Introduction to Computer Arithmetic

2.1 Basic Notions of Floating-Point Arithmetic

The aim of this section is to provide the reader with some basic concepts of floating-point arithmetic, and to define notations that are used throughout the book. For further information, the reader is referred to the IEEE-754-2008 Standard on Floating-Point Arithmetic [245] and to our Handbook on Floating-Point Arithmetic [356]. Interesting and useful material can be found in Goldberg's paper [206] and Kahan's lecture notes [265]. Further information can be found in [55, 95, 102, 103, 180, 217, 232, 266, 277, 286, 370, 373, 378, 469, 477]. Here we mainly focus on the binary formats specified by the 2008 release of the IEEE-754 standard for floating-point arithmetic. The first release of IEEE 754 [6], that goes back to 1985, was a key factor in improving the quality of the computational environment available to programmers. Before the standard, floating-point arithmetic was a mere set of cooking recipes that sometimes worked well and sometimes did not work at all.[1]

2.1.1 Basic Notions

Everybody knows that a radix-β, precision-p floating-point number is a number of the form

$$\pm m \times \beta^e, \tag{2.1}$$

where m is represented with p digits in radix β, $m < \beta$, and e is an integer. However, being able to build trustable algorithms and proofs requires a more formal definition.

A floating-point format is partly[2] characterized by four integers:

- a *radix* (or *base*) $\beta \geq 2$;
- a *precision* $p \geq 2$ (p is the number of "significant digits" of the representation);
- two *extremal exponents* $e_{\min}$ and $e_{\max}$ such that $e_{\min} < e_{\max}$. In all practical cases, $e_{\min} < 0 < e_{\max}$.

---

[1] We should mention a few exceptions, such as some HP pocket calculators and the Intel 8087 coprocessor, that were precursors of the standard.

[2] Partly only, because bit strings must be reserved for representing exceptional values, such as the results of forbidden operations (e.g., 0/0) and infinities.

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

$$x = M \cdot \beta^{e-p+1}, \tag{2.2}$$

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 $e_{\min} \leq e \leq e_{\max}$ is called the *exponent* of the representation of x.

We can now go back to (2.1), and notice that if we define $m = |M| \cdot \beta^{1-p}$ and $s = 0$ if $x \geq 0$, 1 otherwise, then

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

- m is called the *real significand* (or, more simply, the *significand* of the representation). It has one digit before the radix point, and at most $p - 1$ digits after; and
- s is the sign of x.

Notice that for some numbers x, there may exist several possible representations (M, e) or (s, m, e). Just consider the "toy format" $\beta = 10$ and $p = 4$. In that format $M = 4560$ and $e = -1$, and $M = 0456$ and $e = 0$ are valid representations of the number 0.456.

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*. 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 x is a *normal* number.
- Otherwise, one necessarily has $e = e_{\min}$, and the corresponding value x is called a *subnormal* number (the term *denormal number* is often used too). In that case, $|m| < 1$ or, equivalently, $|M| \leq \beta^{p-1} - 1$. Notice that a subnormal number is of absolute value less than $\beta^{e_{\min}}$: subnormal numbers are very tiny numbers.

An interesting consequence of that normalization, when the radix β is equal to 2, is that the first bit of the significand of a normal number must always be "1", and the first bit of the significand of a subnormal number must always be "0". Hence if we have information[3] on the normality of x there is no need to store its first significand bit, and in many computer systems, it is actually not stored (this is called the "hidden bit" or "implicit bit" convention). Table 2.1 gives the basic parameters of the floating-point systems that have been implemented in various machines. Those figures have been taken from references [232, 265, 277, 370]. For instance, the largest representable finite number in the IEEE-754 double-precision/binary64 format [245] is

$$\left(2 - 2^{-52}\right) \times 2^{1023} \approx 1.7976931348623157 \times 10^{308},$$

---

[3]In practice that information is encoded in the exponent field, see Section 2.1.6.

Table 2.1 *Basic parameters of various floating-point systems (p, the precision, is the size of the significand, expressed in number of digits in the radix of the computer system). The "+1" is due to the hidden bit convention. The binary32 and binary64 formats were called "single precision" and "double precision" in the* 1985 *release of the IEEE-754 standard.*

System	β	p	$e_{\min}$	$e_{\max}$	max. value
DEC VAX	2	24	-128	126	$1.7\cdots \times 10^{38}$
(D format)	2	56	-128	126	$1.7\cdots \times 10^{38}$
HP 28, 48G	10	12	-500	498	$9.9\cdots \times 10^{498}$
IBM 370	16	6 (24 bits)	-65	62	$7.2\cdots \times 10^{75}$
and 3090	16	14 (56 bits)	-65	62	$7.2\cdots \times 10^{75}$
IEEE-754 binary32	2	23+1	-126	127	$3.4\cdots \times 10^{38}$
IEEE-754 binary64	2	52+1	-1022	1023	$1.8\cdots \times 10^{308}$
IEEE-754 binary128	2	112+1	-16382	16383	$1.2\cdots \times 10^{4932}$
IEEE-754 decimal64	10	16	-383	384	$9.999\cdots 9 \times 10^{384}$

the smallest positive number is

$$2^{-1074} \approx 4.940656458412465 \times 10^{-324},$$

and the smallest positive normal number is

$$2^{-1022} \approx 2.225073858507201 \times 10^{-308}.$$

Arithmetic based on radix 10 has frequently been used in pocket calculators.[4] Also, it is used in financial calculations, and several decimal formats are specified by the 2008 version of IEEE 754. Decimal arithmetic remains an object of active study [114, 117, 183, 222, 453]. A Russian computer named SETUN [72] used radix 3 with digits $-1, 0$, and 1 (this is called the *balanced ternary system*). It was built[5] at Moscow University, during the 1960s [275]. Almost all other current computing systems use base 2. Various studies [55, 95, 286] have shown that radix 2 *with* the hidden bit convention gives better accuracy than all other radices (by the way, this does not imply that operations—e.g., divisions or square roots—cannot benefit from being done in a higher radix *inside* the arithmetic operators [181]).

2.1.2   Rounding Functions

Let us define a *machine number* to be a number that can be exactly represented in the floating-point system under consideration. In general, the sum, the product, and the quotient of two machine numbers is not a machine number and the result of such an arithmetic operation must be *rounded*.

In a floating-point system that follows the IEEE-754 standard, the user can choose a *rounding function* (also called *rounding mode*) from:

---

[4]A major difference between computers and pocket calculators is that usually computers do much computation between input and output of data, so that the time needed to perform a radix conversion is negligible compared to the whole processing time. If pocket calculators used radix 2, they would perform radix conversions before and after almost every arithmetic operation. Another reason for using radix 10 in pocket calculators is the fact that many simple decimal numbers such as 0.1 are not exactly representable in radix 2.

[5]See http://www.computer-museum.ru/english/setun.htm.

Figure 2.1 *Different possible roundings of a real number x in a radix-β floating-point system. In this example, x > 0.*
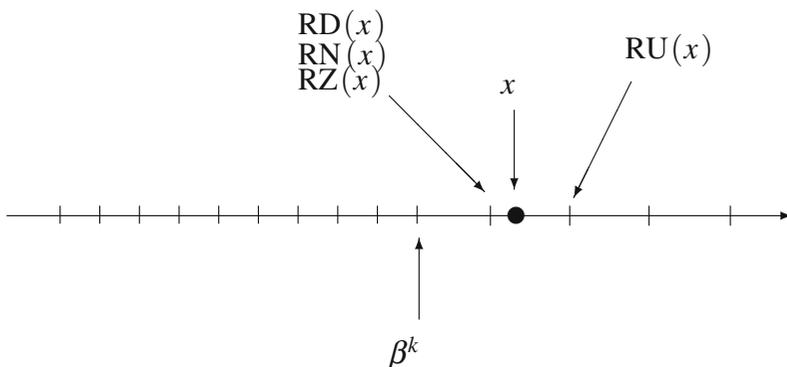
- rounding towards $-\infty$: RD(x) is the largest machine number less than or equal to x;
- rounding towards $+\infty$: RU(x) is the smallest machine number greater than or equal to x;
- rounding towards 0: RZ(x) is equal to RD(x) if $x \geq 0$, and to RU(x) if $x < 0$;
- rounding to nearest: RN(x) is the machine number that is the closest to x (if x is exactly halfway between two consecutive machine numbers, the default convention is to return the "even" one, i.e., the one whose last significand digit is even—a zero in radix 2).

This is illustrated using the example in Figure 2.1.

If the active rounding function is denoted by $\diamond$, and u and v are machine numbers, then the IEEE-754 standard [6, 109] requires that the obtained result should always be $\diamond(u \top v)$ when computing $u \top v$ ($\top$ is $+$, $-$, $\times$, or $\div$). Thus the system must behave as if the result were first computed *exactly*, with infinite precision, and then rounded. Operations that satisfy this property are called "correctly rounded" (or, sometimes, "exactly rounded"). There is a similar requirement for the square root. Such a requirement has a number of advantages:

- it leads to *full compatibility*[6] between computing systems: the same program will give the same values on different computers;
- many algorithms can be designed that use this property. Examples include performing large precision arithmetic [22, 231, 389, 423], designing "compensated" algorithms for evaluating with excellent accuracy the sum of several floating-point numbers [8, 264, 274, 386, 389, 404, 405], or making decisions in computational geometry [341, 387, 423];
- one can easily implement *interval arithmetic* [287, 288, 349], or more generally one can get lower or upper bounds on the exact result of a sequence of arithmetic operations;
- the mere fact that the arithmetic operations become fully specified makes it possible to elaborate formal proofs of programs and algorithms, which is very useful for certifying the behavior of numerical software used in critical applications [39, 41–44, 129, 133, 216–220, 315, 339, 340].

