Computing floating-point square roots via bivariate polynomial evaluation

Claude-Pierre Jeannerod, Hervé Knochel, Christophe Monat, Guillaume Revy

Computing floating-point square roots via bivariate polynomial evaluation

Claude-Pierre Jeannerod, Hervé Knochel, Christophe Monat, Guillaume Revy

Abstract—In this paper we show how to reduce the computation of correctly-rounded square roots of binary floating-point data to the fixed-point evaluation of some particular integer polynomials in two variables. By designing parallel and accurate evaluation schemes for such bivariate polynomials, we show further that this approach allows for high instruction-level parallelism (ILP) exposure, and thus potentially low latency implementations. Then, as an illustration, we detail a C implementation of our method in the case of IEEE 754-2008 *binary32* floating-point data (formerly called *single precision* in the 1985 version of the IEEE 754 standard). This software implementation, which assumes 32-bit unsigned integer arithmetic only, is almost complete in the sense that it supports special operands, subnormal numbers, and all rounding-direction attributes, but not exception handling (that is, status flags are not set). Finally we have carried out experiments with this implementation on the ST231, an integer processor from the STMicroelectronics' ST200 family, using the ST200 family VLIW compiler. The results obtained demonstrate the practical interest of our approach in that context: for all rounding-direction attributes, the generated assembly code is optimally scheduled and has indeed low latency (23 cycles).

Index Terms—Binary floating-point arithmetic, square root, correct rounding, IEEE 754, polynomial evaluation, instruction-level parallelism, rounding error analysis, C software implementation, VLIW integer processor.

1 Introduction

This paper deals with the design and software implementation of an efficient `sqrt` operator for computing square roots of *binary32* floating-point data. As mandated by the IEEE 754 standard (whose initial 1985 version [1] has been revised in 2008 [2]), our implementation supports gradual underflow and the four rounding-direction attributes required for binary format implementations. However, the status flags used for handling exceptions are not set.

As for other basic arithmetic operators, the IEEE 754 standard specifies that `sqrt` operates on and returns floating-point data. Floating-point data are either special data (signed infinities, signed zeros, not-a-numbers (NaN)) or nonzero finite floating-point numbers. In radix two, *nonzero finite floating-point numbers* have the form $x = \pm m \cdot 2^e$, with e an integer such that

$$e_{\min} \leq e \leq e_{\max}, \tag{1}$$

and m a positive rational number having binary expansion

$$m = (\underbrace{0.0\cdots0}_{\lambda \text{ zeros}} 1 m_{\lambda+1} \cdots m_{p-1})_2. \tag{2}$$

- *C.-P. Jeannerod is with INRIA (project-team Arénaire, Lyon, France). E-mail: claude-pierre.jeannerod@ens-lyon.fr*
- *H. Knochel and C. Monat are with STMicroelectronics' Compilation Expertise Center (Grenoble, France). E-mail: {herve.knochel, christophe.monat}@st.com*
- *G. Revy is a member of ParLab within EECS Department at the University of California at Berkeley. This work was done while he was with Université de Lyon - ÉNS Lyon (project-team Arénaire, Lyon, France). E-mail: grevy@eecs.berkeley.edu*

Manuscript last updated March 26, 2010.

Since m is nonzero, the number λ of its leading zeros varies in $\{0, 1, \ldots, p-1\}$. If $|x| < 2^{e_{\min}}$ then x is *subnormal* else it is *normal*. On the one hand, subnormal numbers are such that $e = e_{\min}$ and $\lambda > 0$. On the other hand, normal numbers will be considered only through their normalized representation, that is, the unique representation of the form $\pm m \cdot 2^e$ for which $\lambda = 0$.

The parameters $e_{\min}$, $e_{\max}$, p used so far represent the extremal exponents and the precision of a given binary format. In this paper, we assume they satisfy

$$e_{\min} = 1 - e_{\max} \quad \text{and} \quad 2 \leq p \leq e_{\max}.$$

This assumption is verified for all the binary formats defined in the IEEE 754-2008 standard [2]. This standard further prescribes that the operator $\texttt{sqrt} : x \mapsto r$ specifically behaves as follows:

- If x equals either -0, $+0$, or $+\infty$ then r equals x.
- If x is nonzero negative or NaN then r is NaN.

Those two cases cover what we shall call *special operands*. In all other cases x is a positive nonzero finite floating-point number, that is,

$$x = m \cdot 2^e, \tag{3}$$

with e as in (1) and m as in (2); the result specified by the IEEE 754-2008 standard is then the so-called *correctly-rounded* value

$$r = \circ(\sqrt{x}), \tag{4}$$

where $\circ$ is any of the four rounding-direction attributes: to nearest even (RN), up (RU), down (RD), and to zero (RZ). In fact, since $\sqrt{x} \geq 0$, rounding to zero is the same as rounding down. Therefore considering only the first three rounding-direction attributes is enough:

$$\circ \in \{\text{RN}, \text{RD}, \text{RU}\}.$$

As we will see in Section 2, deducing r from x essentially amounts to taking the square root, up to scaling, of the significand m. For doing this, many algorithms are available (see for example the survey [3] and the reference books [4], [5], [6]).

The method we introduce in this paper is based exclusively on fixed-point evaluation of a suitable bivariate polynomial with integer coefficients. Since polynomial evaluation is intrinsically parallel, this approach allows for very high ILP exposure. Thus, in some contexts such as VLIW integer processors, a significant reduction of latency can be expected. For the ST231, a STMicroelectronics VLIW processor which does not have native floating-point capabilities, we show in this paper that this is indeed the case: the assembly codes generated for optimized implementations of our approach turn out to be optimally scheduled and have latencies reduced by over 30% compared to previously fastest available methods, implemented in [7]; also, the latency overhead for subnormal support is only 2 cycles, yielding a latency of 23 cycles for all $\circ$. This latency for square root compares favourably with the latencies of addition and multiplication which, in the same context, range respectively from 23 to 27 and 18 to 21 cycles, depending on $\circ$ (see http://flip.gforge.inria.fr/ and [8, p. 4]).

The paper is organized as follows. Section 2.1 details three mathematical properties of square roots of binary floating-point numbers, and Section 2.2 shows how to use them to deduce the usual high level algorithmic description of the `sqrt` operator.

In Section 3.1 we then show how to introduce suitable bivariate polynomials that approximate our square root function. In particular, we give some approximation and evaluation error bounds that are sufficient to ensure correct rounding, along with an example of such a polynomial and its error bounds in the case of the *binary32* format. Section 3.2 then details, for each rounding-direction attribute, how to deduce a correctly-rounded value from the approximate polynomial value obtained so far. A summary of our new approach is given in Section 3.3.

A standard C99 implementation of this approach is given in Section 4, for the *binary32* format and assuming that 32-bit unsigned integer arithmetic is available: Section 4.1 shows how to handle special operands; Section 4.2 deals with the computation of the result exponent and a parity bit related to the input exponent (which is needed several times in the rest of the algorithm); Sections 4.3 and 4.4 show how to compute the evaluation point and the value of the polynomial at this evaluation point; there, we also explain how the accuracy of the evaluation scheme has been verified; Finally, Section 4.5 details how to implement correct rounding. This results in three separate square root implementations, one for each rounding-direction attribute.

For these implementations, all that is needed is a C compiler that implements 32-bit arithmetic. However, our design has been guided by a specific target, the ST231 VLIW integer processor from STMicroelectronics' ST200 family. Section 5 is devoted to some experiments carried out with this target and the ST200 family VLIW compiler. After a review of the main features of the ST231 in Section 5.1, the performance results we have obtained in this context are presented and analysed in Section 5.2.

2 Properties of floating-point square roots and general algorithm

The general scheme for moving from (3) to (4) essentially follows from three properties of the square root function in binary floating-point arithmetic.

2.1 Floating-point square root properties

Property 2.1: For x in (3), the real number $\sqrt{x}$ lies in the range of positive normal floating-point numbers, that is,

$$\sqrt{x} \in \left[2^{e_{\min}}, (2-2^{1-p})\cdot 2^{e_{\max}}\right].$$

This first property (see [9] for a proof) implies that $\circ(\sqrt{x})$ is a positive normal floating-point number. Correctly-rounded square roots thus never denormalize nor under/overflow, a fact which will simplify the implementation considerably.

In order to find the normalized representation of $\circ(\sqrt{x})$, let

$$e' = e - \lambda \quad \text{and} \quad m' = m \cdot 2^{\lambda}. \tag{5}$$

It follows that the positive (sub)normal number x defined in (3) satisfies $x = m' \cdot 2^{e'}$ and that $m' \in [1,2)$. Taking the square root then yields

$$\sqrt{x} = \ell \cdot 2^{d},$$

where the real ℓ and the integer d are given by

$$\ell = \sigma\sqrt{m'} \quad \text{with} \quad \sigma = \begin{cases} 1, & \text{if } e' \text{ is even,} \\ \sqrt{2}, & \text{if } e' \text{ is odd,} \end{cases} \tag{6}$$

and, using $\lfloor\ \rfloor$ to denote the usual floor function,

$$d = \lfloor e'/2 \rfloor. \tag{7}$$

It follows from the definition of σ and m' that ℓ is a real number in $[1,2)$. Therefore, rounding $\sqrt{x}$ amounts to round ℓ, and we have shown the following property:

Property 2.2: Let x, ℓ, d be as above. Then

$$\circ(\sqrt{x}) = \circ(\ell)\cdot 2^{d}.$$

In general the fact that $\ell \in [1,2)$ only implies the weaker enclosure $\circ(\ell) \in [1,2]$. This yields two separate cases: if $\circ(\ell) < 2$ then Property 2.2 already gives the normalized representation of the result r; if $\circ(\ell) = 2$ then we must further correct d after rounding, in order to return $r = 1 \cdot 2^{d+1}$ instead of the unnormalized representation $2 \cdot 2^{d}$ given by Property 2.2. The next property, proven in [9], characterizes precisely when such a correction is needed or not. In particular, it is never needed in rounding-to-nearest mode.

