$	15
15	11.1000101000110	$0.10001000000100001^{16}10111\cdots$	17
16	10.11111110001000	$0.100100111111101110^{15}1000\cdots$	16
17	1.0111111000101100	$0.1101000110000101110^{19}1011\cdots$	20
18	11.1111111111111110	$0.10000000000000000010^{17}1100\cdots$	18
19	1.100010000011110011	$0.1100111011010001000010^{23}1100\cdots$	24
20	1.0000101100011111101	$0.11111010100111001111110^{20}1000\cdots$	21
24	10.1110100001100011100011	$0.100101100010000010011110$ $01^{27}0100\cdots$	28
32	1.00011110000011011000101 1 0101101	$0.111100100010110111011101 0$ $100101010^{32}1011\cdots$	33
53	1.101001101010100110011000 00010101101010111100110011 10	$0.1100011100111011110100001000110$ $00101000110000100101010101^{57}0100\cdots$	58

Table 12.7: *Worst cases for the function $1/\sqrt{x}$, for binary floating-point systems and various values of the precision p (including the case of the double-precision/binary64 format: $p = 53$). The input variables are floating-point numbers satisfying $1 \leq x < 4$, which suffices to deduce all worst cases [239], ©IEEE, 2001, with permission.*

By moving the ϵ's to the left-hand sides and squaring, we obtain that

$$2^{2p-2}x - 2^p\sqrt{x}\epsilon + \epsilon^2 = y^2 \quad \text{in the case of a directed rounding mode;}$$
$$2^{2p}x - 2^{p+1}\sqrt{x}\epsilon + \epsilon^2 = (2y+1)^2 \quad \text{in the case of the round-to-nearest mode.}$$

Since y and $2^{p-1}x$ are integers, so must be $-2^p\sqrt{x}\epsilon + \epsilon^2$ in the case of directed rounding modes and $-2^{p+1}\sqrt{x}\epsilon + \epsilon^2$ in the round-to-nearest mode. As a consequence:

- in the case of a directed rounding mode: $y^2 = k \mod 2^{p-1}$ for some small integer k;

- in the case of the rounding-to-nearest mode: $(2y+1)^2 = k \mod 2^{p+1}$ for some small integer k.

It is now clear that if we could find integer solutions z to the equations $z^2 = k \mod 2^{p+e}$ for $e = \pm 1$ and k a small integer, then we would be able to build bad rounding cases for the square root function.

Let us now show how to find solutions to such equations. We consider the Diophantine equation

$$z^2 = k \mod 2^n,$$

for any arbitrary n. We assume that k is a fixed small odd integer. If n is small enough, the equation can be solved by exhaustively trying all integers in $[0, 2^n)$. This quickly stops being efficient as n increases. Kahan [206] described a way to "lift" a solution modulo 2^i to a solution modulo 2^{i+1} when $i \geq 3$. This method is in fact a core ingredient in the theory of p-adic numbers, and is usually referred to as Hensel lifting (see for example the textbook [223] for an introduction to p-adic numbers). After finding a solution z_3 modulo 2^3, Kahan suggests lifting it progressively to obtain solutions $z_4, z_5, z_6, \ldots$ modulo $2^4, 2^5, 2^6, \ldots$ to eventually obtain a solution z_n modulo 2^n.

Let $z_3 \in \{0, 1, \ldots, 7\}$ such that $z_3^2 = k \mod 2^3$. We assume that such a z_3 exists and has been determined by exhaustive search. Kahan's method works as follows:

- if $z_i^2 = k \mod 2^{i+1}$, let $z_{i+1} = z_i$;

- otherwise, let $z_{i+1} = 2^{i-1} - z_i \mod 2^{i+1}$.

Suppose that we are in the second situation and that $z_i^2 = k \mod 2^i$. Then $z_i^2 = 2^i + k \mod 2^{i+1}$. Furthermore, if $i \geq 3$, then $2i - 2 \geq i + 1$ and:

$$\begin{aligned}
z_{i+1}^2 &= 2^{2i-2} + z_i^2 - 2^i z_i \mod 2^{i+1} \\
&= z_i^2 - 2^i z_i \mod 2^{i+1} \\
&= 2^i(1 - z_i) + k \mod 2^{i+1}.
\end{aligned}$$

Since $z_i^2 = k \mod 2^i$ and k is odd, the integer z_i must be odd as well, so that $1 - z_i$ is even, which implies $2^i(1 - z_i) \mod 2^{i+1} = 0$. As a consequence, $z_{i+1}^2 = k \mod 2^{i+1}$.

For example, if we take $k = -7$ and $z_3 = 1$, we obtain the successive z_i's:

$$z_3 = 1, z_4 = 3, z_5 = 5, z_6 = 11, z_7 = 11, \ldots, z_{52} = 919863403429707.$$

At the end, one may define $x = (z_n^2 - k) \cdot 2^{-2n}$ and check if this value provides a bad rounding case for the square root. It could be useful to modify z_n before defining x by adding to it an integer multiple of 2^{n-1}: the quantity z_n remains a solution to the equation $z_n^2 = k \mod 2^n$, and it may help the derived candidate x belong to the desired binade. To continue with the numerical example just above, if we define

$$x = ((z_{52} + 2^{52})^2 - k) \cdot 2^{-104},$$

then

$$x = \frac{6531209183803571}{4503599627370496}$$

is a double-precision/binary64 floating-point number that belongs to $[1, 2)$ and:

$$\sqrt{x} = \overbrace{1.0011010001001001110001100 \cdots 001011}^{53 \text{ bits}}$$
$$\underbrace{00000000000000000 \cdots 000000000000000}_{50 \text{ zeros}} 1011 \cdots.$$

Note that this method efficiently provides *bad* rounding cases that are not necessarily *worst* cases. Hence, it cannot be directly used to find the hardness to round the square root function (this was not its goal).

12.4 Solving the Table Maker's Dilemma for Arbitrary Functions

The methods described for algebraic functions, although sometimes insufficient (they give bounds on the hardness to round that are frequently too coarse), are elegant and efficient. Unfortunately, many functions considered in the IEEE 754-2008 standard are transcendental, including the exponentials, logarithms, trigonometric functions, and inverse trigonometric functions. For these functions, the methods of the previous section become useless.

12.4.1 Lindemann's theorem: application to some transcendental functions

In 1882, Lindemann proved[5] that e^z is transcendental for every nonzero algebraic complex number z. Since floating-point numbers and breakpoints

---

[5]It was that result that showed the transcendence of π.

are algebraic numbers, we easily deduce that, if we except straightforward cases such as $\ln(1) = 0$ or $\cos(0) = 1$, the (radix e) exponential or logarithm, and the sine, cosine, tangent, arctangent, arcsine, and arccosine of a floating-point number cannot be a breakpoint. As a consequence, if f is any of these functions, for any floating-point number x (the straightforward cases being excepted), there exists a number $m_{f,x}$ such that $f(x)$ is not within a distance $\beta^{m_{f,x}}$ from a breakpoint. Since there are finitely many floating-point numbers in the format being used, we easily deduce that there exists a number $m_f = \max_x(m_{f,x})$, such that for any x, $f(x)$ is not within a distance β^{m_f} from a breakpoint.

Unfortunately, this reasoning does not give any hint on the order of magnitude of m_f. For more general functions, such as erf or Γ, we have no equivalent of Lindemann's theorem, so we do not even know if a number m_f does exist.

12.4.2 A theorem of Nesterenko and Waldschmidt

In [297], Nesterenko and Waldschmidt study the smallness of the expression $\left|e^\theta - \alpha\right| + |\theta - \alpha'|$, where α and α' are algebraic numbers and θ is any nonzero real number. They show that for any real number x:

- either x is far away (in a sense to be made precise below) from an algebraic number;

- or so is its exponential.

Restricted to rational numbers, their result provides some useful information for the study of the worst cases related to the TMD, for several elementary transcendental functions. Their result can be interpreted as an effective variant of Liouville's theorem, in the sense that it provides a lower bound on the distance between the transcendental number $f(x)$ and algebraic numbers (including floating-point numbers).

Before giving their result, we need to define the *Weil height* of an algebraic number.

Definition 17 (Weil height)**.** *Let α be an algebraic number of degree d and $P(x) = \sum_{i \leq d} P_i x^i$ be its minimal polynomial. Let $P(x) = P_d \cdot \prod_{i \leq d}(x - \alpha_i)$ be the factorization of P over the complex numbers. Then the Weil height of α is*

$$H(\alpha) = \left(P_d \cdot \prod_{i \leq d} \max(1, |\alpha_i|) \right)^{\frac{1}{d}}.$$

Suppose that α is a nonzero normal radix β floating-point number in precision p. We write $\alpha = x\beta^{e-p+1}$, where x is the integral significand of α, which implies that x is an integer that belongs to $[\beta^{p-1}, \beta^p)$. From the definition above, we derive the two following facts:

- if $|\alpha| \geq 1$, then $H(\alpha) \leq \beta^p |\alpha|$;

- if $0 < |\alpha| < 1$, then $H(\alpha) \leq \beta^{-e+p-1} \leq \beta^p / |\alpha|$.

We thus conclude that, for any nonzero floating-point number α, we have

$$H(\alpha) \leq \beta^p \max(|\alpha|, 1/|\alpha|).$$

Let us now give the result found by Nesterenko and Waldschmidt, in the special case where α and α' are rational numbers.

Theorem 41 (Y. Nesterenko and M. Waldschmidt [297], specialized here to the rational numbers). *Let α and α' be rational numbers. Let θ be an arbitrary nonzero real number. Let A, A', and E be positive real numbers with[6]*

$$E \geq e, \quad A \geq \max\left(H(\alpha), e\right), \quad A' \geq H(\alpha').$$

Then
$$\begin{aligned}
&\left|e^\theta - \alpha\right| + |\theta - \alpha'| \geq \\
&\exp\Big\{-211 \cdot \big(\ln A' + \ln\ln A + 2\ln(E \cdot \max\{1, |\theta|\}) + 10\big) \\
&\cdot \big(\ln A + 2E|\theta| + 6\ln E\big) \cdot \big(3.7 + \ln E\big) \cdot \big(\ln E\big)^{-2}\Big\}.
\end{aligned}$$

Now, suppose that α is a precision-p floating-point number in $[1, \beta)$. Consider the TMD for the exponential function. The exact value $\exp(\alpha)$ belongs to the interval $[e, e^\beta)$. Let k be such that the latter interval is included in $[1, \beta^k)$. We now use the theorem of Nesterenko and Waldschmidt with $E = e$ and $\theta = \alpha'$, where α' is any precision-p floating-point number in $[1, \beta^k)$. We obtain the following:

$$\begin{aligned}
&\left|e^{\alpha'} - \alpha\right| \geq \\
&\exp\Big(-992 \cdot ((3k+p)\ln\beta + \ln\left((p+1)\ln\beta\right) + 12) \cdot \left((p+1)\ln\beta + 2e\beta^k + 6\right)\Big).
\end{aligned}$$

For instance, in the case of binary floating-point arithmetic ($\beta = 2$, $k = 3$), this gives

$$\left|e^{\alpha'} - \alpha\right| \geq 2^{-688p^2 - 992p\ln(p+1) - 67514p - 71824\ln(p+1) - 1283614}.$$

This shows that a precision-m approximation $\hat{f}$ to the exponential function is sufficient to solve the TMD in $[1, \beta)$, with $m \approx Cp^2$ for some rather big constant C. Unfortunately, the constant C that can be derived from Theorem 41 is very large. For the double-precision/binary64 format, we find

$$\left|e^{\alpha'} - \alpha\right| \geq 2^{-7290678}.$$

---

[6]Here, $e = 2.718\cdots$ is the base of the natural logarithm.

For double-precision calculations and the most common functions (exponential, logarithm, sine, etc.) the order of magnitude of the bound on m that can be derived from Theorem 41 is a few millions of bits. Computing with such a precision is feasible, but far too expensive for practical purposes (and remember: from the probabilistic arguments of Section 12.2.1, although we have no proof, we "know" that the actual value of $m_{\max}$ is around 120 for double-precision/binary64 arguments).

One may try to decrease that huge bound on m by improving the proof of Nesterenko and Waldschmidt for the special case where α and α' are floating-point numbers. However, the constant is likely to remain large.

Also, by considering $\theta = \ln \alpha$, one can derive a similar result for the (natural) logarithm function, and get a similar provably sufficient precision $m' \approx C' p^2$, for some big constant C'.

The sufficient precisions obtained using Theorem 41 are too large to be useful in practice for solving the TMD. However, they have the advantage of being easily computable, for any precision p and any radix β.

12.4.3   A first method: tabulated differences

A first solution to find the hardness to round of some function f is to test every possible input. This can be done by evaluating f with a multiple-precision library that guarantees the error bound, but this would be very slow, and solving the TMD in double precision like that (up to about 2^{64} input numbers for each function) would not even be feasible in practice. To speed up such exhaustive tests, one can use the following property of the floating-point numbers: except when the exponent changes, the floating-point numbers are in *arithmetic progression* (e.g., for binary64, $1, 1 + 2^{-52}, 1 + 2 \cdot 2^{-52}, 1 + 3 \cdot 2^{-52}$, ..., up to $2 = 1 + 2^{52} \cdot 2^{-52}$). The exponent change is not a problem in practice, as it occurs rarely. In general, the algorithms that will be described below will not even be able to exploit the full length of the arithmetic progressions.

To be able to use the above property, one will approximate the function f by a polynomial P on some interval I with some error bound ε on the significand of $f(x)$ (this error bound must include the error of the evaluation of P). Then P can be evaluated very quickly on successive floating-point numbers (in arithmetic progression) by using *tabulated differences* [222]: if P has degree d, after a fast initialization, computing each new value only needs d additions.

Let us show how it works. Let $x_1, x_2, x_3, \ldots$ be numbers in arithmetic progression (i.e., $x_{i+1} - x_i = \delta, \forall i$). Define $\Delta^{(0)} P = P$, then

$$\Delta^{(1)} P(x) = P(x + \delta) - P(x),$$

and, for $i \leq d$,

$$\Delta^{(i+1)} P(x) = \Delta^{(i)} P(x + \delta) - \Delta^{(i)} P(x).$$

One can easily show that $\Delta^{(d)} P$ is a constant C. This suggests a method for quickly evaluating the values $P(x_i)$: from a few first values we compute

$\Delta^{(i)}P(x_0)$, for $i = 1, \ldots, d$. Then, each time we want to compute the value of P at a new point x_{j+1}, we evaluate

$$\Delta^{(d-1)}P(x_{j+1}) = \Delta^{(d-1)}P(x_j) + C$$

and, for $i = d - 2, d - 3, \ldots, 0$,

$$\Delta^{(i)}P(x_{j+1}) = \Delta^{(i)}P(x_j) + \Delta^{(i+1)}P(x_j).$$

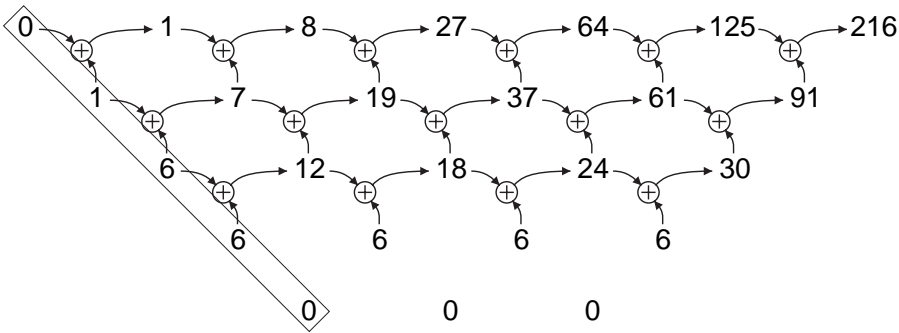A very simple example is shown in Figure 12.3, in the case $\delta = 1$.

Figure 12.3: This figure shows how to compute $P(1), P(2), P(3), \ldots$, for $P(X) = X^3$ with 3 additions per value. The numbers on the left are the coefficients of P in the base $\left\{1, X, \frac{X(X-1)}{2}, \frac{X(X-1)(X-2)}{6}\right\}$.

Testing whether $P(x)$ is within $\beta^{-m} + \varepsilon$ from a breakpoint allows one to keep all the values for which $f(x)$ is within β^{-m} from a breakpoint (the potential worst cases) while eliminating most input values, in particular, all those for which $f(x)$ is not within $\beta^{-m} + 2\varepsilon$ from a breakpoint. This can be regarded as a filter: if m is large enough and ε small enough, then it is possible to test the remaining values in a reasonable time by using a multiple-precision library, so that the hardness to round f can be found.

Since the breakpoints are also in arithmetic progression (except in the rare cases where the exponent changes, which can be worked around easily), one does not need to compute $P(x)$: it suffices to compute $P(x)$ modulo δ, where δ is the distance between two breakpoints. So, the additions mentioned above can be carried out modulo δ, avoiding the computation of the most significant digits.

We have not yet explained how to approximate f by a polynomial P. Again, a naive algorithm may be too slow, especially if I is small, as this would mean many approximations to compute: we would need to split the input domain into too many intervals I. The method suggested in Lefèvre's thesis [249] is to use some computer algebra system to approximate f by a high-degree polynomial Q on a large interval. This interval can be split into

N subintervals (in a regular way) on which Q can be approximated by polynomials P_0, P_1, ..., P_{N-1} of a smaller degree. The work [249] describes how one can deduce an approximation P_i from the previous one P_{i-1} (the idea is to use techniques similar to the tabulated differences). This method can be used recursively.

We have introduced several parameters: the value of m, the error bound, the length of the interval I, and the degree d of the polynomial P. We would like to have a small error bound (so that very few values remain after the tests), a large interval I (to reduce the time needed to compute the approximations), and a small degree d (so that evaluating the next value can be done quickly). However, there are constraints between these parameters. For instance, a small error bound will imply a small interval I and/or a high degree d. Thus, one needs a compromise between these parameters. It is not clear how to make the best choice (it depends very much on the implementation), but the statistical arguments presented in Section 12.2.1 can be useful, e.g., to guess how many values will remain.

Unfortunately, there exist some functions and domains for which the approximation by polynomials with reasonable parameters is not possible. This is the case of the trigonometric functions on very large arguments: due to the cancellation in the range reduction, there is no clear regularity between consecutive values of the function. However, the fact that these functions are periodic can be exploited (see Section 12.4.7).

Finding worst cases in single precision could be done using the naive method, but this first advanced method was a big step toward solving the TMD for most functions in double precision. However, it is still too slow to be used in practice. Two other algorithms will be presented in the upcoming sections: Lefèvre's algorithm (Section 12.4.5) and the SLZ algorithm (Section 12.4.6). These algorithms are complementary, Lefèvre's algorithm being more suitable to low precisions (up to double precision), and SLZ being more suitable to higher precisions, such as the double-extended or quadruple/binary128 (or decimal128) precisions, and to find bounds on the hardness to round the function. Both kinds of algorithms are based on notions introduced in this section.

Before presenting these algorithms, we give a "graphical" presentation of the problem.

12.4.4   From the TMD to the distance between a grid and a segment

We have already used the fact that:

- the input values are in an arithmetic progression;
- the breakpoints are in an arithmetic progression.

These two important properties will be used to design faster algorithms. Basically the problem is to find the points of a regular grid that are close enough to the graph of function f.

But since the input values are the machine numbers, and the breakpoints are (depending on the rounding mode) the machine numbers or the middle points between consecutive machine numbers, these properties also imply a symmetry between input and output: the graph of the inverse[7] function f^{-1} can be regarded as the same as the graph of f, and the grids for f and for f^{-1} can be joined to form a single grid, as shown in Figure 12.4.
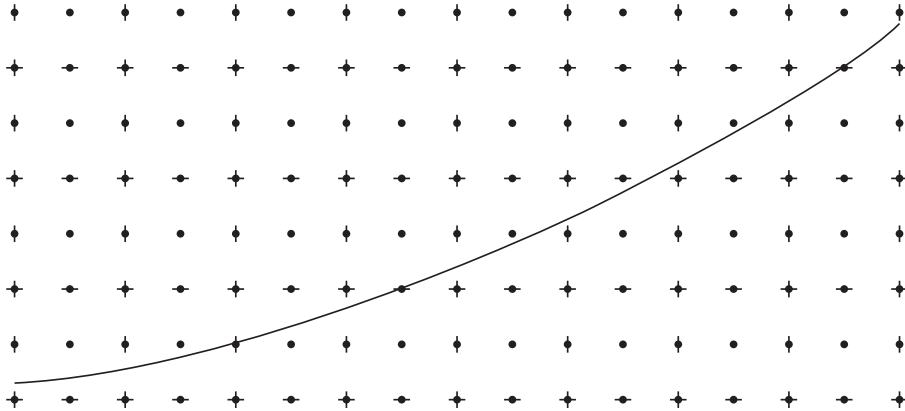

Figure 12.4: *The graph of f (and f^{-1}) and a regular grid consisting of points whose coordinates are the breakpoints. More precisely, the coordinate of a (vertical or horizontal) row having a small segment on all its points is a machine number. An intersection between the graph and a vertical (resp. horizontal) segment corresponds to a bad case for f (resp. f^{-1}). Note that in practice, the segments are chosen much smaller than on the figure: if double-precision/binary64 is at stake, a typical segment length is 2^{-50} times the distance between two points.*

By considering this new grid, searching for the worst cases of the inverse function f^{-1} can be done at the same time as searching for the worst cases of f, with little additional cost. Although this doubles the number of input points to be tested, the approximation part will globally be faster, and by adequately choosing between f and f^{-1}, this will be faster than testing both functions separately.

For instance, in double-precision/binary64 arithmetic, the worst cases of $f(x) = 2^x$ among the $2^{52} = 4,503,599,627,370,496$ input values such that $2^{-39} \leq x < 2^{-38}$ will be quickly obtained by considering the 5680 possible machine numbers y such that:

$$1 + 5678 \cdot 2^{-52} \leq f(2^{-39}) \leq y \leq f(2^{-38}) \leq 1 + 11357 \cdot 2^{-52}.$$

---

[7]In practice, we can always decompose the tested domain into subdomains on which f is invertible, if need be. From the discussion that follows, considering f^{-1} may be useful even when one is not interested in the worst cases for this inverse function.

At the same time, if the factor on the number of input values between f and f^{-1} is very large (around 2^{40} in the example), then the selected lower bound on k (as defined in Section 12.2.1) for the tested function must be much lower to be able to find bad cases for the inverse function. For instance, the worst case of $f(x) = 2^x$ in the above domain,

$$f(1.111001110001011100000100100011001101010110100111100 \times 2^{-39})$$

$$= \overbrace{1.0000000000000000000000000000000000000010101000110100}^{53 \text{ bits}}$$
$$\underbrace{00000000000000000000000000000000000000000000000000000000}_{54 \text{ zeros}}1110\cdots$$

will be found from the following bad case for $\log_2$:

$$\log_2(1.0000000000000000000000000000000000000010101000110100)$$

$$= \overbrace{1.111001110001011100000100100011001101010110100111011}^{53 \text{ bits}}$$
$$\underbrace{11111111111111}_{14 \text{ ones}}0101\cdots.$$

Notice the similar factor on the distance between the exact result and the closest machine number.

Hence, testing f and f^{-1} together will be more efficient than testing them separately. The fact that the worst cases for f and f^{-1} are closely related can be used even if we test both functions separately: we can then compare the results and detect possible bugs.

As additional points are added to the grid (the points for which both coordinates are middles of consecutive machine numbers), spurious bad cases (which correspond to neither f nor f^{-1}) will be found by the various algorithms. But they can be filtered out later, again with little additional cost.

12.4.5   Linear approximation: Lefèvre's algorithm

First, for simplicity, we will scale the input and the output values so that the input values in the considered interval are the first non-negative integers and the breakpoints are integers. More precisely, the scaling has the following form, assuming that the distance between two consecutive input values is β^{-v} and the distance between two consecutive breakpoints is $\beta^{-w}/2$: instead of considering function f on interval I, we will consider g such that $g(k) = 2 \cdot \beta^w \cdot f(u + \beta^{-v}k)$ with non-negative integers $k < N$, for some floating-point number u (the left endpoint of I) and integers v, w, and N.

Now the problem is to find the non-negative integers $k < N$ such that $g(k)$ is very close to an integer, say, such that for some given small positive real number δ_0, one has

$$\exists\, u \in \mathbb{Z},\ |g(k) - u| < \delta_0/2.$$

The algorithm will be based on a linear approximation of the function: g is approximated by a degree-1 polynomial $P(x) = b_0 - ax$, where a and b_0 are real numbers. Taking into account the error of the approximation, the problem becomes:

$$\exists\, u \in \mathbb{Z},\ |(b_0 - k \cdot a) - u| < \delta/2$$

for some positive real number δ, i.e.,

$$\exists\, u \in \mathbb{Z},\ -\delta/2 < (b_0 - k \cdot a) - u < \delta/2.$$

We can translate the segment $y = b_0 - a.x$ upward so that we search for values slightly above the integers: with $b = b_0 + \delta/2$, the condition becomes

$$\exists\, u \in \mathbb{Z},\ 0 < (b - k \cdot a) - u < \delta.$$

Stated differently, the problem reduces to finding the non-negative integers $k < N$ such that $\{b - k \cdot a\} < \delta$, where $\{y\}$ denotes the positive fractional part of y. This transformation will yield a simpler algorithm, because only one comparison is needed instead of two (the fractional part being computable without any comparison).

One can consider the grid modulo 1 ($\{y\}$ is equal to y modulo 1). The upper part of Figure 12.5 shows how the integer grid and the segment $y = b - a.x$ are reduced modulo 1 for both coordinates (x and y). The value of $\{b - k \cdot a\}$ for an integer k is the distance of the point k to the lower left point of the big square. For instance, on this figure, the distance is minimal for $k = 1$.

The points k on the left segment have a particular structure, which will be used by the algorithm. Before going further, notice one of the properties of this structure. The endpoints of the left segment can be joined to form a circle (still from the reduction modulo 1). It can be proved that if one considers all the distances between adjacent points on the circle for given values of N, a, and b, then one obtains at most three different values. This is known as the *three-distance theorem* [382, 399, 401].

In the following, we will work in $\mathbb{R}/\mathbb{Z}$, the additive group of the real numbers modulo 1. We will choose representatives in the interval $[0, 1)$. A number $y \in \mathbb{R}$, such as a and b, can also be regarded as an element of $\mathbb{R}/\mathbb{Z}$, and its canonical representative is the real number $\{y\}$. The operation consisting in adding an element $\alpha \in \mathbb{R}/\mathbb{Z}$ can be regarded as a translation by α on the segment $[0, 1)$, with possible wrapping (since the endpoints 0 and 1 correspond to the same element of $\mathbb{R}/\mathbb{Z}$), or, stated differently, a rotation by α on the corresponding circle. Finally, if k is a non-negative integer, k is said to be the (group) *index* of the element $k \cdot a$ (in the additive subgroup of $\mathbb{R}/\mathbb{Z}$ generated by a).

We now study the configurations $C_n = \{k \cdot a \in \mathbb{R}/\mathbb{Z} : k \in \mathbb{N},\ k < n\}$ for $n \geq 2$. For simplicity, let us assume that, in the following, n is small enough
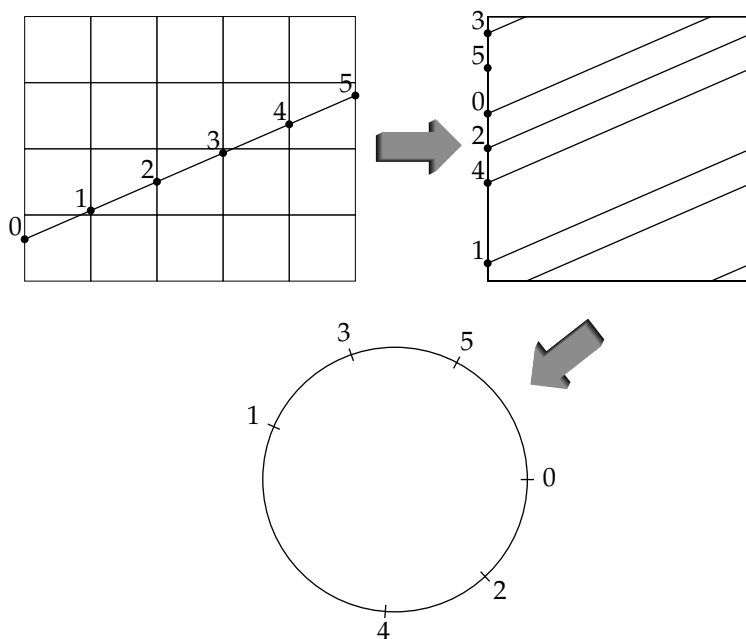
Figure 12.5: An example with $N = 6$ [252], © IEEE 1998, with permission. This figure shows: the integer grid and the segment $y = b - a.x$; the two-dimensional transformation modulo 1; and the representation of the left segment (corresponding to $x \in \mathbb{Z}$) modulo 1 as a circle.

so that $k \cdot a$ is never 0 for $k < n$ (the particular case $k \cdot a = 0$ can occur in practice as implementations work with numbers that are rational, and one must check that the algorithm can handle it correctly).

In any configuration C_n, the n points $k \cdot a$ split the segment $[0, 1)$ into n intervals. These intervals have at most three possible different lengths, which depend on n (this is the three-distance theorem, mentioned above). Moreover, when a is an irrational number, there are infinitely many particular configurations C_n (see Figure 12.6 for an example), for which these intervals have exactly two possible lengths h and ℓ (with $h > \ell$).[8] We will call these configurations *two-length configurations*.

When points are added to a two-length configuration (by increasing n), each new point splits some interval of length h into an interval of length ℓ and an interval of length $h - \ell$, in some fixed order (explained below). Once all the intervals of length h have been split, one obtains a new two-length configuration with intervals of lengths ℓ and $h - \ell$ only. Note that replacing $\{h, \ell\}$ by $\{\ell, h - \ell\}$ corresponds to a step of the well-known subtractive *Euclidean algorithm* for computing GCDs.

_____

[8]Due to accidental equalities when a is a rational number, as is the case in practice, the real property of such a configuration is not the fact that intervals have exactly two possible values, but for the moment, let us explain it this way.
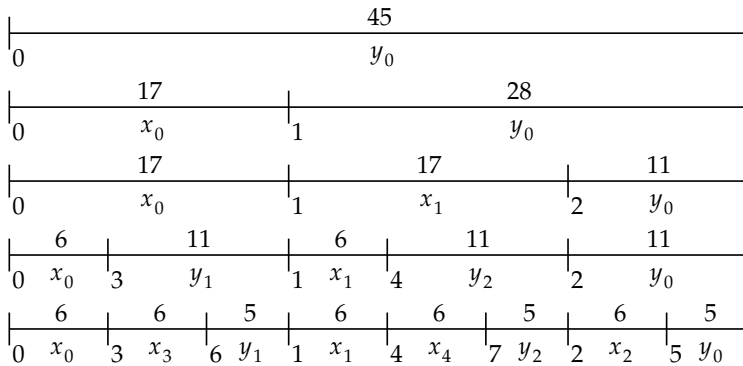
$$
\begin{array}{l}
45 \\
0 \qquad y_0 \\[4pt]
17 \qquad\qquad 28 \\
0 \quad x_0 \qquad 1 \qquad y_0 \\[4pt]
17 \qquad 17 \qquad 11 \\
0 \quad x_0 \qquad 1 \quad x_1 \qquad 2 \quad y_0 \\[4pt]
6 \quad 11 \quad 6 \quad 11 \quad 11 \\
0\ x_0 \ \ 3 \quad y_1 \ \ 1\ x_1 \ \ 4 \quad y_2 \ \ 2 \quad y_0 \\[4pt]
6 \ 6 \ 5 \ 6 \ 6 \ 5 \ 6 \ 5 \\
0\ x_0 \ 3\ x_3 \ 6\ y_1 \ 1\ x_1 \ 4\ x_4 \ 7\ y_2 \ 2\ x_2 \ 5\ y_0
\end{array}
$$

Figure 12.6: Two-length configurations for $a = 17/45$. The points $k \cdot a$ for $0 \le k < n$ are represented by the vertical bars, with the values of k next to them. The name of each interval (x_r or y_r) and the corresponding length (scaled by 45) are given. The initial configuration considered in this section has two points and two intervals, but a virtual configuration with one point (0) and one interval is shown here for completeness, as it may be used in some codes.

More precisely, these two-length configurations C_n satisfy the following properties (the values of u, v, x, $y\ldots$ below all depend on a and n).

- The n points $0 \cdot a$ (the origin), $1 \cdot a$, $2 \cdot a$, $\ldots$, $(n-1) \cdot a$ modulo 1 split the segment $[0,1)$ into u intervals of length x and v intervals of length y, where $n = u + v$.

- The intervals of length x are denoted x_0, x_1, $\ldots$, x_{u-1}, where x_0 is the leftmost interval of $[0,1)$ and $x_r = x_0 + r \cdot a$ (i.e., x_r is x_0 translated by $r \cdot a$ in $\mathbb{R}/\mathbb{Z}$). Moreover, the left endpoint of x_r has index r.

- The intervals of length y are denoted y_0, y_1, $\ldots$, y_{v-1}, where y_0 is the rightmost interval of $[0,1)$ and $y_r = y_0 + r \cdot a$ (i.e., y_r is y_0 translated by $r \cdot a$ in $\mathbb{R}/\mathbb{Z}$). Moreover, the left endpoint of y_r has index $u + r$.

Figure 12.6 shows the segment with its initial point 0 and the first four two-length configurations for $a = 17/45$. The index of each point is written on the figure (here, from 0 to 7 on the last configuration). And for each interval, the name (x_r or y_r) and the length of the interval are also written. The reader can check that these properties are satisfied for each configuration.

Now, let us see on this example of Figure 12.6 how a two-length configuration C_n is transformed into the next two-length configuration C_m, when points n, $n+1$, $\ldots$, $m-1$ are added. First, consider the configuration C_3 (i.e. with 3 points and 3 intervals); one has $u = 2$, $v = 1$, $x = 17$, and $y = 11$. Since $x > y$, each of the two intervals of length $x = 17$ will be split into two intervals of respective lengths $x - y = 17 - 11 = 6$ (on the left) and $y = 11$

(on the right), in the same order given by the indices: first x_0 (point $3 \cdot a$), then x_1 (point $4 \cdot a$). Note that the interval $[0, 3 \cdot a)$ of length 6 becomes the new leftmost interval. Thus, one gets configuration C_5 on the next line of the figure: $u = 2$, $v = 3$, $x = 6$, $y = 11$. Similarly, on C_5, one has $y > x$ this time. So each of the three intervals of length $y = 11$ will be split into two intervals of respective lengths $x = 6$ (on the left) and $y - x = 11 - 6 = 5$ (on the right), in the same order given by the indices: y_0 (point $5 \cdot a$), then y_1 (point $6 \cdot a$), then y_2 (point $7 \cdot a$), and the interval $[5 \cdot a, 1) = [5 \cdot a, 0 \cdot a)$ becomes the new rightmost interval in this new configuration C_8. These properties and transformations are proved in [250].

We have seen that the points $k \cdot a$ modulo 1 have a particular structure. Now let us show how the above properties can be used to efficiently compute the minimum value d of $\{b - k \cdot a\}$ for $0 \le k < N$ and the corresponding value of k, assuming C_N is a two-length configuration. The value d is the distance between b and the closest point of C_N on its left. Stated differently, the quantity b is in an interval $[k \cdot a, k' \cdot a)$ of C_N, where $k \cdot a$ and $k' \cdot a$ are adjacent, and $d = \{b - k \cdot a\}$ for this definition of k. So, in order to determine d and k, we will start from the initial configuration, and run through the successive two-length configurations, updating all data corresponding to the configuration and the position of b. Table 12.8 shows how, from the interval I containing b, its length, and the current position of b in I, one obtains the data for the next two-length configuration, i.e., the new interval I' containing b and the position of b in I'. Table 12.9 shows the various data on the example, with $b = 23.5/45$.

Case	Current configuration			Next configuration	
	Interval I	Length	Test	Interval I'	New d
1	any	ℓ		same	same
2	x_r		$d < x'\ (= x - \ell)$	x_r	same
3		h	$d \ge x'\ (= x - \ell)$	y_{r+v}	$d - x'$
4	y_r		$d < x$	x_{r+u}	same
5			$d \ge x$	y_r	$d - x$

Table 12.8: On the left, data corresponding to the current two-length configuration: the interval I containing b, its length, and the position of b in I. On the right, data one can deduce for the next two-length configuration: the new interval I' containing b and the position of b in I'.

These computations can be rearranged to give Algorithm 12.1.
Algorithm 12.1 returns:

- the first non-negative value $r < N$ and a value d such that $d = \{b - r \cdot a\} < d_0$ if there is such an integer r;

Algorithm 12.1 Returns the first non-negative value $r < N$ and a value d such that $d = \{b - r \cdot a\} < d_0$ if there is such an integer r, else an integer larger or equal to N and a lower bound on $\{b - r \cdot a\}$ for $r < N$. For any bracket (on the right), one has $ux + vy = 1$.

$x \leftarrow \{a\}$
$y \leftarrow 1 - \{a\}$
$d = \{b\}$
$u \leftarrow 1$
$v \leftarrow 1$
$r \leftarrow 0$
if $d < d_0$ **then return** $(0, d)$
loop
  if $d < x$ **then**
    while $x < y$ **do**
      if $u + v \geq N$ **then return** (N, d) $[h = y, b \in x_r]$
      $y \leftarrow y - x$
      $u \leftarrow u + v$
    end while
    if $u + v \geq N$ **then return** (N, d) $[h = x, b \in x_r]$
    $x \leftarrow x - y$
    if $d \geq x$ **then**
      $r \leftarrow r + v$
    end if
    $v \leftarrow v + u$
  else
    $d \leftarrow d - x$
    if $d < d_0$ **then return** $(r + u, d)$ $[b \in y_r]$
    while $y < x$ **do**
      if $u + v \geq N$ **then return** (N, d) $[h = x, b \in y_r]$
      $x \leftarrow x - y$
      $v \leftarrow v + u$
    end while
    if $u + v \geq N$ **then return** (N, d) $[h = y, b \in y_r]$
    $y \leftarrow y - x$
    if $d < x$ **then**
      $r \leftarrow r + u$
    end if
    $u \leftarrow u + v$
  end if
end loop

---

x	y	u	v	I	d	Case	I'	d'
0	45	0	1	y_0	23.5	5	y_0	6.5
17	28	1	1	y_0	6.5	4	x_1	6.5
17	11	2	1	x_1	6.5	3	y_2	0.5
6	11	2	3	y_2	0.5	4	x_4	0.5
6	5	5	3	x_4	0.5	2	x_4	0.5

Table 12.9: Example with $a = 17/45$ and $b = 23.5/45$. For better readability, the values of x, y, d, and d' have been multiplied by 45.