In radix-β, precision-p floating-point arithmetic, if an arithmetic operation is correctly rounded and there is no overflow or underflow[7] then the relative error of that operation is bounded by

---

[6]At least in theory: one must make sure that the order of execution of the operations is not changed by the compiler, that there are no phenomenons of "double roundings" due to the possible use of a wider format in intermediate calculations, and that an FMA instruction is called only if one has decided to use it.

[7]Let us say, as does the IEEE-754 standard, that an operation underflows when the result is subnormal *and* inexact.

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

if the rounding function is round to nearest, and

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

with the other rounding functions.

Very useful algorithms that can be proved assuming correct rounding are the error-free transforms presented in Section 2.2.1 (the first ideas that underlie them go back to Møller [346]).

An important property of the various rounding functions defined by the IEEE-754 standard is that they are *monotonic*. For instance, if $x \leq y$, then $\mathrm{RN}(x) \leq \mathrm{RN}(y)$.

In the 1985 version of the IEEE-754 standard, there was no correct rounding requirement for the elementary functions, probably because it had been believed for many years that correct rounding of the elementary functions would be much too expensive. The situation has changed significantly in the recent years [125, 131, 136] and with the 2008 release of the IEEE-754 standard, correct rounding of some functions becomes recommended (yet not mandatory). These functions are:

$$e^x, e^x - 1, 2^x, 2^x - 1, 10^x, 10^x - 1,$$
$$\ln(x), \log_2(x), \log_{10}(x), \ln(1+x), \log_2(1+x), \log_{10}(1+x),$$
$$\sqrt{x^2 + y^2}, 1/\sqrt{x}, (1+x)^n, x^n, x^{1/n} (n \text{ is an integer}), x^y,$$
$$\sin(\pi x), \cos(\pi x), \arctan(x)/\pi, \arctan(y/x)/\pi,$$
$$\sin(x), \cos(x), \tan(x), \arcsin(x), \arccos(x), \arctan(x), \arctan(y/x),$$
$$\sinh(x), \cosh(x), \tanh(x), \sinh^{-1}(x), \cosh^{-1}(x), \tanh^{-1}(x).$$

We analyze the problem of correctly rounding the elementary functions in Chapter 12. Another frequently used notion is *faithful rounding*: a function is *faithfully rounded* if the returned result is always one of the two floating-point numbers that surround the exact result, and is equal to the exact result whenever this one is exactly representable. Faithful rounding cannot rigorously be called a *rounding* since it is not a deterministic function.

The availability of subnormal numbers (see Section 2.1.1) is a feature of the IEEE-754 standard that offers nice properties at the price of a slight complication of some arithmetic operators. It allows underflow to be gradual (see Figure 2.2). The minimum subnormal positive number in the IEEE-754 double-precision/binary64 floating-point format is

$$2^{-1074} \approx 4.94065645841246544 \times 10^{-324}.$$

In a floating-point system with correct rounding and subnormal numbers, the following theorem holds.

Theorem 1 (Sterbenz Lemma) *In a floating-point system with correct rounding and subnormal numbers, if x and y are floating-point numbers such that*

$$x/2 \leq y \leq 2x,$$

then $x - y$ is a floating-point number, which implies that it will be computed exactly, with any rounding function.
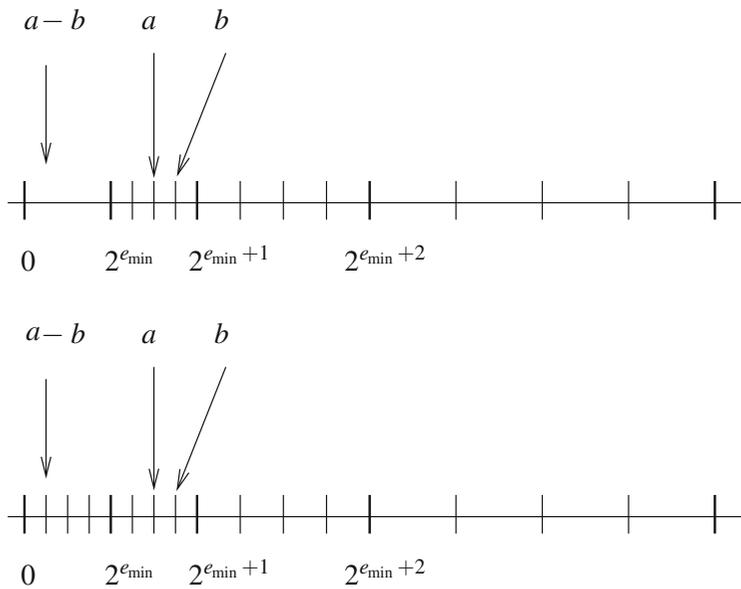
Figure 2.2 *Above is the set of the nonnegative, normal floating-point numbers (assuming radix 2 and 2-bit significands). In that set, $a - b$ is not exactly representable, and the floating-point computation of $a - b$ will return 0 with the round to nearest, round to 0, or round to $-\infty$ rounding functions. Below is the same set with subnormal numbers. Now, $a - b$ is exactly representable, and the properties $a \neq b$ and $a \ominus b \neq 0$ (where $a \ominus b$ denotes the computed value of $a - b$) become equivalent.*

This result is extremely useful when computing accurate error bounds for some elementary function algorithms.

The IEEE-754 standard also defines special representations for exceptions:

- NaN (Not a Number) is the result of an *invalid* arithmetic operation such as $0/0$, $\sqrt{-5}$, ∞/∞, $+\infty + (-\infty)$, …;
- $\pm\infty$ can be the result of an overflow, or the exact result of the division of a nonzero number by zero; and
- ± 0: there are two signed zeroes that can be the result of an underflow, or the exact result of a division by $\pm\infty$.

The reader is referred to [265, 356] for an in-depth discussion on these topics. Subnormal numbers and exceptions must not be neglected by the designer of an elementary function circuit and/or library. They may of course be used as input values, and the circuit/library must be able to produce them as output values when needed.

2.1.3   ULPs

If x is *exactly representable* in a floating-point format and is not an integer power of the radix β, the term ulp (x) (for *unit in the last place*) denotes the magnitude of the last significand digit of x. That is, if,

$$x = \pm x_0.x_1 x_2 \cdots x_{p-1} \times \beta^{e_x}$$

then $\mathrm{ulp}\,(x) = \beta^{e_x - p + 1}$. Defining $\mathrm{ulp}\,(x)$ for all reals x (and not only for the floating-point numbers) is desirable, since the error bounds for functions frequently need to be expressed in terms of ulps. There are several slightly different definitions in the literature [206, 217, 247, 267, 331, 373]. They differ when x is very near a power of β, and they sometimes have counterintuitive properties.

In this book, we will use the following definition.

Definition 1 (*ulp of a real number in radix-β, precision-p arithmetic of minimum exponent $e_{\min}$*) If x is a nonzero number, $|x| \in [\beta^e, \beta^{e+1})$, then $\mathrm{ulp}\,(x) = \beta^{\max(e, e_{\min}) - p + 1}$. Furthermore, $\mathrm{ulp}\,(0) = \beta^{e_{\min} - p + 1}$.

The major advantage of this definition (at least, in radix-2 arithmetic) is that in all cases (even the most tricky), rounding to nearest corresponds to an error of at most $1/2$ ulp of the real value. More precisely, we have

Property 1 *In radix 2, if X is a floating-point number, then*

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

(beware: that property does not always hold in radix-10 arithmetic)

Property 2 *For any radix,*

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

We also have,

Property 3 *For any radix,*

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

(the difference with the previous property is that we have used the ulp of the computed value). Notice that $\mathrm{ulp}\,(t)$ is a monotonic function of $|t|$: if $|t_1| \leq |t_2|$ then $\mathrm{ulp}\,(t_1) \leq \mathrm{ulp}\,(t_2)$. This has an interesting and useful consequence: if we know that the result of a correctly rounded (with round-to-nearest rounding function) arithmetic operation belongs to some interval $[a, b]$, then the rounding error due to that operation is bounded by

$$\frac{1}{2} \cdot \max\{\,\mathrm{ulp}\,(a),\ \mathrm{ulp}\,(b)\} = \frac{1}{2} \cdot \mathrm{ulp}\,(\max\{|a|, |b|\})\,.$$

2.1.4   Infinitely Precise Significand

Implicitly assuming radix 2, we will extend the notion of significand to all real numbers as follows. Let x be a real number. If $x = 0$, then the infinitely precise significand of x equals 0, otherwise, it equals

$$\frac{x}{2^{\lfloor \log_2 |x| \rfloor}}\,.$$

The infinitely precise significand of a nonzero real number has an absolute value between 1 and 2. If x is a normal floating-point number, then its infinitely precise significand is equal to its significand.

2.1.5   Fused Multiply–Add Operations

The FMA instruction evaluates expressions of the form $ab + c$ with one rounding error instead of two (that is, if $\circ$ is the rounding function, $\mathrm{FMA}(a, b, c) = \circ(ab+c)$). That instruction was first implemented on the IBM RS/6000 processor [235, 348]. It was then implemented on several processors such as the IBM PowerPC [256], the HP/Intel Itanium [115], the Fujitsu SPARC64 VI, and the STI Cell, and is available on current processors such as the Intel Haswell and the AMD Bulldozer. More importantly, the FMA instruction is included in the 2008 release of the IEEE-754 standard for floating-point arithmetic [245], so that within a few years, it will probably be available on most general-purpose processors.