Property 2.3: One has $\circ(\ell) = 2$ if and only if $\circ = \mathrm{RU}$ and e is odd and $m = 2 - 2^{1-p}$.

2.2 High level description of square root algorithms

Together with the IEEE 754-2008 specification recalled in Introduction, the properties of Section 2.1 lead to a general algorithm for computing binary floating-point square roots, shown in Figure 1 and Figure 2 for the different rounding-direction attributes. In particular, $\circ(\ell)$ and d are two functions of m and e which can be computed independently from each other.

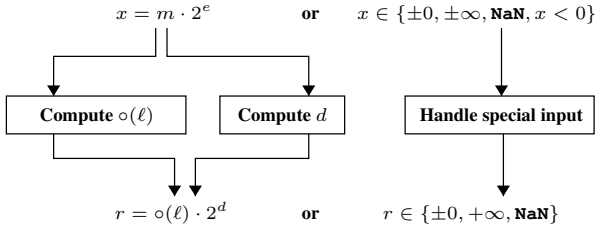


Fig. 1. Square root algorithm for $\circ \in \{\mathsf{RN}, \mathsf{RD}\}$.

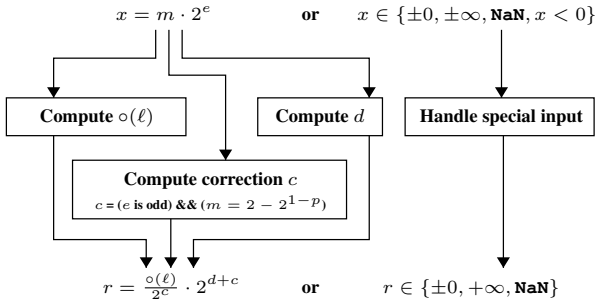


Fig. 2. Square root algorithm for $\circ = \mathsf{RU}$.

(We will see in Section 4.2 that the correction step in Figure 2 can in fact be avoided when using the standard encoding of binary floating-point data.)

Given m and e, computing d is algorithmically easy since by (5) and (7) we have

$$d = \lfloor (e - \lambda)/2 \rfloor . \tag{8}$$

However, computing $\circ(\ell)$ from m and e is far less immediate and typically dominates the cost. In the next section, we present a new way of producing $\circ(\ell)$, which we have chosen because we believe it allows to express the most ILP.

3 Computing $\circ(\ell)$ by Correcting Truncated Approximations

It is useful to start by characterizing the meaning of $\circ(\ell)$ for each rounding-direction attribute $\circ$. Since the real number ℓ belongs to $[1, 2)$, we have the following, where n is a *normal* number:

- $\mathsf{RN}(\ell)$ is the unique n such that

$$-2^{-p} < \ell - n < 2^{-p}, \tag{9}$$

- $\mathsf{RD}(\ell)$ is the unique n such that

$$0 \le \ell - n < 2^{1-p}, \tag{10}$$

- $\mathsf{RU}(\ell)$ is the unique n such that

$$-2^{1-p} < \ell - n \le 0. \tag{11}$$

Both inequalities in (9) are strict, since the square root of a floating-point number cannot be exactly halfway between two consecutive floating-point numbers (see [4, Theorem 9.4], [5, p. 242], or [6, p. 463]). Note also that (9) and (10) both imply $n \in [1, 2)$, while (11) implies the weaker enclosure $n \in [1, 2]$.

Given p, $\circ$, m', and σ, there are many ways of producing the bits of such an n. A way that will allow to express much ILP is by correcting a truncated approximation of ℓ. This approach (detailed for example in [6, p. 459] for division) has three main steps:

- compute a "real number" v approximating ℓ from above with absolute error less than 2^{-p}, that is,

$$-2^{-p} < \ell - v \le 0; \tag{12}$$

- deduce u by truncating v after p fraction bits:

$$0 \le v - u < 2^{-p}; \tag{13}$$

- obtain n by adding, if necessary, a small correction to u and then by truncating after $p - 1$ fraction bits.

Of course the binary expansion of v in (12) will be finite: by "real number" we simply mean a number with precision higher than the target precision p. Typically, v will be representable with at most k bits, with k the register width (for example, $p = 24$ and $k = 32$ for our implementation in Section 4). On the other hand, using $\ell \le \sqrt{2} \cdot \sqrt{2 - 2^{1-p}}$ one may check that if v satisfies (12) then necessarily

$$1 \le v < 2. \tag{14}$$

Therefore, the binary expansion of v has the form

$$v = (1.v_1 \dots v_{p-1} v_p \dots v_{k-1})_2. \tag{15}$$

Our approach for computing v as in (12) and (15) by means of bivariate polynomial evaluation is detailed in Section 3.1 below.

Once v is known, truncation after p fraction bits gives

$$u = (1.v_1 \dots v_{p-1} v_p)_2. \tag{16}$$

The fraction of u is wider than that of n by one bit. We will see in Section 3.2 how to correct u into n by using both this extra bit and the fact that, because of (12) and (13),

$$|\ell - u| < 2^{-p}. \tag{17}$$

3.1 Computing v by bivariate polynomial evaluation

The problem is, given p, m', and σ, to compute an approximation v to ℓ such that (12) holds. Such a value v will in fact be obtained as a solution to

$$\left|(\ell + 2^{-p-1}) - v\right| < 2^{-p-1}. \tag{18}$$

Although slightly more strict than (12), this form will be the natural result of some derivations based on the triangle inequality.

Computing v such that (18) holds usually relies on *iterative refinement* (Newton-Raphson or Goldschmidt method [6, §7.3]), or *univariate polynomial evaluation* [7] (see also [10] for a different context, namely when an FMA instruction is available), or a combination of both [11], [12]. The approach we shall detail now is based exclusively on polynomial evaluation. However, instead of using two polynomials as in [7], we will use a single one, which is bivariate; this makes the approach simpler, more flexible, and eventually faster provided some parallelism is available.

The main steps for producing v via bivariate polynomial evaluation are shown on the diagram below:

$$\begin{array}{ccc} \ell + 2^{-p-1} & \longrightarrow & F(s,t) \\ {\scriptstyle ?}\downarrow & & \downarrow{\scriptstyle \text{function approximation}} \\ v & \underset{\text{polynomial evaluation}}{\longleftarrow} & P(s,t) \end{array}$$

First, using (6) and defining

$$\tau = m' - 1, \tag{19}$$

the real number $\ell + 2^{-p-1}$ can be seen as the value at $(s,t) = (\sigma,\tau)$ of the bivariate function

$$F(s,t) = 2^{-p-1} + s\sqrt{1+t}. \tag{20}$$

Then, let

$$\mathcal{S} = \{1, \sqrt{2}\} \qquad \text{and} \qquad \mathcal{T} = \{h \cdot 2^{1-p}\}_{h=0,1,\ldots,2^{p-1}-1}$$

be the variation domains of σ and τ, respectively, and let

$$1^- = 1 - 2^{1-p}.$$

Since $\mathcal{T} \subset [0, 1^-]$, a second step is the approximation of $F(s,t)$ on $\{1, \sqrt{2}\} \times [0, 1^-]$ by a bivariate polynomial $P(s,t)$. The function F being linear with respect to its first variable, a natural choice for P is

$$P(s,t) = 2^{-p-1} + s \cdot a(t), \tag{21}$$

with $a(t)$ a univariate polynomial that approximates $\sqrt{1+t}$ on $[0, 1^-]$. The third and last step is the evaluation of P at (σ,τ) by a finite-precision, straight-line program $\mathcal{P}$, the resulting value $\mathcal{P}(\sigma,\tau)$ being assigned to v.

Intuitively, if $a(t)$ is "close enough" to $\sqrt{1+t}$ over the whole interval $[0, 1^-]$ and if $\mathcal{P}$ evaluates P "accurately enough" on the whole domain $\mathcal{S} \times \mathcal{T}$ then the resulting value v should be close enough to $\ell + 2^{-p-1}$ in the sense of (18). This claim is made precise by the next lemma.

Lemma 1: Given p, σ, τ, a, P, $\mathcal{P}$ as above, let $\alpha(a)$ be the approximation error defined by

$$\alpha(a) = \max_{t \in [0,1^-]} \left| \sqrt{1+t} - a(t) \right|,$$

and let $\rho(\mathcal{P})$ be the rounding error defined by

$$\rho(\mathcal{P}) = \max_{(s,t) \in \mathcal{S} \times \mathcal{T}} \left| P(s,t) - \mathcal{P}(s,t) \right|.$$

Let further $v = \mathcal{P}(\sigma,\tau)$. If

$$\sqrt{2} \cdot \alpha(a) + \rho(\mathcal{P}) < 2^{-p-1} \tag{22}$$

then v satisfies (18).