- an integer larger than or equal to N and a lower bound on $\{b - r \cdot a\}$ for $r < N$ (which is, in fact, the minimum value of $\{b - r \cdot a\}$ on a larger interval: $r < u + v$) otherwise.

To get all the values $r < N$ such that $\{b - r \cdot a\} < d_0$, one can subtract $(r+1) \cdot a$ from b and $r + 1$ from N, and rerun the algorithm until one reaches N.

   If one is just interested in a lower bound d on $\{b - r \cdot a\}$ for $r < N$, then it suffices to remove all the lines where variable r appears. This simplification is useful in practice when one wants to search for the worst cases of a given function, because generally d is large enough to allow one to deduce that there are no worst cases in the considered interval. In the (rare) case where d is too small for allowing such a deduction, the full algorithm can be rerun to find the potential worst cases.

   This algorithm has several variants. For instance, some loops could be unrolled and/or some tests could be performed at different places. One can also replace sequences of subtractions by divisions, as explained in [250].

   Other works make use of linear approximations as in Lefèvre's algorithm to study the TMD or problems closely related to it. In [124], Elkies used similar linear approximations in an algorithm that finds small rational points (i.e., with small numerators and denominators) near curves. For example, for the curve $x^3 - y^2$, he obtained:

$$5853886516781223^3 - 447884928428402042307918^2 = 161843.$$

This corresponds to a bad rounding case of the function $x^{3/2}$, with an input precision of 53 bits and an output precision of 79 bits. Indeed, for $x = 5853886516781223$, the binary expansion of $x^{3/2}$ is

$$\underbrace{1.011110110\cdots 1001101100001010100110}_{\text{79 bits}}\,\underbrace{0000000\cdots 00000000}_{\text{58 zeros}}\,100\ldots \times 2^{-78}.$$

   In [152], Gonnet described how to find hard-to-round cases by using the Lenstra–Lenstra–Lovász (LLL) lattice reduction algorithm [255], which

looks similar to the SLZ algorithm (see below), due to Stehlé, Lefèvre, and Zimmermann. Gonnet's method corresponds to the SLZ algorithm with degree-1 approximations, which is essentially another way to formulate Lefèvre's algorithm.

12.4.6 The SLZ algorithm

To simplify the presentation of the following section, we will once more restrict ourselves to the case of radix 2 floating-point formats. As demonstrated in [253], the method described here may be adapted to any radix (indeed, one of its first applications was finding the worst case for the exponential function in the decimal64 format of the IEEE 754-2008 standard).

To summarize very quickly, Lefèvre's algorithm consists in replacing the function f under study by a piecewise linear approximation, and then finding the worst cases of each one of the linear approximations on its definition domain. By taking into account the approximation error, we see that all bad rounding cases of f are (possibly slightly less) bad rounding cases for the linear approximations. Furthermore, if the subintervals where each linear approximation is valid are small enough, then the bad rounding cases of the linear approximations can be found efficiently, as described in the previous section. If we are trying to find the worst cases of a regular elementary function f over a given binade, e.g., $[1, 2)$, and if p is the precision, then one can show that we need around $2^{2p/3}$ subintervals of length $2^{-2p/3}$. Each subinterval requires a number of operations that is polynomial in p. Roughly speaking, the overall cost of the computation is around $2^{2p/3}$.

Higher-degree polynomial approximations

The bottleneck of Lefèvre's method lies in the number of linear approximations that are required to approximate the function with the accuracy that is necessary for the tests. In order to decrease the number of subintervals to be considered, and thus the overall cost of the search for bad cases, it is tempting to consider better approximations; namely, polynomial approximations of higher degree. To achieve that goal, Lefèvre, Stehlé, and Zimmermann [389, 390] suggested computing a piecewise, constant-degree, polynomial approximation to the considered function f. Over an interval of width $\tau < 1$, we can expect a good degree-d polynomial approximation to f to have a maximal error around τ^{d+1}: hence, by choosing a higher degree, we can get approximations that work in larger intervals.

Consider the following example, with parameters that correspond to Lefèvre's method. Take the binade $[1, 2)$ and a precision $p = 53$ (which corresponds to the binary64 format or IEEE 754-2008). Suppose we want a piecewise approximation to a regular enough function f on the binade, with absolute error less than around 2^{-72}. With linear functions, i.e., $d = 1$,

we would be allowed subintervals of length around 2^{-36}. In total, we would thus require approximately 2^{36} subintervals. Suppose now that we consider degree-3 polynomials. Then we are allowed subintervals of length around 2^{-18}, thus requiring only approximately 2^{18} subintervals in total. By increasing the degree of the approximations up to around 72, we see that only a small number of subintervals is then needed to approximate f as tightly as we decided.

It thus seems that by increasing the degree sufficiently, we could find all worst rounding cases by studying a constant number of subintervals. Unfortunately, finding the bad rounding cases of a polynomial of degree $d > 1$ seems to be significantly more complicated than finding the bad rounding cases of a linear function. In addition to the quality of the approximation, one has to take into account the feasibility of finding the bad cases of the approximating polynomials. Suppose we consider a precision p. In [389], Stehlé, Lefèvre, and Zimmermann showed how to use degree-2 approximations on subintervals of length around $2^{-3p/5}$ to find the worst cases over a given binade in time $\approx 2^{3p/5}$. They refined their analysis in [390] to obtain a cost of around $2^{4p/7}$, still using degree-2 approximations. Later on, Stehlé [387, 388] strengthened the study further and showed that by using degree-3 polynomials the cost can be decreased to $2^{p/2}$. For the moment, higher degree polynomials seem useless to determine the hardness to round.

Bad rounding cases of polynomials

Suppose a degree-d polynomial P approximates a function f on an interval of length τ. As discussed above, we expect that $|P(x) - f(x)| \leq c\tau^{d+1}$ for any real x in the interval, for some constant c. Suppose we want to find all precision-p floating-point numbers x in that interval such that $f(x)$ is within distance 2^{-m} from a floating-point number (this corresponds to searching bad cases for the directed rounding modes, but can be adapted easily to the round-to-nearest mode). More precisely, we look for the floating-point numbers x such that there exists an integer k with:

$$|2^p \cdot f(x) - k| \leq 2^{-m+p}.$$

If $a \mod 1$ denotes the centered fractional part of a, i.e., the number $a' \in [-1/2, 1/2)$ such that $a - a'$ is an integer, then the preceding equation can be rewritten as:

$$|2^p \cdot f(x) \mod 1| \leq 2^{-m+p}.$$

If we restrict ourselves to the x's belonging to the interval for which P approximates f, then, by the triangular inequality, we obtain:

$$|2^p \cdot P(x) \mod 1| \leq 2^{-m+p} + 2^p c\tau^{d+1}.$$

This means that x is also a somewhat bad rounding case for the polynomial P. If $2^p c\tau^{d+1} \approx 2^{-m+p}$, then it suffices to look for the bad rounding cases of P

that are of a slightly degraded quality (the new m is slightly smaller than the former one).

Suppose now that we are interested in finding the precision-p floating-point numbers x, belonging to a given interval of length τ, that satisfy:

$$|2^p \cdot P(x) \mod 1| \leq 2^{-m+p}.$$

To simplify, suppose that we shift the interval in order to center it around 0. We can then consider the following bivariate polynomial equation:

$$Q(t, u) = 0 \mod 1, \tag{12.3}$$

where $Q(t, u) = 2^p \cdot P(2^{-p}t) + u$. We are interested in solutions (t_0, u_0) such that t_0 is an integer in $[-2^{p-1}\tau, 2^{p-1}\tau]$ and u_0 belongs to the interval $[-2^{-m+p}, 2^{-m+p}]$.

Note that if we had two such bivariate polynomial equations without the "mod 1," then we could possibly eliminate variables in order to find the solutions. One would still need to be lucky enough to have the two polynomials be algebraically independent. The SLZ method relies on such an independence assumption (which seems satisfied most often in practice). But so far, we have only one polynomial equation, modulo 1.

We are interested in the small solutions to Equation (12.3): the ranges of variables in which we are interested are very restricted. Indeed, the range of interest for the variable u is $[-2^{-m+p}, 2^{-m+p}]$, although the modulus would make it natural to consider the larger interval $[-1/2, 1/2]$. The range of interest of the variable t is $[-2^{p-1}\tau, 2^{p-1}\tau]$, although the modulus and the 2^{-p} scaling would make it more natural to consider the (much) bigger interval $[-2^{p-1}, 2^{p-1}]$. Such range-restricted polynomial equations modulo an integer have been extensively studied in the field of cryptology. Informally, today's leader, the RSA Public Key Cryptosystem relying on integer factorization involves several polynomial equations. In some contexts, often deriving from a will to speedup the system, information providing the secret key is contained in the small solutions of these equations. Solving range-restricted polynomial equations thus leads to several cryptanalyses of variants of RSA. With respect to our concerns about the bad rounding cases for elementary functions, this implies that we can try using the methods developed by cryptographers to solve the TMD. For instance, the SLZ algorithm relies on Coppersmith's method to find small roots of polynomials modulo an integer [82, 83].

Let us now describe Coppersmith's method. We mentioned two difficulties while trying to solve Equation (12.3): we would prefer having several polynomial equations instead of a single one, and we would like to eliminate the modulus. To work around the first problem, we use shifts and powers of the initial polynomial. If $Q(t, u) = 0 \mod 1$, then

$$\forall i, j \geq 0, \forall k > 0, \ t^i u^j \cdot Q^k(t, u) = 0 \mod 1.$$

This already provides infinitely many polynomial equations. Furthermore, suppose that $R(t, u)$ is an integer linear combination of finitely many $t^i u^j Q^k$'s: we have

$$R(t, u) = \sum_{i,j,k} n_{i,j,k} \cdot t^i u^j \cdot Q^k(t, u),$$

where the $n_{i,j,k}$'s are integers and only a finite number of them are nonzero. Then any solution (t_0, u_0) to the initial polynomial equation (Equation (12.3)) satisfies $R(t_0, u_0) = 0 \mod 1$. This provides even more polynomial equations. In Coppersmith's method, one considers a finite subset of the possible $t^i u^j Q^k$'s with all their integer linear combinations.

By mapping a polynomial to the vector of its coordinates, we obtain a discrete additive subgroup of some Euclidean space $\mathbb{R}^N$ (for some integer N that is possibly large). Such algebraic objects are called *Euclidean lattices*, and have been studied extensively in mathematics, after the pioneering work of Minkowski [278] at the end of the nineteenth century. The main algorithmic task given a lattice consists in finding a short nonzero vector in the lattice. The LLL algorithm [255] performs such a task efficiently (to be more specific, in time polynomial in the bit size of the input description of the lattice). A brief presentation of LLL is given in Section 16.2, page 524. In our context, by calling the LLL algorithm, we can find a small bivariate polynomial R such that if (t_0, u_0) is a solution to the initial equation, then $R(t_0, u_0) = 0 \mod 1$. This smallness helps us to remove the modulus: indeed, if the coefficients of R are small, since we consider small ranges for the solutions, then $|R(t, u)|$ is itself small. By choosing all parameters carefully, we obtain that, if (t_0, u_0) is a solution to the initial equation, then:

$$R(t_0, u_0) = 0 \mod 1 \text{ and } |R(t_0, u_0)| < 1.$$

This implies that $R(t_0, u_0) = 0$, without the modulus. Note that we now only have one polynomial equation left. In fact, the LLL algorithm not only finds one small vector, but several small vectors, and therefore we can find two polynomial equations $R_1(t, u) = 0$ and $R_2(t, u) = 0$. The solutions to these equations are then computed using standard variable elimination methods (for example, using resultants), and one keeps those that are actually solutions to the initial equation

$$(t, u) = 0 \mod 1.$$

Several technical difficulties arise while trying to apply this general framework. First, Coppersmith's method is usually described with integer polynomials (and a modulus greater than 1). In our case, we have polynomials with real coefficients. In fact, we only have approximately known polynomials with real coefficients. The lattice description is only approximate.

The errors thus created can be analyzed rigorously and can be handled by using multi-precision approximations (see Chapter 14 for an introduction to multi-precision floating-point arithmetic). A second difficulty, which we have already mentioned, is the fact that the two eventually obtained polynomial equations over the reals (without the modulus) may not be algebraically independent. It is not known yet how to avoid that possible annoyance, and thus the SLZ method remains heuristic. Finally, one needs to "set the different parameters adequately" in order to get two sufficiently small polynomials after the call to LLL. The choice of the parameters derives from an analysis of the input given to LLL. This reduces to evaluating determinants of nonsquare matrices, see [388]. There, the determinant of an $n_1 \times n_2$ matrix B with $n_1 \geq n_2$ is defined as the square root of the determinant of the square matrix $B^T \cdot B$ of the pairwise scalar products of the columns of B.

Putting it all together

The SLZ algorithm consists in approximating the function f under scope by using degree-d polynomials on many small subintervals, and then in finding the bad rounding cases for each one of these polynomials using Coppersmith's method.

Taking $d = 1$ gives Lefèvre's algorithm, described in the previous section. If one increases d, then fewer subintervals are needed in the approximation step. Unfortunately, when d increases, the variable range restriction requirements of Coppersmith's method become stronger. In particular, this implies that when d increases, either the widths of the subintervals shall decrease or the parameter m (quantifying the quality of the bad rounding cases) shall increase. We have seen in Section 12.2.1 that one expects the worst rounding case of an elementary function over a given binade to satisfy $m \approx 2p$. Since we are interested in actually finding the exact hardness to round, we are limited to increasing m to essentially $2p$. This implies that when d increases greatly, we should decrease the widths of the subintervals of approximation to ensure that Coppersmith's method works, and thus increase the number of subintervals to be considered.

We see that there is a compromise to be found between increasing d to decrease the number of studied subintervals (because the quality of the approximations increases), and not increasing d too much because then Coppersmith's method requires thinner subintervals. According to the analysis of [387, 388], it seems that in the context of bad rounding cases, the best choice for d is 3, i.e., piecewise cubic approximations to the function f. This provides around $2^{p/2}$ subintervals to be considered, each requiring a small (polynomial in p) number of operations.

Although the complexity of the SLZ algorithm (around $2^{p/2}$) is better than that of Lefèvre's (around $2^{2p/3}$), the improvement is not dramatic for the double-precision (i.e., $p = 53$). This stems from the fact that each subinterval

can be dealt with (much) faster in Lefèvre's algorithm. It seems that the SLZ algorithm could overtake Lefèvre's for $p = 64$. Note that the complexity remains exponential, and thus increasing p further will quickly render the SLZ algorithm too costly.

An implementation of the SLZ algorithm is available under the GNU General Public License at the URL

```
http://perso.ens-lyon.fr/damien.stehle/english.html#software.
```

Extensions of the SLZ algorithm

The SLZ algorithm may be used with a larger value of m. As discussed above, this is useless for disclosing the worst rounding cases in a typical situation. However, it may be used to find exceptionally bad rounding cases (i.e., much worse than predicted by the statistical arguments of Section 12.2.1), or to prove that no such hard-to-round case exists, thus providing a sufficient precision bound for the correctly rounded evaluation of the function f under scope. We are not aware of any such application of the SLZ algorithm at the time of this writing, but it could be worth a further investigation for greater precisions p.

For example, in quadruple/binary128 precision ($p = 113$), both Lefèvre's and the SLZ algorithms (with $m = 2p$) become far too expensive, but knowing a provably sufficient evaluation precision may still be needed. The cost of the SLZ decreases when m increases, and even becomes polynomial in p when the quality parameter m is of the order of p^2.

In his Ph.D. dissertation, Stehlé [387] describes an extension of the SLZ algorithm to functions of several variables. To the best of our knowledge, this has not been used in practice so far. However, it could prove useful to find bad rounding cases (or sufficient evaluation precisions) for common functions like x^y.

12.4.7   Periodic functions on large arguments

A limitation of the various non-naive algorithms lies in the requirement that the studied function f can be closely approximated by small degree polynomials on subintervals. These subintervals should not be too small with respect to the width of the considered binade; otherwise, the number of such subintervals and thus the cost of the bad-case search explodes. This is the case for most elementary functions over most of their domains of definition. The only exceptions are periodic functions, such as the trigonometric functions ($\sin$, $\cos$, $\tan$), on arguments that are far away from the period. In the extreme case, two consecutive floating-point numbers in such a domain belong to different periods of the functions and their evaluations are completely uncorrelated.

When the period is a particularly simple value, such as a power of two, as in the functions $\sin(\pi x)$, $\cos(\pi x)$, and $\tan(\pi x)$, it is possible to deduce the worst cases corresponding to large arguments from those corresponding to small arguments (see Section 12.2.3). But for the conventional trigonometric functions ($\sin$, $\cos$, $\tan$), such deductions are not possible.

In [162], Hanrot, Lefèvre, Stehlé, and Zimmermann describe an algorithm that works around that difficulty. The main idea is to not consider the floating-point numbers themselves, but their reductions modulo the elementary period (2π or π in the case of the conventional trigonometric functions). Then the SLZ algorithm or Lefèvre's algorithm is applied to the studied function on this new range, for which small degree approximations are valid on larger subintervals. A difficulty arises from the fact that both Lefèvre's and the SLZ algorithms require the possible inputs to form an arithmetic progression (e.g., floating-point numbers), whereas here the floating-point numbers reduced modulo 2π may not form an arithmetic progression. By using continued fractions, the authors are able to subdivide the reduced floating-point numbers into several arithmetic progressions that are then considered separately. Overall, this algorithm is slower than SLZ, but it is faster than the only currently known alternative, i.e., the exhaustive search. The (heuristic) complexity bounds of this algorithm range from around $2^{4p/5}$ if it relies on Lefèvre's algorithm to around

$$2^{(6-2\sqrt{10})p} \le 2^{0.676p}$$

if it relies on the SLZ algorithm.

12.5 Some Results

12.5.1 Worst cases for the exponential, logarithmic, trigonometric, and hyperbolic functions

In this section, we give the worst cases for some functions in some domains. These worst cases have been found after several years of computations on small networks of workstations/PCs (several hundreds of thousands of hours, after summing the time on each CPU). These worst cases can be used to implement mathematical functions with correct rounding and to check libraries that claim correct rounding. An optimized implementation may need more bad cases (i.e., a smaller bound on k, with the notation of Section 12.2.1), possibly with different domain splitting. But we had to make a choice and could not use dozens of pages for each function. Some functions have special bad cases that do not follow the probabilistic model because of some mathematical property. If there are not too many of these bad cases, they are all given (e.g., sinh and cosh, in Table 12.13); otherwise the domain is split (e.g., exp, in Table 12.10).

Each bad case has the following format: the argument (binary64 number) and the truncated result (i.e., the result rounded toward zero in the binary64 format), both in C99's hexadecimal format (described below), and then the bits after the significand of the truncated result. The first bit after the significand is the rounding bit. Then, as we have bad cases, the rounding bit is followed by a long run of zeros or ones; the value in exponent (superscript) is the length of this run. Finally, we give the next four bits.

The argument and the truncated result are written in C99's hexadecimal format for concision. Here, each datum consists of:

- a significand, written as a 13-hexadecimal-digit fixed-point number (the first digit is a 1, hence the $1 + 4 \times 13 = 53$ bits of precision), the digits being denoted 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F;

- the letter P (as a separator);

- a binary exponent E, written in decimal, i.e., the significand is to be multiplied by 2^E to get the value.

Let us illustrate this format on an example: the bad case

$$\text{1.9E9CBBFD6080BP-31}$$

of function exp (see fifth bad case of Table 12.10), whose truncated result is 1.000000033D397P0 and the following bits are $1\,0^{57}\,1010\cdots$. The argument x is here in binary:

$$\overbrace{1.\underbrace{1001}_{9}\underbrace{1110}_{E}\underbrace{1001}_{9}\underbrace{1100}_{C}\underbrace{1011}_{B}\underbrace{1011}_{B}\underbrace{1111}_{F}\underbrace{1101}_{D}\underbrace{0110}_{6}\underbrace{0000}_{0}\underbrace{1000}_{8}\underbrace{0000}_{0}\underbrace{1011}_{B}}^{53\text{ bits}} \times 2^{-31}$$

and its exponential is:

$$\overbrace{1.\underbrace{0000}_{0}\underbrace{0000}_{0}\underbrace{0000}_{0}\underbrace{0000}_{0}\underbrace{0000}_{0}\underbrace{0000}_{0}\underbrace{0000}_{0}\underbrace{0011}_{3}\underbrace{0011}_{3}\underbrace{1101}_{D}\underbrace{0011}_{3}\underbrace{1001}_{9}\underbrace{0111}_{7}}^{53\text{ bits}}$$

$$1\underbrace{00000000000000000000\cdots 00000000000000000000}_{57\text{ zeros}}1010\cdots \times 2^0.$$

Note: For technical reasons, the bad cases have been searched assuming precision 53 for *any* exponent, i.e., as if there were no subnormals in the format. But the results given in Tables 12.10 through 12.16 include the subnormal range (down to 2^{-1074}), so that implementers have complete information.

Function	Domain	Argument	Truncated result	Trailing bits
exp	$[\log(2^{-1074}), -2^{-36}]$	-1.12D31A20FB38BP5	1.5B0BF3244820AP-50	$0\,1^{58}\,0010\cdots$
		-1.A2FEFEFD580DFP-13	1.FFE5D0BB7EABFP-1	$0\,0^{57}\,1100\cdots$
		-1.ED318EFB627EAP-27	1.FFFFF84B39C4P-1	$1\,1^{59}\,0001\cdots$
		-1.3475AC05CEAD7P-29	1.FFFFFECB8A54P-1	$0\,0^{57}\,1001\cdots$
	$[2^{-31}, \log(2^{1025}))$	1.9E9CBBFD6080BP-31	1.000000033D397P0	$1\,0^{57}\,1010\cdots$
		1.83D4BCDEBB3F4P2	1.AC50B409C8AEEP8	$0\,0^{57}\,1000\cdots$
exp(x) − 1	$(\log(2^{-1074}), -2^{-54}]$	-1.0000000000001P-51	1.FFFFFFFFFFFCP-1	$0\,0^{100}\,1010\cdots$
	$[2^{-53}, \log(2^{1025}))$	1.FFFFFFFFFFFP-53	1.0000000000000P0	$1\,1^{104}\,0101\cdots$
	$[2^{-35}, \log(2^{1024}))$	1.274BBF1EFB1A2P-10	1.2776572C25129P-10	$1\,0^{58}\,1000\cdots$
	$(-\infty, -2^{-34}]$	-1.19E53FCD490D0P-23	-1.19E53E96DFFA8P-23	$1\,0^{56}\,1110\cdots$
	$[2^{-51}, \log(2^{1024}))$	1.7FFFFFFFFFFDP-49	1.8000000000005P-49	$1\,1^{96}\,0110\cdots$
	$(-\infty, -2^{-51}]$	-1.800000000003P-49	-1.7FFFFFFFFFFAP-49	$0\,0^{96}\,1000\cdots$
2^x	$(-\infty, +\infty)$	1.12B14A318F904P-27	1.000000017CCE02P0	$0\,0^{58}\,1101\cdots$
		1.BFBBDE44EDFC5P-25	1.0000009B2C385P0	$0\,0^{59}\,1011\cdots$
		1.E4596526BF94DP-10	1.0053FC2EC2B53P0	$0\,1^{59}\,0100\cdots$
10^x	$(0, \log10(2^{1024}))$	1.DF760B2CDEED3P-49	1.0000000000022P0	$1\,0^{58}\,1110\cdots$
		1.A83B1CF779890P-26	1.000000F434FAAP0	$0\,1^{60}\,0101\cdots$
		1.7C3DDD23AC8CAP-10	1.00DB40291E4F5P0	$0\,1^{58}\,0010\cdots$
		1.AA6E0810A7C29P-2	1.4DEC173D50B3EP1	$0\,1^{58}\,0001\cdots$
		1.D7D271AB4EEB4P-2	1.71CE472EB84C7P1	$1\,1^{64}\,0111\cdots$
		1.75F49C6AD3BADP0	1.CE41D8FA665F9P4	$1\,1^{64}\,0101\cdots$
	$(\log10(2^{-1074}), 0)$	-1.1416C72A588A6P-1	1.27D838F22D09FP-2	$1\,1^{65}\,0010\cdots$
		-1.F28E0E25574A5P-32	1.FFFFFF70811EP-1	$0\,0^{59}\,1011\cdots$

Table 12.10: Worst cases for functions e^x, $e^x - 1$, 2^x, and 10^x. The worst cases given here and the results given in Tables 12.4 and 12.5 suffice to round functions e^x, 2^x and 10^x correctly in the full binary64/double-precision range (for function e^x the input values between -2^{-53} and 2^{-52} are so small that the results given in Tables 12.4 and 12.5 can be applied, so they are omitted here) [251]. Radix-β exponentials of numbers less than $\log_\beta(2^{-1074})$ are less than the smallest positive machine number. Radix-β exponentials of numbers larger than $\log_\beta(2^{1024})$ are overflows.

Function	Domain	Argument	Truncated result	Trailing bits
ln	$[2^{-1074}, 2^{-1})$	1.EA71D85CEE020P-509	-1.60296A66B42FFP8	1 1^{60} 0000$\cdots$
		1.9476E304CD7C7P-384	-1.09B60CAF47B35P8	1 0^{60} 1010$\cdots$
		1.26E9C4D327960P-232	-1.4155584BCD084P7	0 0^{60} 1001$\cdots$
		1.613955DC802F8P-35	-1.7F02F9BAF6035P4	0 1^{60} 0011$\cdots$
	$[2^{-1}, 2^1)$	1.BADED30CBF1C4P-1	-1.290EA09E36478P-3	1 1^{54} 0110$\cdots$
	$[2^1, 2^{1025})$	1.C90810D354618P245	1.54CD1FEA76639P7	1 1^{63} 0101$\cdots$
		1.62A88613629B6P678	1.D6479EBA7C971P8	0 0^{64} 1110$\cdots$
ln(1 + x)	$[2^{-35}, 2^{98}]$	1.AB50B409C8AEEP8	1.83D4BCDEBB3F3P2	1 1^{60} 0101$\cdots$
		1.8AA92BC84FF91P54	1.2EE70220FB1C4P5	1 1^{60} 0011$\cdots$
		1.0410C95B580B9P71	1.89D56A0C38E6FP5	0 0^{62} 1011$\cdots$
	$[2^{-35}, 2^{1024})$	1.C90810D354618P245	1.54CD1FEA76639P7	1 1^{63} 0101$\cdots$
		1.62A88613629B6P678	1.D6479EBA7C971P8	0 0^{64} 1110$\cdots$
	$(-1, -2^{-35})$	-1.7FFFF3FCFFD03P-30	-1.7FFFF4017FCFEP-30	1 0^{58} 1001$\cdots$
	$(2^{-51}, 2^{1024})$	1.80000000000003P-50	1.7FFFFFFFFFFFEP-50	1 0^{99} 1000$\cdots$
	$(-1, -2^{-51})$	-1.7FFFFFFFFFFFDP-50	-1.8000000000001P-50	0 1^{99} 0110$\cdots$

Table 12.11: Worst cases for functions $\ln(x)$ and $\ln(1+x)$. The worst cases given here suffice to round functions $\ln(x)$ and $\ln(1 + x)$ correctly in the full binary64/double-precision range.

table

Function	Domain	Argument	Truncated result	Trailing bits
$\log_2$	$[2^{-1}, 2^{1024})$	1.B4EBE40C95A01P0	1.8ADEAC981E00DP-1	$1\,0^{53}\,1011\cdots$
		1.1BA39FF28E3EAP2	1.12EECF76D63CDP1	$0\,0^{53}\,1001\cdots$
		1.1BA39FF28E3EAP4	1.097767BB6B1E6P2	$1\,0^{54}\,1001\cdots$
		1.61555F75885B4P128	1.00EE05A07A6E7P7	$1\,1^{53}\,0011\cdots$
		1.D30A43773DD1BP256	1.00DE0E189B724P8	$1\,0^{53}\,1100\cdots$
		1.61555F75885B4P256	1.007702D03D373P8	$1\,1^{54}\,0011\cdots$
		1.61555F75885B4P512	1.003B81681E9B9P9	$1\,1^{55}\,0011\cdots$
	$[2^{-1074}, 2^{-1})$	1.365116686B078P-765	-1.CC68A4AEE240DP7	$0\,1^{61}\,0110\cdots$
		1.83E55C0285C96P-762	-1.CA68A4AEE240DP7	$0\,1^{61}\,0110\cdots$
		1.A8639E89F5E46P-625	-1.77D933C1A88E0P7	$1\,1^{61}\,0101\cdots$
		1.ED8C87C3BF5CFP-49	-1.CEE46399392D6P3	$0\,1^{62}\,0000\cdots$
		1.27D838F22D0A0P-2	-1.1416C72A588A5P-1	$1\,1^{65}\,0101\cdots$
$\log_{10}$	$[2^{-1}, 2^{1})$	1.B0CF736F1AE1DP-1	-1.2AE5057CD8C44P-4	$0\,1^{54}\,0110\cdots$
		1.89825F74AA6B7P0	1.7E646F3FAB0D0P-3	$1\,0^{57}\,1001\cdots$
	$[2^{1}, 2^{1024})$	1.71CE472EB84C8P1	1.D7D271AB4EEB4P-2	$0\,0^{64}\,1010\cdots$
		1.CE41D8FA665FAP4	1.75F49C6AD3BADP0	$0\,0^{66}\,1010\cdots$
		1.E12D66744FF81P429	1.02D4F53729E44P7	$1\,0^{68}\,1001\cdots$

Table 12.12: Worst cases for functions $\log_2(x)$ and $\log_{10}(x)$. The worst cases given here suffice to round functions $\log_2(x)$ and $\log_{10}(x)$ correctly in the full binary64/double-precision range.

Function	Domain	Argument	Truncated result	Trailing bits
sinh	$[2^{-25}, \mathrm{asinh}(2^{1024}))$	1.DFFFFFFFFE3EP-20	1.E0000000000FD1P-20	$1\,1^{72}\,0001\cdots$
		1.DFFFFFFFF8F8P-19	1.E000000000003F47P-19	$1\,1^{66}\,0001\cdots$
		1.DFFFFFFFFE3E0P-18	1.E0000000000FD1FP-18	$1\,1^{60}\,0001\cdots$
		1.67FFFFFFD08AP-17	1.680000001AB25P-17	$1\,1^{57}\,0000\cdots$
		1.897374D74DE2AP-13	1.897374FE073E1P-13	$1\,0^{56}\,1011\cdots$
cosh	$[2^{-25}, 2^{6})$	1.465655F122FF5P-24	1.0000000000000CP0	$1\,1^{61}\,0001\cdots$
		1.7FFFFFFFFFF7P-23	1.00000000000047P0	$1\,1^{89}\,0010\cdots$
		1.7FFFFFFFFFFDCP-22	1.00000000000011FP0	$1\,1^{83}\,0010\cdots$
		1.7FFFFFFFFF70P-21	1.00000000000047FP0	$1\,1^{77}\,0010\cdots$
		1.7FFFFFFFFDC0P-20	1.00000000000011FFP0	$1\,1^{71}\,0010\cdots$
		1.1FFFFFFFF0DP-20	1.0000000000A1FP0	$1\,1^{73}\,0110\cdots$
		1.DFFFFFFFFB9BP-20	1.0000000001C1FP0	$1\,1^{69}\,0010\cdots$
		1.1FFFFFFFFC34P-19	1.0000000000287FP0	$1\,1^{67}\,0110\cdots$
		1.7FFFFFFF700P-19	1.00000000047FFP0	$1\,1^{65}\,0010\cdots$
		1.DFFFFFFFEE6CP-19	1.00000000000707FP0	$1\,1^{63}\,0010\cdots$
		1.1FFFFFFFF0D0P-18	1.0000000000A1FFP0	$1\,1^{61}\,0110\cdots$
		1.4FFFFFFFE7E2P-18	1.000000000DC7FP0	$1\,1^{60}\,0011\cdots$
		1.7FFFFFFFDC00P-18	1.00000000011FFFFP0	$1\,1^{59}\,0010\cdots$
		1.AFFFFFFFFCCBEP-18	1.0000000016C7FP0	$1\,1^{58}\,0010\cdots$
		1.DFFFFFFFB9B0P-18	1.0000000001C1FFP0	$1\,1^{57}\,0010\cdots$
		1.EA5F2F2E4B0C5P1	1.710DB0CD0FED5P4	$1\,0^{57}\,1110\cdots$

Table 12.13: Worst cases for functions $\sinh(x)$ and $\cosh(x)$. The worst cases given here suffice to round these functions correctly in the full binary64/double-precision range. If x is small enough, the results given in Tables 12.4 and 12.5 can be applied. If x is large enough, the results given in Section 12.2.3 allow one to use the results obtained for the exponential function.

Function	Domain	Argument	Truncated result	Trailing bits
asinh	$[2^{-25}, 2^{1024})$	1.E000000000000FD2P-20	1.DFFFFFFFFFE3EP-20	$0\,0^{72}\,1110\cdots$
		1.E000000000003F48P-19	1.DFFFFFFFF8F8P-19	$0\,0^{66}\,1110\cdots$
		1.C90810D354618P244	1.54CD1FEA76639P7	$1\,1^{63}\,0101\cdots$
		1.8670DE0B68CADP655	1.C7206C1B753E4P8	$0\,0^{62}\,1111\cdots$
		1.62A88613629B6P677	1.D6479EBA7C971P8	$0\,0^{64}\,1110\cdots$
acosh	$[1, 2^{91}]$	1.297DE35D02E90P13	1.3B562D2651A5DP3	$0\,1^{61}\,0001\cdots$
		1.91EC4412C344FP85	1.E07E71BFCF06EP5	$1\,1^{61}\,0101\cdots$
	$[1, 2^{1024})$	1.C90810D354618P244	1.54CD1FEA76639P7	$1\,1^{63}\,0101\cdots$
		1.62A88613629B6P677	1.D6479EBA7C971P8	$0\,0^{64}\,1110\cdots$

Table 12.14: Worst cases for inverse hyperbolic functions in binary64/double precision. Concerning function $\sinh^{-1}$, if the input values are small enough, there is no need to compute the worst cases: the results given in Tables 12.4 and 12.5 can be applied.

Function	Domain	Argument	Truncated result	Trailing bits
sin	$[2^{-25}, u)$	1.E00000000001C2P-20	1.DFFFFFFFF02EP-20	$0\,0^{72}\,1110\cdots$
		1.E0000000000708P-19	1.DFFFFFFFFC0B8P-19	$0\,0^{66}\,1110\cdots$
		1.E000000001C20P-18	1.DFFFFFFF02E0P-18	$0\,0^{60}\,1110\cdots$
		1.598BAE9E632F6P-7	1.598A0AEA48996P-7	$0\,1^{59}\,0000\cdots$
		1.FE767739D0F6DP-2	1.E9950730C4695P-2	$1\,1^{65}\,0000\cdots$
cos	$[2^{-17}, \mathrm{acos}(2^{-26}))$ $\cup \left[\mathrm{acos}(-2^{-27}), 2^2\right)$	1.06B505550E6B2P-9	1.FFFFBC9A3FBFEP-1	$0\,0^{58}\,1100\cdots$
		1.34EC2F9FC9C00P1	-1.7E2A5C30E1D6DP-1	$0\,1^{58}\,0110\cdots$
	$[0, \mathrm{acos}(2^{-26}))$ $\cup \left[\mathrm{acos}(-2^{-27}), 2^2\right)$	1.80000000000009P-23	1.FFFFFFFFFF70P-1	$0\,0^{88}\,1101\cdots$
tan	$[2^{-25}, \pi/2]$	1.DFFFFFFFFF1FP-22	1.E0000000000151P-22	$0\,1^{78}\,0100\cdots$
		1.DFFFFFFFFFC7CP-21	1.E00000000000545P-21	$1\,1^{72}\,0100\cdots$
		1.DFFFFFFFFF1F0P-20	1.E0000000001517P-20	$1\,1^{66}\,0100\cdots$
		1.67FFFFFFE845P-19	1.6800000002398P-19	$0\,1^{63}\,0100\cdots$
		1.DFFFFFFFC7C0P-19	1.E0000000000545FP-19	$1\,1^{60}\,0100\cdots$
		1.67FFFFFFA114P-18	1.680000008E61P-18	$1\,1^{57}\,0100\cdots$
		1.50486B2F87014P-5	1.5078CEBFF9C72P-5	$1\,0^{57}\,1001\cdots$

Table 12.15: Worst cases for the trigonometric functions in binary64/double precision. So far, we only have worst cases in the following domains: $\left[2^{-25}, u\right)$ where $u = 1.1001001000011_2 \times 2^1$ for the sine function ($u = 3.141357421875_{10}$ is slightly less than π); $\left[0, \mathrm{acos}(2^{-26})\right) \cup \left[\mathrm{acos}(-2^{-27}), 2^2\right)$ for the cosine function; and $\left[2^{-25}, \pi/2\right]$ for the tangent function. Sines of numbers of absolute value less than 2^{-25} are easily handled using the results given in Tables 12.4 and 12.5.