Such an instruction may be extremely helpful for the designer of arithmetic algorithms:

- it facilitates the exact computation of division remainders, which allows the design of efficient software for correctly rounded division [68, 113, 115, 265, 331, 333];
- it makes the evaluation of polynomials faster and—in general—more accurate: when using Horner's scheme,[8] the number of necessary operations (hence, the number of roundings) is halved. This is extremely important for elementary function evaluation, since polynomial approximations to these functions are frequently used (see Chapter 3). Markstein, and Cornea, Harrison, and Tang devoted very interesting books to the evaluation of elementary functions using the fused multiply–add operations that are available on the HP/Intel Itanium processor [115, 331];
- as noticed by Karp and Markstein [269], it makes it possible to easily get the exact product of two floating-point variables. More precisely, once we have computed the floating-point product π of two variables a and b (which is $\mathrm{RN}(ab)$ if we assume that the rounding function is round to nearest), one FMA operation suffices to compute the error of that floating-point multiplication, namely $ab - \pi$ (see Section 2.2.1).

And yet, as noticed by Kahan [265] a clumsy use (by an unexperienced programmer or a compiler) of a fused multiply–add operation may lead to problems. Depending on how it is implemented, function

$$f(x, y) = \sqrt{x^2 - y^2}$$

may sometimes return an NaN when $x = y$. Consider the following as an example:

$$x = y = 1 + 2^{-52}.$$

In binary64/double-precision arithmetic this number is exactly representable. The binary64 number that is closest to x^2 is

$$S = \frac{2251799813685249}{2251799813685248} = \frac{2^{51} + 1}{2^{51}},$$

and the binary64 number that is closest to $S - y^2$ is

$$-\frac{1}{2028240960365167042394725128601 6} = -2^{-104}.$$

---

[8]Horner's scheme consists in evaluating a degree-n polynomial $a_n x^n + a_{n-1} x^{n-1} + \cdots + a_0$ as $(\cdots (((a_n x + a_{n-1}) x + a_{n-2}) x + a_{n-3}) \cdots) x + a_0$. This requires n multiplications and n additions if we use conventional operations, or n fused multiply–add operations. See Chapter 5 for more information.

Hence, if the floating-point computation of $x^2 - y^2$ is implemented as $RN(RN(x^2) - y \times y)$, then the obtained result will be less than 0 and computing its square root will generate a NaN, whereas the exact result is 0. The problem does not occur if we do not use the FMA operation: the rounding functions are monotonic, so that if $|x| \geq |y|$ then the computed value of x^2 will be larger than or equal to the computed value of y^2.

2.1.6   The Formats Specified by the IEEE-754-2008 Standard for Floating-Point Arithmetic

Table 2.2 gives the widths of the various fields (significand, exponent) and the main parameters of the binary interchange formats specified by IEEE 754, and Table 2.3 gives the main parameters of the decimal formats. Let us describe the internal encoding of numbers represented in the *binary* formats of the Standard (for the internal encodings of decimal numbers, see [356]). 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: what is actually stored is the "trailing significand," i.e., the significand without its leftmost bit). This ordering allows one to compare floating-point numbers as if they were sign-magnitude integers.

Table 2.2 *Widths of the various fields and main parameters of the binary interchange formats of size up to* 128 *bits specified by the 754-2008 standard.* [245]

IEEE 754-2008 name	binary16	binary32	binary64	binary128
Former name	N/A	Single precision	Double precision	Quad precision
Storage width	16	32	64	128
Trailing significand width	10	23	52	112
W_E, exponent field width	5	8	11	15
b, bias	15	127	1023	16383
Precision p	11	24	53	113
e_{max}	$+15$	$+127$	$+1023$	$+16383$
e_{min}	-14	-126	-1022	-16382
Largest finite number	65504	$2^{128} - 2^{104}$ $\approx 3.403 \times 10^{38}$	$2^{1024} - 2^{971}$ $\approx 1.798 \times 10^{308}$	$2^{16384} - 2^{16271}$ $\approx 1.190 \times 10^{4932}$
Smallest positive normal number	$2^{-14} \approx 6.104 \times 10^{-5}$	2^{-126} $\approx 1.175 \times 10^{-38}$	2^{-1022} $\approx 2.225 \times 10^{-308}$	2^{-16382} $\approx 3.362 \times 10^{-4932}$
Smallest positive number	$2^{-24} \approx 5.960 \times 10^{-8}$	2^{-149} $\approx 1.401 \times 10^{-45}$	2^{-1074} $\approx 4.941 \times 10^{-324}$	2^{-16494} $\approx 6.475 \times 10^{-4966}$

Table 2.3 *Main parameters of the decimal interchange formats of size up to* 128 *bits specified by the 754-2008 standard* [245].

IEEE-754-2008 name	decimal32	decimal64 (basic)	decimal128 (basic)
p	7	16	34
e_{max}	$+96$	$+384$	$+6144$
e_{min}	-95	-383	-6143

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), 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 2.2. N_e is called the *biased exponent*. 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 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:

    - 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 [6, 245, 356] for a definition).

This encoding of binary floating-point numbers has a nice property: one obtains the successor of a floating-point number by considering its binary representation as the binary representation of an integer, and adding one to that integer.

2.1.7   Testing Your Computational Environment

The various parameters (radix, significand and exponent lengths, rounding functions…) of the floating-point arithmetic of a computing system may strongly influence the result of a numerical program. An amusing example of this is the following program, given by Malcolm [204, 330], that returns the radix of the floating-point system being used (beware: an aggressively "optimizing" compiler might decide to replace `((A+1.0)-A)-1.0` by `0`).

```
A := 1.0;
B := 1.0;
while ((A+1.0)-A)-1.0 = 0.0 do A := 2*A;
while  ((A+B)-A)-B <> 0.0 do B := B+1.0;
return(B)
```

Similar—yet much more sophisticated—algorithms are used in rather old inquiry programs such as MACHAR [98] and PARANOIA [270], that provide a means for examining your computational environment. Other programs for checking conformity of your computational system to the IEEE Standard for Floating Point Arithmetic are Hough's UCBTEST (available at http://www.netlib.org/fp/ucbtest.tgz), and a more recent tool presented by Verdonk, Cuyt and Verschaeren [462, 463].

2.2   Advanced Manipulation of FP Numbers

2.2.1   Error-Free Transforms: Computing the Error of a FP Addition or Multiplication

Let a and b be two precision-p floating-point numbers, and define $s = \text{RN}(a+b)$, i.e., $a+b$ correctly rounded to the nearest precision-p floating-point number. It can be shown that if the addition of a and b does not overflow, then the error of that floating-point addition, namely $(a+b) - s$, is a precision-p floating-point number.[9] Interestingly enough, that error can be computed by very simple algorithms, as shown below.

Theorem 2   (Fast2Sum algorithm) *([148], and Theorem C of [275], p. 236). Assume the radix β of the floating-point system being considered is less than or equal to* 3*, and that the used arithmetic provides correct rounding with rounding to the 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. Algorithm 1 below computes two floating-point numbers s and t that satisfy:*

- *$s + t = a + b$ exactly;*
- *s is the[10] floating-point number that is closest to $a + b$.*

Algorithm 1 The **Fast2Sum** algorithm [148].

$s \leftarrow \text{RN}(a+b)$
$z \leftarrow \text{RN}(s-a)$
$t \leftarrow \text{RN}(b-z)$

Algorithm 1 requires that the exponent of a should be larger than or equal to that of b. That condition is satisfied when $|a| \geq |b|$. When we do not have preliminary information on a and b that allows us to make sure that the condition is satisfied, using Algorithm 1 requires a preliminary comparison of $|a|$ and $|b|$, followed by a possible swap of these variables. In most modern architectures, this comparison and this swap may significantly hinder performance, so that in general, it is preferable to use Algorithm 2 below, which gives a correct result whatever the ordering of $|a|$ and $|b|$ is.

Algorithm 2 The **2Sum** algorithm.

$s \leftarrow \text{RN}(a+b)$
$a' \leftarrow \text{RN}(s-b)$
$b' \leftarrow \text{RN}(s-a')$
$\delta_a \leftarrow \text{RN}(a-a')$
$\delta_b \leftarrow \text{RN}(b-b')$
$t \leftarrow \text{RN}(\delta_a + \delta_b)$

---

[9]Beware: that property is not always true with rounding functions different from RN. The error of a floating-point addition with one of these other rounding functions may not sometimes be exactly representable by a floating-point number of the same format.

[10]As a matter of fact there can be *two* such numbers (if $a + b$ is the exact middle of two consecutive floating-point numbers).

We have [41, 275, 346],

Theorem 3 *If a and b are normal floating-point numbers, then for any radix β, provided that no overflow occurs, the values returned by Algorithm 2 satisfy $a + b = s + t$.*

One can show that Algorithm 2 is optimal in terms of number of operations [280]. Algorithms 1 and 2 make it possible to compute the error of a floating-point addition. Interestingly enough, it is also possible to compute the error of a floating-point multiplication. Unless overflow occurs, Algorithm 3 below returns two values p and ρ such that p is the floating-point number that is closest to ab, and $p + \rho = ab$ exactly, provided that $e_a + e_b \geq e_{\min} + p - 1$ (where $e_{\min}$ is the minimum exponent of the floating-point format being used, e_a and e_b are the exponents of a and b, and p is the precision). It requires one multiplication and one fused multiply–add (FMA). Although I present it with the round-to-nearest function, it works as well with the other rounding functions.

Algorithm 3 The **Fast2MultFMA** algorithm.

$$\pi \leftarrow \text{RN}(ab);$$
$$\rho \leftarrow \text{RN}(ab - \pi)$$

Performing a similar calculation without a fused multiply–add operation is possible [148] but requires 17 floating-point operations instead of 2. Some other interesting arithmetic functions are easily implementable when a fused multiply–add is available [45, 67, 253].

Algorithms 1, 2, and 3 can be used for building *compensated algorithms*, i.e., algorithms in which the errors of "critical" operations are computed to be later on "re-injected" in the calculation, in order to partly compensate for these errors. For example, several authors have suggested "compensated summation" algorithms (see for instance [261, 367, 386]). Another example is the following (notice that the first two lines are nothing but Algorithm 3):

Algorithm 4 Kahan's way to compute $x = ad - bc$ with fused multiply–adds.