Proof: Using the definitions of F and P, we have

$$\begin{aligned} \left| (\ell + 2^{-p-1}) - v \right| &= |F(\sigma,\tau) - \mathcal{P}(\sigma,\tau)| \\ &\le |F(\sigma,\tau) - P(\sigma,\tau)| \\ &\quad + |P(\sigma,\tau) - \mathcal{P}(\sigma,\tau)| \\ &\le \sigma \left| \sqrt{1+\tau} - a(\tau) \right| + \rho(\mathcal{P}). \end{aligned}$$

Since $1 \le \sigma \le \sqrt{2}$ and $\tau \in [0, 1^-]$, it follows that

$$\left| (\ell + 2^{-p-1}) - v \right| \le \sqrt{2} \cdot \alpha(a) + \rho(\mathcal{P}),$$

which concludes the proof. □

It remains to find a polynomial approximant a together with an evaluation program $\mathcal{P}$ so that $\alpha(a)$ and $\rho(\mathcal{P})$ satisfy (22). In practice, since $\alpha(a)$ and $\rho(\mathcal{P})$ may be real numbers, a certificate will consist in computing two dyadic numbers d_α and d_ρ such that

$$\alpha(a) \le d_\alpha, \quad \rho(\mathcal{P}) \le d_\rho, \quad \sqrt{2} \cdot d_\alpha + d_\rho < 2^{-p-1}.$$

The construction of a and $\mathcal{P}$ is highly context-dependent: it is guided by both the value of p and some features of the target processor (register precision k, instruction set, latencies, degree of parallelism,...). The two paragraphs below illustrate how to choose a and $\mathcal{P}$ in the case where $k = 32$ and $p = 24$.

3.1.1 Constructing a polynomial approximant

Since P in (21) will be evaluated at run-time, a small degree for a is usually preferred. One may guess the smallest possible value of $\deg(a)$ as follows. The rounding error $\rho(\mathcal{P})$ in (22) being non-negative, a must satisfy

$$\alpha(a) < 2^{-p-3/2}. \tag{23}$$

Now let $\mathbb{R}[t]_d$ be the set of univariate real polynomials of degree at most d and recall, for example from [13, §3.2], that the *minimax* degree-d approximation of $\sqrt{1+t}$ on $[0, 1^-]$ is the unique $a^* \in \mathbb{R}[t]_d$ such that

$$\alpha_d^* := \alpha(a^*) \le \alpha(a), \quad \text{for all } a \in \mathbb{R}[t]_d. \tag{24}$$

Thus, by combining (23) and (24), one must have

$$\alpha^*_{\deg(a)} < 2^{-p-3/2}.$$

A lower bound on $\deg(a)$ can then be guessed by estimating α_d^* numerically for increasing values of d until $2^{-p-3/2}$ is reached.

For example, for $p = 24$, the degree of a must satisfy $\alpha^*_{\deg(a)} < 2^{-25.5}$. Comparing to the estimations[1] of α_d^* displayed in Table 1 indicates that a should be of degree at least 8.

Once we have an idea of the smallest possible degree for a, it remains to find a machine representation of a. For this representation, a typical choice is the monomial basis $1, t, t^2, \ldots$ together with some coefficients $a_0, a_1, a_2, \ldots$ that are exactly representable using at most k bits.

1. Such estimations can be computed for example using Remez' algorithm, which is available in *Sollya* (http://sollya.gforge.inria.fr/); see also [13, §3.5], [14], [15].

TABLE 1
Numerical estimations of α_d^* for $5 \le d \le 10$.

d	5	6	7	8	9	10
$-\log_2(\alpha_d^*)$	19.58	22.47	25.31	28.12	30.89	33.65

Continuing the previous example where $k = 32$, it turns out that $a(t) = \sum_{i=0}^{8} a_i t^i$ can be chosen such that

$$a_0 = 1 \quad \text{and} \quad a_i = (-1)^{i+1} A_i \cdot 2^{-31},\ 1 \le i \le 8, \quad (25)$$

with the following values for the A_i's (including $A_0 = a_0 \cdot 2^{31}$): $A_0 = 2^{31}$, $A_1 = 2^{30}$, $A_2 = 2^{28}$, $A_3 = 134124516$, $A_4 = 82769556$, $A_5 = 53306947$, $A_6 = 29806269$, $A_7 = 11452029$, and $A_8 = 2^{21}$. (See Listing 3 for their hexadecimal values). These integers A_i were found by truncating the coefficients obtained after several calls to *Sollya*'s Remez algorithm and by favoring powers of two. Each of them can be stored using only 32 bits and four of them are powers of two. Notice also that

$$A_8 \le \cdots \le A_2 \le A_1 \le A_0. \quad (26)$$

A certified supremum norm computation (implemented for example in *Sollya*; see also [16], [17]) applied to this particular polynomial gives a bound less than $2^{-25.5}$ as required by (23). (The computed bound has the form $2^{-25.972\cdots}$.)

3.1.2 Writing an evaluation program

In our context, the evaluation program $\mathcal{P}$ is typically a piece of C code that implements a finite-precision computation of $P(s,t)$. It should be accurate enough in the sense of (22) and, since we favor latency (rather than throughput, for example), as fast as possible.

Such an implementation will not require using floating-point arithmetic, and fixed-point arithmetic will suffice to evaluate $P(s,t)$ accurately enough. In addition, we have the lemma below, which shows that $P(s,t)$ lies in a fairly small range.

Lemma 2: If $(s,t) \in \mathcal{S} \times \mathcal{T}$ then $P(s,t) \in (1,2)$.

Proof: Let $\epsilon = 2^{-p-3/2}$. It follows from the definition of $\alpha(a)$ and the bound in (23) that

$$\sqrt{1+t} - \epsilon < a(t) < \sqrt{1+t} + \epsilon.$$

Using $0 \le t \le 1 - 2^{1-p}$ and $\sqrt{2-2^{1-p}} \le \sqrt{2} - 2\epsilon$, we get

$$1 - \epsilon < a(t) < \sqrt{2} - \epsilon. \quad (27)$$

Since $1 \le s \le \sqrt{2}$ and $\epsilon < 2^{-p-1}$, we deduce that $P(s,t) = 2^{-p-1} + s \cdot a(t)$ belongs to $(1,2)$. □

With $\alpha(a) \le 2^{-25.972\cdots}$ as in the previous paragraph, a sufficient condition on $\rho(\mathcal{P})$ for (22) to be satisfied is

$$\rho(\mathcal{P}) < 2^{-25} - \sqrt{2} \cdot 2^{-25.972\cdots} = 2^{-26.840\cdots}.$$

Rounding the evaluation point. When designing an evaluation program $\mathcal{P}$ that achieves this accuracy, a preliminary step is to make the input (σ, τ) machine-representable. On the one hand, the binary expansion of τ is $0.m_{\lambda+1} \cdots m_{p-1}$ and thus, since λ is non-negative, τ is already representable using k bits provided $p - 1 \le k$. On the other hand, writing RN_k for rounding-to-nearest in precision k, we shall replace σ defined in (6) by

$$\widehat{\sigma} = \begin{cases} 1, & \text{if } e' \text{ is even,} \\ \mathrm{RN}_k(\sqrt{2}), & \text{if } e' \text{ is odd.} \end{cases} \quad (28)$$

The lemma below gives an upper bound on the loss of accuracy that occurs when rounding (σ, τ) to $(\widehat{\sigma}, \tau)$.

Lemma 3: Given p, k, s, t, a, P, $\mathcal{P}$ as above, let $\widehat{\mathcal{S}} = \{1, \mathrm{RN}_k(\sqrt{2})\}$ and define

$$\widehat{\rho}(\mathcal{P}) = \max_{(s,t) \in \widehat{\mathcal{S}} \times \mathcal{T}} |P(s,t) - \mathcal{P}(s,t)|.$$

Then

$$\rho(\mathcal{P}) < 2^{1/2-k} + \widehat{\rho}(\mathcal{P}).$$

Proof: By definition, the bound $\rho(\mathcal{P})$ is attained for some $(s_0, t_0) \in \mathcal{S} \times \mathcal{T}$. Writing $\hat{s}_0 = \mathrm{RN}_k(s_0)$, one has $\mathcal{P}(\hat{s}_0, t_0) = \mathcal{P}(s_0, t_0)$ and, by the triangle inequality,

$$\rho(\mathcal{P}) \le |s_0 - \hat{s}_0| \cdot |a(t_0)| + \widehat{\rho}(\mathcal{P}).$$

The conclusion then follows from $|s_0 - \hat{s}_0| \le 2^{-k}$ and the fact that (27) implies $|a(t_0)| < \sqrt{2}$. □

Applying Lemma 3 with $k = 32$ shows that we are left with finding an evaluation program $\mathcal{P}$ such that

$$\widehat{\rho}(\mathcal{P}) \le 2^{-26.840\cdots} - 2^{-31.5} = 2^{-26.898\cdots}. \quad (29)$$

Designing an evaluation program. By evaluation program, we mean a division-free straight-line program, that is, roughly, a set of instructions computing $P(s,t)$ from s, t, p and the a_i's by using only additions, subtractions, and multiplications, without branching. In our context we shall assume first unbounded parallelism and some latencies for addition/subtraction and multiplication. Then we parenthesize the expression $P(s,t)$ in order to expose as much ILP as we can and, thus, to decrease the overall latency of polynomial evaluation. An example of such a parenthesization, generated automatically in the same spirit as [18], is

$$\begin{aligned} P(s,t) \quad = \quad & \Big[\Big(\big(2^{-p-1} + s \cdot (a_0 + a_1 t)\big) + a_2 \cdot (s \cdot t^2)\Big) \\ & \qquad + a_3 t \cdot (s \cdot t^2)\Big] \\ & + \Big[\big(t^2 \cdot (s \cdot t^2)\big) \cdot \Big((a_4 + a_5 t) + t^2 \cdot \big((a_6 + a_7 t) + a_8 t^2\big)\Big)\Big]. \end{aligned}$$

Note that t^2 and $s \cdot t^2$ are common subexpressions. With unlimited parallelism and latencies of 1 for addition, of 3 for (pipelined) multiplication, and of 1 for multiplication by a power of two (that is, a shift), such a parenthesization gives a latency of 13 (compared to 34 for Horner's scheme). We have found no parenthesization of smaller latency.