Function	Domain	Argument	Truncated result	Trailing bits
asin	$[2^{-25}, 1]$	1.DFFFFFFFF02EP-20	1.E000000000001C1P-20	$1\,1^{72}\,0001\cdots$
		1.DFFFFFFFC0B8P-19	1.E0000000000707P-19	$1\,1^{66}\,0001\cdots$
		1.DFFFFFFF02E0P-18	1.E0000000001C1FP-18	$1\,1^{60}\,0001\cdots$
		1.67FFFFFE54DAP-17	1.6800000002F75P-17	$1\,1^{57}\,0000\cdots$
		1.C373FF4AAD79BP-14	1.C373FF594D65AP-14	$1\,0^{57}\,1010\cdots$
		1.E9950730C4696P-2	1.FE76739D0F6DP-2	$0\,0^{64}\,1000\cdots$
acos	$[2^{-26}, 1]$	1.7283D529A146EP-19	1.921F86F3C82C5P0	$0\,0^{58}\,1000\cdots$
		1.FD737BE914578P-11	1.91E006D41D8D8P0	$1\,1^{62}\,0010\cdots$
		1.1CDCD1EA2AD3BP-9	1.919146D3F492EP0	$1\,1^{57}\,0010\cdots$
		1.60CB9769D9218P-8	1.90BEE93D2D09CP0	$0\,0^{57}\,1011\cdots$
		1.53EA6C7255E88P-4	1.7CDACB6BBE707P0	$0\,1^{57}\,0101\cdots$
atan	$[-1, -2^{-27}]$	-1.52F06359672CDP-2	1.E87CCC94BA418P0	$1\,0^{56}\,1101\cdots$
		-1.124411A0EC32EP-5	1.9AB23ECDD436AP0	$1\,0^{56}\,1101\cdots$
	$(2^{-25}, +\infty)$	1.E000000000546P-21	1.DFFFFFFFFC7CP-21	$0\,0^{72}\,1011\cdots$
		1.E000000001518P-20	1.DFFFFFFFF1F0P-20	$0\,0^{66}\,1011\cdots$
		1.E000000005460P-19	1.DFFFFFFFC7C0P-19	$0\,0^{60}\,1011\cdots$
		1.680000008E62P-18	1.67FFFFFA114P-18	$0\,0^{57}\,1011\cdots$
		1.22E8D75E2BC7FP-11	1.22E8D5694AD2BP-11	$1\,0^{59}\,1101\cdots$
		1.6298B5896ED3CP1	1.3970E827504C6P0	$1\,0^{63}\,1101\cdots$
		1.C721FD48F4418P19	1.921FA3447AF55P0	$1\,0^{58}\,1011\cdots$
		1.EB19A7B5C3292P29	1.921FB540173D6P0	$1\,1^{59}\,0011\cdots$
		1.CCDA26AD0CD1CP47	1.921FB54442D06P0	$0\,1^{57}\,0111\cdots$

Table 12.16: Worst cases for the inverse trigonometric functions in binary64/double precision. Concerning the arcsine function, the results given in Tables 12.4 and 12.5 and in this table make it possible to correctly round the function in its whole domain of definition.

12.5.2 A special case: integer powers

Concerning the particular case of function x^n (where n is an integer), one has $(2x)^n = 2^n x^n$. Therefore, if two numbers x and y have the same significand, their images x^n and y^n also have the same significand. So only one binade needs to be tested,[9] $[1, 2)$ in practice.

For instance, in double-precision arithmetic, the hardest-to-round case for the function x^{952} corresponds to

$$x = 1.0101110001101001001000000010110101000110100000100001$$

and we have

$$x^{952} = \underbrace{1.0011101110011001001111100000100010101010110100100110}_{53 \text{ bits}} 1$$
$$\underbrace{00000000 \cdots 00000000}_{63 \text{ zeros}} 1001 \cdots \times 2^{423}$$

which means that x^n is extremely close to the exact middle of two consecutive double-precision numbers. There is a run of 63 consecutive zeros after the rounding bit. This case is the worst case for all values of n between 3 and 1035.

Table 12.17 gives the longest runs k of identical bits after the rounding bit (assuming the target precision is double precision) in the worst cases for $3 \leq n \leq 1035$.

These results allow one to design algorithms for computing correctly-rounded integer powers, for values of n small enough (see Section 5.7, page 177).

12.6 Current Limits and Perspectives

We have described several ways to find the hardness to round of elementary functions or bounds thereof. The first approach, related to Liouville's theorem, is efficient but restricted to algebraic functions. Furthermore, the tightness of the bounds it provides degrades quickly with the degree of the considered algebraic function. In contrast, Lefèvre's and the SLZ algorithms apply to most functions. However, although they are significantly faster than the exhaustive search, their running times remain exponential in the considered precision p.

Currently, the worst cases of most elementary functions in the (radix-2) double precision/binary64 format (i.e., $p = 53$) have been found, using Lefèvre's algorithm. This status seems reachable in Intel's double-extended precision format ($p = 64$ for radix-2 floating-point numbers) within

---

[9]We did not take subnormal numbers into account, but one can prove that the worst cases in all rounding modes can also be used to round subnormals correctly.

a few years, using either Lefèvre's algorithm or the SLZ algorithm. Due to their exponential costs, the quadruple precision will remain out of reach for many years with these methods.

The methods presented here can readily be adapted to decimal arithmetic [253] (indeed, the worst cases for the exponential function in decimal64 arithmetic have been computed). Another interesting application of these methods that find worst cases is that they may be used to implement an "accurate tables method" due to Gal [142, 143, 391].

n	k
6, 12, 13, 21, 58, 59, 61, 66, 70, 102, 107, 112, 114, 137, 138, 145, 151, 153, 169, 176, 177, 194, 198, 204, 228, 243, 244, 249, 250, 261, 268, 275, 280, 281, 285, 297, 313, 320, 331, 333, 340, 341, 344, 350, 361, 368, 386, 387, 395, 401, 405, 409, 415, 418, 419, 421, 425, 426, 427, 442, 449, 453, 454, 466, 472, 473, 478, 480, 488, 493, 499, 502, 506, 509, 517, 520, 523, 526, 532, 533, 542, 545, 555, 561, 562, 571, 574, 588, 590, 604, 608, 614, 621, 626, 632, 634, 639, 644, 653, 658, 659, 664, 677, 689, 701, 708, 712, 714, 717, 719, 738, 741, 756, 774, 778, 786, 794, 797, 807, 830, 838, 842, 847, 849, 858, 871, 885, 908, 909, 910, 919, 925, 927, 928, 931, 936, 954, 961, 964, 970, 971, 972, 980, 984, 988, 989, 993, 1006, 1008, 1014, 1024	53
4, 18, 44, 49, 50, 97, 100, 101, 103, 142, 167, 178, 187, 191, 203, 226, 230, 231, 236, 273, 282, 284, 287, 304, 310, 311, 312, 328, 338, 355, 374, 388, 389, 391, 393, 394, 400, 422, 428, 434, 435, 439, 444, 455, 469, 501, 504, 511, 529, 535, 536, 549, 558, 559, 560, 566, 573, 577, 578, 581, 587, 596, 606, 612, 623, 628, 635, 643, 649, 656, 675, 691, 699, 700, 711, 713, 715, 718, 731, 732, 743, 744, 773, 775, 790, 799, 804, 808, 810, 821, 826, 841, 848, 863, 883, 889, 902, 907, 914, 917, 932, 942, 943, 949, 951, 960, 966, 968, 998, 1015, 1023	54
24, 28, 30, 41, 56, 67, 87, 122, 135, 143, 147, 159, 160, 190, 208, 248, 252, 264, 269, 270, 279, 289, 300, 315, 339, 376, 396, 402, 410, 460, 479, 497, 515, 516, 521, 539, 579, 599, 602, 617, 674, 685, 693, 723, 729, 758, 767, 770, 780, 802, 834, 835, 843, 853, 866, 884, 912, 921, 935, 973, 976, 979, 1000, 1004	55
89, 106, 171, 247, 254, 278, 316, 327, 348, 360, 424, 451, 463, 476, 495, 512, 531, 645, 697, 722, 728, 747, 833, 857, 901, 903, 1029, 1033	56
11, 84, 91, 234, 237, 274, 407, 576, 695, 751, 819, 828, 872, 879, 1016	57
35, 144, 233, 337, 733	58
51, 336, 761, 886	59
503, 1017, 1028	60
458	61
878, 1030	62
952	63

Table 12.17: Longest runs k of identical bits after the rounding bit (assuming the target precision is double precision/binary64) in the worst cases of function x^n, for $3 \leq n \leq 1035$. Values of n that are not in the table correspond to $k < 53$.

Part V

Extensions

Chapter 13

Formalisms for Certifying Floating-Point Algorithms

W HILE THE PREVIOUS CHAPTERS have made clear that it is common practice to certify floating-point algorithms with pen-and-paper proofs, this practice can lead to subtle bugs. Indeed, floating-point arithmetic introduces numerous special cases, and examining all the details would be tedious. As a consequence, the certification process tends to focus on the main parts of the correctness proof, so that it does not grow out of reach.

For instance, the proof and even the algorithm may no longer be correct when some value is equal to or near a power of the radix, as being a discontinuity point of the ulp function. Moreover pen-and-paper proofs may be ambiguous, e.g., by being unclear on whether the exact value or its approximation is considered for the ulp.

Unfortunately, experience has shown that simulation and testing may not be able to catch the corner cases this process has ignored. By providing a stricter framework, formal methods provide a means for ensuring that algorithms always follow their specifications.

13.1 Formalizing Floating-Point Arithmetic

In order to perform an in-depth proof of the correctness of an algorithm, its specification must be precisely described and formalized. For floating-point algorithms, this formalization has to encompass the arithmetic: number formats, operators, exceptional behaviors, undefined behaviors, and so on. A new formalization may be needed for any variation in the floating-point environment.

Fortunately, the IEEE 754 standard precisely defines some formats and how the arithmetic functions behave on these formats: "Each operation shall be performed as if it first produced an intermediate result correct to infinite precision and with unbounded range, and then rounded that result [...]"

This definition makes a formalization of floating-point arithmetic both feasible and practical, as long as the implementation (language, compiler, etc.) strictly follows the IEEE-754 requirements, which may not be the case in practice (see Chapter 7 and Section 3.4.6). However, a formalization can still take into account the specificity of the implementation; for the sake of simplicity and because it is not possible to be exhaustive, such a specificity will be ignored in the following.

Moreover, this single formalization can be used for describing the specification of any algorithm whose implementation relies on this standardized arithmetic.

13.1.1   Defining floating-point numbers

The first stage of a formalization lies in a proper definition of the set of floating-point numbers. This definition can be performed at several levels. First of all, one should define the set itself and the values that parameterize it, e.g., radix and precision. Then comes the actual representation of the numbers: how they translate from and to streams of bits. Finally, a semantics of the numbers is needed. It is generally provided by their interpretation as a subset of the real numbers.

Structural definition

The IEEE 754 standard describes five categories of floating-point data: signed zeros, subnormal numbers, normal numbers, signed infinities, and Not a Number (NaN) data. These categories can be used to define the set of data as a disjoint union of sets of data. Any given floating-point datum is described by one and only one of the following branches. For each category, some additional pieces of information represent the datum, e.g., its significand.

> Floating-point data ::=
> | **Zero**:   sign
> | **Subnormal**:   sign, significand
> | **Normal**:   sign, significand, exponent
> | **Infinity**:   sign
> | **NaN**:   payload

While the same disjoint union could be used to define both binary and decimal floating-point numbers, the formalization may be simpler if the radix β is encoded as a parameter of the whole type. The type of the "significand" fields has to be parameterized by the precision p, while the type of the "exponent" is parameterized by $e_{\min}$ and $e_{\max}$. The type of the "payload" could also be parameterized by p; but for clarity, we will assume it is not. The type of "sign" is simply the set $\{+, -\}$, possibly encoded by a Boolean. The fully featured disjoint union has now become:

Floating-point data $(\beta, p, e_{\min}, e_{\max}) ::=$
  | **Zero**:   sign
  | **Subnormal**:   sign, significand $\in M_s(\beta, p)$
  | **Normal**:   sign, significand $\in M_n(\beta, p)$, exponent $\in E(e_{\min}, e_{\max})$
  | **Infinity**:   sign
  | **NaN**:   payload

Notice that the parameters do not necessarily have to be restricted to the set of values mandated by the standard. A generic formalization can be written for any radix, any precision, and any extremal exponents. A specific standard-compliant instantiation of the generic formalization can then be used when certifying an algorithm.

Some other parameters could be added to represent floating-point numbers that do not follow the scheme set by the IEEE 754 standard, e.g., the use of two's complement significands. The "Subnormal" branch could also be removed for implementations that do not support such numbers.

Binary representation

This part of the formalization describes how the numbers are actually stored. It mainly provides two functions for converting the structural definition of a number from/to its physical representation as a bit vector. These functions are indispensable when working at the bit level (What does happen if the 35th bit of a number gets set?), for example, in hardware or software implementation of floating-point operators.

Note that the floating-point operations are better specified as functions operating on the structural definition of numbers, which is a more expressive representation than bit vectors. As a consequence, the conversion functions can be ignored if the algorithm does not perform any bit twiddling. Indeed, the algorithm can then be certified on the structural definitions only. Such a certification is equivalent to the one made on bit vectors, but it allows for the use of a simplified formalism, as shown in Section 13.1.2.

Semantic interpretation

Floating-point arithmetic is designed as a replacement for the arithmetic on real numbers in most algorithms. The specification then has to relate the floating-point results with the corresponding real numbers. This can be achieved by providing functions between the two sets. Unfortunately, neither are they isomorphic, nor is one a subset of the other.

First, normal numbers, subnormal numbers, and signed zeros can easily be converted into real numbers.[1] When the specification focuses on the behavior of the algorithm with respect to real numbers, this conversion

---

[1]Both floating-point signed zeros are mapped to the same real zero, so the conversion function is not injective.

function r is usually sufficient, and its inverse is not needed. For example, expressing that a nonzero real number x has to be close to a floating-point number $\tilde{x}$ can be achieved by the following inequality on real numbers: $|r(\tilde{x})/x - 1| \leq \epsilon$.

As for floating-point infinities and NaNs, the set of real numbers could be extended to include corresponding elements. The r function would then be defined on the whole set of floating-point data. In order for this extended real set to be usable, a coherent arithmetic has to be defined on it. This may prove difficult and, more importantly, unneeded. Indeed, as these exceptional values are usually avoided or handled with special care in algorithms, they do not warrant a special arithmetic. As a consequence, the conversion function can be restricted to a partial domain of floating-point data. When employed in a certification, this formalism will then require the use of this function to be accompanied with a proof that the floating-point datum being converted is a finite number.

This conversion function is usually hidden: finite floating-point numbers are implicitly coerced to real numbers in mathematical formulas.

13.1.2   Simplifying the definition

The more complicated the arithmetic formalization is, the less efficient its use will be when certifying an algorithm, as it requires taking into account many more corner cases. So, a smaller structural definition and simpler rounding operators may ease the certification process. However, care should be taken that the formalization is still meaningful for the algorithm being certified.

First of all, some branches of the disjoint union can be merged: instead of splitting zeros, subnormal, and normal numbers apart, they can all be expressed as a triple (s, m, e) which maps to the real number $(-1)^s \cdot m \cdot \beta^{e-p+1}$.[2] The exponent e is still between $e_{\min}$ and $e_{\max}$, but a normal floating-point number may no longer have a normalized representation since the branch of the disjoint union should also deal with other numbers. In particular, all the numbers with $m = 0$ would now represent a floating-point zero.

Depending on its usage, the formalization can be simplified further. For most algorithms, the actual bit pattern of the computed values does not matter much. Indeed, higher-level properties are usually expected: the accuracy of a result, its being in a given domain, and so on. Therefore, the formalization can be simplified, as long as it does not change the truth of the properties described in the specification of the algorithm. For instance, if the algorithm never accesses the payload of a NaN, then this payload can be removed from the definition. Indeed, from the point of view of the algorithm, NaN data will be indistinguishable from each other.

---

[2]Using an integral significand (Section 2.1) can greatly improve the ability of proof assistants to automatically discharge some proof obligations, hence reducing the amount of work left to the user.

The following formalization may therefore be sufficient to prove the exact same properties as the full formalization on a given algorithm:

Floating-point numbers $(\beta, p, e_{\min}, e_{\max})$ ::=
| **Finite**:   sign, significand $\in M(\beta, p)$, exponent $\in E(e_{\min}, e_{\max})$
| **Infinity**:   sign
| **NaN**:

Let us now consider the case of signed zeros. Their sign mostly matters when a floating-point computations has to return a signed zero. So, if the sign of zero were to be always ignored, it would not have any impact on the interpretation of these computations. However, there are some other cases where the sign has an influence, e.g., a division by zero. In this case, the sign of zero is involved in the choice of the sign of the nonzero (infinite) result. Yet, if the certification is meant to include proofs that no divisions by zero occur during computations, then the sign of zero can be safely discarded from the definition. In particular, it means that the sign can be embedded into the significand: any zero, subnormal, or normal number would therefore be represented by a pair (m, e) with m a signed value.

In order to simplify the definition even further, the exponent bounds $e_{\min}$ and $e_{\max}$ could be removed. It would, however, introduce values that cannot be represented as floating-point datums. Section 13.1.4 details this approach.

13.1.3 Defining rounding operators

Thanks to the framework that the IEEE 754 standard provides, floating-point operators do not have to be formalized to great lengths. They can be described as the composition of a mathematical operation on real numbers ("infinite precision" and "unbounded range") and a rounding operator that converts the result to a floating-point number. As a consequence, assuming arithmetic on real numbers has already been properly formalized, most of the work involved in defining floating-point arithmetic operators will actually focus on defining rounding operators on real numbers.

Range and precision

In our preceding structural definition, the exponent range $[e_{\min}, e_{\max}]$ and the precision p are part of the datum type: they restrict the ranges of the available significands and exponents. As a consequence and by definition, a finite floating-point number is bounded. Moreover, a formal object representing a floating-point number cannot be created unless one proves that its significand and exponent, if any, are properly bounded.

Another approach would be to remove these bounds from the datum type and use them to parameterize the rounding operators only. Then a finite floating-point number would only be a multiple of a power of the radix β,

and hence unbounded *a priori*. The property that it is bounded would instead come from the fact that such a number was obtained by a rounding operation.

Relational and functional definitions

There are two approaches to defining these operators. The first one considers rounding operators as relations between the set of floating-point numbers and the set of real numbers. Two such numbers are related if the first one is the rounded value of the second one.

In addition to standard rounding modes, this approach makes it possible to define nondeterministic rounding modes. For example, one could imagine that, when rounding to nearest, the tie breaking is performed in a random direction, so several floating-point values could be obtained when rounding a real number. Such a property can be expressed with a relation. However, nondeterministic rounding modes are rarely used in practice.

More interestingly, the relational approach can deal with underspecified rounding operators. This allows us to certify more generic properties about an algorithm, without having to change the way the algorithm is described. For instance, an algorithm may compute the correct value, even if some intermediate results are only faithfully rounded. Or some languages may allow a multiplication followed by an addition to be contracted to an FMA on some processors (e.g., see Section 7.2.3), and algorithms may still compute the correct value if this happens.

The other approach applies to deterministic rounding modes only. It describes them by a function from the set of real numbers to the set of floating-point numbers. This approach is especially interesting when the image of the function is actually the set of finite floating-point numbers.[3]

This functional definition allows us to manipulate floating-point expressions as if they were expressions on real numbers, as long as none of the floating-point operations of the expressions invoke an exceptional behavior.

Monotonicity

Powerful theorems can be applied on rounded computations as long as the rounding operators satisfy the following two properties. First, any floating-point number is its own and only rounded value. Second, if $\tilde{u}$ and $\tilde{v}$ are finite rounded values of the real numbers u and v, the ordering $u \leq v$ implies $\tilde{u} \leq \tilde{v}$. As a consequence, the functional version of a rounding operator is useful if it is the identity function on the set of floating-point numbers and a locally constant yet increasing function on the whole set of real numbers—or at least on the subset that rounds to finite floating-point

---

[3]This property arises naturally when the formalism has been extended by removing the upper bound $e_{\max}$. See Section 13.1.4.

numbers. As an example of using these two properties, let us consider the proof of the following simple lemma.

Lemma 42 (Midpoint of two floating-point numbers in radix-2 arithmetic). *Consider the formalism of a binary floating-point arithmetic with no upper bound* $e_{\max}$ *on the exponents of rounded numbers. When evaluated with any useful rounding operator* $\circ$, *the expression* `(a + b) / 2` *returns a floating-point value no less than* $\min(a, b)$ *and no greater than* $\max(a, b)$, *assuming a and b are finite floating-point numbers.*

Proof. We start from the mathematical inequality

$$2 \cdot \min(a, b) \leq a + b \leq 2 \cdot \max(a, b).$$

The rounding operator is an increasing function, so we get

$$\circ(2 \cdot \min(a, b)) \leq \circ(a + b) \leq \circ(2 \cdot \max(a, b)).$$

Since the floating-point radix is 2, the values $2 \cdot \min(a, b)$ and $2 \cdot \max(a, b)$ are representable floating-point numbers. As a consequence,

$$2 \cdot \min(a, b) \leq \circ(a + b) \leq 2 \cdot \max(a, b).$$

Halving the terms and using once again the monotonicity of the rounding operator, we get

$$\circ(\min(a, b)) \leq \circ\left(\frac{\circ(a + b)}{2}\right) \leq \circ(\max(a, b)).$$

As the rounding operator is the identity on $\min(a, b)$ and $\max(a, b)$, we obtain the final inequality:

$$\min(a, b) \leq \circ\left(\frac{\circ(a + b)}{2}\right) \leq \max(a, b). \qquad \square$$

Note that the identity and monotonicity properties of the rounding operator are independent of the constraint $e_{\min}$. As a consequence, the lemma holds, even when subnormal numbers are computed. The correctness of the proof, however, depends on the numbers $2 \cdot \min(a, b)$ and $2 \cdot \max(a, b)$ being in the range of finite floating-point numbers. Otherwise, the rounding operation would no longer be the identity for these two numbers, hence invalidating the proof. Therefore, a hypothesis was added in order to remove the upper bound $e_{\max}$. This means that the lemma no longer applies to a practical floating-point arithmetic. Section 13.1.4 will consider ways to use such a formalization so that it can be applied to a bounded $e_{\max}$.

13.1.4   Extending the set of numbers

As mentioned before, a formalization is sufficient for certifying an algorithm, if it does not change the truth value of the properties one wants to prove on this algorithm. Therefore, if one can prove that no infinities or NaNs can be produced by the algorithm,[4] a formalization without infinities and NaNs is sufficient.

Moreover, the simplified formalization does not need an $e_{\max}$ bound. Indeed, in order to prove that this formalization was sufficient, one has already proved that all the floating-point numbers produced while executing the algorithm are finite. Therefore, embedding $e_{\max}$ in the formalization does not bring any additional confidence in the correctness of the algorithm.

Such a simplification eases the certification process, but it also requires some extra discipline. Formal tools will no longer systematically require proofs of the exponents not overflowing. Therefore, the user has to express these properties explicitly in the specification, in order to prove that the simplified formalization is valid, and hence to obtain a correct certificate.

Let us consider Lemma 42 anew. This lemma assumes there is no $e_{\max}$, so that a value like $2 \cdot \max(a, b)$ can be rounded to a finite number. Notice that this value is not even computed by the algorithm, so whether the product overflows or not does not matter. Only the computation of (a + b) / 2 is important. In other words, if one can prove that no overflows can occur in the algorithm, then the formalization used in Lemma 42 is a valid simplification for the problem at hand. Therefore, an immediate corollary of Lemma 42 states: for a full-fledged binary floating-point arithmetic, (a + b) / 2 is between a and b if the sum $a + b$ does not overflow.

Similarly, if subnormal numbers are proved not to occur, the lower bound $e_{\min}$ can be removed from the description of the floating-point datum and finite floating-point numbers will be left with zero and normal numbers only. These modifications are extensions of the floating-point model, as exponent bounds have been removed. The main consequence is that some theorems are then valid on the whole domain of floating-point finite numbers. For instance, one can assume that $|\mathrm{RN}(x) - x|/x$ is bounded by

$$(n - 1) \times \beta^{1-p}/2$$

for any (nonzero) real x, without having to first prove that x is in the normal range. In particular, x does not have to be a value computed by the studied program; it can be a ghost value that appears in the proof only, in order to simplify it.

---

[4]When arithmetic operations produce a trap in case of exception, the proof is straightforward; but the trap handling must also be proved (the absence of correct trap handling was the cause of the explosion of Ariane 5 in 1996). Another way is to prove that all the infinitely precise values are outside the domain of exceptional behaviors. The proof is more complicated in the latter case.

13.2 Formalisms for Certifying Algorithms by Hand

Once there is a mathematical setting for floating-point arithmetic, one can start to formally certify algorithms that depend on it. Among the early attempts, one can cite Holm's work that combined a floating-point formalism with Hoare's logic in order to check numerical algorithms [184]. Barrett later used the Z notation to specify the IEEE 754 standard and refine it to a hardware implementation [23]. Priest's work is an example of a generic formalism for designing guaranteed floating-point algorithms [336]. All of these works were based on detailed pen-and-paper mathematical proofs.

Nowadays, computer-aided proof systems make it possible, not only to state formalisms on floating-point arithmetic, but also to mechanically check the proofs built on these formalisms. This considerably increases the confidence in the correctness of floating-point hardware or software. The following paragraphs present a survey of some of these formalisms. The choice of a given formalism for certifying an algorithm should depend on the scope of the algorithm (low-level or high-level?) and on the features of the corresponding proof assistant (proof automation, support for real numbers, support for programming constructs, and so on).

13.2.1 Hardware units

Processor designers have been early users of formal certification. If a bug goes unnoticed at design time, it may incur a tremendous cost later, as it may cause its maker to recall and replace the faulty processor. Unfortunately, extensive testing is not practical for floating-point units, as their highly combinatorial nature makes it time consuming, especially at pre-silicon stages: hence the need for formal methods, in order to extend the coverage of these units.

At this level, floating-point arithmetic does not play an important role. Certified theorems mostly state that "given these arrays of bits representing the inputs, and assuming that these logical formulas describe the behavior of the unit, the computed output is the correct result." Floating-point arithmetic is only meaningful here for defining what a "correct result" is, but none of its properties will usually appear in the formal proof.

As an example of such formal certifications, one can cite the various works around the floating-point units embedded in AMD-K5 processors. Moore, Lynch, and Kaufmann were interested in the correctness of the division algorithm [283], while Russinoff tackled the arithmetic operators at the RTL level [355, 357] and the square root at the microcode level [356]. All these proofs are based on a formalism written for the ACL2 first-order proof assistant.[5]

One specificity of this formalism is the use of rational numbers only. Since inputs, outputs, and ideal results are all rational numbers, this

---

[5]http://www.cs.utexas.edu/users/moore/acl2/

restriction does not hinder the certification process of addition, multiplication, and division. But it is an issue for the square root, as the ideal results often happen to be irrational numbers. It implies that the correctness theorem for the square root cannot be stated as follows:

$$\forall x \in \text{Float}, \quad \texttt{sqrt}(x) = \circ(\sqrt{x}).$$

It has to be slightly modified so that the square root can be avoided:

$$\forall x \in \text{Float}, \ \forall a, b \in \text{Rational}, \quad 0 \leq a \leq b \quad \Longrightarrow$$
$$a^2 \leq x \leq b^2 \quad \Longrightarrow \quad \circ(a) \leq \texttt{sqrt}(x) \leq \circ(b).$$

From this formally certified theorem, one can then conclude with a short pen-and-paper proof that the floating-point square root is correctly computed. First of all, rounding operators are monotonic, so $a \leq \sqrt{x} \leq b$ implies $\circ(a) \leq \circ(\sqrt{x}) \leq \circ(b)$. If $\sqrt{x}$ is rational, taking $a = b = \sqrt{x}$ forces $\texttt{sqrt}(x) = \circ(\sqrt{x})$. If $\sqrt{x}$ is not rational, it is not the rounding breakpoint between two floating-point values either.[6] Therefore, there is a neighborhood of $\sqrt{x}$ in which all the real values are rounded to the same floating-point number. Since rational numbers are a dense subset of real numbers, choosing a and b in this neighborhood concludes the proof.

13.2.2   Low-level algorithms

While the addition and multiplication operators are implemented purely in hardware, division and square root are actually microcoded inside AMD-K5 processors. They are computed by small algorithms performing a sequence of additions and multiplications, on floating-point numbers with a 64-bit significand and a 17-bit exponent, with various rounding operators. Performing the certification of these algorithms then relies on properties of floating-point arithmetic, e.g., the notion of rounding errors.

ACL2 is not the only proof assistant with a formalism rich enough to prove such algorithms. Harrison devised a framework for HOL Light[7] [168] and used it for proving the correctness of the division operator on Intel Itanium processors [170]. The floating-point unit of these processors provides the fused multiply-add (FMA) operation only, so division has to be implemented in software by a sequence of floating-point operations. As such, the approach is similar to AMD-K5's microcoded division, and so is the certification.

There is a third formalism for proving small floating-point algorithms. It is available for two proof assistants: Coq[8] and PVS.[9] This effort was started

---

[6]One could certainly devise a rounding operator that has irrational breakpoints. Fortunately, none of the standard modes is that vicious.

[7]http://www.cl.cam.ac.uk/~jrh13/hol-light/

[8]http://coq.inria.fr/

[9]http://pvs.csl.sri.com/

by Daumas, Rideau, and Théry [97]. Boldo then extended it for proving numerous properties of floating-point arithmetic, e.g., Dekker's error-free multiplication for various radices and precisions [30], the faithfulness of Horner's polynomial evaluation [36], and the use of an exotic rounding operator for performing correct rounding [34].

13.2.3 Advanced algorithms

Previous examples of certification are geared toward rounding properties of algorithms. Once the global rounding error has been proved to be small enough, the proofs are mostly complete.

When considering more complicated functions, e.g., elementary functions, a separate issue arises. Indeed, the exact evaluation of these functions cannot even be performed with the basic arithmetic operators on real numbers. Therefore, their evaluation has to involve some additional approximations. For instance, an elementary function may be replaced by a polynomial that approximates it. As a consequence, proving that the global rounding error is small enough is no longer sufficient to check that the operator is correctly evaluated. One must also prove that the chosen polynomial is sufficiently close to the original elementary function.

This property can be proved by finding the extrema of the error function ϵ between the polynomial P and the elementary function f. As this error may be difficult to directly study from a formal point of view, one could first replace the elementary function by a high-degree polynomial F, so that the error function becomes $\epsilon = (P - F) + (F - f)$. The polynomial F is usually chosen so that the part $F - f$ can be symbolically bounded. For instance, when F is a truncation of the alternated Taylor series of f, the bound on $F - f$ is given by the first discarded term.

The other part $Q = P - F$ is simply a polynomial, so it can be automatically bounded by formal methods. In particular, formally locating the roots of the derivative Q' gives guaranteed bounds on the extrema of Q. An example of this approach to certifying elementary functions can be found in Harrison's HOL Light proof of Tang's floating-point implementation of the exponential function [167]. In this proof, small intervals enclosing the roots of Q' were obtained by using its Sturm sequence.

Harrison has also considered using sum-of-square decompositions for proving properties of polynomial systems [172], which is a problem much more general than just bounding Q. The decomposition $\sum R_i^2$ of a polynomial $R = Q + a$ is particularly well suited for formal proofs. Indeed, while the decomposition may be difficult to obtain, the algorithm does not have to be formally proved. Only the equality $R = \sum R_i^2$ has to, which is easy and fast. A straightforward corollary of this equality is the property $Q \geq -a$.

However, both approaches seem to be less efficient than doing a variation analysis of Q in HOL Light [169]. This analysis also considers the roots

of the derivative Q', but instead of using Sturm sequences to obtain the enclosures of the roots, Harrison recursively encloses all the roots of its derivatives. Indeed, a differentiable function can be zero only at one point between two consecutive zeros of its derivative. Since Q is a polynomial, the recursion stops on linear polynomials which have trivial roots.

Another approach is the decomposition of Q' in the Bernstein polynomial basis. Indeed, if the number of sign changes in the sequence of coefficients is zero or one, this is also the number of roots (similar to Descartes' Law of Signs). If it is bigger, De Casteljau's algorithm can be used to efficiently compute the Bernstein decompositions of Q on two subintervals. A bisection can therefore isolate all the extrema of Q. Zumkeller implemented and proved this method in Coq [445].

Another possibility lies in using interval arithmetic and variation analysis to obtain extrema. This time, the analysis is directly performed on the function $\epsilon = P - f$. For this purpose, Melquiond formalized and implemented in Coq an automatic differentiation and an interval arithmetic with floating-point bounds [274]. A similar method was first experimented in PVS [96] but it proved inefficient due to the use of rational numbers as interval bounds.

13.3   Automating Proofs

Due to the wide scope of floating-point arithmetic, there cannot be an automatic process for certifying all the valid floating-point algorithms. Some parts of the certification may have to be manually performed, possibly in a formal environment. Nevertheless, most proofs on floating-point algorithms involve some repetitive, tedious, and error-prone, tasks, e.g., verifying that no overflows occur. Another common task is a forward error analysis in order to prove that the distance between the computed result and the ideal result is bounded by a specified constant.

The Gappa[10] tool is meant to automate most of these tasks. Given a high-level description of a binary floating-point (or fixed-point) algorithm, it tries to prove or exhibit some properties of this algorithm by performing interval arithmetic and forward error analysis. When successful, the tool also generates a formal proof that can be embedded into a bigger development,[11] hence greatly reducing the amount of manual certification needed. The methods used in Gappa depend on the ability to perform numerical computations

---

[10]Gappa is distributed under a GNU GPL license. It can be freely downloaded at

```
http://lipforge.ens-lyon.fr/projects/gappa/
```

[11]Certifying the numerical accuracy of an implementation is only a small part of a certification effort. One may also have to prove that there are no infinite loops, no accesses out of the bounds of an array, and so on. So the user has to perform and combine various proofs in order to get a complete certification.

during the proofs, so there are two main constraints on the floating-point algorithms: their inputs have to be bounded somehow,[12] and the precision of the arithmetic cannot be left unspecified.[13]

While Gappa is meant to help certify floating-point algorithms, it manipulates expressions on real numbers only and proves properties of these expressions. Floating-point arithmetic is expressed using the functional rounding operators presented in Section 13.1.3. As a consequence, infinities, NaNs, and signed zeros are not first-class citizens in this approach. So, Gappa is unable to propagate them through computations, but it is nonetheless able to prove they do not occur during computations.

In developing the tool, several concepts related to the automatic certification of the usual floating-point algorithms were brought to light. The following sections describe them in the context of Gappa.

13.3.1 Computing on bounds

Gappa proves floating-point properties by exhibiting facts on expressions of real numbers. These facts are of several types, characterized by some predicates mixing numerical values with expressions. The main predicate expresses numerical bounds on expressions. These bounds are obtained and can then be verified with interval arithmetic (see Section 2.9 page 51). For example, if the real x is enclosed in the interval $[-3, 7]$, one can easily prove that the real $\sqrt{1 + x^2}$ is well specified and enclosed in the interval $[1, 8]$.

Exceptional behaviors

Enclosures are sufficient for expressing properties usually encountered while certifying numerical algorithms. The first kind of property deals with exceptional behaviors. An exceptional behavior occurs when the input of a function is out of its definition domain. It also occurs when the evaluation causes a trap, e.g., in case of overflow or underflow.

Consider the computation $\mathrm{RN}(\sqrt{\mathrm{RD}(1 + \mathrm{RU}(x^2))})$. Let us assume that the software will not be able to cope with an overflow or an invalid operation if one is caused by this formula. Certifying that no exceptional behavior occurs then amounts to checking that neither $\mathrm{RU}(x^2)$ nor $\mathrm{RD}(1 + \mathrm{RU}(x^2))$ nor $\mathrm{RN}(\sqrt{\mathrm{RD}(\ldots)})$ overflows, and that $\mathrm{RD}(1 + \mathrm{RU}(x^2))$ is non-negative. This last property is easy enough to express with an enclosure: $\mathrm{RD}(1 + \mathrm{RU}(x^2)) \in [0, +\infty)$. What about the ones on overflow?

Let us denote M the biggest representable floating-point number and M^+ the first power of the radix too big to be representable.[14] Expressing that

---

[12] For algorithms evaluating elementary functions, the inputs are naturally bounded, due to the argument reduction step (see Section 11, page 378).

[13] Again, this is hardly an issue, as most algorithms are written with a specific floating-point format in mind, e.g., binary64.

[14] So M^+ would actually be the floating-point number just after M if there was no overflow.

$\mathrm{RU}(x^2)$, or more generally $\mathrm{RU}(y)$ does not overflow may be achieved with the property $y \in (-M^+, +M]$. Negative values outside this half-open interval would be rounded to $-\infty$, and positive values to $+\infty$. This approach has a drawback: it depends on the direction of the rounding mode. The enclosure $y \in (-M^+, +M]$ is used for $\mathrm{RU}(y)$, $y \in [-M, +M^+)$ for $\mathrm{RD}(y)$, $y \in (-M^+, M^+)$ for $\mathrm{RZ}(y)$, and $y \in (-\frac{1}{2}(M + M^+), +\frac{1}{2}(M + M^+))$ for $\mathrm{RN}(y)$.

Let us consider a set of floating-point numbers without an upper bound on the exponent instead, as described in Section 13.1.4. In other words, while $e_{\max}$ still characterizes numbers small enough to be stored in a given format, the rounding operators ignore it and may return real numbers bigger than M, e.g., $\mathrm{RN}(\beta M) = \beta M$. This extended set, therefore, makes it simpler to characterize an operation that does not overflow: one just has to prove that $\circ(y) \in [-M, +M]$, whatever the rounding mode $\circ$ happens to be.

Quantifying the errors

Proving that no exceptional behaviors occur is only a part of the certification process. If the algorithm is not meant (or unable) to produce exact values at each step due to rounding errors, one wants to bound the distance between the computed value $\tilde{v}$ and the infinitely precise value $\hat{v}$ at the end of the algorithm.[15] This value $\hat{v}$ may not even be the mathematical value v that the algorithm tries to approximate. Indeed, some terms of v may have been neglected in order to speed up the algorithm ($1 + x^{-42} \rightsquigarrow 1$ for big x). Or they may have been simplified so that they can actually be implemented ($\cos x \rightsquigarrow 1 - \frac{x^2}{2}$ for small x).

There are two common distances between the computed value and the mathematical value: the absolute error and the relative error. In order to reduce the number of expressions to analyze, Gappa favors errors written as $\tilde{v} - v$ and $\frac{\tilde{v} - v}{v}$, but other forms could be handled as well.

Error expressions are bounded by employing methods from forward error analysis. For instance, the expression $(\tilde{u} \times \tilde{v}) - (u \times v)$ is bounded in two steps. First, the expressions $\tilde{u} - u$ and $\tilde{v} - v$, which are still error expressions, are bounded separately. The same is done for expressions u and v. Second, these four bounds are combined by applying interval arithmetic to the following formula:

$$(\tilde{u} \times \tilde{v}) - (u \times v) = (\tilde{u} - u) \times v + u \times (\tilde{v} - v) + (\tilde{u} - u) \times (\tilde{v} - v).$$

The previous example is the decomposition of the absolute error between two products. Similar decompositions exist for the two kinds of error and for all the arithmetic operators: $+, -, \times, \div, \sqrt{\cdot}$.

---

[15] An expression computing $\hat{v}$ can be obtained by removing all the rounding operators from an expression computing $\tilde{v}$.