$w \leftarrow \text{RN}(bc)$
$e \leftarrow \text{RN}(w - bc)$ // this operation is exact: $e = \hat{w} - bc$.
$f \leftarrow \text{RN}(ad - w)$
$x \leftarrow \text{RN}(f + e)$
return x

In [253], it is shown that in precision-p binary floating-point arithmetic, the relative error of Algorithm 4 is bounded by 2^{-p+1}, and that the error in ulps is bounded by $3/2$ ulps.

2.2.2 Manipulating Double-Word or Triple-Word Numbers

As we will see in Chapter 3, the elementary functions are very often approximated by polynomials. Hence, an important part of the function evaluation reduces to the evaluation of a polynomial. This requires a sequence of additions and multiplications (or a sequence of FMAs). However, when we want a very accurate result, it may not suffice to represent the coefficients of the approximating polynomial in the "target format."[11] Furthermore, to avoid a too large accumulation of rounding errors, it may

---

[11]Throughout the book, we call "target format" the floating-point format specified for the returned result, and "target precision" its precision.

sometimes be necessary to represent intermediate variables of the polynomial evaluation algorithm with a precision larger than the target precision. All this is easily handled when a wider floating-point format is available in hardware. When this is not the case, one can represent some high-precision variables as the unevaluated sum of two or three floating-point numbers. Such unevaluated sums are called "double-word" or "triple-word" numbers. Since the floating-point format used for implementing such numbers is almost always the binary64 format, previously called "double precision," these numbers are often called "double double" or "triple double" numbers in the literature.

Algorithms 1, 2, and 3 are the basic building blocks that allow one to manipulate double-word or triple-word numbers. For instance, Dekker's algorithm for adding two double-word numbers (x_h, x_ℓ) and (y_h, y_ℓ) is shown in Algorithm 5 below.

Algorithm 5 Dekker's algorithm for adding two double-word numbers (x_h, x_ℓ) and (y_h, y_ℓ) [148].

---

if $|x_h| \geq |y_h|$ **then**
  $(r_h, r_\ell) \leftarrow \text{Fast2Sum}(x_h, y_h)$
  $s \leftarrow \text{RN}(\text{RN}(r_\ell + y_\ell) + x_\ell)$
else
  $(r_h, r_\ell) \leftarrow \text{Fast2Sum}(y_h, x_h)$
  $s \leftarrow \text{RN}(\text{RN}(r_\ell + x_\ell) + y_\ell)$
end if
$(t_h, t_\ell) \leftarrow \text{Fast2Sum}(r_h, s)$
return (t_h, t_ℓ)

The most accurate algorithm for double-word addition in Bailey's QD library, as presented in [317], is Algorithm 6 below.

Algorithm 6 The most accurate algorithm implemented in Bailey's QD library for adding two double-word numbers $x = (x_h, x_\ell)$ and $y = (y_h, y_\ell)$ [317].

$(s_h, s_\ell) \leftarrow \text{2Sum}(x_h, y_h)$
$(t_h, t_\ell) \leftarrow \text{2Sum}(x_\ell, y_\ell)$
$c \leftarrow \text{RN}(s_\ell + t_h)$
$(v_h, v_\ell) \leftarrow \text{Fast2Sum}(s_h, c)$
$w \leftarrow \text{RN}(t_\ell + v_\ell)$
$(z_h, z_\ell) \leftarrow \text{Fast2Sum}(v_h, w)$
return (z_h, z_ℓ)

Assuming $|x_\ell| \leq 2^{-p} \cdot |x|$, where p is the precision of the binary floating-point arithmetic being used, one can show that the relative error of Algorithm 6 is bounded by

$$2^{-2p} \cdot \left(3 + 13 \cdot 2^{-p}\right),$$

(the bound is valid provided that $p \geq 3$, which always holds in practice).

Bailey's algorithm for multiplying a double-word number by a floating-point number is given below (here, we assume that an FMA is available, so that we can use Algorithm 3 to represent the product of two floating-point numbers by a double-word number).

Algorithm 7 The algorithm implemented in Bailey's QD library for multiplying a double-word number $x = (x_h, x_\ell)$ by a floating-point number y [317]. Here, we assume that an FMA instruction is available, so that we can use the Fast2MultFMA algorithm (Algorithm 3).

$(c_h, c_{\ell 1}) \leftarrow \text{Fast2MultFMA}(x_h, y)$
$c_{\ell 2} \leftarrow \text{RN}(x_\ell \cdot y)$
$(t_h, t_{\ell 1}) \leftarrow \text{Fast2Sum}(c_h, c_{\ell 2})$
$t_{\ell 2} \leftarrow \text{RN}(t_{\ell 1} + c_{\ell 1})$
$(z_h, z_\ell) \leftarrow \text{Fast2Sum}(t_h, t_{\ell 2})$
return (z_h, z_ℓ)

Assuming $|x_\ell| \leq 2^{-p} \cdot |x|$, one can show that the relative error of Algorithm 7 is bounded by

$$2 \cdot 2^{-2p}$$

Lauter [300] gives basic building blocks for a triple-word arithmetic. These blocks have been used for implementing critical parts in the CRLIBM library for correctly rounded elementary functions in binary64/double-precision arithmetic (see Section 14.4). For instance, here is one of Lauter's algorithms for adding two triple-word numbers and obtaining the result as a triple-word number.

Algorithm 8 An algorithm suggested by Lauter [300] for adding two triple-word numbers $a = (a_h, a_m, a_\ell)$ and $b = (b_h, b_m, b_\ell)$—a and b must satisfy the conditions of Theorem 4 below.

$(r_h, t_1) \leftarrow \text{Fast2Sum}(a_h, b_h)$
$(t_2, t_3) \leftarrow \text{2Sum}(a_m, b_m)$
$(t_7, t_4) \leftarrow \text{2Sum}(t_1, t_2)$
$t_6 \leftarrow \text{RN}(a_\ell + b_\ell)$
$t_5 \leftarrow \text{RN}(t_3 + t_4)$
$t_8 \leftarrow \text{RN}(t_5 + t_6)$
$(r_m, r_\ell) \leftarrow \text{2Sum}(t_7, t_8)$
return (r_h, r_m, r_ℓ)

Lauter showed the following result.

Theorem 4 *If Algorithm 8 is run in binary64 arithmetic with two input triple-word numbers $a = (a_h, a_m, a_\ell)$ and $(b = (b_h, b_m, b_\ell)$ satisfying:*

$$\begin{cases} |b_h| \leq (3/4) \cdot |a_h| \\ |a_m| \leq 2^{-\alpha_0} \cdot |a_h|, \text{ with } \alpha_0 \geq 4 \\ |a_\ell| \leq 2^{-\alpha_u} \cdot |a_m|, \text{ with } \alpha_u \geq 1 \\ |b_m| \leq 2^{-\beta_0} \cdot |b_h|, \text{ with } \beta_0 \geq 4 \\ |b_\ell| \leq 2^{-\beta_u} \cdot |b_m|, \text{ with } \beta_u \geq 1 \end{cases}$$

then the returned result (r_h, r_m, r_ℓ) satisfies

$$\begin{cases} r_h + r_m + r_\ell = ((a_h + a_m + a_\ell) + (b_h + b_m + b_\ell)) \cdot (1 + \epsilon), \\ \quad \text{with } |\epsilon| \leq 2^{-\min(\alpha_0 + \alpha_u, \beta_0 + \beta_u) - 47} + 2^{-\min(\alpha_0, \beta_0) - 98}, \\ |r_m| \leq 2^{-\min(\alpha_0, \beta_0) + 5} \cdot |r_h|, \\ |r_\ell| \leq 2^{-53} \cdot |r_m|. \end{cases}$$

The following is one of Lauter's algorithms for multiplying two double-word numbers and getting the result as a triple-word number.

Algorithm 9 An algorithm suggested by Lauter [300] for multiplying two double-word numbers $a = (a_h, a_\ell)$ and $b = (b_h, b_\ell)$, assuming a binary64 underlying arithmetic, and obtaining the product as a triple-word number—a and b must satisfy the conditions of Theorem 5 below. ADD22 is Algorithm 5.

$(r_h, t_1) \leftarrow \text{Fast2MultFMA}(a_h, b_h)$
$(t_2, t_3) \leftarrow \text{Fast2MultFMA}(a_h, b_\ell)$
$(t_4, t_5) \leftarrow \text{Fast2MultFMA}(a_\ell, b_h)$
$t_6 \leftarrow \text{RN}(a_\ell b_\ell)$
$(t_7, t_8) \leftarrow \text{ADD22}((t_2, t_3), (t_4, t_5))$
$(t_9, t_{10}) \leftarrow \text{2Sum}(t_1, t_6)$
$(r_m, r_\ell) \leftarrow \text{ADD22}((t_7, t_8), (t_9, t_{10}))$
return (r_h, r_m, r_ℓ)

Lauter showed the following result.

Theorem 5 *If Algorithm 9 is run in binary64 arithmetic with two input double-word numbers $a = (a_h, a_\ell)$ and $b = (b_h, b_\ell)$ satisfying $|a_\ell| \leq 2^{-53}|a_h|$ and $|b_\ell| \leq 2^{-53}|b_h|$, then the returned result (r_h, r_m, r_ℓ) satisfies*

$$\begin{cases} r_h + r_m + r_\ell = ((a_h + a_\ell) \cdot (b_h + b_\ell)) \cdot (1 + \epsilon), \\ \quad \text{with } |\epsilon| \leq 2^{-149}, \\ |r_m| \leq 2^{-48} \cdot |r_h|, \\ |r_\ell| \leq 2^{-53} \cdot |r_m|. \end{cases}$$

2.2.3   An Example that Illustrates What We Have Learnt so Far

The following polynomial, generated by the Sollya tool (see Section 4.2) approximates function $\cos(x)$, for $x \in [-0.0123, +0.0123]$ (that domain is a tight enclosure of $[-\pi/256, +\pi/256]$), with an error less than 1.9×10^{-16}:

$$P(x) = 1 + a_2 x^2 + a_4 x^4,$$

where a_2 and a_4 are the following binary64/double-precision numbers:

$$\begin{cases} a_2 = -2251799813622611 \times 2^{-52} \approx -0.499999999986091 \\ a_4 = 1501189276987675 \times 2^{-55} \approx 0.04166637249080271 \end{cases}$$

Here we wish to have a tight upper bound on the error committed if we evaluate P as follows (Algorithm 10) in binary64/double-precision arithmetic. We wish to obtain the result as a double-word number (s_h, s_ℓ).