Accuracy issues. In our example, the rounding error of the program $\mathcal{P}$ that implements in fixed-point arithmetic the above parenthesization must satisfy (29). This requirement will be checked in Section 4.4.

For now, let us notice that this parenthesization can in fact be implemented using 32-bit unsigned integers only, which avoids to loose one bit of precision because of the need to store the sign of the coefficients a_i. Indeed, an appropriate choice of arithmetic operators can be found, that ensures that all intermediate variables are positive:

$$\begin{aligned} P(s,t) \;=\; & \Big[\Big(\big(2^{-p-1} + s\cdot(a_0 + a_1 t)\big) - |a_2|\cdot(s\cdot t^2)\Big) \\ & \qquad + a_3 t\cdot(s\cdot t^2)\Big] \\ & - \Big[\big(t^2\cdot(s\cdot t^2)\big)\cdot \\ & \quad \Big(\big(|a_4| - a_5 t\big) + t^2\cdot\big(\big(|a_6| - a_7 t\big) + |a_8| t^2\big)\Big)\Big]. \end{aligned} \tag{30}$$

3.2 Correction to ensure correct rounding

For each rounding-direction attribute $\circ$ we will now obtain $n = \circ(\ell)$ by correcting u in (16) and (17). How to correct u depends on whether u is above or below ℓ. Thus the rounding algorithms below rely on either $u \geq \ell$ or $u > \ell$, which can both be implemented exactly (see Subsection 4.5).

3.2.1 Rounding to nearest

An algorithm producing n as in (9) is:

> if $u \geq \ell$ then
> $\quad n :=$ truncate u after $p-1$ fraction bits
> else
> $\quad n :=$ truncate $u + 2^{-p}$ after $p-1$ fraction bits

If $v_p = 0$ the above algorithm always returns the value u; this is the desired result, for in this case u is already a floating-point number and thus (17) implies (9). If $v_p = 1$ then u is at least $1 + 2^{-p}$ and is the midpoint between the two consecutive floating-point numbers $u - 2^{-p}$ and $u + 2^{-p}$: the former is returned when $u > \ell$, the latter when $u < \ell$, and (17) implies (9) in both cases; the case $u = \ell$ never happens because ℓ cannot be a midpoint.

3.2.2 Rounding down

An algorithm producing n as in (10) is:

> if $u > \ell$ then
> $\quad n :=$ truncate $u - 2^{-p}$ after $p-1$ fraction bits
> else
> $\quad n :=$ truncate u after $p-1$ fraction bits

If $v_p = 1$ then this algorithm always returns the floating-point number $u - 2^{-p}$ which, because of (17), satisfies (10) as required. If $v_p = 0$ then u is already a floating-point number: if $u > \ell$ then $u \geq 1 + 2^{1-p}$ and the algorithm returns $u - 2^{1-p}$, which is the floating-point predecessor of u, by truncating the midpoint $u - 2^{-p}$; if $u \leq \ell$, the returned value is u; in both cases, using (17) yields (10).

3.2.3 Rounding up

An algorithm producing n as in (11) is:

> if $u \geq \ell$ then
> $\quad n :=$ truncate $u + 2^{-p}$ after $p-1$ fraction bits
> else
> $\quad n :=$ truncate $u + 2^{1-p}$ after $p-1$ fraction bits

If $v_p = 1$ this algorithm always produces the floating-point number $u + 2^{-p}$ which, because of (17), satisfies (11) as required. If $v_p = 0$ then u is already a floating-point number: if $u \geq \ell$, the algorithm returns u; if $u < \ell$, it returns $u + 2^{1-p}$, which is the floating-point successor of $u \leq 2 - 2^{1-p}$; in both cases, using (17) yields (11).

3.3 Summary: main steps of the computation of $\circ(\ell)$

The box "Compute $\circ(\ell)$" in Figures 1 and 2 can be replaced by the ones in Figure 3 below. This figure recapitulates the main steps of the approach we have proposed in Sections 3.1 and 3.2 for deducing the correctly-rounded value $\circ(\ell)$ from m and e.

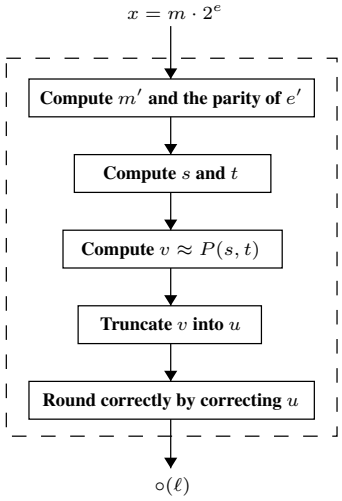


Fig. 3. Computation of $\circ(\ell)$ for $\circ \in \{\mathsf{RN}, \mathsf{RD}, \mathsf{RU}\}$.

4 Implementation details

The above square root method, which is summarized in Figures 1–3, can be implemented by operating exclusively on (unsigned) integers. We will now detail such an implementation, in C and for the *binary32* format of [2]. For this format the storage bit-width, precision, and maximum exponent are, respectively,

$$k = 32, \qquad p = 24, \qquad e_{\max} = 127.$$

When writing the code we have essentially assumed unbounded parallelism and that 32-bit integer arithmetic is available. Additional assumptions on the way input and output are encoded and on which operators are available, are as follows.

Input and output encoding. The operand x and result r of `sqrt` are implemented as integers $X, R \in$

$\{0, 1, \ldots, 2^{32} - 1\}$ whose bit strings correspond to the standard encoding of binary floating-point data (see [2, §3.4]). Our implementation of `sqrt` is thus a C function as in line 1 of Listing 2, which returns the value of R.

Table 2 indicates some useful relationship between x and X that are a consequence of this encoding. (Of course the same would hold for r and R.) Also, the bit string of X must be interpreted as follows: its first bit X_{31} gives the sign of x; the next 8 bits encode the biased exponent E of x as $E = \sum_{i=0}^{7} X_{i+23} 2^i$; the last 23 bits define the trailing significand field. In particular, if x is (sub)normal then

- the biased exponent E is related to the exponent e in (1) as follows:

$$E = e + 127 - [\lambda > 0], \tag{31}$$

  where $[\lambda > 0] = 1$ if x is subnormal, 0 otherwise;
- the trailing significand field carries the bits of m in (2) as follows:

$$X_{22} \ldots X_0 = \underbrace{0 \ldots 01}_{\lambda \text{ bits}} m_{\lambda+1} \ldots m_{23}. \tag{32}$$

Available operators. Besides the usual operators like `+`, `-`, `<<`, `>>`, `&`, `|`, `^`, we assume a fast way to compute the following functions for $A, B \in \{0, 1 \ldots, 2^{32} - 1\}$:

- $\max(A, B)$;
- the number $\mathrm{nlz}(A)$ of leading (that is, leftmost) zeros of the bit string of A;
- $\lfloor AB/2^{32} \rfloor$, whose bit string contains the 32 most significant bits of the product AB.

In our C codes, the operators corresponding to these functions will be respectively written `max`, `nlz`, and `mul` for readability. More precisely, with the ST231 target in mind, we shall assume that the latencies of `max` and `nlz` are of 1 cycle, while the latency of `mul` is of 3 cycles; furthermore, we shall assume that at most 2 instructions of type `mul` can be launched simultaneously at every cycle. (How to implement `max`, `nlz`, and `mul` in C is detailed in Appendix A.)

From Sections 4.2 to 4.5, the operand x will be a positive (sub)normal number. In this case, $X_{31} = 0$ and since the result r is a positive normal number, $R_{31} = 0$ as well. Therefore, it suffices to determine the 8 bits $R_{30}, \ldots, R_{23}$, which give the (biased) result exponent $D = \sum_{i=0}^{7} R_{i+23} 2^i$, and the 23 bits $R_{22}, \ldots, R_0$, which define the trailing significand field of the result.

4.1 Handling special operands

For square root the floating-point operand x is considered special when it is ± 0, $+\infty$, less than zero, or NaN. Table 2 thus implies that x is special if and only if $X \in \{0\} \cup [2^{31} - 2^{23}, 2^{32})$, that is,

$$(X - 1) \bmod 2^{32} \geq 2^{31} - 2^{23} - 1. \tag{33}$$

All special operands can thus be detected using (33).

TABLE 3
Square root results for special operands.

Operand x	$+0$	$+\infty$	-0	less than zero	NaN
Result r	$+0$	$+\infty$	-0	qNaN	qNaN

The results required by the IEEE 754-2008 standard for the square root of such operands are listed in Table 3.

Note that there are essentially only two cases to consider: r is either x or qNaN. The first case occurs when $x \in \{+0, +\infty, -0\}$, which, when x is known to be special, is a condition equivalent to

$$X \leq 2^{31} - 2^{23} \quad \text{or} \quad X = 2^{31}. \tag{34}$$

If condition (34) is not satisfied then a quiet NaN is constructed by setting the bits $X_{30}, \ldots, X_{22}$ to 1 while leaving X_{31} and $X_{21}, \ldots, X_0$ unchanged; this can be done by taking the bitwise OR of X and of the constant

$$2^{31} - 2^{22} = (7FC00000)_{16},$$

whose bit string consists of 1 zero followed by 9 ones followed by 22 zeros. Note that the quiet NaN thus produced keeps as much of the information of X as possible, as recommended in [2, §6.2]; in particular, the payload is preserved when quieting an sNaN, and if x is a qNaN then x is returned.