Another important class of operators is the rounding operators, as presented in Section 13.1.3. The absolute error $\circ(\tilde{u}) - u$ is decomposed into a simpler error $\tilde{u} - u$ and a rounding error $\circ(\tilde{u}) - \tilde{u}$. The rounding error is then bounded depending on the properties of the rounding operator. For floating-point arithmetic, this often requires a crude bound on $\tilde{u}$.

Gappa therefore works by decomposing the target expression into smaller parts by relying on methods related to forward error analysis. Once all these parts can be bounded—either by hypotheses (e.g., preconditions of the studied function) or by properties of rounding operators—the tool performs interval arithmetic to compute an enclosure of the expression.

13.3.2   Counting digits

Representable numbers for a given format are a discrete subset of the real numbers. This is especially noticeable when considering fixed-point arithmetic. Since we do not consider overflow, a fixed-point format can be defined as the weight or position of the least significant bit (LSB) of the numbers. Therefore, given three fixed-point variables a, b, and c with the same number format, the addition $c := \circ(a+b)$ is exact. Such exact operations are inherently frequent when using fixed-point arithmetic, be it in software or in hardware design.

Thus, we would like the tool to be able to prove that $c - (a + b)$ is equal to zero, so that the rounding error is not overestimated. As described above, the tool knows how to bound $\circ(u) - u$ for a real number u. For instance, if $\circ$ rounds toward $+\infty$ and the weight of the LSB is 2^k, then the property $\forall u \in \mathbb{R}, \circ(u) - u \in [0, 2^k)$ holds. Notice that $[0, 2^k)$ is the smallest possible subset, since u can be any real number. So this is not helpful in proving $\circ(a + b) - (a + b) \in [0, 0]$.

The issue comes from $a + b$ not being any real number. It is the sum of two numbers belonging to the fixed-point set $\{m \cdot 2^k \mid m \in \mathbb{Z}\}$, which happens to be closed under addition. Since $\circ$ is the identity on this set, the rounding error is zero. So manipulating enclosures only is not sufficient: the tool has to gather some information on the discreteness of the expressions.

Fixed-point arithmetic

As stated above, reasoning on discrete expressions could be achieved by considering the stability of fixed-point sets. However, it would be overly restrictive, as this approach would not cater to multiplication. Instead, Gappa keeps track of the positions of the LSB of all the expressions, if possible. In the previous example, a and b are multiples of 2^k, so their sum is too. Since the operator $\circ$ rounds at position k, it does not modify the result of the sum $a + b$. More generally, if an expression u is a multiple of 2^ℓ and if an operator $\circ$ rounds at position $h \leq \ell$, then $\circ(u) = u$.

For directed rounding, this proposition can be extended to the opposite case $h > \ell$. Indeed the rounding error $\varepsilon = \circ(u) - u$ satisfies $|\varepsilon| < 2^h$ and is a multiple of 2^ℓ. Thus, $|\varepsilon| \leq 2^h - 2^\ell$. As a consequence, when rounding toward zero at position h a number u multiple of 2^ℓ, the rounding error $\circ(u) - u$ is enclosed in $[-\delta, \delta]$ with $\delta = \max(0, 2^h - 2^\ell)$.

Let us denote $\mathrm{FIX}(u, k)$ the predicate stating that the expression u is a multiple of 2^k: $\exists m \in \mathbb{Z}, \ u = m \cdot 2^k$. Assuming that the properties $\mathrm{FIX}(u, k)$ and $\mathrm{FIX}(v, \ell)$ hold for the expressions u and v, the following properties can be deduced:

$$\mathrm{FIX}(-u, k),$$
$$\mathrm{FIX}(u + v, \min(k, \ell)),$$
$$\mathrm{FIX}(u \times v, k + \ell).$$

Moreover, if the operator $\circ$ rounds at position k, then the property $\mathrm{FIX}(\circ(u), k)$ holds. The predicate is also monotonic:

$$\mathrm{FIX}(u, k) \wedge k \geq \ell \quad \Rightarrow \quad \mathrm{FIX}(u, \ell).$$

Floating-point arithmetic

The error bounds could be handled in a similar way for floating-point arithmetic, but the improvements are expected to be less dramatic. For instance, when rounding toward zero the product of two numbers in the binary64 format, the improved bound on the relative error (assuming a nonunderflowing result) is $2^{-52} - 2^{-105}$. This better bound is hardly more useful than the standard error bound 2^{-52} for proving the correctness of most floating-point algorithms.

As a consequence, Gappa does not try to improve the relative error bound, but it still tries to detect exact operations. Consider that rounding happens at precision p and that the smallest positive number is 2^k. A rounding operator $\circ_{p,k}$ will leave an expression u unmodified when its number w of significant digits is small enough ($w \leq p$) and its least significant digit has a weight 2^e big enough ($e \geq k$). The second property can be expressed with the FIX predicate: $\mathrm{FIX}(u, k)$. The first property requires a new predicate:

$$\mathrm{FLT}(u, p) \quad \equiv \quad \exists m, e \in \mathbb{Z}, \ u = m \cdot 2^e \wedge |m| < 2^p.$$

Therefore, Gappa uses the following theorem:

$$\mathrm{FIX}(u, k) \wedge \mathrm{FLT}(u, p) \quad \Rightarrow \quad \circ_{p,k}(u) - u \in [0, 0].$$

Assuming that the properties $\mathrm{FLT}(u, p)$ and $\mathrm{FLT}(v, q)$ hold for the expressions u and v, the following properties can be deduced:

$$\mathrm{FLT}(-u, p),$$
$$\mathrm{FLT}(u \times v, p + q).$$

As with the fixed-point case, some properties can be deduced from a floating-point rounding operator:

$$\text{FLT}(\circ_{p,k}(u), p) \wedge \text{FIX}(\circ_{p,k}(u), k).$$

There is again a monotonicity property:

$$\text{FLT}(u, p) \wedge p \le q \quad \Rightarrow \quad \text{FLT}(u, q).$$

And finally, both predicates can be related as long as bounds are known on the expressions:

$$\text{FIX}(u, k) \wedge |u| < 2^g \quad \Rightarrow \quad \text{FLT}(u, g - k),$$
$$\text{FLT}(u, p) \wedge |u| \ge 2^g \quad \Rightarrow \quad \text{FIX}(u, g - p + 1).$$

Application

Here is an example showing how the various predicates interact. Consider two expressions $u \in [3.2, 3.3]$ and $v \in [1.4, 1.8]$, both with p significant digits at most. How can we prove that the difference $u - v$ also has p significant digits at most? Notice that Lemma 2 (Sterbenz's lemma, see Section 4.2, page 122) does not apply, since $\frac{3.3}{1.4} > 2$.

First of all, by interval arithmetic, we can prove $|u - v| \le 1.9$. Moreover, since we have $|u| \ge 2^1$ and $\text{FLT}(u, p)$, the property $\text{FIX}(u, 2 - p)$ holds. Similarly, $\text{FIX}(v, 1 - p)$ holds. Therefore, we can deduce a property on their difference: $\text{FIX}(u - v, 1 - p)$. By combining this property with the bound $|u - v| < 2^1$, we get $\text{FLT}(u - v, p)$. So we have proved that $u - v$ has at most p significant digits.

This reasoning could then be extended until we prove that the floating-point subtraction is actually exact. Indeed, if the smallest positive floating-point number is 2^k, we have both $\text{FIX}(u, k)$ and $\text{FIX}(v, k)$. As a consequence, we also have $\text{FIX}(u - v, k)$. By combining this property with $\text{FLT}(u - v, p)$, we conclude that $u - v$ is representable as a floating-point number.

13.3.3 Manipulating expressions

Errors between structurally similar expressions

As presented in Section 13.3.1, Gappa computes bounds on an expression representing an error, either $\tilde{x} - x$ or $\frac{\tilde{x} - x}{x}$, by first rewriting it in another form. This new expression is meant to make explicit the errors on the subterms of $\tilde{x}$ and x. For instance, if $\tilde{x}$ and x are the quotients $\tilde{u}/\tilde{v}$ and u/v, respectively,

then Gappa relies on the following equality:[16]

$$\frac{\tilde{u}/\tilde{v} - u/v}{u/v} = \frac{\epsilon_u - \epsilon_v}{1 + \epsilon_v} \quad \text{with} \quad \epsilon_u = \frac{\tilde{u} - u}{u} \quad \text{and} \quad \epsilon_v = \frac{\tilde{v} - v}{v} .$$

The left-hand side contains the correlated expressions $\tilde{u}$ and u, and $\tilde{v}$ and v, while the right-hand side does not. Therefore, the latter should have a better structure for bounding the error by interval arithmetic. It still has a bit of correlation, since the expression ϵ_v appears twice; however, this does not induce an overestimation of the bounds. Indeed, the whole expression is a homographic transformation with respect to ϵ_v, hence monotonic on the two parts of its domain. Gappa takes advantage of this property when computing its enclosure. Therefore, as long as the errors ϵ_u and ϵ_v are not correlated, tight bounds on the relative error between $\tilde{u}/\tilde{v}$ and u/v are obtained. In fact, unless the partial errors ϵ_u and ϵ_v are purposely correlated, this correlation hardly matters when bounding the whole error.

For this approach to work properly, the expressions $\tilde{x}$ and x have to use the same operator—a division in the example above. Moreover, their subterms should match, so that the error terms ϵ_u and ϵ_v are meaningful. In other words, $\tilde{u}$ should somehow be an approximation to u, and $\tilde{v}$ to v.

For instance, Gappa's approach will fail if the user wants to analyze the error between the expressions $\tilde{x} = (a_1 \times \tilde{u})/(a_2 \times \tilde{v})$ and $x = u/v$. If a_1 and a_2 are close together, then computing $\tilde{x}$ may be a sensible way of getting an approximate value for x, hence the need to bound the error. Unfortunately, $a_1 \times \tilde{u}$ is not an approximation to u, only $\tilde{u}$ is. So the transformation above will fail to produce useful bounds on the error. Indeed, an error expression ϵ_u between $a_1 \times \tilde{u}$ and u does not present any interesting property.

In such a situation, the user should tell Gappa that $\tilde{x}$ and $x' = \tilde{u}/\tilde{v}$ are close, and how to bound the (relative) error between them. When asked about the error between $\tilde{x}$ and x, Gappa will separately analyze its two parts—the error between $\tilde{x}$ and x' and the error between x' and x—and combine them.

Using intermediate expressions

This process of using an intermediate expression in order to get errors between structurally similar expressions is already applied by Gappa for rounding operators. Indeed, while the user may sometimes want to bound the error between $\tilde{x} = \tilde{u}/\tilde{v}$ and $x = u/v$, a more usual case is the error between $\tilde{y} = \circ(\tilde{x})$ and x. The two expressions no longer have a similar structure, since the head symbol of $\tilde{y}$ is the unary operator $\circ$ while the head symbol of x is the binary operator $/$.

---

[16]Using the right-hand-side expression to compute the left-hand-side one requires several hypotheses on u, v, and $\tilde{v}$ not being zero. Section 13.3.4 details how Gappa eliminates these hypotheses by using a slightly modified definition of the relative error.

To avoid this issue, Gappa registers that $\tilde{y}$ is an approximation to $y = \tilde{x}$, by definition and purpose of rounding operators. Whenever it encounters an error expression involving $\tilde{y}$, e.g., $\frac{\circ(\tilde{x})-x}{x}$, Gappa will therefore try to decompose it by using y as an intermediate expression:

$$\frac{\tilde{y} - x}{x} = \epsilon_1 + \epsilon_2 + \epsilon_1 \times \epsilon_2$$

with

$$\epsilon_1 = \frac{\tilde{y} - y}{y} \quad \text{and} \quad \epsilon_2 = \frac{y - x}{x} = \frac{\tilde{x} - x}{x}.$$

Rounded values are not the only ones Gappa considers to be approximations to other values. Whenever the tool encounters a hypothesis of the form $\tilde{x} - x \in I$ or $\frac{\tilde{x}-x}{x} \in I$, it assumes that $\tilde{x}$ is an approximation to x, as the enclosed expression looks like an error between $\tilde{x}$ and x. This heuristic is useful in automatically handling problems where there are method errors in addition to rounding errors. In the example of Section 13.4.1, the user has a polynomial $p(x)$ that approximates $\sin x$. The goal is to get a bound on the error between the computed value $\tilde{p}(x)$ and the ideal value $\sin x$. Since the user provides a bound on the method error between p and $\sin$, Gappa deduces that the following equality may be useful:

$$\frac{\tilde{p}(x) - \sin x}{\sin x} = \epsilon_p + \epsilon_s + \epsilon_p \times \epsilon_s$$

with

$$\epsilon_p = \frac{\tilde{p}(x) - p(x)}{p(x)} \quad \text{and} \quad \epsilon_s = \frac{p(x) - \sin x}{\sin x}.$$

Gappa can then automatically compute a bound on ϵ_p, as this is an error between two expressions with the same structure once rounding operators have been removed from $\tilde{p}$.

Cases of user hints

When Gappa's heuristics are unable to reduce expressions to errors between terms with similar structures, the user can provide additional hints. There are two kinds of hints. The first kind tells Gappa that two expressions have the same bounds under some constraints. In particular, if one of the expressions is composed of errors between structurally similar terms, then Gappa will be able to deduce tight bounds on the other expression. The second kind of hint tells Gappa that a specific term approximates another one, so the tool will try to exhibit an error expression between the two of them.

Matching formula structures   If the developer has decided some terms are negligible and can be ignored in the computations, then the formula of the computed value will have some missing terms with respect to the formula of the exact value. As a consequence, the structures of the two formulas may be different.

As an example, let us consider a double-double multiplication algorithm. The inputs are the pairs of numbers (x_h, x_ℓ) and (y_h, y_ℓ) with x_ℓ negligible with respect to x_h, and y_ℓ to y_h. The output of the algorithm is a pair (z_h, z_ℓ) that should approximate the exact product $p = (x_h + x_\ell) \times (y_h + y_\ell)$. The algorithm first computes the exact result $z_h + t$ of the product $x_h \times y_h$ (see Section 4.4, page 132). The products $x_h \times y_\ell$ and $x_\ell \times y_h$ are then added to the lower part t in order to get z_ℓ:

$$
\begin{aligned}
z_h + t &\;\leftarrow\; x_h \times y_h \\
z_\ell &\;\leftarrow\; \mathrm{RN}(\mathrm{RN}(t + \mathrm{RN}(x_h \times y_\ell)) + \mathrm{RN}(x_\ell \times y_h)).
\end{aligned}
$$

The product $x_\ell \times y_\ell$ is supposedly negligible and was ignored when computing z_ℓ. The goal is to bound the error $(z_h + z_\ell) - p$, but the two subterms do not have the same structure: p is a product, $z_h + z_\ell$ is not. So we rewrite the error by distributing the multiplication and setting aside $x_\ell \times y_\ell$:

$$
(z_h + z_\ell) - p = \underbrace{(z_h + t) - (x_h \times y_h)}_{\delta_1} + \underbrace{z_\ell - (t + (x_h \times y_\ell) + (x_\ell \times y_h))}_{\delta_2} - x_\ell \times y_\ell\,.
$$

In the preceding formula, δ_1 is zero by definition of the exact product, while δ_2 is the difference between two expressions with the same structure once the rounding operators have been removed. An evaluation by interval of the right-hand-side expression will therefore give tight bounds on the left-hand-side expression.

Handling converging formulas   Gappa does not recognize converging expressions. Handling them with the usual heuristics will fail, since their structure is generally completely unrelated to the limit of the sequence. The developer should therefore explain to the tool why a particular formula was used. Let us consider Newton's iteration for computing the multiplicative inverse of a number a:

$$
x_{n+1} = \mathrm{RN}(x_n \times \mathrm{RN}(2 - \mathrm{RN}(a \times x_n))).
$$

This sequence quadratically converges toward a^{-1}, when there are no rounding operators. So the first step is to tell Gappa that the computed value x_{n+1} is an approximation to the same expression without rounding operators:

$$
x_{n+1} \simeq x_n \times (2 - (a \times x_n)) \equiv x'_{n+1}.
$$

Notice that x'_{n+1} is not defined as depending on x'_n but on x_n. Indeed, the iteration is self-correcting, so using x_n will help Gappa to notice that rounding errors are partly compensated at each step. The developer can now tell Gappa that the error with the ideal value decreases quadratically:

$$\frac{x'_{n+1} - a^{-1}}{a^{-1}} = -\left(\frac{x_n - a^{-1}}{a^{-1}}\right)^2.$$

When evaluating the error $(x_{n+1} - a^{-1})/a^{-1}$ at a step $n + 1$, Gappa will split it into two parts: $(x_{n+1} - x'_{n+1})/x'_{n+1}$, which is the total rounding error at this step, and $(x'_{n+1} - a^{-1})/a^{-1}$, which is given by the identity above and depends on the error at step n.

13.3.4 Handling the relative error

Expressing a relative error as $\frac{\tilde{u}-u}{u}$ implicitly puts a constraint $u \neq 0$. Therefore, if the enclosure $\frac{\tilde{u}-u}{u} \in I$ appears in a hypothesis or as a conclusion of a theorem, it is generally unavailable until one has also proved $u \neq 0$. This additional constraint is not an issue when dealing with the relative error of a product, that is $\tilde{u} = \circ(x \times y)$ and $u = x \times y$. Indeed, the relative error $\frac{\tilde{u}-u}{u}$ can usually be bounded only for u outside of the underflow range. As a consequence, the property $u \neq 0$ comes for free.

For floating-point addition, the situation is not that convenient. Indeed, the relative error between $\tilde{u} = \circ(x + y)$ and $u = x + y$ (with x and y floating-point numbers) is commonly admitted to be always bounded, even for subnormal results. So the relative error should be expressed in a way such that its use does not require one to prove that the ideal value u is nonzero.

Gappa achieves this property by introducing another predicate:

$$\text{REL}(\tilde{u}, u, I) \quad \equiv \quad -1 < \text{lower}(I) \wedge \exists \epsilon_u \in I, \ \tilde{u} = u \times (1 + \epsilon_u).$$

Always-bounded relative errors

The relative error of a sum $z = \text{RN}(x) + \text{RN}(y)$ can now be enclosed as follows:[17]

$$\text{REL}(\text{RN}(z), z, [-2^{-53}, 2^{-53}]).$$

The property above is too restrictive, however. The two subterms do not necessarily have to be rounded to nearest in binary64 format. For instance, it would still hold for $z = \text{RD}(x) + \text{RZ}_{\text{binary32}}(y)$. As a matter of fact, z does not even have to be a sum. So, which conditions are sufficient for the relative error to be bounded? For $\text{RN}(z)$ with $z = \text{RN}(x) + \text{RN}(y)$, the bound is a

---

[17]The expression $z = \text{RN}(x) + \text{RN}(y)$ ensures that z is not the sum of any two real numbers, but of two floating-point numbers represented with the same format as $\text{RN}(z)$, i.e., same precision and $e_{\min}$.

consequence of the floating-point addition not losing any information with a magnitude smaller than the smallest positive floating-point number 2^k. This is generalized to the following theorem, which Gappa relies on to bound the relative error when rounding to nearest:

$$\text{FIX}(u,k) \quad\Longrightarrow\quad \text{FLT}(\circ_{p,k}(u), u, [-2^{-p}, 2^{-p}]).$$

The original property can then be recovered with the following proof. Since both $\text{RN}(x)$ and $\text{RN}(y)$ are binary64 numbers, they satisfy the properties $\text{FIX}(\text{RN}(x), -1074)$ and $\text{FIX}(\text{RN}(y), -1074)$. As a consequence (Section 13.3.2), their sum does too: $\text{FIX}(\text{RN}(x) + \text{RN}(y), -1074)$. Applying the theorem above hence gives $\text{REL}(\text{RN}(z), z, [-2^{-53}, 2^{-53}])$.

Propagation of relative errors

Rules for the REL predicate are straightforward. Given the two properties $\text{REL}(\tilde{u}, u, I_u)$ and $\text{REL}(\tilde{v}, v, I_v)$, multiplication and division are as follows:

$\text{REL}(\tilde{u} \times \tilde{v}, u \times v, J)$ with $J \supseteq \{\epsilon_u + \epsilon_v + \epsilon_u \times \epsilon_v \mid \epsilon_u \in I_u, \epsilon_v \in I_v\}$,
$\text{REL}(\tilde{u}/\tilde{v}, u/v, J)$ with $J \supseteq \{\frac{\epsilon_u - \epsilon_v}{1 + \epsilon_v} \mid \epsilon_u \in I_u, \epsilon_v \in I_v\}$.

Assuming $u + v \neq 0$ and $\tilde{v}/(u+v) \in I$, the relative error for addition is given by $\text{REL}(\tilde{u} + \tilde{v}, u + v, J)$ with

$$J \supseteq \{\epsilon_u \times t + \epsilon_v \times (1 - t) \mid \epsilon_u \in I_u, \epsilon_v \in I_v, t \in I\}.$$

Composing relative errors is similar[18] to multiplication: Given the two properties $\text{REL}(z, y, I_1)$ and $\text{REL}(y, x, I_2)$, we have $\text{REL}(z, x, J)$ with

$$J \supseteq \{\epsilon_1 + \epsilon_2 + \epsilon_1 \times \epsilon_2 \mid \epsilon_1 \in I_1, \epsilon_2 \in I_2\}.$$

13.4   Using Gappa

13.4.1   Toy implementation of sine

Let us consider the C code shown in Listing 13.1.

The literal `0x28E9.p-16f` is a C99 hexadecimal notation for the floating-point constant of type `float` equal to $10473/65536$. So the function computes a value $y \simeq x - x^3/6 \simeq \sin x$.

Assuming all the computations are rounded to nearest in the binary32 format,[19] what is the relative error between the computed value y and the

---

[18]Encountering the same formula is not unexpected: taking $z = \tilde{u} \times \tilde{v}$, $y = \tilde{u} \times v$, and $x = u \times v$ makes it appear.

[19]With this C code, this is guaranteed by the ISO C99 standard under the condition that `FLT_EVAL_METHOD` is equal to 0 (see Section 7.2.3, page 213) and that floating expressions are not contracted (in particular because an FMA could be used here, see Section 7.2.3, page 214).

C listing 13.1 Toy implementation of sine around zero.

```
float my_sine(float x)
{
  assert(fabsf(x) <= 1);
  float y = x * (1.f - x*x * 0x28E9.p-16f);
  return y;
}
```

---

mathematical value $\sin x$? The Gappa tool will be used to find some bounds on this error. Note that, for the sake of this example, x will first be assumed not to be too small. So one of the hypotheses will be $|x| \in [2^{-100}, 1]$.

Gappa is designed to prove some logical propositions involving enclosures of real-valued expressions. These expressions can be built around the basic arithmetic operators only. In particular, Gappa does not know about the sine function. So the value $\sin x$ will be represented by a new variable sin_x. Consider the following script:

```
{ |x| in [1b-100,1] -> (y - sin_x) / sin_x in ? }
```

Since it contains an interrogation mark, Gappa will try to find an interval I such that the following statement holds:

$$\forall x \forall y \forall sin\_x \quad |x| \in [2^{-100}, 1] \Rightarrow \frac{y - sin\_x}{sin\_x} \in I.$$

Obviously, there is no such interval I, except $\mathbb{R}$ assuming sin_x is implicitly not zero.[20] The proposition is missing a lot of data. In particular, y should not be a universally quantified variable: it should be an expression depending on x. This definition of y can be added to the Gappa script before the logical proposition:[21]

```
y = ...; # to be filled
{ |x| in [1b-100,1] -> (y - sin_x) / sin_x in ? }
```

The computations are performed in rounding to nearest and the numbers are stored in the binary32 format. In Gappa's formalism, this means that the rounding operator float<24, −149, ne>[22] is applied to each infinitely precise result. Note that the operator float<ieee_32,ne> is a predefined synonym. Still it would be cumbersome to type this operator for each floating-point operation, so a shorter alias rnd is defined beforehand:

```
@rnd = float<ieee_32,ne>;
y = rnd(x * rnd(1 - rnd(rnd(x * x) * 0x28E9p-16)));
```

---

[20]Gappa never assumes that a divisor cannot be zero, unless running in *unconstrained* mode.

[21]It does not have to, though. The term y could be directly replaced by its definition in the logical proposition.

[22]The parameters of the rounding operator float<24, −149, ne> express its properties: target precision is 24 bits, the smallest positive representable number is 2^{-149}, and results are rounded to Nearest Even.

Even with the alias, the definition of y is still obfuscated. Since all the operations are using the same rounding operator, it can be prefixed to the equal symbol, so that the definition is almost identical to the original C code:

```
y rnd= x * (1 - x*x * 0x28E9p-16);
```

Although y is now defined, Gappa is still unable to tell anything interesting about the logical proposition, since sin_x is a universally quantified variable instead of being $\sin x$. It would be possible to define sin_x as the sum of a high-degree polynomial in x with an error term δ. Then a hypothesis stating some bounds on δ would be sufficient from a mathematical point of view.

However, this is not the usual approach with Gappa. Rather, we should look at the original C code again. The infinitely precise expression

$$My = x \times \left(1 - \frac{10473}{65536} \times x^2 \right)$$

was purposely chosen so that it is close to $\sin x$, since this is what we want to compute. Therefore, sin_x should be defined with respect to that infinitely-precise expression. The relative error between My and $sin\_x = \sin x$ can be obtained (and proved) by means external to Gappa; it is added as a hypothesis to the proposition:

```
My = x * (1 - x*x * 0x28E9p-16);
{ |x| in [1b-100,1] /\ |(My - sin_x) / sin_x| <= 1.55e-3
  -> ((y - sin_x) / sin_x) in ? }
```

Note that constants in Gappa are never implicitly rounded. So the literal `1.55e-3` is the real number $155/1000$ and the literal `1b-100` represents the real number $1 \cdot 2^{-100}$—it could also have been written `0x1.p-100`. Similarly, the hexadecimal literal `0x28E9p-16` that appears in y and My is a compact way of expressing the real number $10473/65536$.

At this point, Gappa still cannot give an interval containing the relative error, although all the information seems to be present. In order to find the issue, the tool can be run in *unconstrained* mode with the `-Munconstrained` command-line option. This mode prevents Gappa from noticing that values underflow and that divisors are zero.[23] Gappa then answers that the relative error is bounded by $1.551 \cdot 10^{-3}$.

The tool also explains that it has assumed sin_x is not zero in order to get this bound. Indeed, Gappa is unable to use the hypothesis on the relative error $(My - sin\_x)/sin\_x$ if sin_x is zero. Since this property cannot be deduced from the current hypotheses, a new one needs to be added. Since $|\sin x|$ is known to be bigger than $\frac{|x|}{2}$ for $|x| \leq \frac{\pi}{2}$, one can just assume that $|sin\_x|$ is bigger than 2^{-101}.

---

[23]As a consequence, Gappa no longer generates a formal proof, since it would contain holes.

Gappa script 13.1 Relative error of a toy implementation of sine.

```
# Floating-point format is binary32; operations are rounded to nearest
@rnd = float<ieee_32,ne>;

# Value computed by the floating-point function
y rnd= x * (1 - x*x * 0x28E9p-16);
# Value computed with an infinitely-precise arithmetic (no rounding)
My   = x * (1 - x*x * 0x28E9p-16);

# Proposition to prove
{
  # Input x is smaller than 1 (but not too small)
  |x| in [1b-100,1] /\
  # My is close to sin x and the bound is known
  |(My - sin_x) / sin_x| <= 1.55e-3 /\
  # Helper hypothesis: sin x is not too small either
  |sin_x| in [1b-101,1]
->
  # Target expression to bound
  ((y - sin_x) / sin_x) in ?
}
```

Given Script 13.1, Gappa is able to compute and prove a bound on the relative error between y and sin_x. Since sin_x will appear as a universally quantified variable in a formal development, it can later be instantiated by $\sin x$ without much trouble. This instantiation will simply require the user to prove that the properties

$$|(My - \sin x)/\sin x| \leq 1.55 \cdot 10^{-3}$$

and

$$|\sin x| \in [2^{-100}, 1]$$

hold for $x \in [2^{-100}, 1]$.

In order to get rid of the extraneous hypotheses on x and $\sin x$ not being too small, one can express the relative errors directly with the predicate REL (Section 13.3.4). In Gappa syntax, the relative error between two expressions u and v is written u -/ v. It amounts to saying that there exists ϵ such that $u = v \times (1 + \epsilon)$; and enclosures on u -/ v are actually enclosures on ϵ. This leads to Script 13.2.

Two additional informations were given to Gappa in Script 13.2. Firstly, x is not just any real number, it is a binary32 number. This is expressed by saying that x is the rounded value of a dummy number $x\_$. Therefore x is no longer universally quantified, $x\_$ is. This is hardly an issue, as this dummy variable can be instantiated by the actual argument X of the my_sine function in a formal development. Since $x = \text{RN}(x\_) = \text{RN}(X) = X$ holds by virtue of X being a representable floating-point number, the new proposition is equivalent to the old one.

Gappa script 13.2 Relative error of a toy implementation of sine (variant).

```
@rnd = float<ieee_32,ne>;

# x is a binary32 floating-point number
x = rnd(x_);

y rnd= x * (1 - x*x * 0x28E9p-16);
My   = x * (1 - x*x * 0x28E9p-16);

{
  |x| <= 1 /\
  |My -/ sin_x| <= 1.55e-3
->
  |y -/ sin_x| <= 1.551e-3
}

# Separate the cases |x| <= 1b-100 and |x| >= 1b-100
$ |x| in (1b-100);
```

Secondly, there is not a single proof scheme (at least not in Gappa setting) that fits all of the values $x \in [-1, 1]$. So, the script tells the tool to first consider the cases $|x|$ small and $|x|$ big separately and then to merge the two resulting proofs.

13.4.2   Integer division on Itanium

The second example is taken from the software algorithm for performing 16-bit unsigned integer division with floating-point operations on Itanium [89, 170]. A specificity of this algorithm is that all the operations are without rounding error and this is important for proving the correctness of the algorithm. So, the execution of this script heavily relies on Gappa keeping count of the significant digits of computed expressions.

The inputs of the algorithm are a and b. They are 16-bit positive integers, so the logical proposition assumes $a, b \in [1, 65535]$. Moreover, in order to state their integer status, these variables are defined as the integer part of some dummy real numbers:[24]

```
a = int<dn>(a_);
b = int<dn>(b_);
```

The first step of the algorithm is to get an approximation to the reciprocal b^{-1}. This approximation y_0 is computed by the Itanium assembly instruction `frcpa`, which returns an 11-bit value with a relative error ϵ_0 of at most $2^{-8.886}$. So we add a hypothesis on ϵ_0 too: $|\epsilon_0| \le 0.00211373$. Moreover, as previously,

---

[24]Rounding direction dn is toward $-\infty$, but the actual direction does not matter. Indeed, as with Script 13.2, the universally quantified variable $a\_$ is meant to be instantiated by the integer a, so $a = \lfloor a \rfloor = \lceil a \rceil = \lceil a \rceil$.

y_0 is expressed as the rounded value of a dummy real number, so that the number of significant digits is known to Gappa.

For this purpose, an unusual rounding operator `float<11,-1000,dn>` is defined. The fact that it rounds toward $-\infty$ does not matter; only the 11-bit precision does. The choice of the smallest positive value 2^{-1000} possibly returned by the operator does not matter either; it is just chosen so that it does not interfere with the algorithm, whose smallest reciprocal is 65535^{-1}.

```
y0 = float<11,-1000,dn>(y0_);
```

The algorithm then follows with three FMA operations performed in extended precision (operator `rnd`). These operations could be expressed as

```
@rnd = float<x86_80,ne>;
q0 = rnd(a * y0);
e0 = rnd(1 + 1b-17 - b * y0);
q1 = rnd(q0 + e0 * q0);
```

Yet all these operations are exact by design of the algorithm. So using nonrounded formulas will make the later hints simpler:

```
q0 = a * y0;
e0 = 1 + 1b-17 - b * y0;
q1 = q0 + e0 * q0;
```

In order to still get a complete proof, Gappa will instead have to prove that rounding each of the values does not change the final result. In other words, the conclusion of the logical proposition asks for a proof that $\mathsf{rnd}(q_0) - q_0$, $\mathsf{rnd}(e_0) - e_0$, and $\mathsf{rnd}(q_1) - q_1$ are zero.

Ultimately, we would like to prove that $\lfloor q_1 \rfloor$, which is the output of the algorithm, is equal to $\lfloor a/b \rfloor$. This cannot be directly achieved in Gappa; but a small mathematical lemma can get us closer to it. Let us define `err` as the relative error between q_1 and a/b:

```
err = (q1 - a / b) / (a / b);
```

The product $a \times$ err is equal to $q_1 \times b - a$. If this product is in the interval $[0, 1)$, then q_1 is in the interval $[\frac{a}{b}, \frac{a+1}{b})$, which does not contain any integer except for a/b potentially. As a consequence, the integer $\lfloor q_1 \rfloor$ is equal to $\lfloor a/b \rfloor$. Therefore, we need the fact $0 \leq a \times$ err < 1, which we obtain by asking Gappa to prove the stronger property $a \times$ err $\in [0, 0.99999]$.

In order to simplify the expressions, we also introduce the relative error ϵ_0 between y_0 and b^{-1}:

```
eps0 = (y0 - 1 / b) / (1 / b);
```

Since the algorithm relies on error compensation, its Gappa proof requires two mathematical identities—they are the reasons why the algorithm actually computes the integer quotient:

```
e0 -> -eps0 + 1b-17                        { b <> 0 };
err -> -(eps0 * eps0) + (1 + eps0) * 1b-17   { a <> 0, b <> 0 };
```

The first rule says that, when Gappa has computed an enclosure

$$-\epsilon_0 + 2^{-17} \in I,$$

it can assume the enclosure $e_0 \in I$ holds (but only if it also proves $b \neq 0$). This assumptions is valid because the left-hand side and the right-hand side of the rule are equal:

$$e_0 = 1 + 2^{-17} - b \times y_0 = -(y_0 - b^{-1})/b^{-1} + 2^{-17} = -\epsilon_0 + 2^{-17}$$

trivially holds when b is not zero. The second rule is trivial to validate too, since the rounding operators were purposely omitted from the definition of the computed values q_0, e_0, and e_1.

At this stage, the Gappa script is almost complete. If run, the tool proves all the enclosures, except for $\mathrm{rnd}(q_1) - q_1 \in [0,0]$. It is unable to prove that 64 bits are sufficient for representing q_1. As a matter of fact, naive interval arithmetic tends to overestimate bounds when the input intervals are wide. So, a simple way to help Gappa is by splitting the interval containing b into smaller subintervals. Thus, we ask Gappa to perform a bisection until it can prove the bounds on $\mathrm{rnd}(q_1) - q_1$ for each subinterval:

```
rnd(q1) - q1 $ b;
```

Gappa then succeeds in proving the four properties. A look at the generated formal proof shows that the tool decided to study two cases: $b \in [1, 4096]$ and $b \in [4097, 65535]$.

Another way of helping Gappa to succeed would be to search where Gappa overestimates intervals, and then add rules (or modify formulas) in order to tighten the bounds. Here, the overestimation comes from the variable q_0 appearing twice in the definition of q_1. Changing the definition to $q_1 = q_0 \times (1 + e_0)$ is then sufficient. But this also means that the definition of q_1 is a bit farther from the usual definition of a floating-point FMA (yet provably equivalent).

Note that this last step is superfluous when writing a script. Looking for overestimations does not make the proposition any truer; it just reduces the size of the generated formal proof, since the tool no longer studies several cases. Script 13.3 is the complete example.

Gappa script 13.3 Correctness of the 16-bit integer division on IA-64.

```
@rnd = float<x86_80,ne>;
a = int<dn>(a_);
b = int<dn>(b_);

y0 = float<11,-1000,dn>(y0_);

# Values computed by FMA, with rounding operators omitted
q0 = a * y0;
e0 = 1 + 1b-17 - b * y0;
q1 = q0 * (1 + e0);

# Relative errors between computed and ideal values
eps0 = (y0 - 1 / b) / (1 / b);
err = (q1 - a / b) / (a / b);

{
  # Inputs are unsigned 16-bit integers
  a in [1,65535] /\ b in [1,65535] /\
  # Error is bounded by specification of frcpa
  |eps0| <= 0.00211373
->
  # Prove the quotient is correct
  a * err in [0,0.99999] /\
  # Prove the intermediate computations are exact
  rnd(q0) - q0 in [0,0] /\
  rnd(e0) - e0 in [0,0] /\
  rnd(q1) - q1 in [0,0]
}

# Rules for error compensation
e0 -> -eps0 + 1b-17                        { b <> 0 };
err -> -(eps0 * eps0) + (1 + eps0) * 1b-17   { a <> 0, b <> 0 };

# Perform bisection along b; needed when q1 is defined as q0 + q0 * e0
# rnd(q1) - q1 $ b;
```

Chapter 14

Extending the Precision

THOUGH VERY USEFUL in many situations, the fixed-precision floating-point formats that are available in hardware or software in our computers may sometimes prove insufficient. There are reasonably rare cases when the binary64/decimal64 or binary128/decimal128 floating-point numbers of the IEEE 754 standard are too crude as approximations of the real numbers. This may occur for example when dealing with ill-conditioned numerical problems: internal computations with very high precision may be needed to obtain a meaningful final result.

Another situation where the usual basic precisions are obviously insufficient is when the result of the computation itself is required with very high precision. This frequently happens in "experimental mathematics" [38]; for example, when one wants billions of decimals of mathematical constants such as π.

Of course, big precisions may be costly, both in terms of time and memory consumption, so one must find convenient tradeoffs between cost and precision. Precision is not the only problem; the dynamic range of usual formats may be a problem as well. The exponent range may be too restricted for the application under scope, giving rise to frequent overflows and underflows.

The scope of this chapter is to describe usual ways of extending the precision and the exponent range.

Examples

Usual applications of extended precision and exponent range include many algorithmic geometry algorithms [377, 335], the computation of the digits of important mathematical constants such as π [358, 19], the constant e and the zeros of Riemann's zeta function, linear algebra [258], and the determination of integer relations between real or complex numbers [20], etc.

For instance, a famous formula such as the Bailey–Borwein–Plouffe formula for π [21]:

$$\pi = \sum_{k=0}^{\infty} \frac{1}{16^k} \left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6} \right),$$

which allows one to compute the zillionth bit of π without needing to compute the previous ones, was first found using "experimental mathematics" techniques. Another example from number theory is the work on the Mertens conjecture [312, 230].

Also, for designing and/or testing libraries that evaluate mathematical functions in the largest basic formats (e.g., binary64/decimal64 or binary128/decimal128), it is often necessary to perform preliminary calculations (e.g., computation of coefficients of polynomial or rational approximations, and of values stored in tables) in precisions that are significantly higher than those of the "target" formats. To compute worst cases for the rounding of mathematical functions (see Chapter 12) we also sometimes require high precisions.