Algorithm 10 This algorithm returns an approximation to $P(x)$ as a double-word number (s_h, s_ℓ). We assume that an FMA instruction is available to compute $\text{RN}(a_2 + a_4 y)$ in line 2.

$y \leftarrow \text{RN}(x^2)$
$s_1 \leftarrow \text{RN}(a_2 + a_4 y)$
$s_2 \leftarrow \text{RN}(s_1 y)$
$(s_h, s_\ell) \leftarrow \text{Fast2Sum}(1, s_2)$
return (s_h, s_ℓ)

Since roundings are monotonic functions, we have

$$0 \leq y \leq \text{RN}(0.0123^2).$$

Let us call b_y that bound, i.e.,

$$b_y = \frac{1395403955455759}{9223372036854775808}.$$

We have $\text{ulp}(y) \leq \text{ulp}(b_y) = 2^{-65}$, therefore, since the computation of y is the result of a correctly rounded floating-point multiplication:

$$|y - x^2| \leq \frac{1}{2}\,\text{ulp}(b_y) = 2^{-66}. \tag{2.3}$$

So we have bounded the error committed at the first line of Algorithm 10. Let us now deal with the second line. We have

$$a_2 + a_4 y \in [a_2, a_2 + a_4 b_y]$$

therefore

$$
\begin{aligned}
s_1 &= \text{RN}(a_2 + a_4 y) \\
&\in [\text{RN}(a_2), \text{RN}(a_2 + a_4 b_y)] = \left[a_2, -\frac{4503542848513793}{2^{53}}\right].
\end{aligned} \tag{2.4}
$$

Thus $\text{ulp}(s_1) \leq \text{ulp}(\max\{|a_2|, |a_2 + a_4 b_y|\}) = \text{ulp}(|a_2|) = 2^{-54}$, from which we deduce

$$|s_1 - (a_2 + a_4 y)| \leq 2^{-55}.$$

This gives

$$
\begin{aligned}
|s_1 - (a_2 + a_4 x^2)| &\leq |s_1 - (a_2 + a_4 y)| + |(a_2 + a_4 y) - (a_2 + a_4 x^2)| \\
&\leq 2^{-55} + a_4 \cdot |y - x^2|,
\end{aligned}
$$

from which we deduce, using (2.3),

$$|s_1 - (a_2 + a_4 x^2)| \leq 2^{-55} + a_4 \cdot 2^{-66}. \tag{2.5}$$

We are now ready to tackle the third line of the algorithm. We have,

$$s_1 y \in \left[a_2 y, -\frac{4503542848513793}{2^{53}} \cdot y\right],$$

which implies,

$$s_1 y \in \left[a_2 \cdot b_y, 0\right],$$

therefore,

$$s_2 = \text{RN}(s_1 \cdot y) \in \left[\text{RN}\left(a_2 \cdot b_y\right), 0\right],$$

therefore,

$$s_2 \in \left[-\frac{5581615821667775}{2^{66}}, 0 \right]. \tag{2.6}$$

from this we deduce that $\text{ulp}\,(s_2) \leq 2^{-66}$, which implies

$$|s_2 - s_1 y| \leq 2^{-67}. \tag{2.7}$$

We now have

$$\begin{aligned}
|s_2 - (a_2 x^2 + a_4 x^4)| &\leq |s_2 - s_1 y| + |s_1 y - s_1 x^2| + |s_1 x^2 - (a_2 x^2 + a_4 x^4)| \\
&\leq 2^{-67} + |s_1| \cdot |y - x^2| + x^2 \cdot |s_1 - (a_2 + a_4 y)| \\
&\leq 2^{-67} + |a_2| \cdot 2^{-66} + 0.0123^2 \cdot (2^{-55} + a_4 \cdot 2^{-66})
\end{aligned}$$

using (2.3), (2.4), and (2.5).

Finally, (2.6) implies $|s_2| < 1$, therefore Algorithm Fast2Sum is legitimately used at line 4 of the algorithm, so that $s_h + s_\ell = 1 + s_2$. We can therefore deduce a bound on the error committed when evaluating polynomial P using Algorithm 10:

$$\begin{aligned}
|(s_h + s_\ell) - P(x)| &\leq 2^{-67} + |a_2| \cdot 2^{-66} + 0.0123^2 \cdot (2^{-55} + a_4 \cdot 2^{-66}) \\
&\leq 1.77518 \times 10^{-20}.
\end{aligned} \tag{2.8}$$

The obtained bound (2.8) is rather tight: for instance, if we apply Algorithm 10 to the input value $x = 1772616811707781/2^{57}$, the evaluation error is 1.76697×10^{-20}.

The "toy" example we have considered here can be generalized to the evaluation of polynomials of larger degree, possibly using different evaluation schemes (see Chapter 5): the underlying idea is to compute interval enclosures of all intermediate variables, which allows to compute bounds on the rounding errors of the arithmetic operations. It can be adapted to compute relative error bounds instead of absolute ones. However, the idea of computing evaluation errors as we just have done here has some limitations:

- the process was already tedious and error prone with our toy example. In practical cases (degrees of polynomials that can be as large as a few tens, with some coefficients that can be double-word numbers), it may become almost impractical. Furthermore, to avoid inherent overestimations of enclosures that occur in interval arithmetic, one may need to split the input domain into many subintervals and redo the calculation for each of them;
- as we will see in Chapter 5, when some parallelism is available on the target processor (pipelined operators, several FPUs)—which is always the case with recent processors—many different evaluation schemes are possible (when the degree of the polynomial is large, the number of possible schemes is huge). In general, choosing which evaluation scheme will be implemented results from a compromise between the latency or throughput and accuracy. To find a good compromise, one may wish to get, in reasonable time, a tight bound on the evaluation error for several tens of evaluation schemes;
- if the function we are implementing is to be used in critical applications, one needs confidence in the obtained error bounds, which is not so obvious when they are derived from long and tedious calculations. One may even wish *certified* error bounds.

These remarks call for an automation of the calculation of error bounds for small "straight-line" numerical programs (such as those used for evaluating elementary functions), and for the possibility of using proof checkers for certifying these bounds. These needs are fulfilled by the Gappa tool, presented in the next section.

2.2.4   The GAPPA Tool

Thanks to the IEEE-754 standard, we now have an accurate definition of floating-point formats and operations. This allows the use of formal proofs to verify pieces of mathematical software. For instance, Harrison used HOL Light to formalize floating-point arithmetic [217] and check floating-point trigonometric functions [218] for the Intel-HP IA64 architecture. Russinoff [406] 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 arithmetic algorithms [42, 315].

The Gappa tool [128, 339] can be downloaded at http://gappa.gforge.inria.fr. It was designed by Melquiond to help to prove properties of small (up to a few hundreds of operations) yet complicated floating-point programs. Typical useful properties Gappa helps to prove are the fact that some value stays within a given range (which is important in many cases, for instance if we wish to guarantee that there will be no overflow), or that it is computed with a well-bounded relative error. The paper [134] explains how Gappa has been used to certify functions of the CRLIBM library of correctly rounded elementary functions. Gappa uses interval arithmetic, a database of rewriting rules, and hints given by the user to prove a property, and generates a formal proof that can be mechanically checked by an external proof checker. This was considered important by the authors of CRLIBM: as explained by de Dinechin et al. [134], in the first versions of the library, the complete paper and pencil proof of a single function required tens of pages, which inevitably cast some doubts on the trustability of the proof.

The following Gappa file automatically computes a bound on the error committed when evaluating the polynomial P of the previous section using Algorithm 10, with an input value in $[-0.0123, +0.0123]$. It is made up of three parts: the first one describes the numerical algorithm, the second one describes the exact value we are approximating, and the third one describes the theorem we wish to prove. For more complex algorithms, we may need a fourth part that describes hints given to Gappa.

```
@RN = float<ieee_64,ne>;
# defines RN as round-to-nearest in binary64 arithmetic

x = RN(xx);
a2 = -2251799813622611b-52;
a4 = 1501189276987675b-55;

# description of the program

y RN = x * x;
# now, we describe the action of the FMA
s1beforernd = a2 + a4*y;
s1 = RN(s1beforernd);
s2 RN = s1*y;
s = 1 + s2; # no rounding: Fast2Sum is an exact transformation

# description of the exact value we are approximating
# convention: an "M" as a prefix of the names of "exact" variables
```

```
My = x * x;
Ms1 = a2 + a4 * My;
Ms2 = Ms1 * My;
Ms = 1 + Ms2;
epsilon = (Ms-s);

# description of what we want to prove

{
# input hypothesis
|x| <= 1.23e-02

->
# goal to prove
|epsilon| in ?
/\ |s2| <= 1

# first line of goal: bound we wish to obtain
# second line: necessary to allow one to use Fast2Sum
}
```

When running this file with Gappa, we obtain

```
Results:
   |epsilon| in [0, 94384511554069319b-122 {1.77518e-20, 2^(-65.6106)}]
```

Concerning the first goal `|epsilon| in ?`, Gappa found the same error bound as the one we have computed in the previous section (which is not surprising: it probably uses the same method). There is no answer to our second goal `|s2| <= 1` which, in Gappa's syntax, just means that the answer was `true`.