Using the fact that the addition of two 32-bit unsigned integers is done modulo 2^{32} and taking the hexadecimal values of the constants in (33) and (34), we finally get the C code shown in Listing 1 for handling special operands. In this code, notice that the four operations `+`, `<=`, `==`, and `|` on `X` are independent of each other.

Listing 1
Code for handling special operands.

```
if ((X - 1) >= 0x7F7FFFFF) {
 if ((X <= 0x7F800000) || (X == 0x80000000))
    return X;
 else
    return X | 0x7FC00000;          // qNaN
}
else
{
 ... // Code for non-special operands,
     // detailed in Sections 4.2 to 4.5
     // as well as in Listings 2 and 3.
}
```

Remark that (33) and (34) extend immediately to other standard formats like binary$k$ with $k = 16, 64, 128, \ldots$. Thus, if integer arithmetic modulo 2^k is available and if the standard encoding is used for binary$k$ data, special values can be handled in the same way as shown here.

4.2 Computing the biased value of d and parity of e'

By Property 2.1 the result r cannot be subnormal. Therefore, by applying (31) we deduce that the biased value

TABLE 2
Relationship between floating-point datum x and its encoding into integer $X = \sum_{i=0}^{31} X_i 2^i$.

Value or range of integer X	Floating-point datum x	Bit string $X_{31} \dots X_0$
0	$+0$	00000000000000000000000000000000
$(0, 2^{23})$	positive subnormal number	$00000000X_{22} \dots X_0$ with some $X_i = 1$
$[2^{23}, 2^{31} - 2^{23})$	positive normal number	$0\underbrace{X_{30}X_{29}X_{28}X_{27}X_{26}X_{25}X_{24}X_{23}}_{\text{not all ones, not all zeros}}X_{22} \dots X_0$
$2^{31} - 2^{23}$	$+\infty$	01111111100000000000000000000000
$(2^{31} - 2^{23}, 2^{31} - 2^{22})$	sNaN	$0111111110X_{21} \dots X_0$ with some $X_i = 1$
$[2^{31} - 2^{22}, 2^{31})$	qNaN	$0111111111X_{21} \dots X_0$
2^{31}	-0	10000000000000000000000000000000
$(2^{31}, 2^{31} + 2^{23})$	negative subnormal number	$10000000X_{22} \dots X_0$ with some $X_i = 1$
$[2^{31} + 2^{23}, 2^{32} - 2^{23})$	negative normal number	$1\underbrace{X_{30}X_{29}X_{28}X_{27}X_{26}X_{25}X_{24}X_{23}}_{\text{not all ones, not all zeros}}X_{22} \dots X_0$
$2^{32} - 2^{23}$	$-\infty$	11111111100000000000000000000000
$(2^{32} - 2^{23}, 2^{32} - 2^{22})$	sNaN	$1111111110X_{21} \dots X_0$ with some $X_i = 1$
$[2^{32} - 2^{22}, 2^{32})$	qNaN	$1111111111X_{21} \dots X_0$

D of the exponent d of r is always given by

$$D = d + 127.$$

In order to compute D from X, we use first the expression of d in (8) and the relation (31) to obtain

$$D = \lfloor (E - \lambda + [\lambda > 0] + 127)/2 \rfloor. \quad (35)$$

Then, using (32) and the second and third rows of Table 2, we deduce that the number of leading zeros of X is $\lambda + 8$ when $\lambda > 0$, and at most 8 when $\lambda = 0$. Hence

$$\lambda = M - 8, \qquad M = \max(\mathrm{nlz}(X), 8). \quad (36)$$

An immediate consequence of this is that $[\lambda > 0] = [M > 8]$. However, more instruction-level parallelism can be obtained by observing in Table 2 that

$$[\lambda > 0] = [X < 2^{23}] \quad \text{for } x \text{ positive (sub)normal.} \quad (37)$$

The formula (35) for the biased exponent D thus becomes

$$D = \lfloor (E - M + [X < 2^{23}] + 135)/2 \rfloor. \quad (38)$$

A possible C code implementing (38) is as follows:

```
Z = nlz(X);    E = X >> 23;  B = X < 0x800000;
M = max(Z,8);                C = B + 135;
D = (E - M + C) >> 1;
```

Remark that in *rounding-up* mode Property 2.3 requires that the integer D obtained so far be further incremented by 1 when

$$m = 2 - 2^{-23} \text{ and } e \text{ is odd.} \quad (39)$$

(This correction has also been illustrated in Figure 2.) Since (39) implies that x is a normal number, D must be replaced by $D + 1$ if and only if $X \geq 2^{23}$ and the last 24 bits of X are 1 zero followed by 23 ones. An implementation of this update is thus:

```
d1 = 0x00FFFFFF; d2 = 0x007FFFFF;
D = D + ((X >= 0x800000) & ((X & d1) == d2));
```

In fact, this treatment specific to rounding up can be avoided by exploiting the standard encoding as follows. Recall that $n = \circ(\ell)$ is in $[1, 2]$ and has at most 23 fraction bits, and that we want the bit string $0R_{30} \dots R_{23}R_{22} \dots R_0$ of the result r. Instead of concatenating the bit string $R_{30} \dots R_{23}$ of D and the bit string $R_{22} \dots R_0$ of the fraction field of r, one can add to $(D-1) \cdot 2^{23}$ the integer $n \cdot 2^{23}$:

- If $n = (1.n_1 \dots n_{23})_2$ then this addition corresponds to $(D-1) \cdot 2^{23} + 2^{23} + (0.n_1 \dots n_{23})_2 \cdot 2^{23}$ and since no carry propagation occurs, it simply concatenates the bit string of D and the bit string $n_1 \dots n_{23}$.
- If $n = (10.0 \dots 0)_2$ then this addition corresponds to $(D-1) \cdot 2^{23} + 2 \cdot 2^{23} = (D+1) \cdot 2^{23}$. Hence R encodes the normal number $r = (1.0 \dots 0)_2 \cdot 2^{d+1}$.

Hence, we implemented the computation of $D-1$ using the formula below, which directly follows from (38):

$$D - 1 = \lfloor (E - M + [X < 2^{23}] + 133)/2 \rfloor. \quad (40)$$

This corresponds to the computation of variable `Dm1` at line 7 of Listing 2. The only difference with the implementation of D given right after (38) occurs at line 6, where we perform `B + 133` instead of `B + 135`.

Let us now turn to the parity of e' in (5), which will be needed in Sections 4.3 and 4.5. Using (31), (36), and (37) we deduce that e' is even if and only if the last bit of $E - M + [X < 2^{23}]$ is equal to 1. Since the latter expression already appears in (38) or (40), an implementation follows immediately:

```
even = (E - M + B) & 0x1;
```

An alternative code, which uses only logical operators and unsigned integers, consists in taking the XOR of the last bit of $E + [X < 2^{23}]$ and of the last bit of M:

```
even = ((E & 0x1) | B) ^ (M & 0x1);
```

Listing 2
Square root implementation for the *binary32* format, assuming a non-special operand and rounding to nearest.

```
1  uint32_t binary32sqrt(uint32_t X)
2  {
3    uint32_t B, C, Dm1, E, even, M, S, T, Z, P, Q, U, V;
4
5    Z = nlz(X);                                E = X >> 23;            B = X < 0x800000;
6    M = max(Z,8);                              C = B + 133;
7    even = ((E & 0x1) | B) ^ (M & 0x1);        T = (X << 1) << M;      Dm1 = (E - M + C) >> 1;
8    S = 0xB504F334 & (0xBFFFFFFF + even);
9
10   V = biv_poly_eval(S, T); // Bivariate polynomial evaluation: S [1.31], T [0.32], V [2.30]
11
12   U = V & 0xFFFFFFC0;      // Truncation after 24 fraction bits: U [2.24]
13
14   P = mul(U, U);                             Q = ((T >> 1) | 0x80000000) >> (even + 2);
15   if (P >= Q)
16     return (Dm1 << 23) + (U >> 7);
17   else
18     return (Dm1 << 23) + ((U + 0x00000040) >> 7);
19 }
```

4.3 Computing the evaluation point $(\widehat{\sigma}, \tau)$

In precision $k = 32$, rounding $\sqrt{2}$ to nearest gives the Q1.31 number $1 + 889516852 \cdot 2^{-31}$. Thus $\widehat{\sigma}$ in (28) is given by

$$\widehat{\sigma} = S \cdot 2^{-31}, \tag{41}$$

with S the integer in $[0, 2^{32})$ such that $S = 2^{31}$ if e' is even, and $S = 2^{31} + 889516852 = (B504F334)_{16}$ if e' is odd. This integer S will be used in our code to encode $\widehat{\sigma}$, and its bit string has the form

$$S_{31}S_{30}S_{29}\ldots S_0 = \begin{cases} 100\ldots0, & \text{if } e' \text{ is even,} \\ 10*\ldots*, & \text{if } e' \text{ is odd.} \end{cases}$$

Since $S_{31}S_{30} = 10$ in both cases, selecting the right bit string can be done by taking the bitwise AND of the constant $(B504F334)_{16} = (10*\ldots*)_2$ and of

$$2^{31} + 2^{30} - 1 + [e' \text{ is even}] = \begin{cases} 110\ldots0, & \text{if } e' \text{ is even,} \\ 101\ldots1, & \text{if } e' \text{ is odd.} \end{cases}$$

Therefore, since $2^{31} + 2^{30} - 1 = (BFFFFFFF)_{16}$ and given the value of the integer `even` (see Subsection 4.2), computing S can be done as shown at line 8 in Listing 2.