Multiple-precision arithmetic is out of the scope of this book. However, since it has useful applications for people who design floating-point algorithms, we will quickly present a few basic methods, give some references, and list a few useful packages. We will mainly focus on methods that allow a programmer to roughly double, triple, or even quadruple the precision for critical parts of programs, without requiring the use of multiple-precision packages.

14.1   Double-Words, Triple-Words...

When the need for increased precision is limited to twice (or thrice, or maybe even four times) the largest precision of the available floating-point arithmetic, then a possible solution is to resort to arithmetics that are usually called "double-double" or "triple-double" arithmetics in the literature. These clumsy names come from the fact that, as we are writing this book, the format with the largest precision that is available on all platforms of commercial significance is the double-precision format of IEEE 754-1985 (now much better named binary64 in the IEEE 754-2008 standard). We will call them "double-word" and "triple-word" arithmetics, since there is no special reason for necessarily having the underlying floating-point format being double precision—it makes sense to choose, as the underlying format, the largest one available in hardware—and because the format that was once called "double precision" now has another name in the IEEE 754-2008 standard.

14.1.1 Double-word arithmetic

In Chapters 4 and 5, we have studied algorithms[1] that allow one to represent the sum or product of two floating-point numbers as the unevaluated sum of two floating-point numbers $x_h + x_\ell$, where x_h is nearest the exact result. A natural idea is to manipulate such unevaluated sums. This is the underlying principle of *double-word arithmetic*. It consists in representing a number x as the unevaluated sum of two basic precision floating-point numbers:

$$x = x_h + x_\ell,$$

such that the significands of x_h and x_ℓ do not overlap,[2] which means here that

$$|x_\ell| \leq \frac{1}{2}\,\mathrm{ulp}(x_h).$$

If the basic precision floating-point numbers are of precision p, double-words are not equivalent to precision-$2p$ floating-point numbers. For instance, if the basic precision is double precision/binary64, the double-word that best approximates π is constituted by

$p_1 = 11.0010010000111111011010101000100010000101101000011000_2,$

and

$p_2 = 1.0001101001100010011000110011000101000101110000000111_2 \times 2^{-53}.$

Here, $p_1 + p_2$ is equivalent to a precision-107 binary floating-point approximation to π. Now, $\ln(12)$ will be approximated by

$\ell_1 = 10.011111000010001011010111100110100111001111001111101_2,$

and

$\ell_2 = -1.1001100011100100000011111000010110111101011110010111_2 \times 2^{-55}.$

Here, $\ell_1 + \ell_2$ is equivalent to a precision-109 binary floating-point approximation to $\ln(12)$.

Due to this "wobbling precision" and the fact that the arithmetic algorithms will be either quite complex or slightly inaccurate, double-word arithmetic does not exhibit the "clean, predictable behavior" of a correctly rounded, precision-$2p$, floating-point arithmetic. It might sometimes be very useful (there are nice applications in computational geometry, for instance),

---

[1] Such as 2Sum (Algorithm 4.4, page 130), Fast2Sum (Algorithm 4.3, page 126), Dekker product (Algorithm 4.7, page 135), and 2MultFMA (Algorithm 5.1, page 152).

[2] Of course, when we write $x_h + x_\ell$, the addition symbol corresponds to the exact, mathematical addition.

but it also frequently is a stopgap, and must be used with caution. Li et al. [258] qualify double-double arithmetic as an "attractive nuisance except for the BLAS"[3] and even compare it to an unfenced backyard swimming pool!

When Dekker introduced the *Dekker product* and *Fast2Sum* algorithms in his seminal paper [108], he also suggested ways of performing operations on double-word numbers (addition, multiplication, division, and square root), and he provided an analysis of the rounding error for these operations. For instance, Dekker's algorithm for adding two double-word numbers (x_h, x_ℓ) and (y_h, y_ℓ) is shown in Algorithm 14.1.

Algorithm 14.1 Dekker's algorithm for adding two double-word numbers (x_h, x_ℓ) and (y_h, y_ℓ) [108]. We assume radix 2.

> **if** $|x_h| \geq |y_h|$ **then**
> $\quad (r_h, r_\ell) \leftarrow \text{Fast2Sum}(x_h, y_h)$
> $\quad s \leftarrow \text{RN}(\text{RN}(r_\ell + y_\ell) + x_\ell)$
> **else**
> $\quad (r_h, r_\ell) \leftarrow \text{Fast2Sum}(y_h, x_h)$
> $\quad s \leftarrow \text{RN}(\text{RN}(r_\ell + x_\ell) + y_\ell)$
> **end if**
> $(t_h, t_\ell) \leftarrow \text{Fast2Sum}(r_h, s)$
> **return** (t_h, t_ℓ)

Notice that since the Fast2Sum algorithm is used, we need the radix β to be less than or equal to 3 (which means, in practice, $\beta = 2$).

Dekker's algorithm for multiplying two double-word numbers is shown in Algorithm 14.2 (again, it requires radix 2).

Algorithm 14.2 Dekker's algorithm for multiplying two double-word numbers (x_h, x_ℓ) and (y_h, y_ℓ) [108]. If a fused multiply-add (FMA) instruction is available, it is advantageous to replace DekkerProduct(x_h, y_h) by 2MultFMA(x_h, y_h) (defined in Section 5.1). The radix must be 2.

> $(c_h, c_\ell) \leftarrow \text{DekkerProduct}(x_h, y_h)$
> $p_1 \leftarrow \text{RN}(x_h \cdot y_\ell)$
> $p_2 \leftarrow \text{RN}(x_\ell \cdot y_h)$
> $c_\ell \leftarrow \text{RN}(c_\ell + \text{RN}(p_1 + p_2))$
> $(t_h, t_\ell) \leftarrow \text{Fast2Sum}(c_h, c_\ell)$
> **return** (t_h, t_ℓ)

It is important to understand that these operations are not "correctly

---

[3]BLAS is an acronym for *Basic Linear Algebra Subroutines*.

rounded."[4] As stated above, in most cases, double-word arithmetics do not behave like a precision-$2p$ correctly rounded floating-point arithmetic.

When the IEEE 754-1985 standard for floating-point arithmetic was released, it became much easier to establish simple algorithms that work as soon as an IEEE 754 floating-point arithmetic is available. The operations on double-words became more portable, and libraries were developed to implement this arithmetic. Among the first ones, one can cite Bailey's [180] and Briggs's [47] libraries for "double-double" arithmetic (Briggs has ceased maintaining his library). Recent and efficient functions for double-double arithmetic are included in the QD library by Hida, Li, and Bailey [180].[5]

Several addition and multiplication algorithms are implemented in the QD library. This makes it possible to choose different tradeoffs between speed and accuracy. For instance, the most accurate algorithm for double-word addition in QD, as presented in [258], is shown in Algorithm 14.3.

Algorithm 14.3 The most accurate algorithm implemented in QD for adding two double-word numbers (x_h, x_ℓ) and (y_h, y_ℓ) [258]. A comment in the program at `http://crd.lbl.gov/~dhbailey/mpdist/` attributes this algorithm to Briggs and Kahan. The radix must be 2 (otherwise, the Fast2Sum algorithm cannot be used).

$(s_h, s_\ell) \leftarrow 2\text{Sum}(x_h, y_h)$
$(t_h, t_\ell) \leftarrow 2\text{Sum}(x_\ell, y_\ell)$
$c \leftarrow \text{RN}(s_\ell + t_h)$
$(v_h, v_\ell) \leftarrow \text{Fast2Sum}(s_h, c)$
$w \leftarrow \text{RN}(t_\ell + v_\ell)$
$(z_h, z_\ell) \leftarrow \text{Fast2Sum}(v_h, w)$
return (z_h, z_ℓ)

The multiplication algorithm is shown as Algorithm 14.4.[6] We must warn the reader that it will not be as accurate as a "real" floating-point multiplication of precision twice the basic precision.

---

[4]By the way, "correct rounding" is not clearly defined for double-words: Does this mean that we get the double-word closest to the exact value, or that we get the $2p$-digit number closest to the exact value (if the underlying arithmetic is of precision p)? This can be quite different. For instance, in radix-2, precision-p arithmetic, the double-word closest to $a = 2^p + 2^{-p} + 2^{-p-1}$ is a itself, whereas the precision-$2p$ number closest to a is $2^p + 2^{-p+1}$.

[5]QD is a library for quad-word (as a matter of fact, quad-double) arithmetic. It also includes functions for double-word arithmetic. As we are writing these lines, the QD software is available at `http://crd.lbl.gov/~dhbailey/mpdist/`.

[6]As implemented in `http://crd.lbl.gov/~dhbailey/mpdist/qd-2.3.7.tar.gz`.

Algorithm 14.4 The algorithm implemented in QD for multiplying two double-word numbers (x_h, x_ℓ) and (y_h, y_ℓ). Here, 2Prod is either the Dekker product or 2MultFMA, depending on the availability of an FMA instruction. The radix must be 2 (otherwise, the Fast2Sum algorithm cannot be used).

$(p_h, p_\ell) \leftarrow 2\mathrm{Prod}(x_h, y_h)$
$p_\ell \leftarrow \mathrm{RN}(p_\ell + \mathrm{RN}(x_h \cdot y_\ell))$
$p_\ell \leftarrow \mathrm{RN}(p_\ell + \mathrm{RN}(x_\ell \cdot y_h))$
$(z_h, z_\ell) \leftarrow \mathrm{Fast2Sum}(p_h, p_\ell)$
$\mathrm{return}(z_h, z_\ell)$

14.1.2   Static triple-word arithmetic

For triple-word arithmetic, where numbers are represented as unevaluated sums of three floating-point numbers, the implementation by Lauter [244, 245] is worth mentioning. It is used for handling critical parts in the CRlibm library for correctly rounded elementary functions in double-precision/binary64 floating-point arithmetic. It was specifically designed for the implementation of such functions, which typically require about 120 bits of accuracy. For such accuracies, double-double (i.e., double-word arithmetic based on the double-precision/binary64 format) is not enough, but triple-double, with $3 \times 53 = 159$ bits, is an overkill (if double-extended precision is available, double-word arithmetic based on that format is an interesting alternative). The originality of Lauter's implementation is therefore to allow some overlap in the significands of the three double-precision numbers. Here, two floating-point numbers are said to overlap if their exponent difference is smaller than their significand size.[7] Overlap in a triple-double means that it is not as accurate as it could be. This is acceptable in static code such as polynomial-based elementary function evaluation, for which one knows in advance the number of operations to perform and the accuracy required for each operation.

As the following will show, removing overlap (an operation called renormalization) is expensive. It requires several invocations of the Fast2Sum algorithm, which removes overlap from a double-double. The approach of Lauter is to provide a separate renormalization procedure, so that renormalizations are invoked explicitly in the code, and as rarely as possible.

Each operation is provided with a theorem that expresses a bound on its relative accuracy, as a function of the overlaps of the inputs. For illustration, we give in Algorithm 14.5 and in Theorem 43 one example of an operation implemented by Lauter with its companion theorem.

---

[7]Note that even if the general idea remains the same, the notion of overlap slightly changes depending on the authors!

Algorithm 14.5 Evaluating the sum of a triple-double and a double-double as a triple-double [244].

Require: $a_h + a_\ell$ is a double-double number and $b_h + b_m + b_\ell$ is a triple-double number such that

$$\begin{aligned} |b_h| &\leq 2^{-2} \cdot |a_h| \\ |a_\ell| &\leq 2^{-53} \cdot |a_h| \\ |b_m| &\leq 2^{-\beta_o} \cdot |b_h| \\ |b_\ell| &\leq 2^{-\beta_u} \cdot |b_m| \end{aligned}$$

Ensure: $r_h + r_m + r_\ell$ is a triple-double number approximating $a_h + a_\ell + b_h + b_m + b_\ell$ with a relative error given by Theorem 43.

$(r_h, t_1) \leftarrow \text{Fast2Sum}\,(a_h, b_h)$
$(t_2, t_3) \leftarrow \text{Fast2Sum}\,(a_\ell, b_m)$
$(t_4, t_5) \leftarrow \text{Fast2Sum}\,(t_1, t_2)$
$t_6 \leftarrow \text{RN}(t_3 + b_\ell)$
$t_7 \leftarrow \text{RN}(t_6 + t_5)$
$(r_m, r_\ell) \leftarrow \text{Fast2Sum}\,(t_4, t_7)$

Here, β_o and β_u measure the possible overlap of the significands of the inputs.

Theorem 43 (Relative error of Algorithm 14.5)**.** *If $a_h + a_\ell$ and $b_h + b_m + b_\ell$ are the values in the argument of Algorithm 14.5, such that the preconditions hold, then the values r_h, r_m, and r_ℓ returned by the algorithm satisfy*

$$r_h + r_m + r_\ell = ((a_h + a_\ell) + (b_h + b_m + b_\ell)) \cdot (1 + \epsilon),$$

where ϵ is bounded by

$$|\epsilon| \leq 2^{-\beta_o - \beta_u - 52} + 2^{-\beta_o - 104} + 2^{-153}.$$

The values r_m and r_ℓ will not overlap at all, and the overlap of r_h and r_m will be bounded by

$$|r_m| \leq 2^{-\gamma} \cdot |r_h|$$

with

$$\gamma \geq \min\left(45, \beta_o - 4, \beta_o + \beta_u - 2\right).$$

Lauter's library of triple-double operators is freely available as part of the CRlibm project.[8] It offers about 30 operators that have turned out to be useful for the development of CRlibm functions [100]. Some operators have inputs of different types, as in the previous example. Several implementations of the correct rounding of a triple-double to a double-precision number are also provided. The technical report that describes these operators [244] has been updated and is available as part of the documentation of CRlibm.

---

[8]http://www.ens-lyon.fr/LIP/Arenaire/Ware/

14.1.3   Quad-word arithmetic

Quad-word arithmetic has been implemented in the QD library by Hida, Li, and Bailey [180] (each"word" being a double-precision/binary64 number). A quad-word number is the unevaluated sum of four floating-point numbers (a_0, a_1, a_2, a_3), with the requirement that

$$a_{i+1} \leq \frac{1}{2} \, \mathrm{ulp}(a_i). \tag{14.1}$$

This means that these numbers are "nonoverlapping" in a sense slightly stronger than what we will consider later in Section 14.2. Most of the algorithms of Hida, Li, and Bailey first produce an intermediate result in the form of an unevaluated sum of *five* floating-point numbers (i.e., a five-term "expansion," see Section 14.2). Moreover, these five numbers do not satisfy a requirement such as (14.1): they may have a few "overlapping bits."

   To produce a final quad-word result, it is therefore necessary to perform a "renormalization" step. The following renormalization procedure, presented in [180], is a variant of the renormalization method for expansions introduced by Priest [336]. See Algorithm 14.6.

Algorithm 14.6 Renormalize$(a_0, a_1, a_2, a_3, a_4)$ [180]. The result is $(b_0, b_1, \ldots, b_k)$, with $k \leq 3$ (and almost always $k = 3$). ©IEEE, 2001, with permission.

$(s, t_4) \leftarrow \mathrm{Fast2Sum}(a_3, a_4)$
$(s, t_3) \leftarrow \mathrm{Fast2Sum}(a_2, s)$
$(s, t_2) \leftarrow \mathrm{Fast2Sum}(a_1, s)$
$(s, t_1) \leftarrow \mathrm{Fast2Sum}(a_0, s)$
$k \leftarrow 0$
for $i = 1$ to 4 **do**
  $(s, e) \leftarrow \mathrm{Fast2Sum}(s, t_i)$
  if $e \neq 0$ **then**
    $b_k \leftarrow s$
    $s \leftarrow e$
    $k \leftarrow k + 1$
  end if
end for
return $(b_0, b_1, \ldots, b_k)$

   We should warn the reader that the domain of validity of this renormalization algorithm is not known exactly.

- It was shown by Priest that, in double-precision/binary64 arithmetic, if the terms a_i do not overlap by more than 51 bits,[9] then the terms b_i of the result will satisfy (14.1), i.e., we will get a quad-word.

---

[9]Actually, Priest showed that in precision p, it is sufficient that the terms a_i do not overlap

- And yet, that condition is not a necessary condition, and it is only conjectured that the renormalization algorithm works with the intermediate results produced by the algorithms presented by Hida, Li, and Bailey in [180]. Again, like double-word arithmetic (and probably even more), quad-word arithmetic must be used with much caution.

It is easier to explain these algorithms with drawings (using the notation introduced by Hida, Li, and Bailey). First, the 2Sum algorithm will be represented as shown in Figure 14.1.
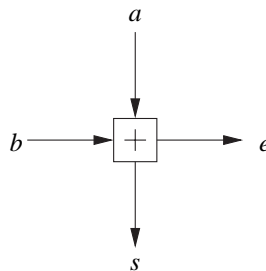
Figure 14.1: The representation of Algorithm 2Sum [180]. Here, $s = \mathrm{RN}(a+b)$, and $s + e = a + b$ exactly.

The rounded-to-nearest floating-point addition and multiplication of two numbers will be represented as shown in Figure 14.2.
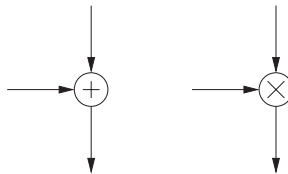
Figure 14.2: The representation of rounded-to-nearest floating-point addition and multiplication [180].

Figure 14.3 represents the simplest of the two algorithms for adding two quad-words presented by Hida, Li, and Bailey in [180].

Concerning the accuracy of that algorithm, Hida, Li, and Bailey have shown the following result.

---

by more than $p - 2$ digits. This result also works in radix β, provided that we replace the calls to Fast2Sum by calls to 2Sum (remember that Fast2Sum works if $\beta \leq 3$).
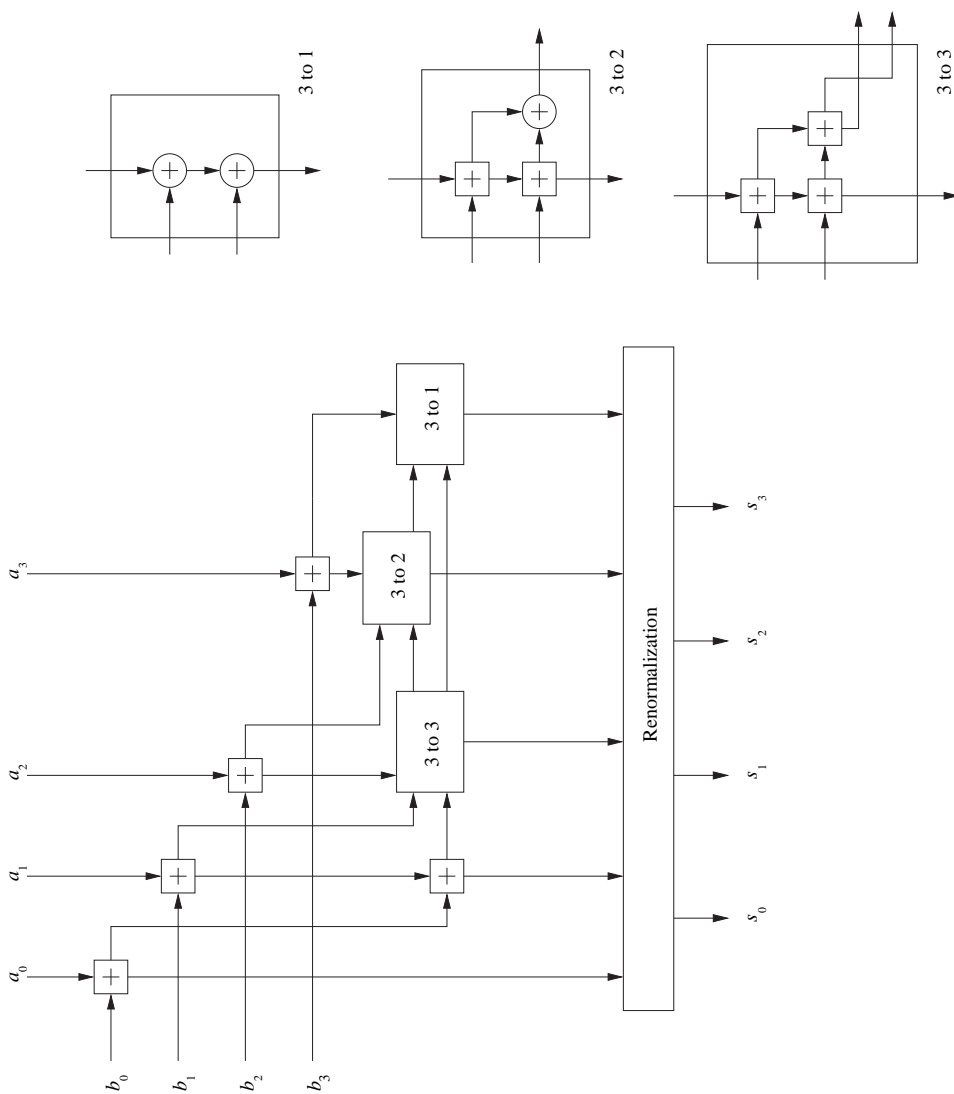
Figure 14.3: SimpleAddQD, the simplest of the two quadword+quadword algorithms presented by Hida, Li, and Bailey [180]. It computes the sum of two quad-words $a = (a_0, a_1, a_2, a_3)$ and $b = (b_0, b_1, b_2, b_3)$. ©IEEE, 2001, with permission.

Theorem 44 (Hida, Li, and Bailey [180])**.** *If the underlying floating-point arithmetic is the double-precision/binary64 format of the IEEE 754 standard, then the five-term expansion σ produced before the normalization step in algorithm SimpleAddQD (described in Figure 14.3) satisfies*

$$|\sigma - (a + b)| \leq 2^{-211} \max\{|a|, |b|\}.$$

If there is much cancellation (i.e., if $|a + b|$ is very small compared to $\max\{|a|, |b|\}$), the relative error on the result may therefore become large. A more accurate algorithm, due to Shewchuk and Boldo, is also presented by Hida, Li, and Bailey in [180]. That algorithm is rather similar to Shewchuk's Fast-Expansion-Sum Algorithm (Algorithm 14.9) presented in the next section.

14.2 Floating-Point Expansions

In the spirit of double-word, triple-word, and quad-word arithmetics, the arithmetic on floating-point expansions was first developed by Priest [336], and in a slightly different way by Shewchuk [377]. To be fair, we must say that Priest's expansion arithmetic actually preceded quad-word arithmetic: for instance, the renormalization algorithm of the QD library (Algorithm 14.6) is directly inspired from a technique due to Priest.

If, starting from some set of floating-point inputs, we only perform exact additions and multiplications, then the values we obtain are always equal to finite sums of floating-point numbers. Such finite sums are called *expansions*. Hence, a natural idea is to try to manipulate such expansions,[10] for performing calculations that are either exact, or approximate yet very accurate. An example of application [377] is the robust implementation of computational geometry algorithms; and another one is the implementation of a few critical parts in very accurate function libraries. Let us now give some definitions.

Definition 18 (Expansion—adapted from Shewchuk's definition [377])**.** *An expansion x is a set of n floating-point numbers $x_1, x_2, \ldots, x_n$ used for representing the real number*

$$x_1 + x_2 + \cdots + x_n.$$

Each x_i is called a component *of x.*

The notion of expansion is intrinsically redundant: a number whose radix-β expansion is finite can be represented by many different expansions. To make expansions useful in practice and easy to manipulate, we must somewhat reduce that amount of redundancy by requiring that the components of an expansion do not *overlap*. There are several slightly different notions of overlapping. One of them is the one that was required for quad-words (Equation (14.1)). Two other definitions of interest are given below.

---

[10]Preferably, *nonoverlapping* expansions, to avoid too much redundancy, as we will see later.

504   Chapter 14. Extending the Precision

Definition 19 (Nonoverlapping floating-point numbers). *Two radix-β, precision-p, floating-point numbers x and y are $\mathcal{S}$-nonoverlapping (that is, nonoverlapping according to Shewchuk's definition [377]) if there exist integers r and s such that $x = r \cdot \beta^s$ and $|y| < \beta^s$, or $y = r \cdot \beta^s$ and $|x| < \beta^s$.*

Assuming x and y have normal representations $m_x \cdot \beta^{e_x}$ and $m_y \cdot \beta^{e_y}$ (with $1 \leq m_x, m_y < \beta$), they are $\mathcal{P}$-nonoverlapping (that is, nonoverlapping according to Priest's definition [337]) if $|e_y - e_x| \geq p$.

Notice that zero is $\mathcal{S}$-nonoverlapping with any nonzero floating-point number.

For example, in a floating-point system of radix $\beta = 10$ and precision $p = 4$, the numbers 1.200×10^3 and 1.500×10^1 are $\mathcal{S}$-nonoverlapping, whereas they are not $\mathcal{P}$-nonoverlapping.

Definition 20 (Nonoverlapping expansions). *An expansion is $\mathcal{S}$-nonoverlapping (that is, nonoverlapping according to Shewchuk's definition [377]) if all of its components are mutually $\mathcal{S}$-nonoverlapping. It is $\mathcal{P}$-nonoverlapping (that is, nonoverlapping according to Priest's definition [337]) if all of its components are mutually $\mathcal{P}$-nonoverlapping.*

In general, a $\mathcal{P}$-nonoverlapping expansion with m components carries more information than an $\mathcal{S}$-nonoverlapping expansion with m components, since Priest's condition for nonoverlapping is stronger than Shewchuk's condition. In a pinch, in extreme cases, in radix 2, an $\mathcal{S}$-nonoverlapping expansion with 53 components may not contain more information than one double-precision number (it suffices to put each bit of a floating-point number in a separate component). And yet, $\mathcal{S}$-nonoverlapping expansions are of much interest, because the arithmetic algorithms that make it possible to manipulate them are generally simpler than the algorithms used for $\mathcal{P}$-nonoverlapping expansions.

Definition 21 (Nonadjacent expansions). *Following Shewchuk's definition [377], we say that two floating-point numbers x and y are adjacent if they $\mathcal{S}$-overlap, or if βx and y $\mathcal{S}$-overlap, or if x and βy $\mathcal{S}$-overlap. An expansion is nonadjacent if no two of its components are adjacent.*

In radix 2, every expansion can be transformed into a nonadjacent expansion that represents the same real number. For instance, with $\beta = 2$ and $p = 5$, the $\mathcal{S}$-nonoverlapping expansion:

$$\left(1.0111_2 \times 2^{12}\right) + \left(1.01_2 \times 2^7\right) + \left(1.1001_2 \times 2^4\right) = 6073_{10}$$

can be transformed into the following nonadjacent expansion:

$$\left(1.1_2 \times 2^{12}\right) - \left(1.0_2 \times 2^6\right) - \left(1.11_2 \times 2^2\right) = 6073_{10}.$$

Shewchuk also designates a *strongly nonoverlapping expansion* (in radix 2) an expansion such that no two of its components are $\mathcal{S}$-overlapping, no component is adjacent to two other components, and whenever there is a pair of adjacent components, both are powers of 2.

Now let us give examples of algorithms for performing arithmetic operations on expansions. These algorithms are based on the 2Sum, Fast2Sum, and Dekker product or 2MultFMA algorithms presented in Chapters 4 and 5. We first show Algorithms 14.7 and 14.8.

Algorithm 14.7 Shewchuk's Grow-Expansion algorithm [377]. Input values: an $\mathcal{S}$-nonoverlapping expansion e of m components and a floating-point number b. Output value: an $\mathcal{S}$-nonoverlapping expansion e of $m + 1$ components. We assume that the radix is 2 and that the precision p satisfies $p \geq 3$.

$Q_0 \leftarrow b$
for $i = 1$ to m **do**
$\quad (Q_i, h_i) \leftarrow 2\text{Sum}(Q_{i-1}, e_i)$
end for
$h_{m+1} \leftarrow Q_m$
return h

Algorithm 14.8 Shewchuk's Expansion-Sum algorithm [377]. Input values: an $\mathcal{S}$-nonoverlapping m-component expansion e and an $\mathcal{S}$-nonoverlapping n-component expansion f. Output value: an $\mathcal{S}$-nonoverlapping $m + n$-component expansion h.

$h \leftarrow e$
for $i = 1$ to n **do**
$\quad (h_i, h_{i+1}, \ldots, h_{i+m}) \leftarrow \text{Grow-Expansion}((h_i, h_{i+1}, \ldots, h_{i+m-1}), f_i)$
end for
return h

Notice that the sum of an m-component expansion and an n-component expansion, if calculated with Algorithm 14.8, will be an $(m + n)$-component expansion. Sometimes, this growth in the number of components is unavoidable. For instance, if the expansions e and f satisfy

$$\forall i, |e_i| < \tfrac{1}{2}\,\text{ulp}(e_{i+1}),$$

$$|e_m| < \tfrac{1}{2}\,\text{ulp}(f_1),$$

$$\forall i, |f_i| < \tfrac{1}{2}\,\text{ulp}(f_{i+1}),$$

then the sum of e and f cannot be expressed with fewer than $m + n$ components. And yet, in most cases, that sum could be expressed with a much smaller number of components: to avoid a useless growth in the number of

components that would make expansion arithmetic very inefficient after a
very few operations, one must frequently *compress* the expansions. This can
be done using Algorithm 14.11, page 508.

Theorem 45 (Shewchuk [377]). *Assume the radix is 2, and that the rounding mode
is round to nearest even. Let $e = \sum_{i=1}^{m} e_i$ and $f = \sum_{i=1}^{n} f_i$ be $\mathcal{S}$-nonoverlapping
expansions of m and n floating-point numbers of precision $p \geq 3$. Suppose that
the components of e and f are sorted in order of* increasing magnitude, *except that
any of these components may be zero. Algorithm 14.8 produces an $\mathcal{S}$-nonoverlapping
expansion h such that*

$$h = \sum_{i=1}^{m+n} h_i = e + f,$$

*where the components of h are also sorted in order of increasing magnitude, except
that any of them may be zero. Furthermore, if e and f are nonadjacent, then h is
nonadjacent.*

Beware: the components of the quad-words, in the previous section,
were numbered in order of decreasing magnitude. Here, we assume that the
components of Shewchuk's expansions are numbered in order of increasing
magnitude, mainly because it simplifies the presentation of the algorithms: in
most cases, the least significant component is calculated first, and it is much
simpler to immediately assign it the number 1.

Let us now give a faster algorithm for adding expansions. Algo-
rithm 14.9 is similar to an algorithm due to Priest [336]. Notice that if the
two input expansions are $\mathcal{S}$-nonoverlapping, we do not necessarily get an
$\mathcal{S}$-nonoverlapping result (Priest showed that in radix 2 the components of the
result may overlap by at most one bit). And yet, if the two input expansions
are *strongly nonoverlapping*, then the result will be strongly nonoverlapping
too. Again, the number of components of the result is equal to the sum of the
number of components of the input expansions: it is necessary to "compress"
the expansions from time to time. This can be done using Algorithm 14.11,
page 508.

Algorithm 14.9 Shewchuk's Fast-Expansion-Sum algorithm [377].

Merge e and f into a single sequence g, in order of nondecreasing
magnitude
$(Q_2, h_1) \leftarrow \text{Fast2Sum}(g_2, g_1)$
for $i = 3$ to $m + n$ **do**
    $(Q_i, h_{i-1}) \leftarrow \text{2Sum}(Q_{i-1}, g_i)$
end for
$h_{m+n} \leftarrow Q_{m+n}$
return h

Theorem 46 (Shewchuk [377]). *Assume the radix is 2 and that the rounding mode is round to nearest even. Let $e = \sum_{i=1}^{m} e_i$ and $f = \sum_{i=1}^{n} f_i$ be strongly nonoverlapping expansions of m and n floating-point numbers of precision $p \geq 4$. Suppose that the components of e and f are sorted in order of increasing magnitude, except that any of these components may be zero. Algorithm 14.9 produces a strongly nonoverlapping expansion h such that*

$$h = \sum_{i=1}^{m+n} h_i = e + f,$$

where the components of h are also sorted in order of increasing magnitude, except that any of them may be zero.

Algorithm 14.10 computes the product of an expansion by one floating-point number. It is the basic building block of an expansion multiplication algorithm.

Algorithm 14.10 Shewchuk's Scale-Expansion algorithm [377]. Here, 2Prod is either the Dekker product or 2MultFMA, depending on the availability of an FMA instruction.

$(Q_2, h_1) \leftarrow 2\mathrm{Prod}(e_1, b)$
for $i = 2$ to m **do**
    $(T_i, t_i) \leftarrow 2\mathrm{Prod}(e_i, b)$
    $(Q_{2i-1}, h_{2i-2}) \leftarrow 2\mathrm{Sum}(Q_{2i-2}, t_i)$
    $(Q_{2i}, h_{2i-1}) \leftarrow \mathrm{Fast2Sum}(T_i, Q_{2i-1})$
end for
$h_{2m} \leftarrow Q_{2m}$
return h

Algorithm 14.10 may seem quite complex. It is more understandable using a graphic representation. Such a representation (due to Shewchuk) is given in Figure 14.4.

Shewchuk shows that if $e = \sum_{i=1}^{m} e_i$ is a nonoverlapping expansion of m (binary) floating-point numbers sorted in order of increasing magnitude,[11] if b is a floating-point number, and if the precision p is larger than or equal to 4 (which always holds in practice), then Algorithm 14.10 produces a nonoverlapping expansion h of $2m$ components, equal to $b \cdot e$.

An algorithm such as 14.10 may be used iteratively to produce the product of two expansions. However, this would create an expansion with too many terms. Again, we need to an algorithm for "compressing" expansions. Algorithm 14.11 was introduced by Shewchuk [377].

---

[11]Except that any of the e_i may be zero.

Figure 14.4: Graphic representation of Shewchuk's Scale-Expansion Algorithm [377]
(Algorithm 14.10).

Algorithm 14.11 Shewchuk's compression algorithm [377] (there was a small
typo in [377]: line 15 of this algorithm was written $h_t \leftarrow Q$ instead of $h_t \leftarrow q$).
The input value is an m-component expansion e, and the output value is an
n-component expansion h that represent the same real number.

$Q \leftarrow e_m$
$b \leftarrow m$
for $i = m - 1$ **downto** 1 **do**
   $(Q, q) \leftarrow \text{Fast2Sum}(Q, e_i)$
   if $q \neq 0$ **then**
      $g_b \leftarrow Q$
      $b \leftarrow b - 1$
      $Q \leftarrow q$
   end if
end for
$t \leftarrow 1$
for $i = b + 1$ **to** m **do**
   $(Q, q) \leftarrow \text{Fast2Sum}(g_i, Q)$
   if $q \neq 0$ **then**
      $h_t \leftarrow q$
      $t \leftarrow t + 1$
   end if
end for
$h_t \leftarrow Q$
$n \leftarrow t$
return h, n

Theorem 47 (Shewchuk [377]). *If $e = \sum_{i=1}^{m} e_i$ is a nonoverlapping expansion
of $m \geq 3$ (binary) floating-point numbers, sorted in order of increasing magnitude
(except that any of the e_i may be zero), then Algorithm 14.11 produces a nonoverlap-*

ping, n-component expansion[12] h, such that $\sum_{i=1}^{n} h_i = e$, where the h_i are sorted in order of increasing magnitude. If $e \neq 0$ then none of the h_i is zero, and h_n is within one $\mathrm{ulp}(h_n)$ from e.

Let us now give an example that illustrates these algorithms.

Example 13 (Manipulation of expansions using Shewchuk's algorithms)**.**
Assume that the underlying basic precision floating-point arithmetic is the single-precision/binary32 format of the IEEE 754 standard. Consider the 3-component, nonoverlapping expansion e:

$$
\begin{aligned}
e_1 &= -1.11101110010110011101101_2 \times 2^{-49} \\
e_2 &= -1.01110111011111010010111_2 \times 2^{-24} \\
e_3 &= 11.0010010000111111011011_2
\end{aligned}
$$

and the 2-component, nonoverlapping expansion f:

$$
\begin{aligned}
f_1 &= 1.10011111110011101111101_2 \times 2^{-25} \\
f_2 &= 1.0110101000001001110011_2.
\end{aligned}
$$

If we add e and f using the Fast-Expansion-Sum algorithm (Algorithm 14.9, page 506), we get the following 5-component expansion h:

$$
\begin{aligned}
h_1 &= -1.11101110010110011101101_2 \times 2^{-49} \\
h_2 &= 1.0 \times 2^{-48} \\
h_3 &= 1.111000001111001011_2 \times 2^{-25} \\
h_4 &= 0 \\
h_5 &= 100.10001110010010010101_2.
\end{aligned}
$$

Now, if we "compress" this expansion using Algorithm 14.11, we get the following 3-component expansion z:

$$
\begin{aligned}
z_1 &= 1.00011010011000010011_2 \times 2^{-53} \\
z_2 &= 1.111000001111001011_2 \times 2^{-25} \\
z_3 &= 100.10001110010010010101_2.
\end{aligned}
$$

One can easily check that $z = e + f$.

14.3 Floating-Point Numbers with Batched Additional Exponent

Another limitation of the basic-precision floating-point formats is their restricted exponent range. For instance, in double-precision/binary64

---

[12]Furthermore, it is a nonadjacent expansion if the round-to-nearest-even rounding mode is used.

arithmetic, the smallest representable number is

$$\alpha = 2^{-1074}$$

and the largest one is

$$\Omega = 2^{1023}(2 - 2^{-52}).$$

For example, this restriction may prevent one from converting very large integers into floating-point numbers. If this is deemed necessary, there are two main ways to work around that restriction. The easiest way is to use an arbitrary-precision floating-point library, since for these libraries, the exponent of the considered floating-point number is most often stored in a 32- or 64-bit integer, which allows much larger numbers. However, this is an expensive solution since arbitrary-precision numbers can be significantly more expensive to manipulate than basic-precision floating-point numbers, and in our case, a larger precision may not be needed.

Another usual approach is to batch each floating-point number f with an integer e. The represented number is then $f \cdot \beta^e$ (where β is the radix of the floating-point system), which we will denote by (f, e). A given number has several representations, due to the presence of two exponents, the one contained in f, and the additional one, i.e., e. For example, assuming $\beta = 2$, $(1.0, 0) = (0.5, 1)$. To make the representation unique, one may require f to be between two given consecutive powers of β (and thus make the internal exponent useless). Normalizing a pair (f, e) into an equivalent one (f', e') such that f' belongs to the prescribed domain can be performed by using a simple exponent manipulation; for example, with the C function ldexp. This normalization has the advantage of making all basic arithmetic operations simpler. Basic arithmetic operations are rather simple to program. Such numbers are available in the dpe library of Pélissier and Zimmermann [329], in the INTLAB[13] library of Rump, and internally in many libraries, including Magma [40] and NTL [379][14] (in which it corresponds to the XD class). There is also a staggered interval arithmetic implemented in the C-XSC language [28].