For more complex programs, the way we have written the previous Gappa file is dangerous. We wanted to have an estimate of the error committed when evaluating $1 + a_2 x^2 + a_4 x^4$ using Algorithm 10. Imagine we have committed an error in the description of the program (hence quite possibly in the program itself), and that we have written

```
s1beforernd = a2 + a4*x;
```

instead of

```
s1beforernd = a2 + a4*y;
```

since the part of the Gappa file that describes the exact value was just obtained by directly rewriting, without roundings, the description of the program, we would very likely have also written

```
Ms1 = a2 + a4 * x;
```

instead of

```
Ms1 = a2 + a4 * My;
```

so that Gappa would have concluded that the computation is very accurate, although we do not at all compute what we wished to compute! The solution to that problem is to have a very simple description of the exact value, as close as possible to the mathematical definition and as independent as possible from the algorithm being used. We could for instance replace the four lines that describe the exact value by

```
Ms = 1 + a2*x*x + a4*x*x*x*x;
```

Unfortunately, if we just do that, we obtain a poor error bound. Gappa returns

```
Results:
  |epsilon| in [0, 174426593954067b-60 {0.000151291, 2^(-12.6904)}]
```

The solution is to give a hint to Gappa, i.e., to explain how the mathematical definition and the algorithm are related. This is done very simply, just by adding the line

```
Ms -> 1 + (a2 + a4*(x*x))*(x*x);
```

Gappa will try to check that the expressions $1 + a2*x*x + a4*x*x*x*x$ and $1 + (a2 + a4*(x*x))*(x*x)$ are equivalent, warn us if it does not succeed, and use the hint to compute a much better error bound, very close to the first one:

```
Results:
  |epsilon| in [0, 94391651810570331b-122 {1.77531e-20, 2^(-65.6105)}]
```

Hence, our final Gappa file is as follows.

```
@RN = float<ieee_64,ne>;
# defines RN as round to the nearest in binary64 arithmetic

x = RN(xx);
a2 = -2251799813622611b-52;
a4 = 1501189276987675b-55;

# description of the program

y RN = x * x;
# now, we describe the action of the FMA
s1beforernd = a2 + a4*y;
s1 = RN(s1beforernd);
s2 RN = s1*y;
s = 1 + s2; # no rounding: Fast2Sum is an exact transformation

# description of the exact value we are approximating
# convention: an "M" as a prefix of the names of "exact" variables

Ms = 1 + a2*x*x + a4*x*x*x*x;
epsilon = (Ms-s);

# description of what we want to prove

{
# input hypothesis
|x| <= 1.23e-02

->
# goal to prove
|epsilon| in ?
/\ |s2| <= 1

# first line of goal: bound we wish to obtain
# second line: necessary to allow use of Fast2Sum

}

# Now some hints to help Gappa

Ms -> 1 + (a2 + a4*(x*x))*(x*x);
```

As we can see, compared to our approach of the previous section, Gappa frees us from long and error-prone calculations. Furthermore, once the initial Gappa input file is written, small modifications

allow one to easily explore variants of the evaluation scheme. The most important issue, however, is that if called with option −Bcoq, Gappa generates a formal proof of the returned result. That proof can then be verified by the Coq proof checker.[12]

2.2.5   Maple Programs that Compute binary32 and binary64 Approximations

The following Maple programs implement the round-to-nearest-even rounding functions in binary32/single-precision and binary64/double-precision. They compute the binary32 and binary64 floating-point numbers that are closest to t for any real number t (and they use the "round-to-nearest ties to even" tie-breaking rule).

RN function, binary32 arithmetic

```
nearest_binary32 := proc(xx)
local x, sign, logabsx, exponent, mantissa, infmantissa, powermin,
expmin, powermax, expmax, powermiddle, expmiddle;
Digits := 100;
x := evalf(xx);
if (x=0) then sign, exponent, mantissa := 1, -126, 0
else
  if (x < 0) then sign := -1
  else sign := 1
  fi:
  x := abs(x);
  if x >=  2^(127)*(2-2^(-24)) then mantissa := infinity; exponent := 127
  else if x <= 2^(-150) then mantissa := 0; exponent := -126
     else
        if x <= 2^(-126) then exponent := -126
        else
# x is between 2^(-126) and 2^(128)
        powermin := 2^(-126); expmin := -126;
        powermax := 2^128; expmax := 128;
        while (expmax-expmin > 1) do
            expmiddle := round((expmax+expmin)/2);
            powermiddle := 2^expmiddle;
            if x >= powermiddle then
                powermin := powermiddle;
                expmin := expmiddle
            else
                powermax := powermiddle;
                expmax := expmiddle
            fi
          od;
# now, expmax - expmin = 1
# and powermin <= x < powermax
# powermin = 2^expmin
# and powermax = 2^expmax
# so expmin is the exponent of x
        exponent := expmin;
        fi;
        infmantissa := x*2^(23-exponent);
        if frac(infmantissa) <> 0.5 then mantissa := round(infmantissa)
          else
              mantissa := floor(infmantissa);
              if type(mantissa,odd) then mantissa := mantissa+1 fi
```

---

[12]Coq can be downloaded at https://coq.inria.fr.

```
            fi;
         mantissa := mantissa*2^(-23);
      fi;
   fi;
fi;
sign*mantissa*2^exponent;
end:
```

RN function, binary64 arithmetic

```
nearest_binary64 := proc(xx)
local x, sign, logabsx, exponent, mantissa, infmantissa,
powermin, expmin, powermax,expmax, powermiddle, expmiddle;
Digits := 100;
x := evalf(xx);
if (x=0) then sign, exponent, mantissa := 1, -1022, 0
else
  if (x < 0) then sign := -1
  else sign := 1
  fi:
  x := abs(x);
  if x >=  2^(1023)*(2-2^(-53)) then mantissa := infinity; exponent := 1023
  else if x <= 2^(-1075) then mantissa := 0; exponent := -1022
     else
        if x <= 2^(-1022) then exponent := -1022
        else
# x is between 2^(-1022) and 2^(1024)
        powermin := 2^(-1022); expmin := -1022;
        powermax := 2^1024; expmax := 1024;
        while (expmax-expmin > 1) do
           expmiddle := round((expmax+expmin)/2);
           powermiddle := 2^expmiddle;
           if x >= powermiddle then
               powermin := powermiddle;
               expmin := expmiddle
           else
               powermax := powermiddle;
               expmax := expmiddle
           fi
        od;
# now, expmax - expmin = 1
# and powermin <= x < powermax
# powermin = 2^expmin
# and powermax = 2^expmax
# so expmin is the exponent of x
        exponent := expmin;
        fi;
        infmantissa := x*2^(52-exponent);
        if frac(infmantissa) <> 0.5 then mantissa := round(infmantissa)
           else
              mantissa := floor(infmantissa);
              if type(mantissa,odd) then mantissa := mantissa+1 fi
           fi;
        mantissa := mantissa*2^(-52);
     fi;
  fi;
fi;
sign*mantissa*2^exponent;
end:
```

The following programs evaluates ulp (t) for any real number t, in binary32 and binary64 floating-point arithmetic.

ULP function, binary32 arithmetic

```
ulp_in_binary_32 := proc(t)
 local x, res, expmin, expmax, expmiddle;
 x := abs(t);
 if x < 2^(-125) then res := 2^(-149)
    else if x > (1-2^(-24))*2^(128) then res := 2^104
    else
      expmin := -125; expmax := 128;
 # x is between 2^expmin and 2^expmax
      while (expmax-expmin > 1) do
        expmiddle := round((expmax+expmin)/2);
        if x >= 2^expmiddle then
              expmin := expmiddle
        else  expmax := expmiddle
        fi;
       od;
 # now, expmax - expmin = 1
 # and 2^expmin <= x < 2^expmax
        res := 2^(expmin-23)
 fi;
 fi; res;
 end:
```

ULP function, binary64 arithmetic

```
ulp_in_binary_64 := proc(t)
 local x, res, expmin, expmax, expmiddle;
 x := abs(t);
 if x < 2^(-1021) then res := 2^(-1074)
    else if x > (1-2^(-53))*2^(1024) then res := 2^971
    else
      expmin := -1021; expmax := 1024;
 # x is between 2^expmin and 2^expmax
      while (expmax-expmin > 1) do
        expmiddle := round((expmax+expmin)/2);
        if x >= 2^expmiddle then
              expmin := expmiddle
        else  expmax := expmiddle
        fi;
       od;
 # now, expmax - expmin = 1
 # and 2^expmin <= x < 2^expmax
        res := 2^(expmin-52)
 fi;
 fi:
 res;
 end:
```

2.2.6   The Future of Floating-Point Arithmetic

Floating-point arithmetic, as it is known nowadays, results from a compromise between several requirements, in terms of range, accuracy, ease of use, ease of implementation, ease of verification/certification,

speed, memory consumption…. As technology evolves, many parameters involved in that compromise change with time. A simple example is the ratio between the delay of a memory access and the delay of an arithmetic operation. This ratio has considerably increased during the last years. Ultimately, this evolution will almost certainly lead to changes in the way we represent and manipulate real numbers on computers. However, it is difficult to forecast the magnitude of these changes: will we use slightly modified versions of our current floating-point systems, or will we use very different number systems? Over the years, various alternatives to conventional floating-point arithmetic have been suggested: tapered floating-point arithmetic [13, 350], level index arithmetic [91, 369], logarithmic number systems [272, 441], slash number systems [344], etc. Recently, Gustafson [211] suggested an interesting variant of tapered floating-point arithmetic, called the Unum number system, with an "exact" bit added to the representation of the numbers, and a subtle interval interpretation of the nonexact representations.