The number τ in (19) satisfies $\tau = (0.m_{\lambda+1}\ldots m_{23})_2$. Therefore, it can be viewed as a Q0.32 number

$$\tau = T \cdot 2^{-32}, \tag{42}$$

where $T = \sum_{i=0}^{31} T_i 2^i$ is the integer in $[0, 2^{32})$ such that

$$T_{31}\ldots T_0 = m_{\lambda+1}\ldots m_{23}\underbrace{0\cdots0}_{\lambda+9}. \tag{43}$$

By (32) and (36), we see that T can be computed by shifting X left $M + 1$ positions. Since M is not immediately available, more ILP can be exposed by implementing this shift as in line 7 of Listing 2. Also, for the shift by $M = \max(\mathrm{nlz}(X), 8)$ to be well defined here, the C standard [19] requires that $0 \le M < 32$. One may check that this is indeed the case: since by assumption x is nonzero, the number of leading zeros of X is less than 32 and thus $8 \le M < 32$.

4.4 Computing the approximate polynomial value v

Listing 3 below shows an implementation of the evaluation scheme (30) using 32-bit unsigned integers and the identities in (25). Multiplications by coefficients a power of two (like A_1, A_2, A_8) are implemented as simple shifts.

Assuming a latency of 1 for additions, subtractions, and shifts, a latency of 3 for (pipelined) multiplications, and that at most 2 multiplications can be started simultaneously, this code can be scheduled in at most 13 cycles, as shown in Table 4. There the dashes '—' indicate that an instruction requires an additional slot because it uses an extended immediate; see Section 5.1.

The numerical quality of the code in Listing 3 has been verified using the *Gappa* software (see http://gappa.gforge.inria.fr/ and [20], [21]; see also [22, §4] for some guidelines on how to translate a C code into *Gappa* syntax). With this software, we first checked that all variables $r_0, \ldots, r_{21}$ are indeed integers in the range $[0, 2^{32})$. Then we used *Gappa* to compute a certified upper bound on the final rounding error; the bound produced is less than $2^{-27.93}$ and thus less than the sufficient bound in (29). The *Gappa* script we wrote to perform this certification step is contained in the file `binary32sqrt.gappa` available at http://prunel.ccsd.cnrs.fr/ensl-00335792.

4.5 Implementing the rounding tests

There the only non-trivial part is to evaluate the expressions $u \ge \ell$ (used when rounding to nearest and rounding up; see §3.2.1 and §3.2.3), and $u > \ell$ (used when rounding down; see § 3.2.2). It turns out that such comparisons can be implemented *exactly* by introducing three integers P, Q, Q' which we shall define below by considering u^2 and ℓ^2 instead of u and ℓ.

Listing 3
Bivariate polynomial evaluation code.

```
// A0 = 0x80000000; A1 = 0x40000000;
// A2 = 0x10000000; A3 = 0x07fe93e4;
// A4 = 0x04eef694; A5 = 0x032d6643;
// A6 = 0x01c6cebd; A7 = 0x00aebe7d;
// A8 = 0x00200000;

static inline
uint32_t biv_poly_eval(uint32_t S, uint32_t T)
{
  uint32_t r0  = T >> 2;
  uint32_t r1  = 0x80000000 + r0;
  uint32_t r2  = mul(S, r1);
  uint32_t r3  = 0x00000020 + r2;
  uint32_t r4  = mul(T, T);
  uint32_t r5  = mul(S, r4);
  uint32_t r6  = r5 >> 4;
  uint32_t r7  = r3 - r6;
  uint32_t r8  = mul(T, 0x07fe93e4);
  uint32_t r9  = mul(r5, r8);
  uint32_t r10 = r7 + r9;
  uint32_t r11 = mul(r4, r5);
  uint32_t r12 = mul(T, 0x032d6643);
  uint32_t r13 = 0x04eef694 - r12;
  uint32_t r14 = mul(T, 0x00aebe7d);
  uint32_t r15 = 0x01c6cebd - r14;
  uint32_t r16 = r4 >> 11;
  uint32_t r17 = r15 + r16;
  uint32_t r18 = mul(r4, r17);
  uint32_t r19 = r13 + r18;
  uint32_t r20 = mul(r11, r19);
  uint32_t r21 = r10 - r20;
  return r21;
}
```

TABLE 4
Feasible scheduling on ST231.

	Issue 1	Issue 2	Issue 3	Issue 4
Cycle 0	r_4	r_0	r_{14}	—
Cycle 1	r_1	—	r_8	—
Cycle 2	r_{12}	—		
Cycle 3	r_5	r_{15}	—	r_{16}
Cycle 4	r_2	r_{17}		
Cycle 5	r_{13}	—	r_{18}	
Cycle 6	r_9	r_6	r_{11}	
Cycle 7	r_3			
Cycle 8	r_7	r_{19}		
Cycle 9	r_{20}	r_{10}		
Cycle 10				
Cycle 11				
Cycle 12	r_{21}			

Truncating $v = V \cdot 2^{-30}$ after 24 fraction bits yields

$$u = U \cdot 2^{-30}, \tag{44}$$

with U the integer in $[0, 2^{32})$ whose bit string is

$$[\,0\,1\,v_1\,\cdots\,v_{24}\,0\,0\,0\,0\,0\,0\,].$$

Let P be the integer in $[0, 2^{32})$ such that

$$P = \texttt{mul}(U, U).$$

It follows from (44) and the definition of `mul` that

$$u^2 - 2^{-28} < P \cdot 2^{-28} \leq u^2. \tag{45}$$

With σ either 1 or $\sqrt{2}$, we see that $\ell^2 = \sigma^2 m'$ is either

$$m' = (1.m_{\lambda+1}m_{\lambda+2}\ldots m_{23})_2$$

or

$$2m' = (1m_{\lambda+1}.m_{\lambda+2}\ldots m_{23})_2,$$

and can be represented exactly with $24 - \lambda$ bits. Since $24 - \lambda \leq 32$, several encodings into a 32-bit unsigned integer are possible. Because of (45) and the need to compare ℓ^2 with u^2, a natural choice is to encode ℓ^2 into the integer $Q \in [0, 2^{32})$ such that

$$\ell^2 = Q \cdot 2^{-28}. \tag{46}$$

An implementation of the computation of Q using the integer T defined in (42-43) and the parity of e' can be found at line 14 of Listing 2.

4.5.1 Rounding to nearest and rounding up

Once the values of P and Q are available, the condition $u \geq \ell$ used when $\circ \in \{\mathrm{RN}, \mathrm{RU}\}$ can be evaluated thanks to the following characterization:

Property 4.1: The inequality $u \geq \ell$ holds if and only if the condition `P >= Q` is true.

Proof: Since u and ℓ are non-negative, $u \geq \ell$ is equivalent to $u^2 \geq \ell^2$. If $u^2 \geq \ell^2$ then, by (46) and the left inequality in (45), we deduce that $P + 1 > Q$, which is equivalent to $P \geq Q$ for P and Q integers. Conversely, if $P \geq Q$ then multiplying both sides by 2^{-28} gives $P \cdot 2^{-28} \geq \ell^2$ and, using the right inequality in (45), $u^2 \geq \ell^2$. To sum up, $u \geq \ell$ if and only if $P \geq Q$, that is, if and only if the C condition `P >= Q` is true. □

Together with the algorithm of Section 3.2.1, this property accounts for the implementation of rounding to nearest at lines 14 to 18 in Listing 2.

Since rounding up depends on the condition $u \geq \ell$ as well (see Section 3.2.3), this rounding mode can be implemented easily: simply replace lines 15 to 18 in Listing 2 with the following code fragment:

```
15 if(P >= Q)
16  return (Dm1 << 23) + ((U + 0x00000040) >> 7);
17 else
18  return (Dm1 << 23) + ((U + 0x00000080) >> 7);
```

4.5.2 Rounding down

According to the algorithm in Section 3.2.2 rounding down does not rely on the condition $u \geq \ell$ but on the condition $u > \ell$ instead.

In order to implement the condition $u > \ell$, let $Q' \in \{0, 1\}$ be such that $Q' = 1$ if and only if equality $P \cdot 2^{-28} = u^2$ occurs in (45), that is, if and only if $P = U^2 \cdot 2^{-32}$. The latter equality means that U has at least 16 trailing zeros. Since the bit string of U is $[\,0\,1\,v_1\,\cdots\,v_{24}\,0\,0\,0\,0\,0\,0\,]$, this is equivalent to deciding whether $v_{15} = v_{16} = \cdots = v_{24} = 0$ or not. Hence the code below for computing Q':

```
Qprime = (V & 0x0000FFC0) == 0x0;
```

Property 4.2: The inequality $u > \ell$ holds if and only if the condition `P >= Q + Qprime` is true.