14.4 Large Precision Relying on Processor Integers

The most common approach to obtain arbitrary-precision arithmetic consists in representing a floating-point number as an integer of arbitrary length along with an exponent. The bit length of the integer is usually chosen to

---

[13]INTLAB is the Matlab toolbox for self-validating algorithms, it is available at

http://www.ti3.tu-harburg.de/rump/intlab/.

[14]NTL is a library for performing number theory, available at http://www.shoup.net/ntl/.

be a multiple of the size of the machine words slightly larger than the desired precision. Typically, in that setting, an arbitrary-precision floating-point number consists of a sign, an exponent, an array or a list of machine integers[15] that encode the significand, and possibly a small integer that encodes the precision of that particular floating-point number (if the precision is not a global attribute). Multiple-precision integer arithmetic is thus used to manipulate the significands, and usually dominates the costs, in particular when the precision is large (i.e., a significant number of times the length of machine words).

This approach has been used since the early days of programming. For instance, Brent [45] used it in the mid-1970s for his MP library written in FORTRAN. Bailey et al. [22] pursued MP in ARPREC.[16] Nowadays, the MPFR[17] library [137] also relies on this principle. MPFR extends the spirit of the IEEE 754 standard (correct rounding, special data such as infinities and NaNs, and exceptions) to arbitrary-precision floating-point arithmetic. It provides the basic arithmetic operations, many elementary functions, and even several special functions, with correct rounding to any precision.

In that context, arbitrary-precision integer arithmetic is used to implement the floating-point arithmetic operations. Then these operations are used to implement other functions, for example, elementary or special functions. After a brief discussion on the specifications, we describe how to use integer arithmetic to perform the basic floating-point arithmetic operations. Then we quickly review arbitrary-precision integer arithmetic and finally describe some general techniques used to implement mathematical functions in the context of arbitrary-precision floating-point arithmetic.

Specifications

Often, on recent packages, arbitrary-precision floating-point numbers have been implemented in such a way that they can be viewed as "smooth extensions" of the fixed precisions of the IEEE 754-1985 standard. In particular, if we require the precision to be the same as in one of the basic formats of the standard, one should essentially see the behavior described in the standard. For instance, the RR module of NTL implements multiple-precision real numbers with correctly rounded-to-nearest arithmetic operations. For the transcendental functions, correct rounding is not guaranteed, but the computed result has a relative error less than 2^{-p+1}, where p is the current precision. However, in NTL, there are no special values such as $\pm\infty$ or NaN. The MPFR library is a multiple-precision library that offers correct rounding even

---

[15]In a very similar way, native floating-point data representing integers can be used instead of machine integers, e.g., double-precision numbers, whose value is an integer between 0 and $2^{53} - 1$.

[16]ARPREC is available at `http://crd.lbl.gov/~dhbailey/mpdist/`.

[17]MPFR is available at `http://www.mpfr.org/`.

for the transcendental functions (this is possible using Ziv's technique; see Chapter 12, page 408).

In the future, this similarity of behavior should become even more true now that the new IEEE 754-2008 standard defines interchange formats for extended and extendable precision (see Section 3.4.5, page 92).

As a basic-precision floating-point number, an arbitrary-precision floating point number is made of a sign, a significand, and an exponent. Note that a clear difference from conventional precisions is that underflows and overflows are less likely to occur since the exponent is usually stored in a separate integer. However, the exceptional cases may occur anyway, and are usually handled in a way similar to what is suggested by the IEEE 754-1985 standard.

Two strategies exist for handling the precision p. First, it may be a global variable. In that case, at a given time, all floating-point numbers have the same precision, which may be changed over time. This strategy is used for NTL's RR class [379]. The second strategy (available for example in MPFR [137]) consists in batching each floating-point number with a local precision. This can be significantly more efficient, because the user can allocate large precisions only for the variables for which it is deemed necessary, and low precisions for the other ones. However, this makes the semantics and the implementation more complicated: the precisions of the operands as well as the precision of the target must be considered, with different cases that must be handled carefully.

The major purpose of multiple-precision libraries is often to allow for diverse and reliable mathematical computations, which explains why they frequently offer implementations of many mathematical functions. The most frequent ones, of course, are trigonometric functions, hyperbolic functions, exponentials and logarithms, and more elaborate special functions such as the gamma and zeta functions, Bessel functions, and generalizations thereof.

14.4.1   Using arbitrary-precision integer arithmetic for arbitrary-precision floating-point arithmetic

Suppose we are trying to perform a basic operation op $\in \{+, \times, \div\}$ on arbitrary-precision floating-point numbers. We first perform a related integer operation on the significands, and then postprocess the integer result to obtain the floating-point result.

Let a and b be two radix-2 floating-point numbers of precisions p_a and p_b, respectively. Suppose we want to compute $c = \circ(a \text{ op } b)$ for some precision p_c, some rounding mode $\circ$, and some basic operation op $\in \{+, \times, \div\}$. The numbers $a, b,$ and c can be written:

$$x = (-1)^{s_x} \cdot m_x \cdot 2^{e_x}, \quad \text{for } x \in \{a, b, c\},$$

where m_x is an integer whose bit length is close to p_x, s_x belongs to $\{0, 1\}$, and e_x is a small integer.

As in Chapter 8, the addition of a and b reduces to the integer addition or subtraction of m_a with a shifted value of m_b, the amount of the shift being determined by the difference between e_a and e_b. Similarly, the multiplication of a and b reduces to the integer multiplication of m_a and m_b. Finally, the division of a by b can be reduced to the division of a shifted value of m_a by m_b.

Note that, in all cases, only the first p_c bits of the result of the integer operation are actually needed to produce most of m_c. A few additional bits may be needed to determine the correctly rounded result. In the case of floating-point multiplication, this remark can be used to save a possibly significant proportion of the overall cost. Suppose for example that the input and target precisions match. Then only the most significant half (plus a few) of the bits of the integer product of the two significands is actually needed. Krandick and Johnson [231] designed an efficient algorithm to compute that half of the bits. It was later improved by Mulders [288] and then by Hanrot and Zimmermann [164]. In the last two references, the authors concentrate on the lower half of the product of two polynomials, but their analyses extend to the upper half of the product of two integers (with complications due to the propagation of carries).

14.4.2 A brief introduction to arbitrary-precision integer arithmetic

Arbitrary-precision integer arithmetic has been extensively studied for more than 45 years [214, 409, 368, 366, 367, 141]. A big integer is most often stored as a list or an array of processor integers. Big integer arithmetic is usually much slower than processor integer arithmetic, even when the "big integers" turn out to be small enough to be representable by processor integers. A way of using the best of both worlds is the following. Consider a machine integer. If its first (or last) bit is 1, the other bits encode the address of a multiple-precision integer; otherwise, they encode a small integer. This trick is used, e.g., for the integers of the Magma[18] computational algebra system [40]. The overhead for small integers can thus be significantly decreased.

Adding or subtracting arbitrary-precision integers is relatively straight-forward. The algorithmic aspects become significantly more involved when one considers multiplication and division. Here we will consider only multiplication. Note that any multiplication algorithm can be turned into a division algorithm via the Newton–Raphson iteration (see Section 5.3, page 155), with a constant factor overhead in the asymptotic complexity, thanks to the quadratic convergence (see [43] for an extensive discussion). There exists a full hierarchy of multiplication algorithms. By decreasing complexity upper bounds, with n being the maximum of the bit lengths of the two integers to be multiplied, we have:

---

[18]Magma is available at `http://magma.maths.usyd.edu.au/magma/`.

- the "school-book" multiplication, with complexity $O(n^2)$;

- Karatsuba's multiplication [214], with complexity $O(n^{\log_2 3})$;

- the Toom-Cook-3 multiplication [409], with complexity $O(n^{\log_3 5})$;

- its higher-degree generalizations, with complexities $O(n^{\log_4 7})$, $O(n^{\log_5 9})$, ...;

- and finally the fast multiplication of Schönhage and Strassen, which relies on the discrete Fourier transform [368], with complexity $O(n \log n \log \log n)$.

Note that integer multiplication is still the focus of active research, and has been recently improved (at least theoretically) by Fürer [141]. By changing the ring to which the Fourier transform maps the integers to multiply, the author was able to obtain an $O(n \log n \log^* n)$ complexity bound, where $\log^* n$ is the number of times the $\log$ function has to be applied recursively to obtain a number below 1. Several practical improvements over the Schönhage–Strassen algorithm have also been reported recently by Gaudry, Kruppa, and Zimmermann [144].

In Table 14.1, we give a list of some multiplication algorithms, with their asymptotic complexities. We also give the approximate numbers of machine words that the involved integers require for any given algorithm to become more efficient in practice than the algorithms above it in the table. These thresholds derive from the GNU MP library [156]. See also some experiments by Zuras [446].

Algorithm	Asymptotic complexity	GNU MP thresholds
School-book	$O(n^2)$	-
Karatsuba	$O(n^{\log_2 3})$	$[10, 20]$
Toom–Cook-3	$O(n^{\log_3 5})$	$[100, 200]$
Schönhage–Strassen	$O(n \log n \log \log n)$	$[5000, 10000]$
Fürer	$O(n \log n \log^* n)$	unknown

Table 14.1: Asymptotic complexities of multiplication algorithms and approximate practical thresholds in numbers of machine words.

The fast multiplication algorithms rely on the evaluation-interpolation paradigm. We explain it briefly with Karatsuba's multiplication, and refer to [27] for a detailed survey on integer multiplication algorithms. Suppose we want to multiply two positive integers a and b, that are both n bits long. Karatsuba's algorithm first splits a and b into almost equal sizes: a is written $a_1 2^{\lceil n/2 \rceil} + a_0$ and similarly b is written $b_1 2^{\lceil n/2 \rceil} + b_0$, where a_0 and b_0 are

both in $[0, 2^{\lceil n/2 \rceil} - 1]$. The product $c = a \cdot b$ can be expressed as:

$$c = c_2 \cdot 2^{2 \lceil n/2 \rceil} + c_1 \cdot 2^{\lceil n/2 \rceil} + c_0,$$

where $c_2 = a_1 \cdot b_1$, $c_1 = a_1 \cdot b_0 + a_0 \cdot b_1$, and $c_0 = a_0 \cdot b_0$. Using these formulas as such leads to an algorithm for multiplying two n-bit-long integers that uses four multiplications of numbers of size around $n/2$. Applying this idea recursively, i.e., replacing (a, b) by (a_1, b_1), (a_1, b_0), (a_0, b_1), and (a_0, b_0) respectively, leads to a multiplication algorithm of complexity $O(n^2)$, which is the complexity of the school-book algorithm. Indeed, if C_n is the cost of computing the product of two n-bit-long integers with this algorithm, then we just proved that $C_n = 4C_{n/2} + O(n)$, which leads to $C_n = O(n^2)$. Karatsuba's contribution was to notice that in order to obtain c_0, c_1, and c_2, we only need three multiplications of numbers of size around $n/2$ instead of four, because of the formulas:

$$c_2' = a_1 \cdot b_1 \quad c_1' = (a_1 + a_0) \cdot (b_1 + b_0) \quad c_0' = a_0 \cdot b_0$$
$$c_2 = c_2' \qquad c_1 = c_1' - c_2' - c_0' \qquad c_0 = c_0'.$$

We therefore readily deduce that only three multiplications of numbers of size around $n/2$ are needed to perform the product of two numbers of size n (as well as a few additions, which turn out to be negligible in the cost). If we apply these formulas recursively (for the computation of the c_i''s), then we obtain $C_n = 3C_{n/2} + O(n)$, which leads to an $O\left(n^{\log_2 3}\right) \approx O\left(n^{1.585}\right)$ asymptotic complexity bound.

These formulas may seem extraordinary at first sight, but they can be interpreted in an elegant mathematical way. Replace $2^{\lceil n/2 \rceil}$ by an indeterminate x. Then the integers a, b, and c become three polynomials:

$$a(x) = a_1 x + a_0,$$
$$b(x) = b_1 x + b_0,$$

and

$$c(x) = c_2 x^2 + c_1 x + c_0.$$

If we evaluate $c(x)$ for $x = 0$, $x = 1$, and $x = +\infty$ (more precisely, at $+\infty$, we take the dominant coefficient), then we obtain:

$$
\begin{aligned}
c(0) = c_0 &= c_0', \\
c(1) = c_2 + c_1 + c_0 &= c_1', \\
c(+\infty) = c_2 &= c_2'.
\end{aligned}
$$

The c_i''s are the evaluations of $c(x)$ at the three points we selected. Furthermore, since $c = a \cdot b$ and $c(x) = a(x) \cdot b(x)$, we have:

$$
\begin{aligned}
c_0' = a(0) \cdot b(0) &= a_0 \cdot b_0, \\
c_1' = a(1) \cdot b(1) &= (a_0 + a_1) \cdot (b_0 + b_1), \\
c_2' = a(+\infty) \cdot b(+\infty) &= a_1 \cdot b_1.
\end{aligned}
$$

Overall, Karatsuba's algorithm can be summarized as follows: evaluate a and b at points 0, 1, and $+\infty$; by taking the products, obtain the evaluations of c at these points; finally, interpolate the evaluations to recover c. The costly step in Karatsuba's multiplication is the computation of the evaluations of c from the evaluations of a and b.

This evaluation-interpolation principle was also generalized with more than three evaluation points. The Toom-Cook-3 multiplication algorithm consists in splitting a and b into three equal parts (instead of two as in the previous algorithm), and using five evaluation points, for example, 0, 1, -1, -2, and $+\infty$. The multiplication of n-bit-long operands can then be performed with 5 multiplications between operands of bit-size around $n/3$. This provides an asymptotic complexity bound $O(n^{\log_3 5}) \approx O(n^{1.465})$. By letting the number of evaluation points grow to infinity, one can obtain multiplication algorithms of complexity $O(n^{1+\epsilon})$ for any $\epsilon > 0$. However, when n becomes large, it is more interesting to consider particular evaluation points, namely, roots of unity, which gives rise to the Fourier transform-based multiplication of Schönhage and Strassen [368].

Part VI

Perspectives and Appendix

Chapter 15

Conclusion and Perspectives

With the recent IEEE standard update resulting in IEEE 754-2008, computer arithmetic will soon see important changes: the fused multiply-add (FMA) instruction will probably be available on most processors, and correctly rounded functions (at least in some domains) and decimal arithmetic will be provided on most systems. Also, it should be easier to specify whether one wishes the implicit intermediate variables of an arithmetic expression to be computed and stored in the largest available format (to improve the accuracy of the calculations) or in a format clearly specified in the program (to enhance software portability and provability).

We hope that future language standards will allow the users to easily and efficiently take advantage of the various possibilities offered by IEEE 754-2008.

Some instructions might ease the task of programmers. For instance, if, instead of having to use algorithms such as 2Sum (Algorithm 4.4, page 130) or 2MultFMA (Algorithm 5.1, page 152), the corresponding operations (i.e., the exact error of a floating-point addition or multiplication) were implemented on the floating-point units, we could much more efficiently compute accurate sums of many numbers or implement multiple-precision arithmetic. This should not complicate the existing architectures too much, since, to guarantee correctly rounded results, a conventional floating-point adder or multiplier already has to do a large part of the task. Also (but this might be slightly more complex to implement in hardware), having instructions for $\circ(x+y+z)$ or $\circ(xy+zt)$ would make very accurate functions much easier to program.

Concerning the Table Maker's Dilemma, the current methods will make it possible to know the worst cases for most univariate functions in the binary64 or decimal64 formats. For much wider formats (typically, binary128 or decimal128), unless new algorithms are found, we have no hope of determining the worst cases in the foreseeable future. And yet, methods based on the Lenstra–Lenstra–Lovász (LLL) algorithm (see the Appendix) might soon allow us to prove information of the kind "we do not know the hardness to round, but it is less than 400."

Chapter 16

Appendix: Number Theory Tools for Floating-Point Arithmetic

16.1 Continued Fractions

Continued fractions make it possible to build very good (indeed, the best possible, in a sense that will be made explicit by Theorems 49 and 50) rational approximations to real numbers. As such, they naturally appear in many problems of number theory, discrete mathematics, and computer science. Since floating-point numbers are rational approximations to real numbers, it is not surprising that continued fractions play a role in some areas of floating-point arithmetic.

Excellent surveys can be found in [166, 384, 331, 218]. Here, we will just present some general definitions, as well as the few results that are needed in this book, especially in Chapters 5 and 11.

Let α be a real number. From α, consider the two sequences (a_i) and (r_i) defined by

$$
\begin{cases}
r_0 & = \quad \alpha, \\[2mm]
a_i & = \quad \lfloor r_i \rfloor, \\[2mm]
r_{i+1} & = \quad \dfrac{1}{r_i - a_i},
\end{cases}
\tag{16.1}
$$

where "$\lfloor . \rfloor$" is the usual floor function. Notice that the a_i'(s are integers and that the r_i's are real numbers.

If α is an irrational number, then these sequences are defined for any

$i \geq 0$ (i.e., r_i is never equal to a_i), and the rational number

$$\frac{P_i}{Q_i} = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \cfrac{1}{\ddots + \cfrac{1}{a_i}}}}}$$

is called the i-th *convergent* of α. The a_i's constitute the *continued fraction expansion* of α. We write

$$\alpha = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \cdots}}}$$

or, to save space,

$$\alpha = [a_0; a_1, a_2, a_3, a_4, \ldots].$$

If α is rational, then these sequences terminate for some k, and $P_k/Q_k = \alpha$ exactly. The P_i's and the Q_i's can be deduced from the a_i's using the following recurrences:

$$\begin{cases} P_0 &= a_0, \\ P_1 &= a_1 a_0 + 1, \\ Q_0 &= 1, \\ Q_1 &= a_1, \\ P_k &= P_{k-1} a_k + P_{k-2} \text{ for } k \geq 2, \\ Q_k &= Q_{k-1} a_k + Q_{k-2} \text{ for } k \geq 2. \end{cases}$$

Note that these recurrences give irreducible fractions P_i/Q_i: the values P_i and Q_i that are deduced from them satisfy $\gcd(P_i, Q_i) = 1$.

The major interest in the continued fractions lies in the fact that P_i/Q_i is the best rational approximation to α among all rational numbers of denominator less than or equal to Q_i. More precisely, we have the following two results [166].

Theorem 48. *([166, p.151]) Let $(P_j/Q_j)_{j \geq 0}$ be the convergents of α. If a rational number P/Q is a better approximation to α than P_k/Q_k (namely, if $|P/Q - \alpha| < |P_k/Q_k - \alpha|$), then $Q > Q_k$.*

Theorem 49. *([166, p.151]) Let $(P_j/Q_j)_{j \geq 0}$ be the convergents of α. If Q_{k+1} exists, then for any $(P, Q) \in \mathbb{Z} \times \mathbb{N}^*$, with $Q < Q_{k+1}$, we have*

$$|P - \alpha Q| \geq |P_k - \alpha Q_k|.$$

If Q_{k+1} does not exist (which implies that α is rational), then the previous inequality holds for any $(P, Q) \in \mathbb{Z} \times \mathbb{N}^*$.

Interestingly enough, a kind of converse result exists: if a rational approximation to some number α is extremely good, then it must be a convergent of its continued fraction expansion.

Theorem 50. *([166, p.153]) Let P, Q be integers, $Q \neq 0$. If*

$$\left| \frac{P}{Q} - \alpha \right| < \frac{1}{2Q^2},$$

then P/Q is a convergent of α.

An example of continued fraction expansion of an irrational number is

$$e = \exp(1) = 2 + \cfrac{1}{1 + \cfrac{1}{2 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{4 + \cfrac{1}{\ddots}}}}}} = [2; 1, 2, 1, 1, 4 \ldots]$$

which gives the following rational approximations to e:

$$\frac{P_0}{Q_0} = 2, \quad \frac{P_1}{Q_1} = 3, \quad \frac{P_2}{Q_2} = \frac{8}{3}, \quad \frac{P_3}{Q_3} = \frac{11}{4}, \quad \frac{P_4}{Q_4} = \frac{19}{7}, \quad \frac{P_5}{Q_5} = \frac{87}{32}.$$

Other examples are

$$\pi = 3 + \cfrac{1}{7 + \cfrac{1}{15 + \cfrac{1}{1 + \cfrac{1}{292 + \cfrac{1}{1 + \cfrac{1}{\ddots}}}}}} = [3; 7, 15, 1, 292, 1 \ldots]$$

and

$$\sqrt{2} = 1 + \cfrac{1}{2 + \cfrac{1}{2 + \cfrac{1}{2 + \cfrac{1}{2 + \cfrac{1}{\ddots}}}}} = [1; 2, 2, 2, 2, \ldots] = [1; \overline{2}].$$
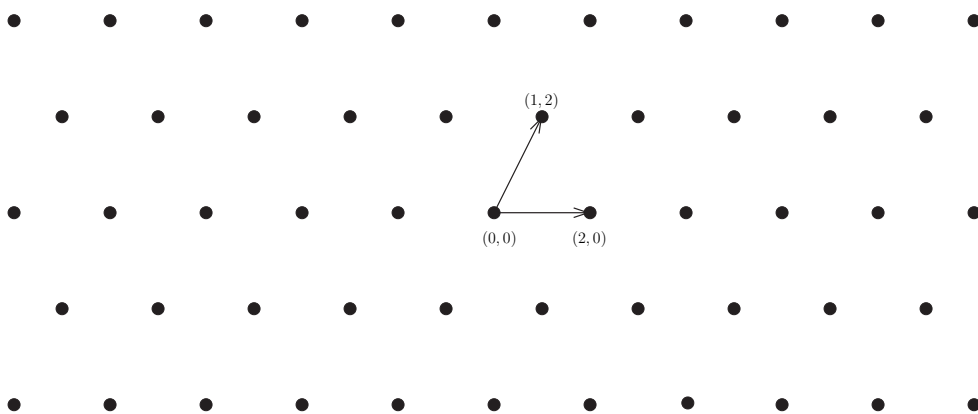
Figure 16.1: The lattice $\mathbb{Z}(2,0) \oplus \mathbb{Z}(1,2)$.

16.2   The LLL Algorithm

A Euclidean lattice is a set of points that are regularly spaced in the space $\mathbb{R}^n$ (see Definition 22). It is a discrete algebraic object that is encountered in several domains of various sciences, including mathematics, computer science, electrical engineering, and chemistry. It is a rich and powerful modeling tool, thanks to the deep and numerous theoretical results, algorithms, and implementations available (see [64, 80, 158, 267, 379] for example).

Let $x = (x_1, \ldots, x_n) \in \mathbb{R}^n$. We set

$$\|x\|_2 = (x|x)^{1/2} = \left(x_1^2 + \cdots + x_n^2\right)^{1/2} \text{ and } \|x\|_\infty = \max_{1 \leq i \leq n} |x_i|.$$

Definition 22. *Let L be a nonempty subset of $\mathbb{R}^n$. The set L is a (Euclidean) lattice if there exists a set of $\mathbb{R}$-linearly independent vectors $b_1, \ldots, b_d$ such that*

$$L = \mathbb{Z} \cdot b_1 \oplus \cdots \oplus \mathbb{Z} \cdot b_d = \left\{ \sum_{i \leq d} x_i \cdot b_i, x_i \in \mathbb{Z} \right\}.$$

The family $(b_1, \ldots, b_d)$ is a basis of the lattice L and d is called the rank of the lattice L.

For example, the set $\mathbb{Z}^n$ and all of its additive subgroups are lattices. These lattices play a central role in computer science since they can be represented exactly. We say that a lattice L is integer (resp. rational) when $L \subseteq \mathbb{Z}^n$ (resp. $\mathbb{Q}^n$). An integer lattice of rank 2 is given in Figure 16.1, as well as one of its bases.

A lattice is often given by one of its bases (in practice, a matrix whose rows or columns are the basis vectors). Unfortunately, as soon as the rank of the lattice is greater than 1, there are infinitely many such representations
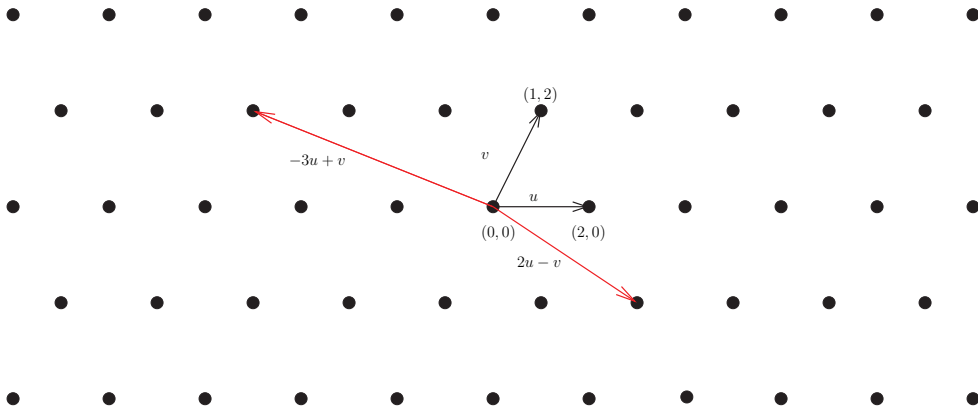
Figure 16.2: Two bases of the lattice $\mathbb{Z}(2,0) \oplus \mathbb{Z}(1,2)$.

for any given lattice. In Figure 16.2, we give another basis of the lattice of Figure 16.1.

Proposition 1. *If $(e_1, \ldots, e_k)$ and $(f_1, \ldots, f_j)$ are two families of $\mathbb{R}$-linearly independent (column) vectors that generate the same lattice, then $k = j$ (this is the rank of the lattice), and there exists a $(k \times k)$-dimensional matrix M with integer coefficients and determinant equal to ± 1 such that $(e_i) = (f_i) \cdot M$.*

Among the infinitely many bases of a specified lattice (if $k \geq 2$), some are more interesting than others. One can define various notions of what a "good" basis is, but most of the time it is required to consist of somewhat short vectors.

The two most famous computational problems related to lattices are the shortest and closest vector problems (SVP and CVP). Since a lattice is discrete, it contains a vector of smallest nonzero norm. That norm is denoted by λ and is called the *minimum* of the lattice. Note that the minimum is reached at least twice (a vector and its opposite), and may be reached more times. The discreteness also implies that, given an arbitrary vector of the space, there always exists a lattice vector closest to it (note that there can be several such vectors). We now state the search versions of the SVP and CVP problems.

Problem 1. Shortest vector problem (SVP). *Given a basis of a lattice $L \subseteq \mathbb{Q}^n$, find a shortest nonzero vector of L, i.e., a vector of norm $\lambda(L)$.*

SVP naturally leads to the following approximation problem, which we call γ-SVP, where γ is a function of the rank only: given a basis of a lattice $L \subseteq \mathbb{Q}^n$, find $b \in L$ such that

$$0 < \|b\| \leq \gamma \cdot \lambda(L).$$

Problem 2. Closest vector problem (CVP). *Given a basis of a lattice $L \subseteq \mathbb{Q}^n$ and a target vector $t \in \mathbb{Q}^n$, find $b \in L$ such that $\|b - t\| = \operatorname{dist}(t, L)$.*

CVP naturally leads to the following approximation problem, which we call γ-CVP, where γ is a function of the rank only: given a basis of a lattice $L \subseteq \mathbb{Q}^n$ and a target vector $t \in \mathbb{Q}^n$, find $b \in L$ such that

$$\|b - t\| \leq \gamma \cdot \mathrm{dist}(t, L).$$

Note that SVP and CVP can be defined with respect to any norm of $\mathbb{R}^n$, and we will write SVP_2 and CVP_2 to explicitly refer to the Euclidean norm. These two computational problems have been studied extensively. We describe very briefly some of the results, and refer to [277] for more details.

Ajtai [4] showed in 1998 that the decisional version of SVP_2 (i.e., given a lattice L and a scalar x, compare $\lambda(L)$ and x) is NP-hard under randomized polynomial reductions. The NP-hardness had been conjectured in the early 1980s by van Emde Boas [412], who proved the result for the infinity norm instead of the Euclidean norm. Khot [219] showed that Ajtai's result still holds for the decisional version of the relaxed problem γ-SVP_2, where γ is an arbitrary constant. Goldreich and Goldwasser [150] proved that, under very plausible complexity theory assumptions, approximating SVP_2 within a factor $\gamma = \sqrt{d/\ln d}$ is not NP-hard, where d is the rank of the lattice. No polynomial-time algorithm is known for approximating SVP_2 within a factor $f(d)$ with f a polynomial in d. On the constructive side, Kannan [213] described an algorithm that solves SVP_2 in time

$$d^{\frac{d(1+o(1))}{2e}} \approx d^{0.184 \cdot d},$$

and in polynomial space (the complexity bound is proved in [163]). Ajtai, Kumar and Sivakumar [5] gave an algorithm of complexity $2^{O(d)}$ both in time and space that solves SVP with high probability. Similar results hold for the infinity norm instead of the Euclidean norm.

In 1981, van Emde Boas [412] proved that the decisional version of CVP_2 is NP-hard (see also [276]). On the other hand, Goldreich and Goldwasser [150] showed, under very plausible assumptions, that approximating CVP_2 within a factor $\sqrt{d/\ln d}$ is not NP-hard. Their result also holds for the infinity norm (see [328]). Unfortunately, no polynomial-time algorithm is known for approximating CVP to a polynomial factor. On the constructive side, Kannan [213] described an algorithm that solves CVP in time

$$d^{\frac{d(1+o(1))}{2}}$$

for the Euclidean norm and

$$d^{d(1+o(1))}$$

for the infinity norm (see [163] for the proofs of the complexity bounds).

If we sufficiently relax the parameter γ, the situation becomes far better. In 1982, Lenstra, Lenstra, and Lovász [255] gave an algorithm that allows one to get relatively short vectors in polynomial time. Their algorithm is now commonly referred to by the acronym LLL.

Theorem 51 (LLL [255]). *Given an arbitrary basis $(a_1, \ldots, a_d)$ of a lattice $L \subseteq \mathbb{Z}^n$, the LLL algorithm provides a basis $(b_1, \ldots, b_d)$ of L that is made of relatively short vectors. Among others, we have $\|b_1\| \leq 2^{(d-1)/2} \cdot \lambda(L)$. Furthermore, LLL terminates within $O(d^5 n \ln^3 A)$ bit operations with $A = \max_i \|a_i\|$.*

More precisely, the LLL algorithm computes what is called a δ-LLL-reduced basis, where δ is a fixed parameter which belongs to $(1/4, 1)$ (if δ is omitted, then its value is $3/4$, which is the historical choice). To define what a LLL-reduced basis is, we need to recall the Gram–Schmidt orthogonalization of a basis. Consider a basis $(b_1, \ldots, b_d)$. Its Gram–Schmidt orthogonalization $(b_1^*, \ldots, b_d^*)$ is defined recursively as follows:

- the vector b_1^* is b_1;

- for $i > 1$, we set $b_i^* = b_i - \sum_{j<i} \mu_{i,j} b_j^*$, where $\mu_{i,j} = \frac{\langle b_i, b_j^* \rangle}{\|b_j^*\|^2}$.

Geometrically, the vector b_i^* is the projection of the vector b_i orthogonally to the span of the previous basis vectors $b_1, \ldots b_{i-1}$. We say that the basis $(b_1, \ldots, b_d)$ is δ-LLL-reduced if the following two conditions are satisfied:

- for any $i > j$, the quantity $\mu_{i,j}$ has magnitude less than or equal to $1/2$. This condition is called the *size-reduction condition*;

- for any i, we have $\delta \|b_i^*\|^2 \leq \|b_{i+1}^*\|^2 + \mu_{i+1,i}^2 \|b_i^*\|^2$. This condition is called Lovász's condition. It means that orthogonally to the first $i-1$ vectors, the $(i+1)$-th vector cannot be arbitrarily small compared to the i-th vector.

LLL-reduced bases have many interesting properties. The most important one is probably that the first basis vector cannot be more than $2^{(d-1)/2}$ times longer than the lattice minimum. The LLL algorithm computes an LLL-reduced basis $(b_1, \ldots, b_d)$ of the lattice spanned by $(a_1, \ldots, a_d)$ by incrementally trying to make the LLL conditions satisfied. It uses an index k which starts at 2 and eventually reaches $d+1$. At any moment, the first $k-1$ vectors satisfy the LLL conditions, and we are trying to make the first k vectors satisfy the conditions. To make the size-reduction condition satisfied for the k-th vector, one subtracts from it an adequate integer linear combination of the vectors $b_1, \ldots, b_{k-1}$. This is essentially the same process as Babai's nearest plane algorithm, described below. After that, one tests Lovász's condition. If it is satisfied, then the index k can be incremented. If not, the vectors b_k and b_{k-1} are swapped, and the index k is decremented. The correctness of the LLL algorithm is relatively simple to prove, but the complexity analysis is significantly more involved. We refer the interested reader to [255].

The LLL algorithm has been extensively studied since its invention [211, 365, 395, 303]. Very often in practice, the returned basis is of better quality than the worst-case bound given above and is obtained faster than expected.

We refer to [304] for more details about the practical behavior of the LLL algorithm.

Among many important applications of the LLL algorithm, Babai [17] derived from it a polynomial-time algorithm for solving CVP with an exponentially bounded approximation factor. We present it in Algorithm 16.1.

Theorem 52 (Babai [17])**.** *Given an arbitrary basis* $(b_1, \ldots, b_d)$ *of a lattice* $L \subseteq \mathbb{Z}^n$, *and a target vector* $t \in \mathbb{Z}^n$, *Babai's nearest plane algorithm (Algorithm 16.1) finds a vector* $b \in L$ *such that*

$$\|b - t\|_2 \leq 2^d \cdot dist_2(t, L).$$

Moreover, it finishes in polynomial time in d, n, $\ln A$, *and* $\ln \|t\|$, *where* $A = \max_i \|a_i\|$.

Algorithm 16.1 Babai's nearest plane algorithm. The inputs are an LLL-reduced basis $(b_i)_{1 \leq i \leq d}$, its Gram–Schmidt orthogonalization $(b_i^*)_{1 \leq i \leq d}$, and a target vector t. The output is a vector in the lattice spanned by the b_i's that is close to t.

$v \leftarrow t$
for ($j = d$; $j \geq 1$; j--) **do**
  $v \leftarrow v - \left\lfloor \frac{\langle v, b_j^* \rangle}{\langle b_j^*, b_j^* \rangle} \right\rceil b_j$
end for
return $(t - v)$

Babai's algorithm may also be described with the LLL algorithm directly. This may be simpler to implement, in particular, if one has access to an implementation of LLL. We give that variant in Algorithm 16.2.

Algorithm 16.2 Babai's nearest plane algorithm, using LLL. The inputs are an LLL-reduced basis $(b_i)_{1 \leq i \leq d}$ and a target vector t. The output is a vector in the lattice spanned by the b_i's that is close to t.

---
for ($j = 0$; $j \leq d$; j++) **do**
  $c_i \leftarrow (b_i, 0)$
end for
$B \leftarrow \max_i \|b_i\|$; $c_{d+1} \leftarrow (t, B)$
$(c_1', \ldots, c_{d+1}') \leftarrow \text{LLL}(c_1, \ldots, c_{d+1})$
return $(c_{d+1} - c_{d+1}')$

Bibliography

[1] E. Abu-Shama and M. Bayoumi. A new cell for low power adders. In *Int. Midwest Symposium on Circuits and Systems*, pages 1014–1017, 1995.

[2] R. C. Agarwal, F. G. Gustavson, and M. S. Schmookler. Series approximation methods for divide and square root in the Power3 microprocessor. In Koren and Kornerup, editors, *Proceedings of the 14th IEEE Symposium on Computer Arithmetic (Adelaide, Australia)*, pages 116–123. IEEE Computer Society Press, Los Alamitos, CA, April 1999.

[3] T. Ahrendt. Fast high-precision computation of complex square roots. In *Proceedings of ISSAC'96 (Zurich, Switzerland)*, 1996.

[4] M. Ajtai. The shortest vector problem in L_2 is NP-hard for randomized reductions (extended abstract). In *Proceedings of the annual ACM symposium on Theory of computing (STOC)*, pages 10–19, 1998.

[5] M. Ajtai, R. Kumar, and D. Sivakumar. A sieve algorithm for the shortest lattice vector problem. In *Proceedings of the annual ACM symposium on Theory of computing (STOC)*, pages 601–610, 2001.

[6] L. Aksoy, E. Costa, P. Flores, and J. Monteiro. Optimization of area in digital FIR filters using gate-level metrics. In *Design Automation Conference*, pages 420–423, 2007.

[7] 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, 2004. ACM Press.

[8] Altera Corporation. *FFT/IFFT Block Floating Point Scaling*, 2005. Application note 404-1.0.

[9] B. Amedro, V. Bodnartchouck, D. Caromel, C. Delbé, F. Huet, and G. L. Taboada. Current state of Java for HP. Preprint, Technical Report 0353, INRIA, 2008. Available at `http://hal.inria.fr/inria-00312039/en`.

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

[11] 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.

[12] C. Anderson, N. Astafiev, and S. Story. Accurate math functions on the Intel IA-32 architecture: A performance-driven design. In Hanrot and Zimmermann, editors, *Real Numbers and Computers*, pages 93–105. INRIA, July 2006.

[13] ARM. *ARM Developer Suite: Compilers and Libraries Guide*. ARM Limited, November 2001. Available at `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0067d/index.html` or in PDF at `http://infocenter.arm.com/help/topic/com.arm.doc.dui0067d/DUI0067.pdf`.

[14] ARM. *ARM Developer Suite: Developer Guide*. ARM Limited, 1.2 edition, November 2001. Document available at `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0056d/index.html` or in PDF at `http://infocenter.arm.com/help/topic/com.arm.doc.dui0056d/DUI0056.pdf`.

[15] 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`.

[16] A. Avizienis. Signed-digit number representations for fast parallel arithmetic. *IRE Transactions on Electronic Computers*, 10:389–400, 1961. Reprinted in E. E. Swartzlander, *Computer Arithmetic*, Vol. 2, IEEE Computer Society Press, Los Alamitos, CA, 1990.

[17] L. Babai. On Lovász' lattice reduction and the nearest lattice point problem. *Combinatorica*, 6(1):1–13, 1986.

[18] I. Babuška. Numerical stability in mathematical analysis. In *Proceedings of the 1968 IFIP Congress*, volume 1, pages 11–23, 1969.