2.3   Redundant Number Systems

In general, when we represent numbers in radix r, we use the digits 0, 1, 2, … $r - 1$. And yet, sometimes, number systems using a different set of digits naturally arise. In 1840, Cauchy suggested the use of digits -5 to $+5$ in radix 10 to simplify multiplications [73]. Booth recoding [47] (a technique sometimes used by multiplier designers) generates numbers represented in radix 2, with digits -1, 0, and $+1$. Digit-recurrence algorithms for division and square root [179, 399] also generate results in a "signed-digit" representation.

Some of these exotic number systems allow carry-free addition. This is what we are going to investigate in this section.

First, assume that we want to compute the sum $s = s_n s_{n-1} s_{n-2} \ldots s_0$ of two integers $x = x_{n-1} x_{n-2} \ldots x_0$ and $y = y_{n-1} y_{n-2} \ldots y_0$ represented in the conventional binary number system. By examining the well-known equation that describes the addition process ("$\vee$" is the boolean "or" and "$\oplus$" is the "exclusive or"):

$$
\begin{aligned}
c_0 \quad &= 0 \\
s_i \quad &= x_i \oplus y_i \oplus c_i \\
c_{i+1} &= x_i y_i \vee x_i c_i \vee y_i c_i
\end{aligned}
\tag{2.9}
$$

we see that there is a dependency relation between c_i, the *incoming carry* at position i, and c_{i+1}. This does not mean that the addition process is intrinsically sequential, and that the sum of two numbers is computed in a time that grows linearly with the size of the operands: the addition algorithms and architectures proposed in the literature [180, 191, 277, 370, 378] and implemented in current microprocessors are much faster than a straightforward, purely sequential, implementation of (2.9). Nevertheless, the dependency relation between the carries makes a fully parallel addition impossible in the conventional number systems.

2.3.1   Signed-Digit Number Systems

In 1961, Avizienis [12] studied different number systems called *signed-digit* number systems. Let us assume that we use radix r. In a signed-digit number system, the numbers are no longer represented using digits between 0 and $r - 1$, but with digits between $-a$ and a, where $a \leq r - 1$. Every number is representable in such a system, if $2a \geq r - 1$. For instance, in radix 10 with digits between -5 and

Figure 2.3 *Computation of $1\overline{5}31\overline{2}0 + 1\overline{1}\overline{2}61\overline{6}$ using Avizienis' algorithm in radix $r = 10$ with $a = 6$.*

x_i	1	$\overline{5}$	3	1	$\overline{2}$	0
y_i	1	$\overline{1}$	$\overline{2}$	6	1	$\overline{6}$
$x_i + y_i$	2	-6	1	7	-1	-6
t_{i+1}	0	-1	0	1	0	-1
w_i	2	4	1	-3	-1	4
s_i	1	4	2	$\overline{3}$	$\overline{2}$	4

$+5$, every number is representable. The number 15725 can be represented by the digit chain $2\overline{4}3 25$ (we use $\overline{4}$ to represent the digit -4); i.e., $15725 = 2 \times 10^4 + (-4) \times 10^3 + (-3) \times 10^2 + 2 \times 10^1 + 5$.

The same number can also be represented by the digit chain $2\overline{4}33\overline{5}$. If $2a \geq r$, then some numbers have several possible representations, which means that the number system is *redundant*. As shown later, this is an important property.

Avizienis also proposed addition algorithms for these number systems. Algorithm 11 performs the addition of two n-digit numbers $x = x_{n-1}x_{n-2} \cdots x_0$ and $y = y_{n-1}y_{n-2} \cdots y_0$ represented in radix r with digits between $-a$ and a, where $a \leq r - 1$ and[13] $2a \geq r + 1$.

Algorithm 11 Avizienis' algorithm

Input : $x = x_{n-1}x_{n-2} \cdots x_0$ and $y = y_{n-1}y_{n-2} \cdots y_0$
Output : $s = s_n s_{n-1} s_{n-2} \cdots s_0 = x + y$

1. in parallel, for $i = 0, \ldots, n - 1$, compute t_{i+1} (carry) and w_i (intermediate sum) satisfying:

$$
\begin{cases}
t_{i+1} = \begin{cases} +1 \text{ if } x_i + y_i \geq a \\ 0 \text{ if } -a + 1 \leq x_i + y_i \leq a - 1 \\ -1 \text{ if } x_i + y_i \leq -a \end{cases} \\
w_i = x_i + y_i - r \times t_{i+1}.
\end{cases}
\tag{2.10}
$$

2. in parallel, for $i = 0, \ldots, n$, compute $s_i = w_i + t_i$, with $w_n = t_0 = 0$.

By examining the algorithm, we can see that the carry t_{i+1} does not depend on t_i. There is no longer any carry propagation: all digits of the result can be generated simultaneously. The conditions "$2a \geq r + 1$" and "$a \leq r - 1$" cannot be simultaneously satisfied in radix 2. Nevertheless, it is possible to perform parallel, carry-free additions in radix 2 with digits equal to -1, 0, or 1, by using another algorithm, also due to Avizienis (or by using the *borrow-save adder* presented in the following).

Figure 2.3 presents an example of the execution of Avizienis' algorithm in the case $r = 10, a = 6$.

Redundant number systems are used in many instances: recoding of multipliers, quotients in division and division-like operations, online arithmetic [182], etc. Redundant additions are commonly used within arithmetic operators such as multipliers and dividers (the input and output data of such operators are represented in a nonredundant number system, but the internal calculations are performed in a redundant number system). For instance, most multipliers use (at least implicitly) the carry-save number system, whereas digit-recurrence dividers actually use two different number systems: the partial remainders are represented, in general, in carry-save, and the quotient digits are represented in

---

[13]This condition is stronger than the condition $2a \geq r - 1$ that is required to represent every number.

a signed-digit number system of radix 2^k, where k is a small integer [179]. The reader interested in redundant number systems can find useful information in [12, 180, 375, 376, 377, 382].

2.3.2   The Carry-Save and Borrow-Save Number Systems

Now let us focus on the particular case of radix 2. In this radix, the two common redundant number systems are the *carry-save* (CS) number system, and the signed-digit number system. In the carry-save number system, numbers are represented with digits 0, 1, and 2, and each digit d is represented by two bits $d^{(1)}$ and $d^{(2)}$ whose sum equals d. In the signed-digit number system, numbers are represented with digits $-1, 0,$ and 1. In that system, we can represent the digits with the *borrow-save* (BS) encoding, also called (p, n) encoding [375]: each digit d is represented by two bits d^+ and d^- such that $d^+ - d^- = d$ (different encodings of the digits also lead to fast and simple arithmetic operators [88, 443]). Those two number systems allow very fast additions and subtractions. The *carry-save adder* (see, for instance, [277]) is a very well-known structure used for adding a number represented in the carry-save system and a number represented in the conventional binary system. It consists of a row of full-adder cells, where a full-adder cell computes two bits t and u, from three bits x, y, and z, such that $2t + u$ equals $x + y + z$ (see Figure 2.4). A carry-save adder is presented in Figure 2.5.

$$u = x \oplus y \oplus z$$
$$t = xy + xz + yz$$

Figure 2.4   *A full-adder (FA) cell. From three bits x, y, and z, it computes two bits t and u such that $x + y + z = 2t + u$.*
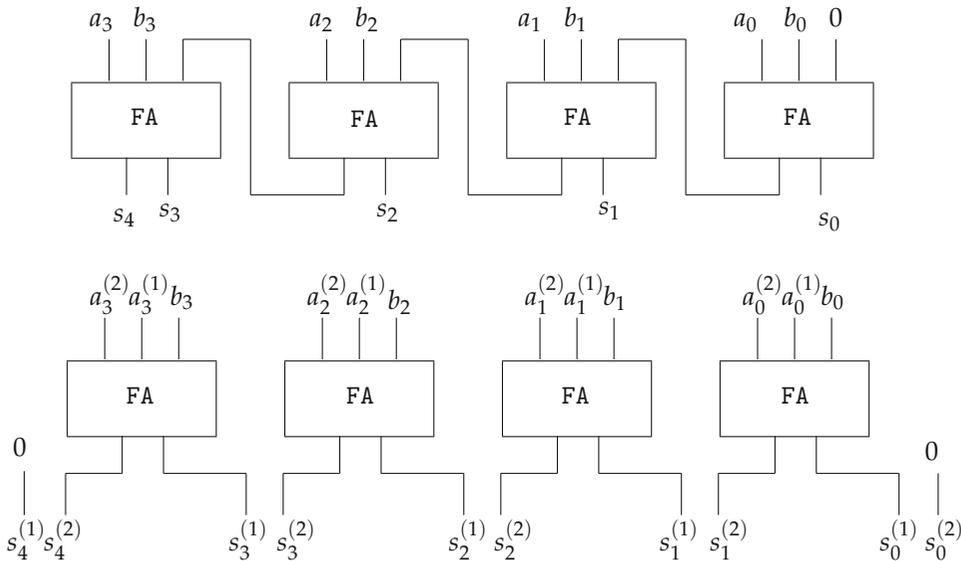

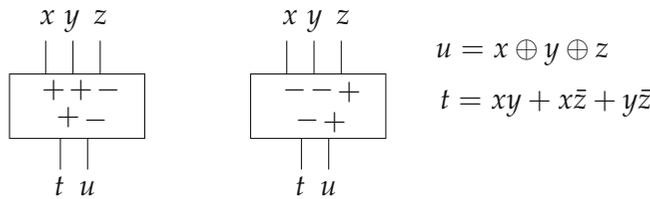**Figure 2.5**   *A carry-save adder (bottom), compared to a carry-propagate adder (top).*

$$u = x \oplus y \oplus z$$

$$t = xy + x\bar{z} + y\bar{z}$$

Figure 2.6 *A PPM cell. From three bits x, y, and z, it computes two bits t and u such that $x + y - z = 2t - u$.*

An adder structure for the borrow-save number system can easily be built using elementary cells slightly different from the FA cell. Algorithm 12 adds two BS numbers.