Proof: Since u and ℓ are non-negative, $u > \ell$ is equivalent to $u^2 > \ell^2$. We consider the two cases $Q' = 1$ and $Q' = 0$ separately. Assume first that $Q' = 1$. Then, using the equality in (45) together with (46), we see that $u > \ell$ is equivalent to $P > Q$, that is, since P and Q are integers, to $P \geq Q + 1 = Q + Q'$. Assume now that $Q' = 0$. In this case $P \cdot 2^{-28} < u^2$ and thus $P \geq Q$ implies $u^2 > Q \cdot 2^{-28} = \ell^2$; conversely, if $u^2 > Q \cdot 2^{-28}$ then, using the left inequality in (45), we find $P + 1 > Q$ and thus $P \geq Q$. Therefore, $u > \ell$ if and only if $P \geq Q + Q'$. Now, recalling that $\ell \in [1, 2)$, we deduce from (46) that $Q < 2^{30}$. Hence $Q + Q'$ always fits in a 32-bit unsigned integer. Consequently, the mathematical condition $P \geq Q + Q'$ is equivalent to the C condition `P >= Q + Qprime`. □

Together with the algorithm of Section 3.2.2, the above property gives the following implementation of rounding down:

```
15 Qprime = (V & 0x0000FFC0) == 0x0;
16 if(P >= Q + Qprime)
17  return (Dm1 << 23) + ((U - 0x00000040) >> 7);
18 else
19  return (Dm1 << 23) + (U >> 7);
```

5 Experiments with the ST231 core

Combining the codes of the previous section leads immediately to a standard C implementation of `sqrt` for the *binary32* format, with subnormal support and correct rounding for each rounding-direction attribute RN, RU, RD.

For validation purposes, each of these three versions has been compiled with Gcc under Linux (using the software given in Appendix A for emulating `max`, `nlz`, `mul`). This allowed for an exhaustive comparison, within a few minutes, against the square root functions of GNU C (glibc)[2] and GNU MPFR.[3]

We also compiled these three versions with the Open 64-based ST200 family VLIW compiler from STMicroelectronics, in -O3 and for the ST231 core. After a review of the main features of the ST231 in Section 5.1, the performance results we have obtained in this context are presented and analysed in Section 5.2.

5.1 Some features of the ST231

The ST200 family of VLIW microprocessors originates from the joint design of the Lx by HP Labs and STMicroelectronics [23]. The ST231 is the most recently designed core of this family, and is widely used in STMicroelectronics SOCs for multimedia acceleration.

2. http://www.gnu.org/software/libc/
3. http://www.mpfr.org/mpfr-current/

In this processor, that executes up to four integer instructions per cycle, all arithmetic instructions operate on the 64 32-bit register file and on the 8 one-bit *branch* register file.

Resource constraints must be observed to form proper instruction *bundles* containing 1 to 4 instructions: only one control instruction, one memory instruction, and up to two $32 \times 32 \rightarrow 32$ multiplications of type `mul` are enabled. Other instructions can be freely used, but are limited to integer only arithmetic, without division.

Another specificity of this architecture is that any immediate form of an instruction is by default encoded to use small immediates (9-bit signed), but can be extended to use extended immediates (32-bit), at the cost of one instruction per immediate extension. For instance, up to two multiplications `mul` each using a 32-bit immediate can be encoded in one bundle. This makes the usage of long immediate constants such as polynomial coefficients very efficient from a memory system standpoint. On the contrary, the absence of a sophisticated data cache model (such as L2 cache) implies a quite important cost of accessing a data table (up to 130 cycles in the case of a data cache miss).

To enable the reduction of conditional branches, the architecture provides partial predication support in the form of conditional selection instructions. In the assembly line below, `$q`, `$r`, `$s` are 32-bit integer registers, `$b` is a one-bit branch register that can be defined through comparison or logical instructions:

```
slct $s = $b, $q, $r
```

This fragment of assembly code writes `$q` in `$s` if `$b` is true, `$r` otherwise. An efficient *if-conversion* algorithm based on the ψ-SSA representation is used in the Open64 compiler to generate partially-predicated code based on the `slct` instruction [24].

The retargeting of the Open64 compiler technology to the ST200 family is able to generate efficient, dense, branch-free code for all the codes described in Section 4, requiring only the usage of one specific intrinsic to select the `nlz` instruction (number of leading zeros).

5.2 Performance results for square root on ST231

The performances obtained when compiling for ST231 with the ST200 compiler are summarized in Table 5. Versions of our codes that do not support subnormals have been implemented too, whose performances are within square brackets.

TABLE 5
Performances of our approach on ST231.

	Subnormal numbers [not] supported		
◦	Latency L	Number N of instructions	IPC = N/L
RN	23 [21]	62 [56]	2.70 [2.67]
RU	23 [21]	63 [57]	2.74 [2.71]
RD	23 [21]	65 [59]	2.83 [2.81]

Here are some observations:

- Thanks to *if-conversion*, the generated assembly codes are fully if-converted, straight-line programs. Thus, all latencies (numbers of cycles) and instruction numbers are independent of the input values.
- Depending on $\circ$, the value N varies as expected: compared to RN, Section 4.5.1 suggests one more addition for RU, while Section 4.5.2 suggests 3 more instructions for RD (get `Qprime` and add it to `Q`).
- On the contrary, the value of L turns out to be independent of $\circ$. In fact, with subnormal support, the value 23 is exactly the latency that can be expected from the codes of Section 4 when assuming *unbounded* parallelism. Indeed, the critical path consists of the following four sub-tasks:
  - ⋆ compute the value `T` in 3 cycles;
  - ⋆ compute the value `V` by bivariate polynomial evaluation in 13 cycles;
  - ⋆ round correctly (truncation, squaring, comparison, and selection) in $1+3+1+1=6$ cycles;
  - ⋆ select the final result in 1 cycle.

  In other words, our implementations with subnormal support are scheduled *optimally* by the ST200 compiler. The same occurs for our versions without subnormals: those were obtained by replacing

```
Z = nlz(X);
M = max(Z,8);          with   T = X << 9;
T = (X << 1) << M;
```

  The cost of `T` drops from 3 cycles to 1 cycle, and one may check that the critical path is defined by the same sub-tasks as before. Hence a new theoretical latency of 21 cycles, which again is achieved in practice. Finally, note that the overhead due to subnormal support is only of 2 cycles.
- In all cases, the number of instructions per cycle (IPC) is larger than 2.6. A more precise description of bundle occupancy is given in Figure 4, assuming $\circ$ = RN and subnormal support. Among the 23 bundles used, 4 contain four instructions and 12 contain three instructions. The instructions used for handling *special* operands are in *light grey*, while those corresponding to *non-special* operands are in *black*; instructions in *dark grey* are shared or used for selection. Note that, from the point of view of latency, handling special operands comes for free up to the last cycle, used for selecting the final result.

Table 6 shows the influence of the evaluation scheme used for $P(s,t) = 2^{-25} + s \cdot a(t)$ on the performances of the complete square root code. Instead of evaluating $P(s,t)$ as in (30) and Listing 3, we evaluate $a(t)$ by three methods and then multiply by s and add 2^{-25}. "Best univariate" here is one giving the lowest latency (12 cycles) we have found assuming unbounded parallelism. Comparing with the first line of Table 5, we remark that:

- Despite high-ILP exposure, our bivariate evaluation scheme increases the total number of instructions only by 4 (less than 7%) compared to Horner's rule.
- Our scheme yields a complete code that is about twice faster than when using Horner's rule.
- We use 4 cycles less than when evaluating $a(t)$ as fast as possible and then performing in sequence the multiplication by s and the addition by 2^{-25} ("Best univariate"). This shows the advantage of evaluating the bivariate polynomial $P(s,t)$ directly, allowing for s to be distributed within $a(t) = \sum_i a_i t^i$.

Fig. 4. Typical bundle occupancy on ST231.

TABLE 6
Performances with other evaluation schemes on ST231.

	$\circ = RN$ and with [without] subnormals		
Evaluation scheme	L	N	IPC = N/L
Horner's rule	44 [42]	58 [52]	1.32 [1.24]
Estrin's method	29 [27]	60 [54]	2.07 [2.00]
Best univariate	27 [24]	62 [56]	2.30 [2.33]

Table 7 gives the latencies obtained in [7] (for rounding to nearest and without subnormals) by implementing various other methods on ST231. We denote by R and nR the classical restoring and non-restoring algorithms; N2, G2, G1 refer to some variants of Newton-Raphson/Goldschmidt methods, based on low-degree polynomial evaluation (for the initial approximation) followed by 1 or 2 iterations; P5 and P6 are methods based exclusively on piecewise, univariate polynomial approximants (three of degree 5 or two of degree 6).

Our version in 21 cycles is about 7 times faster than the restoring method and twice faster than Goldschmidt's method G2. It is also 1.43 times faster than P6, which was the previously fastest known implementation of

TABLE 7
Latencies obtained with other square root methods on ST231, assuming $\circ = \text{RN}$ and *no* subnormal support.

Method	R	nR	N2	G2	G1	P5	P6
Latency L	148	133	45	42	36	33	30

correctly-rounded square root on ST231.

6 Conclusion

After detailing some properties of the square root function in binary floating-point arithmetic, we proposed, for its computation with correct rounding, a high-ILP approach based on bivariate polynomial evaluation. To demonstrate the effectiveness of this approach, we further provided and analysed complete C implementations, one for each rounding-direction attribute. Some experimental results finally showed that these codes yield optimal schedules and low latencies on a VLIW integer processor like the ST231. Also, the latency overhead for subnormal support is only 2 cycles.

Our approach can be extended in several ways. First, one may check that a faithfully-rounded square root follows from replacing lines 12 to 18 of Listing 2 with

```
return (Dm1 << 23) + (V >> 7);
```

The latency then is 19 cycles with subnormals and 17 cycles without subnormals, leading to a speed-up by a factor of more than 1.2 in both cases.