[19] 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`.

[20] D. H. Bailey and J. M. Borwein. Experimental mathematics: examples, methods and implications. *Notices of the AMS*, 52(5):502–514, May 2005.

[21] D. H. Bailey, J. M. Borwein, P. B. Borwein, and S. Plouffe. The quest for pi. *Mathematical Intelligencer*, 19(1):50–57, 1997.

[22] D. H. Bailey, Y. Hida, X. S. Li, and B. Thompson. ARPREC: an arbitrary precision computation package. Technical report, Lawrence Berkeley National Laboratory, 2002. Available at `http://crd.lbl.gov/~dhbailey/dhbpapers/arprec.pdf`.

[23] G. Barrett. Formal methods applied to a floating-point system. *IEEE Transactions on Software Engineering*, 15(5):611–621, May 1989.

[24] F. Benford. The law of anomalous numbers. *Proceedings of the American Philosophical Society*, 78(4):551–572, 1938.

[25] M. Bennani and M. C. Brunet. Precise: simulation of round-off error propagation model. In *Proceedings of the 12th World IMACS Congress*, July 1988.

[26] C. Berg. *Formal Verification of an IEEE Floating-Point Adder*. Master's thesis, Universität des Saarlandes, Germany, 2001.

[27] D. J. Bernstein. Multidigit multiplication for mathematicians. Available at `http://cr.yp.to/papers.html#m3`, 2001.

[28] F. Blomquist, W. Hofschuster, and W. Krämer. Real and complex staggered (interval) arithmetics with wide exponent range (in German). Technical Report 2008/1, Universität Wuppertal, Germany, 2008.

[29] G. Bohlender, W. Walter, P. Kornerup, and D. W. Matula. Semantics for exact floating point operations. In P. Kornerup and D. W. Matula, editors, *Proceedings of the 10th IEEE Symposium on Computer Arithmetic*, pages 22–26. IEEE Computer Society Press, Los Alamitos, CA, June 1991.

[30] S. Boldo. Pitfalls of a full floating-point proof: example on the formal proof of the Veltkamp/Dekker algorithms. In U. Furbach and N. Shankar, editors, *Proceedings of the 3rd International Joint Conference on Automated Reasoning*, volume 4130 of *Lecture Notes in Computer Science*, pages 52–66, 2006.

[31] S. Boldo and M. Daumas. Representable correcting terms for possibly underflowing floating point operations. In J.-C. Bajard and M. Schulte, editors, *Proceedings of the 16th Symposium on Computer Arithmetic*, pages 79–86. IEEE Computer Society Press, Los Alamitos, CA, 2003.

[32] S. Boldo, M. Daumas, C. Moreau-Finot, and L. Théry. Computer validated proofs of a toolset for adaptable arithmetic. Technical report, École Normale Supérieure de Lyon, 2001. Available at `http://arxiv.org/pdf/cs.MS/0107025`.

[33] S. Boldo, M. Daumas, and L. Théry. Formal proofs and computations in finite precision arithmetic. In T. Hardin and R. Rioboo, editors, *Proceedings of the 11th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning*, 2003.

[34] S. Boldo and G. Melquiond. Emulation of FMA and correctly rounded sums: proved algorithms using rounding to odd. *IEEE Transactions on Computers*, 57(4):462–471, April 2008.

[35] S. Boldo and J.-M. Muller. Some functions computable with a fused-mac. In *Proceedings of the 17th IEEE Symposium on Computer Arithmetic (ARITH-17)*. IEEE Computer Society Press, Los Alamitos, CA, June 2005.

[36] S. Boldo and C. Muñoz. Provably faithful evaluation of polynomials. In *Proceedings of the 2006 ACM Symposium on Applied Computing*, pages 1328–1332, New York, NY, 2006. ACM Press.

[37] A. D. Booth. A signed binary multiplication technique. *Quarterly Journal of Mechanics and Applied Mathematics*, 4(2):236–240, 1951. Reprinted in E. E. Swartzlander, *Computer Arithmetic*, Vol. 1, IEEE Computer Society Press, Los Alamitos, CA, 1990.

[38] J. Borwein and D. H. Bailey. *Mathematics by Experiment: Plausible Reasoning in the* 21*st Century*. A. K. Peters, Natick, MA, 2004.

[39] P. Borwein and T. Erdélyi. *Polynomials and Polynomial Inequalities*. Graduate Texts in Mathematics, 161. Springer-Verlag, New York, 1995.

[40] W. Bosma, J. Cannon, and C. Playoust. The Magma algebra system. I. The user language. *Journal of Symbolic Computation*, 24(3–4):235–265, 1997.

[41] N. Boullis and A. Tisserand. Some optimizations of hardware multiplication by constant matrices. *IEEE Transactions on Computers*, 54(10):1271–1282, 2005.

[42] R. T. Boute. The euclidean definition of the functions div and mod. *ACM Trans. Program. Lang. Syst.*, 14(2):127–144, 1992.

[43] R. Brent and P. Zimmermann. *Modern Computer Arithmetic*. March 2009. Version 0.2.1. Available at `http://www.loria.fr/~zimmerma/mca/mca-0.2.1.pdf`.

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

[45] R. P. Brent. A FORTRAN multiple-precision arithmetic package. *ACM Transactions on Mathematical Software*, 4(1):57–70, 1978.

[46] R. P. Brent, C. Percival, and P. Zimmermann. Error bounds on complex floating-point multiplication. *Mathematics of Computation*, 76:1469–1481, 2007.

[47] K. Briggs. The `doubledouble` library, 1998. Available at `http://www.boutell.com/fracster-src/doubledouble/doubledouble.html`.

[48] N. Brisebarre and S. Chevillard. Efficient polynomial L^∞ approximations. In *ARITH '07: Proceedings of the 18th IEEE Symposium on Computer Arithmetic*, pages 169–176, Washington, DC, 2007. IEEE Computer Society.

[49] N. Brisebarre, F. de Dinechin, and J.-M. Muller. Integer and floating-point constant multipliers for FPGAs. In *Application-specific Systems, Architectures and Processors*, pages 239–244. IEEE, 2008.

[50] N. Brisebarre and J.-M. Muller. Correct rounding of algebraic functions. *Theoretical Informatics and Applications*, 41:71–83, Jan–March 2007.

[51] N. Brisebarre and J.-M. Muller. Correctly rounded multiplication by arbitrary precision constants. *IEEE Transactions on Computers*, 57(2):165–174, February 2008.

[52] N. Brisebarre, J.-M. Muller, and S.-K. Raina. Accelerating correctly rounded floating-point division when the divisor is known in advance. *IEEE Transactions on Computers*, 53(8):1069–1072, August 2004.

[53] N. Brisebarre, J.-M. Muller, and A. Tisserand. Sparse-coefficient polynomial approximations for hardware implementations. In *Proc. 38th IEEE Conference on Signals, Systems and Computers*. IEEE, November 2004.

[54] N. Brisebarre, J.-M. Muller, and A. Tisserand. Computing machine-efficient polynomial approximations. *ACM Transactions on Mathematical Software*, 32(2):236–256, June 2006.

[55] W. S. Brown. A simple but realistic model of floating-point computation. *ACM Transactions on Math. Software*, 7(4), December 1981.

[56] W. S. Brown and P. L. Richman. The choice of base. *Communications of the ACM*, 12(10):560–561, October 1969.

[57] C. Bruel. If-conversion SSA framework for partially predicated VLIW architectures. In *Digest of the 4th Workshop on Optimizations for DSP and Embedded Systems (Manhattan, New York, NY)*, March 2006.

[58] J. D. Bruguera and T. Lang. Leading-one prediction with concurrent position correction. *IEEE Transactions on Computers*, 48(10):1083–1097, October 1999.

[59] J. D. Bruguera and T. Lang. Floating-point fused multiply-add: Reduced latency for floating-point addition. In *Proceedings of the 17th IEEE Symposium on Computer Arithmetic (ARITH-17)*. IEEE Computer Society Press, Los Alamitos, CA, June 2005.

[60] M. C. Brunet and F. Chatelin. A probabilistic round-off error propagation model, application to the eigenvalue problem. In M. G. Cox and S. Hammarling, editors, *Reliable Numerical Software*. Oxford University Press, London, 1987. Available at `http://www.boutell.com/fracster-src/doubledouble/doubledouble.html`.

[61] H. T. Bui, Y. Wang, and Y. Jiang. Design and analysis of low-power 10-transistor full adders using novel XORXNOR gates. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 49(1), 2003.

[62] R. G. Burger and R. Kent Dybvig. Printing floating-point numbers quickly and accurately. In *Proceedings of the SIGPLAN'96 Conference on Programming Languages Design and Implementation*, pages 108–116, June 1996.

[63] 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.

[64] J. W. S. Cassels. *An introduction to the geometry of numbers*. Classics in Mathematics. Springer-Verlag, Berlin, 1997. Corrected reprint of the 1971 edition.

[65] A. Cauchy. Sur les moyens d'éviter les erreurs dans les calculs numériques. *Comptes Rendus de l'Académie des Sciences, Paris*, 11:789–798, 1840. Republished in: Augustin Cauchy, oeuvres complètes, 1ère série, Tome V, pages 431–442. Available at `http://gallica.bnf.fr/scripts/ConsultationTout.exe?O=N090185`.

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

[67] K. D. Chapman. Fast integer multipliers fit in FPGAs (EDN 1993 design idea winner). *EDN Magazine*, May 1994.

[68] T. C. Chen and I. T. Ho. Storage-efficient representation of decimal data. *Communications of the ACM*, 18(1):49–52, 1975.

[69] E. W. Cheney. *Introduction to Approximation Theory*. AMS Chelsea Publishing. Providence, RI, 2nd edition, 1982.

[70] S. Chevillard and C. Q. Lauter. A certified infinite norm for the implementation of elementary functions. In *Seventh International Conference on Quality Software (QSIC 2007)*, pages 153–160. IEEE, 2007.

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

[72] W. D. Clinger. How to read floating-point numbers accurately. *ACM SIGPLAN Notices*, 25(6):92–101, June 1990.

[73] W. D. Clinger. Retrospective: how to read floating-point numbers accurately. *ACM SIGPLAN Notices*, 39(4):360–371, April 2004.

[74] D. Cochran. Algorithms and accuracy in the HP 35. *Hewlett Packard Journal*, 23:10–11, June 1972.

[75] W. Cody and W. Waite. *Software Manual for the Elementary Functions*. Prentice-Hall, Englewood Cliffs, NJ, 1980.

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

[77] W. J. Cody. Implementation and testing of function software. In P. C. Messina and A. Murli, editors, *Problems and Methodologies in Mathematical Software Production*, Lecture Notes in Computer Science 142. Springer-Verlag, Berlin, 1982.

[78] W. J. Cody. MACHAR: a subroutine to dynamically determine machine parameters. *ACM Transactions on Mathematical Software*, 14(4):301–311, December 1988.

[79] S. Collange, M. Daumas, and D. Defour. État de l'intégration de la virgule flottante dans les processeurs graphiques. *Technique et Science Informatiques*, 27(6):719–733, 2008. In French.

[80] J. H. Conway and N. J. A. Sloane. *Sphere Packings, Lattices and Groups*. Springer-Verlag, New York, 1988.

[81] J. Coonen. *Contributions to a Proposed Standard for Binary Floating-Point Arithmetic*. Ph.D. thesis, University of California at Berkeley, 1984.

[82] D. Coppersmith. Finding a small root of a univariate modular equation. In U. M. Maurer, editor, *Proceedings of EUROCRYPT*, volume 1070 of *Lecture Notes in Computer Science*, pages 155–165. Springer-Verlag, Berlin, 1996.

[83] D. Coppersmith. Finding small solutions to small degree polynomials. In J. H. Silverman, editor, *Proceedings of Cryptography and Lattices (CaLC)*, volume 2146 of *Lecture Notes in Computer Science*, pages 20–31. Springer-Verlag, Berlin, 2001.

[84] M. Cornea. Proving the IEEE correctness of iterative floating-point square root, divide and remainder algorithms. *Intel Technology Journal*, Q2:1–11, 1998. Available at `http://download.intel.com/technology/itj/q21998/pdf/ieee.pdf`.

[85] 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 P. Kornerup and J.-M. Muller, editors, *Proceedings of the 18th IEEE Symposium on Computer Arithmetic (ARITH-18)*, pages 29–37. IEEE Computer Society Conference Publishing Services, June 2007.

[86] M. Cornea, R. A. Golliver, and P. Markstein. Correctness proofs outline for Newton–Raphson-based floating-point divide and square root algorithms. In Koren and Kornerup, editors, *Proceedings of the 14th IEEE Symposium on Computer Arithmetic (Adelaide, Australia)*, pages 96–105. IEEE Computer Society Press, Los Alamitos, CA, April 1999.

[87] M. Cornea, J. Harrison, C. Anderson, P. T. P. Tang, E. Schneider, and E. Gvozdev. A software implementation of the IEEE 754R decimal floating-point arithmetic using the binary encoding format. *IEEE Transactions on Computers*, 58(2):148–162, 2009.

[88] M. Cornea, J. Harrison, and P. T. P. Tang. *Scientific Computing on Itanium®-based Systems*. Intel Press, Hillsboro, OR, 2002.

[89] M. Cornea, C. Iordache, J. Harrison, and P. Markstein. Integer divide and remainder operations in the IA-64 architecture. In J.-C. Bajard, C. Frougny, P. Kornerup, and J.-M. Muller, editors, *Proceedings of the 4th Conference on Real Numbers and Computers*, 2000.

[90] 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.

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

[92] O. Creţ, F. de Dinechin, I. Trestian, R. Tudoran, L. Creţ, and L. Văcariu. FPGA-based acceleration of the computations involved in transcranial magnetic stimulation. In *Southern Programmable Logic Conference*, pages 43–48. IEEE, 2008.

[93] A. Cuyt, B. Verdonk, S. Becuwe, and P. Kuterna. A remarkable example of catastrophic cancellation unraveled. *Computing*, 66:309–320, 2001.

[94] A. Dahan-Dalmedico and J. Pfeiffer. *Histoire des Mathématiques*. Editions du Seuil, Paris, 1986. In French.

[95] C. Daramy-Loirat, D. Defour, F. de Dinechin, M. Gallet, N. Gast, C. Q. Lauter, and J.-M. Muller. CR-LIBM, a library of correctly-rounded elementary functions in double-precision. Technical report, LIP Laboratory, Arenaire team, Available at `https://lipforge.ens-lyon.fr/frs/download.php/99/crlibm-0.18beta1.pdf`, December 2006.

[96] M. Daumas, G. Melquiond, and C. Muñoz. Guaranteed proofs using interval arithmetic. In P. Montuschi and E. Schwarz, editors, *Proceedings of the 17th IEEE Symposium on Computer Arithmetic*, pages 188–195, Cape Cod, MA, 2005.

[97] M. Daumas, L. Rideau, and L. Théry. A generic library of floating-point numbers and its application to exact computing. In *Proceedings of the 14th International Conference on Theorem Proving in Higher Order Logics*, pages 169–184, Edinburgh, Scotland, 2001.

[98] B. de Dinechin. From machine scheduling to VLIW instruction scheduling. *ST Journal of Research*, 1(2), September 2004.

[99] F. de Dinechin. The price of routing in FPGAs. *Journal of Universal Computer Science*, 6(2):227–239, 2000.

[100] F. de Dinechin, C. Q. Lauter, and J.-M. Muller. Fast and correctly rounded logarithms in double-precision. *Theoretical Informatics and Applications*, 41:85–102, 2007.

[101] F. de Dinechin and V. Lefèvre. Constant multipliers for FPGAs. In *Parallel and Distributed Processing Techniques and Applications*, pages 167–173, 2000.

[102] F. de Dinechin, C. Loirat, and J.-M. Muller. A proven correctly rounded logarithm in double-precision. In *RNC6, Real Numbers and Computers*, 2004.

[103] F. de Dinechin, E. McIntosh, and F. Schmidt. Massive tracking on heterogeneous platforms. In *9th International Computational Accelerator Physics Conference (ICAP)*, October 2006.

[104] F. de Dinechin and B. Pasca. Large multipliers with less DSP blocks. Research report 2009-03, Laboratoire LIP, École Normale Supérieure de Lyon, Lyon, France, 2009. Available at `http://prunel.ccsd.cnrs.fr/ensl-00356421/`.

[105] F. de Dinechin, B. Pasca, O. Creţ, and R. Tudoran. An FPGA-specific approach to floating-point accumulation and sum-of-products. In *Field-Programmable Technologies*. IEEE, 2008.

[106] F. de Dinechin and A. Tisserand. Multipartite table methods. *IEEE Transactions on Computers*, 54(3):319–330, March 2005.

[107] A. DeHon and N. Kapre. Optimistic parallelization of floating-point accumulation. In *18th Symposium on Computer Arithmetic*, pages 205–213. IEEE, June 2007.

[108] T. J. Dekker. A floating-point technique for extending the available precision. *Numerische Mathematik*, 18(3):224–242, 1971.

[109] M. deLorimier and A. DeHon. Floating-point sparse matrix-vector multiply for FPGAs. In *Field-Programmable Gate Arrays*, pages 75–85. ACM, 2005.

[110] J. Demmel. Underflow and the reliability of numerical software. *SIAM Journal on Scientific and Statistical Computing*, 5(4):887–919, 1984.

[111] A. G. Dempster and M. D. Macleod. Constant integer multiplication using minimum adders. *Circuits, Devices and Systems, IEE Proceedings*, 141(5):407–413, 1994.

[112] J. Detrey and F. de Dinechin. Table-based polynomials for fast hardware function evaluation. In *Application-Specific Systems, Architectures and Processors*, pages 328–333. IEEE, 2005.

[113] J. Detrey and F. de Dinechin. Floating-point trigonometric functions for FPGAs. In *Intl Conference on Field-Programmable Logic and Applications*, pages 29–34. IEEE, August 2007.

[114] J. Detrey and F. de Dinechin. Parameterized floating-point logarithm and exponential functions for FPGAs. *Microprocessors and Microsystems, Special Issue on FPGA-based Reconfigurable Computing*, 31(8):537–545, 2007.

[115] J. Detrey and F. de Dinechin. A tool for unbiased comparison between logarithmic and floating-point arithmetic. *Journal of VLSI Signal Processing*, 49(1):161–175, 2007.

[116] J. Detrey, F. de Dinechin, and X. Pujol. Return of the hardware floating-point elementary function. In *18th Symposium on Computer Arithmetic*, pages 161–168. IEEE, June 2007.

[117] W. R. Dieter, A. Kaveti, and H. G. Dietz. Low-cost microarchitectural support for improved floating-point accuracy. *IEEE Computer Architecture Letters*, 6(1): 13–16, January 2007.

[118] V. Dimitrov, L. Imbert, and A. Zakaluzny. Multiplication by a constant is sublinear. In *18th Symposium on Computer Arithmetic*. IEEE, June 2007.

[119] C. Doss and R. L. Riley, Jr. FPGA-based implementation of a robust IEEE-754 exponential unit. In *Field-Programmable Custom Computing Machines*, pages 229–238. IEEE, 2004.

[120] Y. Dou, S. Vassiliadis, G. K. Kuzmanov, and G. N. Gaydadjiev. 64-bit floating-point FPGA matrix multiplication. In *Field-Programmable Gate Arrays*, pages 86–95. ACM, 2005.

[121] P. Echeverría and M. López-Vallejo. An FPGA implementation of the powering function with single precision floating-point arithmetic. In *Proceedings of the 8th Conference on Real Numbers and Computers*, Santiago de Compostela, Spain, 2008.

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

[123] L. Eisen, J. W. Ward, H. W. Tast, N. Mäding, J. Leenstra, S. M. Mueller, C. Jacobi, J. Preiss, E. M. Schwarz, and S. R. Carlough. IBM POWER6 accelerators: VMX and DFU. *IBM Journal of Research and Development*, 51(6):1–21, November 2007.

[124] N. D. Elkies. Rational points near curves and small nonzero $|x^3 - y^2|$ via lattice reduction. In Wieb Bosma, editor, *Proceedings of the 4th Algorithmic Number Theory Symposium (ANTS IV)*, volume 1838 of *Lecture Notes in Computer Science*, pages 33–63. Springer-Verlag, Berlin, 2000.

[125] M. D. Ercegovac and T. Lang. *Division and Square Root: Digit-Recurrence Algorithms and Implementations*. Kluwer Academic Publishers, Boston, MA, 1994.

[126] M. D. Ercegovac and T. Lang. *Digital Arithmetic*. Morgan Kaufmann Publishers, San Francisco, CA, 2004.

[127] M. D. Ercegovac and J.-M. Muller. Complex division with prescaling of the operands. In *14th IEEE Conference on Application-Specific Systems, Architectures and Processors*, pages 304–314. IEEE Computer Society, June 2003.

[128] M. A. Erle and M. J. Schulte. Decimal multiplication via carry-save addition. In *Application-specific Systems, Architectures and Processors*, pages 348–355. IEEE, 2003.

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

[130] M. A. Erle, M. J. Schulte, and J. M. Linebarger. Potential speedup using decimal floating-point hardware. In *Thirty-Sixth Asilomar Conference on Signals, Systems, and Computers*, volume 2, pages 1073–1077, November 2002.

[131] M. A. Erle, E. M. Schwarz, and M. J. Schulte. Decimal multiplication with efficient partial product generation. In *Proceedings of the 17th IEEE Symposium on Computer Arithmetic (ARITH-17)*. IEEE Computer Society Press, Los Alamitos, CA, June 2005.

[132] G. Even and W. J. Paul. On the design of IEEE compliant floating-point units. *IEEE Transactions on Computers*, 49(5):398–413, May 2000.

[133] G. Even and P.-M. Seidel. A comparison of three rounding algorithms for IEEE floating-point multiplication. *IEEE Transactions on Computers*, 49(7):638–650, 2000.

[134] H. A. H. Fahmy, A. A. Liddicoat, and M. J. Flynn. Improving the effectiveness of floating point arithmetic. In *Thirty-Fifth Asilomar Conference on Signals, Systems, and Computers*, volume 1, pages 875–879, November 2001.

[135] B. P. Flannery, W. H. Press, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C*, 2nd edition. Cambridge University Press, New York, NY, 1992.

[136] G. E. Forsythe and C. B. Moler. *Computer Solution of Linear Algebraic Systems*. Prentice-Hall, Englewood Cliffs, NJ, 1967.

[137] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann. MPFR: A Multiple-Precision Binary Floating-Point Library with Correct Rounding. *ACM Transactions on Mathematical Software*, 33(2), 2007. available at `http://www.mpfr.org/`.

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

[139] W. Fraser. A survey of methods of computing minimax and near-minimax polynomial approximations for functions of a single independent variable. *Journal of the ACM*, 12(3):295–314, July 1965.

[140] Fujitsu. *SPARC64^{TM} VI Extensions*. Fujitsu Limited, 1.3 edition, March 2007.

[141] M. Fürer. Faster integer multiplication. In D. S. Johnson and U. Feige, editors, *Proceedings of the 39th Annual ACM Symposium on Theory of Computing, San Diego, CA*, pages 57–66. ACM, June 2007.

[142] S. Gal. Computing elementary functions: a new approach for achieving high accuracy and good performance. In *Accurate Scientific Computations. Lecture Notes in Computer Science*, volume 235, pages 1–16. Springer-Verlag, Berlin, 1986.

[143] S. Gal and B. Bachelis. An accurate elementary mathematical library for the IEEE floating point standard. *ACM Transactions on Mathematical Software*, 17(1):26–45, March 1991.

[144] P. Gaudry, A. Kruppa, and P. Zimmermann. A GMP-based implementation of Schönhage-Strassen's large integer multiplication algorithm. In *Proceedings ISSAC*, pages 167–174, 2007.

[145] D. M. Gay. Correctly-rounded binary-decimal and decimal-binary conversions. Technical Report Numerical Analysis Manuscript 90-10, ATT & Bell Laboratories (Murray Hill, NJ), November 1990.

[146] W. M. Gentleman and S. B. Marovitch. More on algorithms that reveal properties of floating-point arithmetic units. *Communications of the ACM*, 17(5):276–277, May 1974.

[147] G. Gerwig, H. Wetter, E. M. Schwarz, J. Haess, C. A. Krygowski, B. M. Fleischer, and M. Kroener. The IBM eServer z990 floating-point unit. *IBM Journal of Research and Development*, 48(3/4):311–322, 2004.

[148] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–47, March 1991. An edited reprint is available at `http://www.physics.ohio-state.edu/~dws/grouplinks/floating_point_math.pdf` from Sun's Numerical Computation Guide; it contains an addendum *Differences Among IEEE 754 Implementations*, also available at `http://www.validlab.com/goldberg/addendum.html`.

[149] I. B. Goldberg. 27 bits are not enough for 8-digit accuracy. *Commun. ACM*, 10(2):105–106, 1967.

[150] O. Goldreich and S. Goldwasser. On the limits of non-approximability of lattice problems. In *Proceedings of 30th Annual ACM Symposium on Theory of Computing (STOC)*, pages 1–9. ACM, May 1998.

[151] R. E. Goldschmidt. Applications of division by convergence. Master's thesis, Dept. of Electrical Engineering, Massachusetts Institute of Technology, Cambridge, MA, June 1964.

[152] G. Gonnet. A note on finding difficult values to evaluate numerically. Available at `http://www.inf.ethz.ch/personal/gonnet/FPAccuracy/NastyValues.ps`, 2002.

[153] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, New York, 3rd edition, 1999.

[154] F. Goualard. Fast and correct SIMD algorithms for interval arithmetic. In *Proceedings of PARA '08*, May 2008.

[155] S. Graillat, P. Langlois, and N. Louvet. Algorithms for accurate, validated and fast computations with polynomials. *Japan Journal of Industrial and Applied Mathematics*, Special issue on Verified Numerical Computation, 2009. (To appear).

[156] T. Granlund. The GNU multiple precision arithmetic library, release 4.1.4. Accessible electronically at `http://gmplib.org/gmp-man-4.1.4.pdf`, September 2004.

[157] B. Greer, J. Harrison, G. Henry, W. Li, and P. Tang. Scientific computing on the Itanium[TM] processor. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE Conference on Supercomputing (CDROM)*, pages 41–41, New York, NY, 2001. ACM.

[158] P. M. Gruber and C. G. Lekkerkerker. *Geometry of numbers*, volume 37 of *North-Holland Mathematical Library*. North-Holland Publishing Co., Amsterdam, second edition, 1987.

[159] A. Guntoro and M. Glesner. High-performance FPGA-based floating-point adder with three inputs. In *Field Programmable Logic and Applications*, pages 627–630, 2008.

[160] O. Gustafsson, A. G. Dempster, K. Johansson, and M. D. Macleod. Simplified design of constant coefficient multipliers. *Circuits, Systems, and Signal Processing*, 25(2):225–251, 2006.

[161] R. W. Hamming. On the distribution of numbers. *Bell Systems Technical Journal*, 49:1609–1625, 1970. Reprinted in E. E. Swartzlander, *Computer Arithmetic*, Vol. 1, IEEE Computer Society Press, Los Alamitos, CA, 1990.

[162] G. Hanrot, V. Lefèvre, D. Stehlé, and P. Zimmermann. Worst cases of a periodic function for large arguments. In *ARITH '07: Proceedings of the 18th IEEE Symposium on Computer Arithmetic*, pages 133–140, Washington, DC, 2007. IEEE Computer Society.

[163] G. Hanrot and D. Stehlé. Improved analysis of Kannan's shortest lattice vector algorithm. In *CRYPTO*, volume 4622 of *Lecture Notes in Computer Science*, pages 170–186. Springer-Verlag, 2007.

[164] G. Hanrot and P. Zimmermann. A long note on Mulders' short product. *J. Symb. Comput.*, 37(3):391–401, 2004.

[165] E. R. Hansen and W. Walster. *Global optimization using interval analysis*. MIT Press, Cambridge, MA, 2004.

[166] G. H. Hardy and E. M. Wright. *An Introduction to the Theory of Numbers*. Oxford University Press, London, 1979.

[167] J. Harrison. Floating-point verification in HOL light: The exponential function. Technical Report 428, University of Cambridge Computer Laboratory, 1997.

[168] J. Harrison. A machine-checked theory of floating-point arithmetic. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *Theorem Proving in Higher Order Logics: 12th International Conference, TPHOLs'99*, volume 1690 of *Lecture Notes in Computer Science*, pages 113–130. Springer-Verlag, Berlin, September 1999.

[169] J. Harrison. Formal verification of floating-point trigonometric functions. In W. A. Hunt and S. D. Johnson, editors, *Proceedings of the 3rd International Conference on Formal Methods in Computer-Aided Design, FMCAD 2000*, number 1954 in Lecture Notes in Computer Science, pages 217–233. Springer-Verlag, Berlin, 2000.

[170] J. Harrison. Formal verification of IA-64 division algorithms. In M. Aagaard and J. Harrison, editors, *Proceedings of the 13th International Conference on Theorem Proving in Higher Order Logics, TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*, pages 234–251. Springer-Verlag, 2000.

[171] J. Harrison. Floating-point verification using theorem proving. In M. Bernardo and A. Cimatti, editors, *Formal Methods for Hardware Verification, 6th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2006*, volume 3965 of *Lecture Notes in Computer Science*, pages 211–242, Bertinoro, Italy, 2006. Springer-Verlag.

[172] J. Harrison. Verifying nonlinear real formulas via sums of squares. In K. Schneider and J. Brandt, editors, *Proceedings of the 20th International Conference on Theorem Proving in Higher Order Logics, TPHOLs 2007*, volume 4732 of *Lecture Notes in Computer Science*, pages 102–118, Kaiserslautern, Germany, 2007. Springer-Verlag.

[173] J. Harrison, T. Kubaska, S. Story, and P. T. P. Tang. The computation of transcendental functions on the IA-64 architecture. *Intel Technology Journal*, Q4, 1999. Available at `http://developer.intel.com/technology/itj/archive/1999.htm`.

[174] J. F. Hart, E. W. Cheney, C. L. Lawson, H. J. Maehly, C. K. Mesztenyi, J. R. Rice, H. G. Thacher, and C. Witzgall. *Computer Approximations*. John Wiley & Sons, New York, 1968.

[175] J. Hauser. The SoftFloat and TestFloat Packages. Available at `http://www.jhauser.us/arithmetic/`.

[176] J. R. Hauser. Handling floating-point exceptions in numeric programs. *ACM Trans. Program. Lang. Syst.*, 18(2):139–174, 1996.

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

[178] C. He, G. Qin, M. Lu, and W. Zhao. Group-alignment based accurate floating-point summation on FPGAs. In *Engineering of Reconfigurable Systems and Algorithms*, pages 136–142, 2006.

[179] T. J. Hickey, Q. Ju, and M. H. van Emden. Interval arithmetic: From principles to implementation. *Journal of the ACM*, 48(5):1038–1068, 2001.

[180] Y. Hida, X. S. Li, and D. H. Bailey. Algorithms for quad-double precision floating-point arithmetic. In N. Burgess and L. Ciminiera, editors, *Proceedings of the 15th IEEE Symposium on Computer Arithmetic (ARITH-16)*, pages 155–162, Vail, CO, June 2001.

[181] N. J. Higham. The accuracy of floating point summation. *SIAM J. Sci. Comput.*, 14(4):783–799, 1993.

[182] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, Philadelphia, PA, 2nd edition, 2002.

[183] E. Hokenek, R. K. Montoye, and P. W. Cook. Second-generation RISC floating point with multiply-add fused. *IEEE Journal of Solid-State Circuits*, 25(5):1207–1213, October 1990.

[184] J. E. Holm. *Floating-Point Arithmetic and Program Correctness Proofs*. Ph.D. thesis, Cornell University, 1980.

[185] M. S. Hrishikesh, D. Burger, N. P. Jouppi, S. W. Keckler, K. I. Farkas, and P. Shivakumar. The optimal logic depth per pipeline stage is 6 to 8 FO4 inverter delays. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 14–24. IEEE Computer Society, 2002.

[186] T. E. Hull and J. R. Swenson. Test of probabilistic models for propagation of round-off errors. *Communications of the ACM*, 9:108–113, 1966.

[187] 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`.

[188] International Organization for Standardization. *ISO/IEC Standard IEC-60559: Binary floating-point arithmetic for microprocessor systems*. International Electrotechnical Commission, 2nd edition, 1989.

[189] International Organization for Standardization. *Information technology — Language independent arithmetic — Part 1: Integer and floating point arithmetic*. ISO/IEC standard 10967-1, 1994.

[190] International Organization for Standardization. *Programming Languages – C*. ISO/IEC Standard 9899:1999, Geneva, Switzerland, December 1999.

[191] International Organization for Standardization. *Information technology — Language independent arithmetic — Part 2: Elementary numerical functions*. ISO/IEC standard 10967-2, 2001.

[192] International Organization for Standardization. *Programming languages – Fortran – Part 1: Base language*. International Standard ISO/IEC 1539-1:2004, 2004.

[193] International Organization for Standardization. *Information technology — Language independent arithmetic — Part 3: Complex integer and floating point arithmetic and complex elementary numerical functions*. ISO/IEC standard 10967-3, 2006.

[194] International Organization for Standardization. *Extension for the programming language C to support decimal floating-point arithmetic*. ISO/IEC Draft Standard 24732–N1312, May 2008.

[195] C. Iordache and D. W. Matula. On infinitely precise rounding for division, square root, reciprocal and square root reciprocal. In Koren and Kornerup, editors, *Proceedings of the 14th IEEE Symposium on Computer Arithmetic (Adelaide, Australia)*, pages 233–240. IEEE Computer Society Press, Los Alamitos, CA, April 1999.

[196] C.-P. Jeannerod, H. Knochel, C. Monat, and G. Revy. Faster floating-point square root for integer processors. In *IEEE Symposium on Industrial Embedded Systems (SIES'07)*, 2007.

[197] C.-P. Jeannerod, H. Knochel, C. Monat, and G. Revy. Computing floating-point square roots via bivariate polynomial evaluation. Technical Report RR2008-38, LIP, October 2008.

[198] C.-P. Jeannerod, H. Knochel, C. Monat, G. Revy, and G. Villard. A new binary floating-point division algorithm and its software implementation on the ST231 processor. In *Proceedings of the 19th IEEE Symposium on Computer Arithmetic (ARITH-19)*, Portland, OR, June 2009.

[199] R. M. Jessani and C. H. Olson. The floating-point unit of the PowerPC 603e microprocessor. *IBM Journal of Research and Development*, 40(5):559–566, 1996.

[200] K. Johansson, O. Gustafsson, and L. Wanhammar. A detailed complexity model for multiple constant multiplication and an algorithm to minimize the complexity. In *Circuit Theory and Design*, pages 465–468, 2005.

[201] W. Kahan. Pracniques: further remarks on reducing truncation errors. *Commun. ACM*, 8(1):40, 1965.

[202] W. Kahan. Why do we need a floating-point standard? Technical report, Computer Science, UC Berkeley, 1981. Available at `http://www.cs.berkeley.edu/~wkahan/ieee754status/why-ieee.pdf`.

[203] W. Kahan. Minimizing q*m-n. Text accessible electronically at `http://http.cs.berkeley.edu/~wkahan/`. At the beginning of the file "nearpi.c", 1983.

[204] W. Kahan. Branch cuts for complex elementary functions. In A. Iserles and M. J. D. Powell, editors, *The State of the Art in Numerical Analysis*, pages 165–211. Clarendon Press, Oxford, 1987.

[205] W. Kahan. Lecture notes on the status of IEEE-754. PDF file accessible at `http://www.cs.berkeley.edu/~wkahan/ieee754status/IEEE754.PDF`, 1996.

[206] W. Kahan. A test for correctly rounded sqrt. Postscript version accessible electronically at `http://www.cs.berkeley.edu/~wkahan/SQRTest.ps`, 1996.

[207] W. Kahan. IEEE 754: An interview with William Kahan. *Computer*, 31(3):114–115, March 1998.

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

[209] W. Kahan. A logarithm too clever by half. Available at `http://http.cs.berkeley.edu/~wkahan/LOG10HAF.TXT`, 2004.

[210] W. Kahan and J. Darcy. How Java's floating-point hurts everyone everywhere. Available at `http://www.cs.berkeley.edu/~wkahan/JAVAhurt.pdf`, 1998.

[211] E. Kaltofen. On the complexity of finding short vectors in integer lattices. In *EUROCAL'83*, volume 162 of *Lecture Notes in Computer Science*, pages 236–244. Springer, Berlin, 1983.

[212] G. Kane. *PA-RISC 2.0 Architecture*. Prentice Hall PTR, Upper Saddle River, NJ, 1995.

[213] R. Kannan. Improved algorithms for integer programming and related lattice problems. In *Proceedings of the annual ACM symposium on Theory of computing (STOC)*, pages 193–206, 1983.

[214] A. Karatsuba and Y. Ofman. Multiplication of many-digital numbers by automatic computers. *Doklady Akad. Nauk SSSR*, 145:293–294, 1962. Translation in Physics-Doklady 7, 595–596, 1963.

[215] R. Karpinsky. PARANOIA: a floating-point benchmark. *BYTE*, 10(2), 1985.

[216] R. B. Kearfott. *Rigorous global search: continuous problems*. Kluwer, Dordrecht, 1996.

[217] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, NJ, 1978.

[218] A. Ya. Khinchin. *Continued Fractions*. Dover, New York, 1997.

[219] S. Khot. Hardness of approximating the shortest vector problem in lattices. In *FOCS*, pages 126–135, 2004.

[220] N. G. Kingsbury and P. J. W. Rayner. Digital filtering using logarithmic arithmetic. *Electronic Letters*, 7:56–58, 1971. Reprinted in E. E. Swartzlander, *Computer Arithmetic*, Vol. 1, IEEE Computer Society Press, Los Alamitos, CA, 1990.

[221] A. Klein. A generalized Kahan-Babuška-summation-algorithm. *Computing*, 76:279–293, 2006.

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

[223] N. Koblitz. *p-adic Numbers, p-adic Analysis, and Zeta Functions*, volume 58 of *Graduate Texts in Mathematics*. Springer–Verlag, New York, 1984.

[224] I. Koren. *Computer Arithmetic Algorithms*. Prentice-Hall, Englewood Cliffs, NJ, 1993.

[225] P. Kornerup, C. Q. Lauter, N. Louvet, V. Lefèvre, and J.-M. Muller. Computing correctly rounded integer powers in floating-point arithmetic. Research report 2008-15, Laboratoire LIP, École Normale Supérieure de Lyon, Lyon, France, May 2008. Available at `http://prunel.ccsd.cnrs.fr/ensl-00278430/`.