Algorithm 12 Borrow-Save addition

- input: two BS numbers $a = a_{n-1}a_{n-2}\cdots a_0$ and $b = b_{n-1}b_{n-2}\cdots b_0$, where the digits a_i and b_i belong to $\{-1, 0, 1\}$, each digit d being represented by two bits d^+ and d^- such that $d^+ - d^- = d$.
- output: a BS number $s = s_n s_{n-1}\cdots s_0$ satisfying $s = a + b$.

For each $i = 0, \ldots, n-1$, compute two bits c_{i+1}^+ and c_i^- such that $2c_{i+1}^+ - c_i^- = a_i^+ + b_i^+ - a_i^-$;

For each $i = 0, \ldots, n-1$, compute s_{i+1}^- and s_i^+ such that $2s_{i+1}^- - s_i^+ = c_i^- + b_i^- - c_i^+$ (with $c_0^+ = c_n^- = 0$, and $s_n^+ = c_n^+$).

Both steps of this algorithm require the same elementary computation: from three bits x, y, and z we must find two bits t and u such that $2t - u = x + y - z$. This can be done using a *PPM cell* ("PPM" stands for "Plus Plus Minus"), depicted in Figure 2.6, which is very similar to the FA cell previously described. Using PPM cells, one can easily derive the borrow-save adder of Figure 2.7 from the algorithm. It is possible to add a number represented in the borrow-save system and a number represented in the conventional, nonredundant, binary system by using only one row of PPM cells.[14] This is described in Figure 2.8. More details on borrow-save based arithmetic operators can be found in [24].

2.3.3   Canonical Recoding

Multiplying a given number a by a binary number $b = b_{n-1}b_{n-2}\cdots b_0 = \sum_{i=0}^{n-1} b_i 2^i$ reduces to computing

$$\sum_{i=0}^{n-1} b_i \cdot (a2^i),$$

i.e., we need to add as many shifted copies of a as there are nonzero values b_i. Very soon (first for accelerating multiplications when they were performed in software, using additions and shifts, and later on for accelerating hardwired multiplication and reducing the area of multipliers), authors tried to *recode* the operand b in order to reduce its number of nonzero digits. With the conventional binary representation (for which the only values allowed for b_i are 0 and 1), we have only one possible

---

[14]The carry-save and borrow-save systems are roughly equivalent: everything that is computable using one of these systems is computable at approximately the same cost as with the other one.
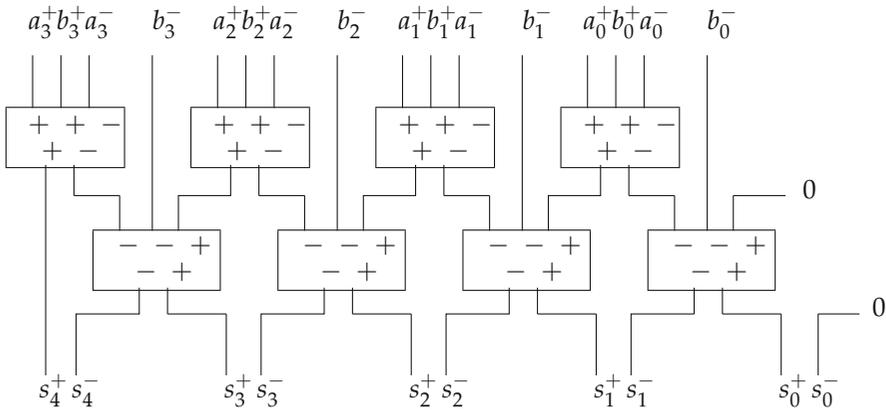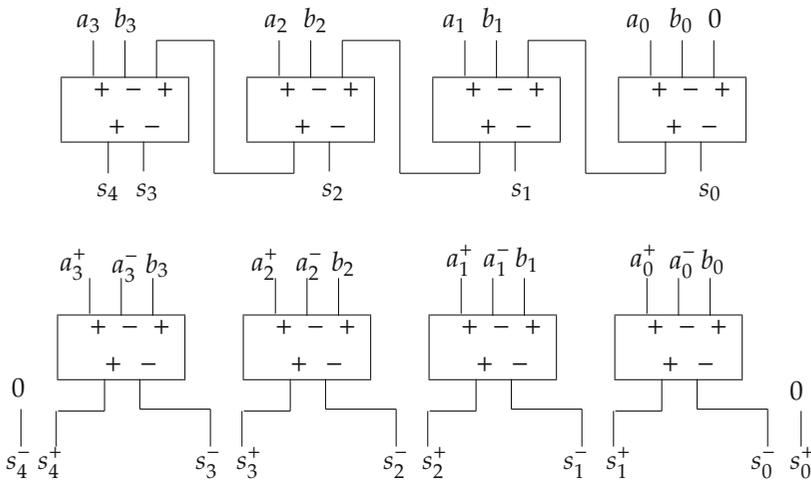
Figure 2.7 *A borrow-save adder.*

Figure 2.8 *A structure for adding a borrow-save number and a nonredundant number* (*bottom*), *compared to a carry-propagate subtractor* (*top*).

representation for each integer b, so attempting to reduce the number of nonzero digits makes no sense, but as we have seen before, if we allow digits from the larger digit set $\{-1, 0, 1\}$, we obtain a *redundant* number system: numbers have several possible representations, so that it makes sense to try to minimize the number of nonzero digits. In our initial multiplication problem, when $b_i = -1$, the number $a2^i$ is subtracted, which can be done with approximately the same delay and/or silicon area as an addition. For instance (still using the symbol $\bar{1}$ for representing the digit "-1"), the binary number

$$11101001111$$

can be "recoded"

$$100\bar{1}0101000\bar{1},$$

and it can be shown (it is a consequence of Theorem 6, below) that the number of nonzero digits of this last representation is minimal. Booth [47] first suggested to recode the initial binary chain by replacing all sub-strings of the form

$$0\underbrace{1\cdots 111}_{k \text{ ones}}$$

by

$$1\underbrace{0\cdots 00}_{k-1 \text{ zeros}}\overline{1}.$$

More formally, assuming that the initial n-bit binary chain is $d_{n-1}d_{n-2}\cdots d_0$, the Booth-recoded, $n+1$-digit chain, $f_n f_{n-1} f_{n-2}\cdots f_0$ is obtained using the set of rules given in Table 2.4.

Unfortunately, that set of rules does not always generate a digit string with a minimal number of nonzero digits. Indeed, the "recoded" digit chain may even have more nonzero digits than the initial one. Just consider the input chain

$$10101010101,$$

whose Booth recoding is

$$1\overline{1}1\overline{1}1\overline{1}1\overline{1}1\overline{1}1\overline{1}.$$

Several authors suggested different recodings that are guaranteed to have less than $\lceil n/2 \rceil$ nonzero digits (where n is the length of the original binary chain). In his seminal paper [393], Reitwiesner suggested a recoding algorithm that generates digit chains that are "minimal," i.e., they have the smallest possible number of nonzero digits. Such digit chains are called *Canonical recodings*. Reitwiesner's algorithm consists in performing the transformation presented in Table 2.5. Looking at the table, one easily checks that we always have

$$2c_{i+1} + f_i = c_i + d_i,$$

from which we immediately deduce that

$$\sum_{i=0}^{n} f_i 2^i = \sum_{i=0}^{n-1} d_i 2^i,$$

i.e., the algorithm effectively generates a digit string that represents the input number.

Looking at the table, we can immediately find the following basic property of the digit strings generated by Reitwiesner's algorithm (which explains why the generated digit string is sometimes called *nonadjacent form* [208]).

Table 2.4 *The set of rules that generate the Booth recoding $f_n f_{n-1} f_{n-2}\cdots f_0$ of a binary number $d_{n-1}d_{n-2}\cdots d_0$ (with $d_i \in \{0,1\}$ and $f_i \in \{-1,0,1\}$). By convention $d_n = d_{-1} = 0$.*

d_n	d_{n-1}	f_n
0	0	0
0	1	1
1	0	$\overline{1}$
1	1	0

Table 2.5 *Reitwiesner's algorithm: the set of rules that generate the canonical recoding $f_n f_{n-1} f_{n-2} \cdots f_0$ of a binary number $d_{n-1} d_{n-2} \cdots d_0$ (with $d_i \in \{0, 1\}$ and $f_i \in \{-1, 0, 1\}$). The process is initialized by setting $c_0 = 0$, and by convention $d_n = 0$.*

c_i	d_{i+1}	d_i	f_i	c_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	0	0
0	1	1	$\bar{1}$	1
1	0	0	1	0
1	0	1	0	1
1	1	0	$\bar{1}$	1
1	1	1	0	1

Property 4 (The nonzero digits of a digit string generated by Reitwiesner's algorithm are *nonadjacent*.) *If $f_n f_{n-1} \cdots f_0$ is the recoding of the binary string $d_{n-1} \cdots d_0$ deduced from the set of rules presented in Table 2.5 then for any i, $f_i \cdot f_{i+1} = 0$.*

A first consequence of this is that the canonical recoding of an n-bit binary string has at most $\lceil n/2 \rceil$ nonzero digits. A second consequence, due to Theorem 6 below, is that the digit strings generated by Reitwiesner's algorithm are minimal: they have the smallest possible number of nonzero digits.

Theorem 6 (Reitwiesner's theorem [393]: minimality of nonadjacent digit chains) *Assume an integer x is represented by a nonadjacent binary digit chain, i.e.,*

$$x = f_n f_{n-1} f_{n-2} \cdots f_0$$

with

$$\forall i, \begin{cases} f_i \in \{-1, 0, 1\}, \\ f_i \cdot f_{i+1} = 0 \end{cases}$$

Any binary representation of x with digits in $\{-1, 0, 1\}$ will contain as least as many nonzero digits as the digit chain $f_n f_{n-1} f_{n-2} \cdots f_0$.

Springer