Second, our approach is not restricted to square rooting and can be adapted to other operators: it has already been employed[4] on ST231 for accelerating division [25], reciprocal square root [26], and more generally $x^{1/n}$ for $|n|$ a "small" positive integer [8]. In fact, our approach is restricted to neither algebraic functions nor the bivariate case: it could in principle be used as soon as argument reduction yields an identity of the form $\ell = F(s, t, \ldots)$, for some multivariate function F. But it remains to determine which functions other than the ones above could benefit significantly from our approach.

Third, although we have focused in this paper on implementations for the *binary32* format (single precision) and 32-bit architectures, similar codes could be derived for the *binary64* format (double precision) and 64-bit architectures. The only major modification would be a new polynomial approximant together with a new bivariate evaluation scheme. However, for application reasons, we are currently mostly interested in supporting double precision on ST231. How to best use the underlying 32-bit integer arithmetic available on this chip in order to emulate efficiently a *binary64* square root seems to be nontrivial. It thus remains to investigate whether the approach we have introduced here can be adapted to that context.

4. The first draft of this article is [9] and dates back to October 2008.

Appendix A
Software implementation of the `max`, `nlz`, and `mul` operators

Maximum of two unsigned integers

```
static inline uint32_t max(uint32_t A,
                           uint32_t B)
{ return A > B ? A : B; }
```

Number of leading zeros of a nonzero unsigned integer

```
static inline uint32_t nlz(uint32_t X)
{ // Input X is assumed to be nonzero.
 uint32_t Z = 0;
 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;
}
```

Higher half of a 32-bit integer product

```
static inline uint32_t mul(uint32_t A,
                           uint32_t B)
{
  uint64_t t0 = A;
  uint64_t t1 = B;
  uint64_t t2 = (t0 * t1) >> 32;
  return t2;
}
```

Acknowledgments

This work was supported by "Pôle de compétitivité mondial" Minalogic and by the ANR project EVA-Flo. Thanks also to Nicolas Jourdan for several interesting discussions and for suggesting the use of Equation (33).

References

[1] American National Standards Institute and Institute of Electrical and Electronic Engineers, "IEEE standard for binary floating-point arithmetic," *ANSI/IEEE Standard, Std 754-1985*, New York, 1985.

[2] "IEEE standard for floating-point arithmetic," IEEE Std. 754-2008, pp.1-58, Aug. 29 2008.

[3] P. Montuschi and P. M. Mezzalama, "Survey of square rooting algorithms," *Computers and Digital Techniques, IEE Proceedings-*, vol. 137, no. 1, pp. 31–40, 1990.

[4] P. Markstein, *IA-64 and Elementary Functions: Speed and Precision*, ser. Hewlett-Packard Professional Books. Prentice Hall, 2000.

[5] M. Cornea, J. Harrison, and P. T. P. Tang, *Scientific Computing on Itanium®-based Systems*. Intel Press, Hillsboro, OR, 2002.

[6] M. D. Ercegovac and T. Lang, *Digital Arithmetic*. Morgan Kaufmann, 2004.

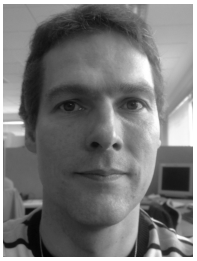
[7] C.-P. Jeannerod, H. Knochel, C. Monat, and G. Revy, "Faster floating-point square root for integer processors," in *IEEE Symposium on Industrial Embedded Systems (SIES'07)*, 2007.

[8] G. Revy, "Implementation of binary floating-point arithmetic on embedded integer processors - polynomial evaluation-based algorithms and certified code generation," Ph.D. dissertation, Université de Lyon - École Normale Supérieure de Lyon, December 2009.

[9] C.-P. Jeannerod, H. Knochel, C. Monat, and G. Revy, "Computing floating-point square roots via bivariate polynomial evaluation," LIP, Tech. Rep. RR2008-38, Oct. 2008. [Online]. Available: http://prunel.ccsd.cnrs.fr/ensl-00335792

[10] R. C. Agarwal, F. G. Gustavson, and M. S. Schmookler, "Series approximation methods for divide and square root in the Power3™ processor," in *Proceedings of the 14th IEEE Symposium on Computer Arithmetic*, I. Koren and P. Kornerup, Eds., Adelaide, Australia, 1999, pp. 116–123.
[11] J.-A. Piñeiro and J. D. Bruguera, "High-speed double-precision computation of reciprocal, division, square root and inverse square root," *IEEE Trans. Computers*, vol. 51, no. 12, pp. 1377–1388, 2002.
[12] S.-K. Raina, "FLIP: a floating-point library for integer processors," Ph.D. dissertation, ÉNS Lyon, France, 2006. [Online]. Available: http://www.ens-lyon.fr/LIP/Pub/Rapports/PhD/PhD2006/PhD2006-02.pdf
[13] J.-M. Muller, *Elementary functions: algorithms and implementation*, 2nd ed. Birkhäuser, 2006.
[14] C. Q. Lauter, "Arrondi correct de fonctions mathématiques - fonctions univariées et bivariées, certification et automatisation," Ph.D. dissertation, ÉNS Lyon, France, 2008.
[15] S. Chevillard, "Évaluation efficace de fonctions numériques - outils et exemples," Ph.D. dissertation, ÉNS Lyon, France, 2009.
[16] S. Chevillard and C. Lauter, "A certified infinite norm for the implementation of elementary functions," in *Proceedings of the 7th IEEE International Conference on Quality Software (QSIC'07)*, A. Mathur, W. E. Wong, and M. F. Lau, Eds. Portland, OR, USA: IEEE Computer Society, 2007, pp. 153–160.
[17] S. Chevillard, M. Joldes, and C. Lauter, "Certified and fast computation of supremum norms of approximation errors," in *Proceedings of the 19th IEEE Symposium on Computer Arithmetic (ARITH-19)*, Portland, OR, Jun. 2009.
[18] J. Harrison, T. Kubaska, S. Story, and P. Tang, "The computation of transcendental functions on the IA-64 architecture," *Intel Technology Journal*, vol. 1999-Q4, pp. 1–7, 1999.
[19] International Organization for Standardization, *Programming Languages – C*. Geneva, Switzerland: ISO/IEC Standard 9899:1999, Dec. 1999.
[20] G. Melquiond, "De l'arithmétique d'intervalles à la certification de programmes," Ph.D. dissertation, École Normale Supérieure de Lyon, Lyon, France, 2006. [Online]. Available: http://www.msr-inria.inria.fr/~gmelquio/doc/06-these.pdf
[21] M. Daumas and G. Melquiond, "Certification of bounds on expressions involving rounded operators," *Transactions on Mathematical Software*, vol. 37, no. 1, 2009.
[22] F. de Dinechin, C. Lauter, and G. Melquiond, "Assisted verification of elementary functions using Gappa," in *Proceedings of the 2006 ACM Symposium on Applied Computing*, Dijon, France, 2006, pp. 1318–1322. [Online]. Available: http://www.msr-inria.inria.fr/~gmelquio/doc/06-mcms-article.pdf
[23] J. A. Fisher, P. Faraboschi, and C. Young, *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*. Morgan Kaufmann, 2005.
[24] 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.
[25] 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, Jun. 2009.
[26] C.-P. Jeannerod and G. Revy, "Optimizing correctly-rounded reciprocal square roots for embedded VLIW cores," in *Proceedings of the 43rd Asilomar Conference on Signals, Systems and Computers*, Pacific Grove, CA, Nov. 2009.

Claude-Pierre Jeannerod received the PhD degree in applied mathematics from Institut National Polytechnique de Grenoble in 2000. After being a postdoctoral fellow in the Symbolic Computation Group at the University of Waterloo, he is now a researcher at INRIA Grenoble - Rhône-Alpes and a member of the LIP laboratory (LIP is a joint laboratory of CNRS, École Normale Supérieure de Lyon, INRIA and Université Claude Bernard Lyon 1). His research interests include computer algebra, linear algebra, and floating-point arithmetic. He is a member of the ACM and the IEEE.



Hervé Knochel received the MS degree in signal processing from the École Nationale Supérieure des Ingénieurs Électriciens de Grenoble (ENSIEG/INPG) in 1994, and the PhD degree in numerical analysis from the Université Joseph Fourier de Grenoble (UJF Grenoble 1) in 1998. He joined STMicroelectronics, Grenoble, in 2000, and is now a Senior Compiler Software Engineer in the Compilation Expertise Center. His main interests are compiler arithmetic optimization, with a special focus on floating-point arithmetic.

Christophe Monat graduated from École Nationale Supérieure de Techniques Avancées (ENSTA ParisTech) in 1989. He worked for EADS developing flight control software, and for Thales Communications designing cryptographic and signal processing intensive electronic warfare systems. He has been with STMicroelectronics, Grenoble, since 1996. He is currently a Principal Engineer in the Compilation Expertise Center. He specializes in highly optimizing compilers and floating-point arithmetic. He is a member of the ACM and the IEEE.

Guillaume Revy received the MS degree in Computer Science from the École Normale Supérieure de Lyon in 2006 and the PhD degree in Computer Science from the Université de Lyon - École Normale Supérieure de Lyon in 2009. He is now postdoctoral fellow in the ParLab (Parallel Computing Laboratory) at the University of California at Berkeley. His research interests include floating-point arithmetic, automatic generation and certification of floating-point programs, and automatic debugging of applications using floating-point arithmetic.