[226] P. Kornerup, V. Lefèvre, N. Louvet, and J.-M. Muller. On the computation of correctly-rounded sums. In *Proceedings of the 19th IEEE Symposium on Computer Arithmetic (ARITH-19)*, Portland, OR, June 2009.

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

[228] 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.

[229] P. Kornerup and J.-M. Muller. Choosing starting values for certain Newton–Raphson iterations. *Theoretical Computer Science*, 351(1):101–110, February 2006.

[230] T. Kotnik and H. J. J. te Riele. The Mertens conjecture revisited. In *Algorithmic Number Theory*, volume 4076/2006 of *Lecture Notes in Computer Science*, pages 156–167, 2006.

[231] W. Krandick and J. R. Johnson. Efficient multiprecision floating-point multiplication with optimal directional rounding. In E. E. Swartzlander, M. J. Irwin, and J. Jullien, editors, *Proceedings of the 11th IEEE Symposium on Computer Arithmetic*, pages 228–233. IEEE Computer Society Press, Los Alamitos, CA, June 1993.

[232] 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.

[233] U. W. Kulisch. Circuitry for generating scalar products and sums of floating-point numbers with maximum accuracy. United States Patent 4622650, 1986.

[234] U. W. Kulisch. *Advanced Arithmetic for the Digital Computer: Design of Arithmetic Units*. Springer-Verlag, Berlin, 2002.

[235] U. W. Kulisch. *Computer Arithmetic and Validity: Theory, Implementation, and Applications*. de Gruyter, Berlin, 2008.

[236] A. Kumar. The HP PA-8000 RISC CPU. *IEEE Micro*, 17(2):27–32, March/April 1997.

[237] B. Lambov. Interval arithmetic using SSE-2. In *Reliable Implementation of Real Number Algorithms: Theory and Practice*, volume 5045 of *Lecture Notes in Computer Science*, pages 102–113. Springer, Berlin, August 2008.

[238] T. Lang and J. D. Bruguera. Floating-point multiply-add-fused with reduced latency. *IEEE Transactions on Computers*, 53(8):988–1003, 2004.

[239] T. Lang and J.-M. Muller. Bound on run of zeros and ones for algebraic functions. In N. Burgess and L. Ciminiera, editors, *Proceedings of the 15th IEEE Symposium on Computer Arithmetic (ARITH-16)*, pages 13–20, June 2001.

[240] T. Lang and A. Nannarelli. A radix-10 combinational multiplier. In *Fortieth Asilomar Conference on Signals, Systems, and Computers*, pages 313–317, October/November 2006.

[241] P. Langlois. Automatic linear correction of rounding errors. *BIT Numerical Algorithms*, 41(3):515–539, 2001.

[242] P. Langlois and N. Louvet. How to ensure a faithful polynomial evaluation with the compensated Horner algorithm. In P. Kornerup and J.-M. Muller, editors, *Proceedings of the 18th IEEE Symposium on Computer Arithmetic (ARITH-18)*, pages 141–149. IEEE Computer Society Conference Publishing Services, June 2007.

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

[244] C. Q. Lauter. Basic building blocks for a triple-double intermediate format. Technical Report 2005-38, LIP, École Normale Supérieure de Lyon, September 2005.

[245] C. Q. Lauter. *Arrondi Correct de Fonctions Mathématiques*. Ph.D. thesis, École Normale Supérieure de Lyon, Lyon, France, October 2008. In French, available at `http://www.ens-lyon.fr/LIP/web/`.

[246] C. Q. Lauter and V. Lefèvre. An efficient rounding boundary test for pow(x, y) in double precision. *IEEE Transactions on Computers*, 58(2):197–207, February 2009.

[247] B. Lee and N. Burgess. Parameterisable floating-point operations on FPGA. In *Thirty-Sixth Asilomar Conference on Signals, Systems, and Computers*, volume 2, pages 1064–1068, November 2002.

[248] V. Lefèvre. Multiplication by an integer constant. Technical Report RR1999-06, Laboratoire de l'Informatique du Parallélisme, Lyon, France, 1999.

[249] V. Lefèvre. *Moyens Arithmétiques Pour un Calcul Fiable*. Ph.D. thesis, École Normale Supérieure de Lyon, Lyon, France, 2000.

[250] V. Lefèvre. New results on the distance between a segment and $\mathbb{Z}^2$. Application to the exact rounding. In *Proceedings of the 17th IEEE Symposium on Computer Arithmetic (ARITH-17)*, pages 68–75. IEEE Computer Society Press, Los Alamitos, CA, June 2005.

[251] V. Lefèvre and J.-M. Muller. Worst cases for correct rounding of the elementary functions in double precision. In N. Burgess and L. Ciminiera, editors, *Proceedings of the 15th IEEE Symposium on Computer Arithmetic (ARITH-16)*, Vail, CO, June 2001.

[252] V. Lefèvre, J.-M. Muller, and A. Tisserand. Toward correctly rounded transcendentals. *IEEE Transactions on Computers*, 47(11):1235–1243, November 1998.

[253] V. Lefèvre, D. Stehlé, and P. Zimmermann. Worst cases for the exponential function in the IEEE 754r decimal64 format. In *Reliable Implementation of Real Number Algorithms: Theory and Practice*, volume 5045 of *Lecture Notes in Computer Sciences*, pages 114–126. Springer, Berlin, 2008.

[254] V. Lefèvre. The Euclidean division implemented with a floating-point division and a floor. Research report RR-5604, INRIA, June 2005.

[255] A. K. Lenstra, H. W. Lenstra, Jr., and L. Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261:515–534, 1982.

[256] R.-C. Li, S. Boldo, and M. Daumas. Theorems on efficient argument reduction. In J.-C. Bajard and M. J. Schulte, editors, *Proceedings of the 16th IEEE Symposium on Computer Arithmetic (ARITH16)*, pages 129–136. IEEE Computer Society Press, Los Alamitos, CA, June 2003.

[257] X. Li, J. Demmel, D. H. Bailey, G. Henry, Y. Hida, J. Iskandar, W. Kahan, A. Kapur, M. Martin, T. Tung, and D. J. Yoo. Design, implementation

and testing of extended and mixed precision BLAS. Technical Report 45991, Lawrence Berkeley National Laboratory, 2000. `http://crd.lbl.gov/~xiaoye/XBLAS`.

[258] X. Li, J. Demmel, D. H. Bailey, G. Henry, Y. Hida, J. Iskandar, W. Kahan, A. Kapur, M. Martin, T. Tung, and D. J. Yoo. Design, implementation and testing of extended and mixed precision BLAS. *ACM Transactions on Mathematical Software*, 28(2):152–205, 2002.

[259] Y. Li and W. Chu. Implementation of single precision floating-point square root on FPGAs. In *FPGAs for Custom Computing Machines*, pages 56–65. IEEE, 1997.

[260] G. Lienhart, A. Kugel, and R. Männer. Using floating-point arithmetic on FPGAs to accelerate scientific N-body simulations. In *FPGAs for Custom Computing Machines*. IEEE, 2002.

[261] W. B. Ligon, S. McMillan, G. Monn, K. Schoonover, F. Stivers, and K. D. Underwood. A re-evaluation of the practicality of floating-point operations on FPGAs. In *FPGAs for Custom Computing Machines*. IEEE, 1998.

[262] T. Lindholm and F. Yellin. *The Java^{TM}Virtual Machine Specification, Second Edition*. Sun Microsystems, 1999.

[263] S. Linnainmaa. Software for doubled-precision floating-point computations. *ACM Transactions on Mathematical Software*, 7(3):272–283, 1981.

[264] J. Liouville. Sur des classes très étendues de quantités dont la valeur n'est ni algébrique ni même réductible à des irrationnelles algébriques. *J. Math. Pures Appl.*, 16:133–142, 1851.

[265] J. Liu, M. Chang, and C.-K. Cheng. An iterative division algorithm for FPGAs. In *Field-Programmable Gate Arrays*, pages 83–89. ACM, 2006.

[266] N. Louvet. *Algorithmes Compensés en Arithmétique Flottante: Précision, Validation, Performances*. Ph.D. thesis, Université de Perpignan, Perpignan, France, November 2007. In French.

[267] L. Lovász. *An algorithmic theory of numbers, graphs and convexity*, volume 50 of *CBMS-NSF Regional Conference Series in Applied Mathematics*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1986.

[268] E. Low and W. Walster. Rump's example revisited. *Reliable Computing*, 8(3):245–248, 2002.

[269] M. A. Malcolm. Algorithms to reveal properties of floating-point arithmetic. *Communications of the ACM*, 15(11):949–951, November 1972.

[270] P. Markstein. *IA-64 and Elementary Functions: Speed and Precision*. Hewlett-Packard Professional Books. Prentice-Hall, Englewood Cliffs, NJ, 2000.

[271] P. W. Markstein. Computation of elementary functions on the IBM RISC System/6000 processor. *IBM Journal of Research and Development*, 34(1):111–119, January 1990.

[272] D. W. Matula. In-and-out conversions. *Communications of the ACM*, 11(1):47–50, January 1968.

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

[274] G. Melquiond. Proving bounds on real-valued functions with computations. In A. Armando, P. Baumgartner, and G. Dowek, editors, *Proceedings of the 4th International Joint Conference on Automated Reasoning*, volume 5195 of *Lecture Notes in Artificial Intelligence*, pages 2–17, Sydney, Australia, 2008.

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

[276] D. Micciancio. The hardness of the closest vector problem with preprocessing. *IEEE Trans. Inform. Theory*, 47(3):1212–1215, 2001.

[277] D. Micciancio and S. Goldwasser. *Complexity of lattice problems: a cryptographic perspective*, volume 671 of *The Kluwer International Series in Engineering and Computer Science*. Kluwer Academic Publishers, Boston, MA, 2002.

[278] H. Minkowski. *Geometrie der Zahlen*. Teubner, Leipzig, 1896. In German.

[279] O. Møller. Quasi double-precision in floating-point addition. *BIT*, 5:37–50, 1965.

[280] D. Monniaux. The pitfalls of verifying floating-point computations. *ACM TOPLAS*, 30(3):1–41, 2008. A preliminary version is available at http://hal.archives-ouvertes.fr/hal-00128124.

[281] R. K. Montoye, E. Hokonek, and S. L. Runyan. Design of the IBM RISC System/6000 floating-point execution unit. *IBM Journal of Research and Development*, 34(1):59–70, 1990.

[282] P. Montuschi and P. M. Mezzalama. Survey of square rooting algorithms. *Computers and Digital Techniques, IEE Proceedings E.*, 137(1):31–40, 1990.

[283] J. S. Moore, T. Lynch, and M. Kaufmann. A mechanically checked proof of the correctness of the kernel of the AMD5K86 floating point division algorithm. *IEEE Transactions on Computers*, 47(9):913–926, September 1998.

[284] R. E. Moore. *Interval Analysis*. Prentice-Hall, Englewood Cliffs, NJ, 1963.

[285] R. E. Moore and F. Bierbaum. *Methods and applications of interval analysis*. SIAM Studies in Applied Mathematics, Philadelphia, PA, 1979.

[286] R. E. Moore, R. B. Kearfott, and M. J. Cloud. *Introduction to Interval Analysis*. SIAM, Philadelphia, PA, 2009.

[287] S. K. Moore. Intel makes a big jump in computer math. *IEEE Spectrum*, February 2008.

[288] T. Mulders. On short multiplications and divisions. *Appl. Algebra Eng. Commun. Comput.*, 11(1):69–88, 2000.

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

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

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

[292] J.-M. Muller. On the definition of ulp(x). Technical Report 2005-09, LIP Laboratory, ENS Lyon, `ftp://ftp.ens-lyon.fr/pub/LIP/Rapports/RR/RR2005/RR2005-09.pdf`, 2005.

[293] J.-M. Muller. *Elementary Functions, Algorithms and Implementation*. Birkhäuser Boston, MA, 2nd edition, 2006.

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

[295] A. Naini, A. Dhablania, W. James, and D. Das Sarma. 1-GHz HAL SPARC64 dual floating-point unit with RAS features. In N. Burgess and L. Ciminiera, editors, *Proceedings of the 15th IEEE Symposium on Computer Arithmetic (ARITH-16)*, pages 174–183, Vail, CO, June 2001.

[296] R. Nave. Implementation of transcendental functions on a numerics processor. *Microprocessing and Microprogramming*, 11:221–225, 1983.

[297] Y. V. Nesterenko and M. Waldschmidt. On the approximation of the values of exponential function and logarithm by algebraic numbers (in Russian). *Mat. Zapiski*, 2:23–42, 1996. Available in English at `http://www.math.jussieu.fr/~miw/articles/ps/Nesterenko.ps`.

[298] H. Neto and M. Véstias. Decimal multiplier on FPGA using embedded binary multipliers. In *Field Programmable Logic and Applications*, pages 197–202, 2008.

[299] A. Neumaier. Rundungsfehleranalyse einiger Verfahren zur Summation endlicher Summen. *ZAMM*, 54:39–51, 1974. In German.

[300] A. Neumaier. *Interval methods for systems of equations*. Cambridge University Press, Cambridge, UK, 1990.

[301] I. Newton. *Methodus Fluxionem et Serierum Infinitarum*. 1664–1671.

[302] K. C. Ng. Argument reduction for huge arguments: Good to the last bit. Technical report, SunPro, 1992.

[303] P. Nguyen and D. Stehlé. Floating-point LLL revisited. In *Proceedings of Eurocrypt 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 215–233. Springer, Berlin, 2005.

[304] P. Nguyen and D. Stehlé. LLL on the average. In *Proceedings of ANTS VII*, volume 4078 of *Lecture Notes in Computer Science*, pages 238–256. Springer, Berlin, 2006.

[305] K. R. Nichols, M. A. Moussa, and S. M. Areibi. Feasibility of floating-point arithmetic in FPGA based artificial neural networks. In *CAINE*, pages 8–13, 2002.

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

[307] S. F. Oberman. Floating point division and square root algorithms and implementation in the AMD-K7 microprocessor. In I. Koren and P. Kornerup, editors, *Proceedings of the 14th IEEE Symposium on Computer Arithmetic (Adelaide, Australia)*, pages 106–115. IEEE Computer Society Press, Los Alamitos, CA, April 1999.

[308] S. F. Oberman, H. Al-Twaijry, and M. J. Flynn. The SNAP project: design of floating-point arithmetic units. In *13th Symposium on Computer Arithmetic*. IEEE Computer Society, 1997.

[309] S. F. Oberman and M. J. Flynn. Design issues in division and other floating-point operations. *IEEE Transactions on Computers*, 46(2):154–161, February 1997.

[310] S. F. Oberman and M. J. Flynn. Division algorithms and implementations. *IEEE Transactions on Computers*, 46(8):833–854, 1997.

[311] S. F. Oberman and M. Y. Siu. A high-performance area-efficient multifunction interpolator. In *Proceedings of the 17th IEEE Symposium on Computer Arithmetic (ARITH-17)*. IEEE Computer Society Press, June 2005.

[312] A. M. Odlyzko and H. J. J. te Riele. Disproof of the Mertens conjecture. *J. Reine Angew. Math.*, 357:138–160, 1985.

[313] T. Ogita, S. M. Rump, and S. Oishi. Accurate sum and dot product. *SIAM Journal on Scientific Computing*, 26(6):1955–1988, 2005.

[314] T. Ogita, S. M. Rump, and S. Oishi. Verified solutions of linear systems without directed rounding. Technical Report 2005-04, Advanced Research Institute for Science and Engineering, Waseda University, Tokyo, Japan, 2005.

[315] H.-J. Oh, S. M. Mueller, C. Jacobi, K. D. Tran, S. R. Cottier, B. W. Michael, H. Nishikawa, Y. Totsuka, T. Namatame, N. Yano, T. Machida, and S. H. Dhong. A fully pipelined single-precision floating-point unit in the synergistic processor element of a CELL processor. *IEEE Journal of Solid-State Circuits*, 41(4):759–771, April 2006.

[316] V. G. Oklobdzija. An algorithmic and novel design of a leading zero detector circuit: Comparison with logic synthesis. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2, March 1994.

[317] F. W. J. Olver. Error analysis of complex arithmetic. In *Computational Aspects of Complex Analysis*, volume 102 of *NATO Advanced Study Institute Series C*, 1983.

[318] 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.

[319] F. Ortiz, J. Humphrey, J. Durbano, and D. Prather. A study on the design of floating-point functions in FPGAs. In *Field Programmable Logic and Applications*, volume 2778 of *LNCS*, pages 1131–1135. Springer, Berlin, 2003.

[320] M. A. Overton. *Numerical Computing with IEEE Floating-Point Arithmetic*. SIAM, Philadelphia, PA, 2001.

[321] A. Panhaleux. *Génération d'itérations de type Newton-Raphson pour la division de deux flottants à l'aide d'un FMA, MSc Thesis*. Technical report, École Normale Supérieure de Lyon, Lyon, France, 2008. In French.

[322] B. Parhami. On the complexity of table lookup for iterative division. *IEEE Transactions on Computers*, C-36(10):1233–1236, 1987.

[323] B. Parhami. *Computer Arithmetic: Algorithms and Hardware Designs*. Oxford University Press, New York, NY, 2000.

[324] M. Parks. Number-theoretic test generation for directed roundings. *IEEE Transactions on Computers*, 49(7):651–658, 2000.

[325] M. Parks. Unifying tests for square root. In Hanrot and Zimmermann, editors, *Real Numbers and Computers*, pages 125–133, July 2006.

[326] D. Patil, O. Azizi, M. Horowitz, R. Ho, and R. Ananthraman. Robust energy-efficient adder topologies. In P. Kornerup and J.-M. Muller, editors, *Proceedings of the 18th IEEE Symposium on Computer Arithmetic (ARITH-18)*, pages 29–37. IEEE Computer Society Conference Publishing Services, June 2007.

[327] M. Payne and R. Hanek. Radian reduction for trigonometric functions. *SIGNUM Newsletter*, 18:19–24, 1983.

[328] C. Peikert. Limits on the hardness of lattice problems in l_p norms. *Computational Complexity*, 17(2):300–351, 2008.

[329] P. Pélissier and P. Zimmermann. The DPE library. Available at `http://www.loria.fr/~zimmerma/free/dpe-1.4.tar.gz`.

[330] C. Percival. Rapid multiplication modulo the sum and difference of highly composite numbers. *Mathematics of Computation*, 72:387–395, 2002.

[331] O. Perron. *Die Lehre von den Kettenbrüchen, 3. verb. und erweiterte Aufl.* Teubner, Stuttgart, 1954–57.

[332] M. Pichat. Correction d'une somme en arithmétique à virgule flottante. *Numerische Mathematik*, 19:400–406, 1972. In French.

[333] R. V. K Pillai, D. Al-Khalili, and A. J. Al-Khalili. A low power approach to floating point adder design. In *International Conference on Computer Design*. IEEE Computer Society, 1997.

[334] J. A. Pineiro and J. D. Bruguera. High-speed double-precision computation of reciprocal, division, square root, and inverse square root. *IEEE Transactions on Computers*, 51(12):1377–1388, December 2002.

[335] S. Pion. *De la Géométrie Algorithmique au Calcul Géométrique*. Ph.D. thesis, Université de Nice Sophia-Antipolis, France, November 1999. In French.

[336] D. M. Priest. Algorithms for arbitrary precision floating point arithmetic. In P. Kornerup and D. W. Matula, editors, *Proceedings of the 10th IEEE Symposium on Computer Arithmetic (Arith-10)*, pages 132–144. IEEE Computer Society Press, Los Alamitos, CA, June 1991.

[337] D. M. Priest. *On Properties of Floating-Point Arithmetics: Numerical Stability and the Cost of Accurate Computations*. Ph.D. thesis, University of California at Berkeley, 1992.

[338] D. M. Priest. Efficient scaling for complex division. *ACM Transactions on Mathematical Software*, 30(4), December 2004.

[339] E. Quinnell, E. E. Swartzlander, and C. Lemonds. Floating-point fused multiply-add architectures. In *Forty-First Asilomar Conference on Signals, Systems, and Computers*, pages 331–337, November 2007.

[340] S.-K. Raina. *FLIP: a Floating-point Library for Integer Processors*. Ph.D. thesis, École Normale Supérieure de Lyon, September 2006. Available at `http://www.ens-lyon.fr/LIP/Pub/PhD2006.php`.

[341] 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.

[342] E. Remez. Sur un procédé convergent d'approximations successives pour déterminer les polynômes d'approximation. *C.R. Académie des Sciences, Paris*, 198:2063–2065, 1934.

[343] T. J. Rivlin. *Chebyshev polynomials. From approximation theory to algebra*. Pure and Applied Mathematics. John Wiley & Sons, New York, 2nd edition, 1990.

[344] J. E. Robertson. A new class of digital division methods. *IRE Transactions on Electronic Computers*, EC-7:218–222, 1958. Reprinted in E. E. Swartzlander, *Computer Arithmetic*, Vol. 1, IEEE Computer Society Press, Los Alamitos, CA, 1990.

[345] E. Roesler and B. Nelson. Novel optimizations for hardware floating-point units in a modern FPGA architecture. In *Field Programmable Logic and Applications*, volume 2438 of *LNCS*, pages 637–646. Springer, Berlin, 2002.

[346] R. Rojas. Konrad Zuse's legacy: the architecture of the Z1 and Z3. *IEEE Annals of the History of Computing*, 19(2), 1997.

[347] 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.

[348] K. F. Roth. Rational approximations to algebraic numbers. *Mathematika*, 2:1–20, 1955.

[349] S. Rump. Solving algebraic problems with high accuracy (habilitationsschrift). In Kulisch and Miranker, editors, *A New Approach to Scientific Computation*, pages 51–120. Academic Press, New york, NY, 1983.

[350] S. Rump and H. Böhm. Least significant bit evaluation of arithmetic expressions in single-precision. *Computing*, 30:189–199, 1983.

[351] S. Rump, P. Zimmermann, S. Boldo, and G. Melquiond. Interval operations in rounding to nearest. *BIT*, 2009. To appear.

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

[353] S. M. Rump, T. Ogita, and S. Oishi. Accurate floating-point summation part II: Sign, K-fold faithful and rounding to nearest. *SIAM Journal on Scientific Computing*, 2005–2008. Submitted for publication.

[354] S. M. Rump, T. Ogita, and S. Oishi. Accurate floating-point summation part I: Faithful rounding. *SIAM Journal on Scientific Computing*, 31(1):189–224, 2008.

[355] D. M. Russinoff. A mechanically checked proof of IEEE compliance of a register-transfer-level specification of the AMD-K7 floating-point multiplication, division, and square root instructions. *LMS Journal of Computation and Mathematics*, 1:148–200, 1998.

[356] D. M. Russinoff. A mechanically checked proof of correctness of the AMD K5 floating point square root microcode. *Formal Methods in System Design*, 14(1):75–125, 1999.

[357] D. M. Russinoff. A case study in formal verification of register-transfer logic with ACL2: The floating point adder of the AMD Athlon processor. *Lecture Notes in Computer Science*, 1954:3–36, 2000.

[358] E. Salamin. Computation of π using arithmetic-geometric mean. *Mathematics of Computation*, 30:565–570, 1976.

[359] J.-L. Sanchez, A. Jimeno, H. Mora, J. Mora, and F. Pujol. A CORDIC-based architecture for high performance decimal calculation. In *IEEE International Symposium on Industrial Electronics*, pages 1951–1956, June 2007.

[360] D. Das Sarma and D. W. Matula. Measuring the accuracy of ROM reciprocal tables. *IEEE Transactions on Computers*, 43(8):932–940, August 1994.

[361] D. Das Sarma and D. W. Matula. Faithful bipartite ROM reciprocal tables. In Knowles and McAllister, editors, *Proceedings of the 12th IEEE Symposium on Computer Arithmetic (ARITH-12)*, pages 17–28. IEEE Computer Society Press, Los Alamitos, CA, June 1995.

[362] D. Das Sarma and D. W. Matula. Faithful interpolation in reciprocal tables. In T. Lang, J.-M. Muller, and N. Takagi, editors, *Proceedings of the 13th IEEE Symposium on Computer Arithmetic*, pages 82–91. IEEE Computer Society Press, Los Alamitos, CA, July 1997.

[363] H. Schmid and A. Bogacki. Use decimal CORDIC for generation of many transcendental functions. *EDN*, pages 64–73, February 1973.

[364] M. M. Schmookler and K. J. Nowka. Leading zero anticipation and detection – a comparison of methods. In N. Burgess and L. Ciminiera, editors, *Proceedings of the 15th IEEE Symposium on Computer Arithmetic (ARITH-16)*, pages 7–12, Vail, CO, June 2001.

[365] C.-P. Schnorr. A more efficient algorithm for lattice basis reduction. *J. Algorithms*, 9(1):47–62, 1988.

[366] A. Schönhage. Schnelle Berechnung von Kettenbruchentwicklungen. *Acta Informatica*, 1:139–144, 1971. In German.

[367] A. Schönhage, A. F. W. Grotefeld, and E. Vetter. *Fast algorithms: a Multitape Turing Machine Implementation*. Bibliographisches Institut, Mannheim, 1994.

[368] A. Schönhage and V. Strassen. Schnelle Multiplikation Grosser Zahlen. *Computing*, 7:281–292, 1971. In German.

[369] M. J. Schulte and E. E. Swartzlander. Exact rounding of certain elementary functions. In E. E. Swartzlander, M. J. Irwin, and G. Jullien, editors, *Proceedings of the 11th IEEE Symposium on Computer Arithmetic*, pages 138–145. IEEE Computer Society Press, Los Alamitos, CA, June 1993.

[370] E. M. Schwarz, M. M. Schmookler, and S. D. Trong. Hardware implementations of denormalized numbers. In *ARITH '03: Proceedings of the 16th IEEE Symposium on Computer Arithmetic (ARITH-16'03)*, page 70, Washington, DC, 2003. IEEE Computer Society.

[371] E. M. Schwarz, M. M. Schmookler, and S. D. Trong. FPU implementations with denormalized numbers. *IEEE Transactions on Computers*, 54(7):825–836, 2005.

[372] P. Sebah and X. Gourdon. Newton's method and high-order iterations. Technical report, 2001. `http://numbers.computation.free.fr/Constants/Algorithms/newton.html`.

[373] P.-M. Seidel. Multiple path IEEE floating-point fused multiply-add. In *46th International Midwest Symposium on Circuits and Systems*, pages 1359–1362. IEEE, 2003.

[374] P.-M. Seidel and G. Even. How many logic levels does floating-point addition require. In *International Conference on Computer Design*, pages 142–149, 1998.

[375] P.-M. Seidel and G. Even. On the design of fast IEEE floating-point adders. In Burgess and Ciminiera, editors, *Proceedings of the 15th IEEE Symposium on Computer Arithmetic (ARITH-16)*, pages 184–194, Vail, CO, June 2001.

[376] A. M. Shams and M. A. Bayoumi. A novel high-performance CMOS 1-bit full-adder cell. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 47(5), 2000.

[377] J. R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete Computational Geometry*, 18:305–363, 1997.

[378] N. Shirazi, A. Walters, and P. Athanas. Quantitative analysis of floating point arithmetic on FPGA based custom computing machine. In *FPGAs for Custom Computing Machines*, pages 155–162. IEEE, 1995.

[379] V. Shoup. NTL, a library for doing number theory, version 5.4. `http://shoup.net/ntl/`, 2005.

[380] R. A. Smith. A continued-fraction analysis of trigonometric argument reduction. *IEEE Transactions on Computers*, 44(11):1348–1351, November 1995.

[381] R. L. Smith. Algorithm 116: Complex division. *Communications of the ACM*, 5(8):435, 1962.

[382] V. T. Sós. On the distribution mod 1 of the sequence $n\alpha$. *Ann. Univ. Sci. Budapest, Eötvös Sect. Math.*, 1:127–134, 1958.

[383] E. Sprangle and D. Carmean. Increasing processor performance by implementing deeper pipelines. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 25–34. IEEE Computer Society, 2002.

[384] H. M. Stark. *An Introduction to Number Theory*. MIT Press, Cambridge, MA, 1981.

[385] G. L. Steele Jr. and J. L. White. How to print floating-point numbers accurately. *ACM SIGPLAN Notices*, 25(6):112–126, June 1990.

[386] G. L. Steele Jr. and J. L. White. Retrospective: how to print floating-point numbers accurately. *ACM SIGPLAN Notices*, 39(4):372–389, april 2004.

[387] D. Stehlé. *Algorithmique de la Réduction de Réseaux et Application à la Recherche de Pires Cas pour l'Arrondi de Fonctions Mathématiques*. Ph.D. thesis, Université Henri Poincaré – Nancy 1, France, December 2005.

[388] D. Stehlé. On the randomness of bits generated by sufficiently smooth functions. In F. Hess, S. Pauli, and M. E. Pohst, editors, *Proceedings of the 7th Algorithmic Number Theory Symposium, ANTS VII*, volume 4078 of *Lecture Notes in Computer Science*, pages 257–274. Springer-Verlag, Berlin, 2006.

[389] D. Stehlé, V. Lefèvre, and P. Zimmermann. Worst cases and lattice reduction. In J.-C. Bajard and M. J. Schulte, editors, *Proceedings of the 16th IEEE Symposium on Computer Arithmetic (ARITH-16)*, pages 142–147. IEEE Computer Society Press, Los Alamitos, CA, June 2003.

[390] D. Stehlé, V. Lefèvre, and P. Zimmermann. Searching worst cases of a one-variable function. *IEEE Transactions on Computers*, 54(3):340–346, March 2005.

[391] D. Stehlé and P. Zimmermann. Gal's accurate tables method revisited. In *Proceedings of the 17th IEEE Symposium on Computer Arithmetic (ARITH-17)*. IEEE Computer Society Press, Los Alamitos, CA, June 2005.

[392] P. H. Sterbenz. *Floating-Point Computation*. Prentice-Hall, Englewood Cliffs, NJ, 1974.

[393] G. W. Stewart. A note on complex division. *ACM Transactions on Mathematical Software*, 11(3):238–241, September 1985.

[394] J. E. Stine and M. J. Schulte. The symmetric table addition method for accurate function approximation. *Journal of VLSI Signal Processing*, 21:167–177, 1999.

[395] A. Storjohann. Faster algorithms for integer lattice basis reduction. Technical report, ETH Zürich, 1996.

[396] Sun. *Numerical Computation Guide – Sun^{TM} Studio 11*, 2005. Available at `http://docs.sun.com/source/819-3693/`.

[397] Sun Microsystems. Interval arithmetic in high performance technical computing. Technical report, 2002.

[398] D. A. Sunderland, R. A. Strauch, S. W. Wharfield, H. T. Peterson, and C. R. Cole. CMOS/SOS frequency synthesizer LSI circuit for spread spectrum communications. *IEEE Journal of Solid State Circuits*, SC-19(4):497–506, 1984.

[399] J. Surányi. Über die Anordnung der Vielfachen einer reellen Zahl mod 1. *Ann. Univ. Sci. Budapest, Eötvös Sect. Math.*, 1:107–111, 1958.

[400] 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.

[401] S. Swierczkowski. On successive settings of an arc on the circumference of a circle. *Fundamenta Math.*, 46:187–189, 1958.

[402] N. Takagi. A hardware algorithm for computing the reciprocal square root. In N. Burgess and L. Ciminiera, editors, *Proceedings of the 15th IEEE Symposium on Computer Arithmetic (ARITH-16)*, pages 94–100, Vail, CO, June 2001.

[403] N. Takagi and S. Kuwahara. A VLSI algorithm for computing the Euclidean norm of a 3D vector. *IEEE Transactions on Computers*, 49(10):1074–1082, October 2000.

[404] P. T. P. Tang. Table-driven implementation of the exponential function in IEEE floating-point arithmetic. *ACM Transactions on Mathematical Software*, 15(2):144–157, June 1989.

[405] P. T. P. Tang. Table-driven implementation of the logarithm function in IEEE floating-point arithmetic. *ACM Transactions on Mathematical Software*, 16(4):378–400, December 1990.

[406] P. T. P. Tang. Table lookup algorithms for elementary functions and their error analysis. In P. Kornerup and D. W. Matula, editors, *Proceedings of the 10th IEEE Symposium on Computer Arithmetic*, pages 232–236, Los Alamitos, CA, June 1991. IEEE Computer Society Press.

[407] P. T. P. Tang. Table-driven implementation of the expm1 function in IEEE floating-point arithmetic. *ACM Transactions on Mathematical Software*, 18(2):211–222, June 1992.

[408] K. D. Tocher. Techniques of multiplication and division for automatic binary computers. *Quarterly Journal of Mechanics and Applied Mathematics*, 11(3):364–384, 1958.

[409] A. L. Toom. The complexity of a scheme of functional elements realizing the multiplication of integers. *Soviet Mathematics Doklady*, 3:714–716, 1963.

[410] S. D. Trong, M. Schmookler, E. M. Schwarz, and M. Kroener. P6 binary floating-point unit. In *18th Symposium on Computer Arithmetic*, pages 77–86. IEEE, 2007.

[411] A. Tyagi. A reduced-area scheme for carry-select adders. *IEEE Transactions on Computers*, 42(10):1163–1170, 1993.

[412] P. van Emde Boas. Another NP-complete problem and the complexity of computing short vectors in a lattice. Technical Report 81-04, Mathematisch Instituut, University of Amsterdam, 1981.

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

[414] 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.

[415] L. Veidinger. On the numerical determination of the best approximations in the Chebyshev sense. *Numerische Mathematik*, 2:99–105, 1960.

[416] G. W. Veltkamp. ALGOL procedures voor het berekenen van een inwendig product in dubbele precisie. Technical Report 22, RC-Informatie, Technishe Hogeschool Eindhoven, 1968.

[417] G. W. Veltkamp. ALGOL procedures voor het rekenen in dubbele lengte. Technical Report 21, RC-Informatie, Technishe Hogeschool Eindhoven, 1969.

[418] B. Verdonk, A. Cuyt, and D. Verschaeren. A precision- and range-independent tool for testing floating-point arithmetic. I: Basic operations, square root, and remainder. *ACM Transactions on Mathematical Software*, 27(1):92–118, 2001.

[419] B. Verdonk, A. Cuyt, and D. Verschaeren. A precision- and range-independent tool for testing floating-point arithmetic. II: Conversions. *ACM Transactions on Mathematical Software*, 27(1):119–140, 2001.

[420] D. Villeger and V. G. Oklobdzija. Evaluation of Booth encoding techniques for parallel multiplier implementation. *Electronics Letters*, 29(23):2016–2017, November 1993.

[421] J. E. Volder. The CORDIC computing technique. *IRE Transactions on Electronic Computers*, EC-8(3):330–334, 1959. Reprinted in E. E. Swartzlander, *Computer Arithmetic*, Vol. 1, IEEE Computer Society Press, Los Alamitos, CA, 1990.

[422] J. E. Volder. The birth of CORDIC. *Journal of VLSI Signal Processing Systems*, 25(2):101–105, June 2000.

[423] Y. Voronenko and M. Püschel. Multiplierless multiple constant multiplication. *ACM Trans. Algorithms*, 3(2), 2007.

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

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

[426] J. S. Walther. A unified algorithm for elementary functions. In *Joint Computer Conference Proceedings*, 1971. Reprinted in E. E. Swartzlander, *Computer Arithmetic*, Vol. 1, IEEE Computer Society Press, Los Alamitos, CA,1990.

[427] J. S. Walther. The story of unified CORDIC. *Journal of VLSI Signal Processing Systems*, 25(2):107–112, June 2000.

[428] 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.

[429] L.-K. Wang, M. J. Schulte, J. D. Thompson, and N. Jairam. Hardware designs for decimal floating-point addition and related operations. *IEEE Transactions on Computers*, 58(2):322–335, March 2009.

[430] X. Wang, S. Braganza, and M. Leeser. Advanced components in the variable precision floating-point library. In *Field-Programmable Custom Computing Machines*, pages 249–258, 2006.

[431] Wikipedia. Elementary function — wikipedia, the free encyclopedia, 2008. [Online; accessed 8-April-2008].

[432] Wikipedia. Endianness — wikipedia, the free encyclopedia, 2008. [Online; accessed 29-August-2008].

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

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

[435] J. H. Wilkinson. *Rounding Errors in Algebraic Processes*. Prentice-Hall, Englewood Cliffs, NJ, 1963.

[436] J. H. Wilkinson. *The Algebraic Eigenvalue Problem*. Oxford University Press, London, 1965.

[437] M. J. Wirthlin. Constant coefficient multiplication using look-up tables. *VLSI Signal Processing*, 36(1):7–15, 2004.

[438] W. F. Wong and E. Goto. Fast hardware-based algorithms for elementary function computations using rectangular multipliers. *IEEE Transactions on Computers*, 43(3):278–294, March 1994.

[439] X. Y. Yu, Y.-H. Chan, B. Curran, E. Schwarz, M. Kelly, and B. Fleischer. A 5GHz+ 128-bit binary floating-point adder for the POWER6 processor. In *European Solid-State Circuits Conference*, pages 166–169, 2006.

[440] N. Zhuang and H. Wu. A new design of the CMOS full adder. *IEEE Journal on Solid-State Circuits*, 27:840–844, 1992.

[441] L. Zhuo and V. Prasanna. Scalable and modular algorithms for floating-point matrix multiplication on FPGAs. In *Reconfigurable Architecture Workshop*. IEEE, 2004.

[442] L. Zhuo and V. K. Prasanna. High performance linear algebra operations on reconfigurable systems. In *Supercomputing*. IEEE, 2005.

[443] R. Zimmermann. *Binary Adder Architectures for Cell-Based VLSI and Their Synthesis*. Ph.D. thesis, Swiss Federal Institute of Technology, Zurich, 1997.

[444] A. Ziv. Fast evaluation of elementary mathematical functions with correctly rounded last bit. *ACM Transactions on Mathematical Software*, 17(3):410–423, September 1991.

[445] R. Zumkeller. Formal global optimisation with Taylor models. In U. Furbach and N. Shankar, editors, *Proceedings of the 3th International Joint Conference on Automated Reasoning*, volume 4130 of *Lecture Notes in Computer Science*, pages 408–422, Seattle, WA, August 2006. Springer, Berlin.

[446] D. Zuras. More on squaring and multiplying large integers. *IEEE Transactions on Computers*, 43(8):899–908, August 1994.

[447] V. Zyuban, D. Brrok, V. Srinivasan, M. Gschwind, P. Bose, P. N. Strenski, and P. G. Emma. Integrated analysis of power and performance for pipelined microprocessor. *IEEE Transactions on Computers*, 53(8):1004–1016, 2004.

Index