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 integer arithmetic only, is almost complete in the sense that it supports special operands, subnormal numbers, and all rounding modes, but not exception handling (that is, status flags are not set). Finally we have carried out experiments with this implementation using the ST200 VLIW compiler from STMicroelectronics. The results obtained demonstrate the practical interest of our approach in that context: for all rounding modes, the generated assembly code is optimally scheduled and has indeed low latency (23 cycles).

Index Terms—Binary floating-point arithmetic, square root, correct rounding, 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 binary 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 modes. 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) 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

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

Since m is nonzero, the number λ of its leading zeros varies in $\{0, 1, \ldots, p-1\}$. If $x \in (-2^{e_{\min}}, 2^{e_{\min}})$ then x is *subnormal*, else x 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

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

This assumption is verified for all the binary formats defined in [2].

The IEEE 754-2008 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 usual four rounding modes: 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 modes is enough:

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

As we will recall 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

- *C.-P. Jeannerod and G. Revy are with INRIA Rhône-Alpes and Université de Lyon (Lyon, France).*
- *H. Knochel and C. Monat are with STMicroelectronics' Compilation Expertise Center (Grenoble, France).*

Manuscript last updated October 29, 2008.

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 the evaluation of a suitable bivariate polynomial. Since polynomial evaluation is intrinsically parallel, this approach allows for very high ILP exposure. Thus, in some contexts such as VLIW processors, a significant reduction of latency can be expected.

The paper is organized as follows. Section 2.1 reviews 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 mode, 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 detailed C implementation of this approach is given in Section 4, for the binary32 format and assuming that 32-bit 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 each of the rounding modes.

For this implementation, all 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. Thus, Section 5 is devoted to some experiments carried out with this target and the ST200 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 usual scheme for moving from (3) to (4) follows from three basic properties which we recall now.

2.1 Floating-point square root properties

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

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

Proof: Using (1), (2) and (3) gives $2^{1-p+e_{\min}} \le x < 2^{1+e_{\max}}$. The square root function being monotonically increasing, we obtain $2^{(1-p+e_{\min})/2} \le \sqrt{x} < 2^{(1+e_{\max})/2}$. The assumption $p \le 1 - e_{\min}$ gives the lower bound $2^{e_{\min}} \le \sqrt{x}$. The upper bound on $\sqrt{x}$ follows from the fact that $p \ge 1$ implies $1 \le 2 - 2^{1-p}$ and from the fact that $e_{\max} \ge 1$ implies $(1+e_{\max})/2 \le e_{\max}$. □

This first property 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 $r = 2^{d+1}\cdot 1$ instead of the unnormalized representation $2^{d}\cdot 2$ given by Property 2.2. The next property 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 $m = 2 - 2^{1-p}$.

Proof: Assume first that $\circ(\ell) = 2$. Then necessarily $\ell > 2 - 2^{1-p}$. Recalling that $\ell = \sigma\sqrt{m'}$ and using the fact that $1 \le \sqrt{m'} \le m' \le 2 - 2^{1-p}$, we get $\sigma > 1$ and thus $\sigma = \sqrt{2}$. Therefore, $\circ(\ell) = 2$ implies that e is odd and that $\sqrt{m'} > \sqrt{2}\cdot(1-2^{-p})$. It follows that $m' > 2 - 2\cdot 2^{1-p}$, which is the floating-point predecessor of $2-2^{1-p}$. Then m' must be equal to $2-2^{1-p}$, and thus $\lambda = 0$ and $m = m'$. With $\sigma = \sqrt{2}$ and $m' = m = 2-2^{1-p}$, one may check that $\ell < 2 - 2^{-p}$. Since $2 - 2^{-p}$ is the midpoint between the two consecutive floating-point numbers $2 - 2^{1-p}$ and 2, we have that $\circ(\ell) = 2$ implies $\circ = \mathrm{RU}$. Conversely, if e is odd and $m = 2 - 2^{1-p}$ then $\ell > 2 - 2^{1-p}$. Taking $\circ = \mathrm{RU}$ further gives $\circ(\ell) = 2$. □

2.2 High level description of square root algorithms

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

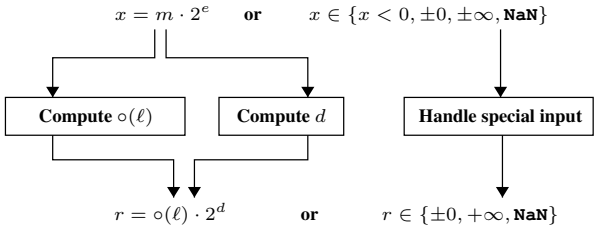

Fig. 1. Square root algorithm for $\circ \in \{\mathsf{RN}, \mathsf{RD}\}$.

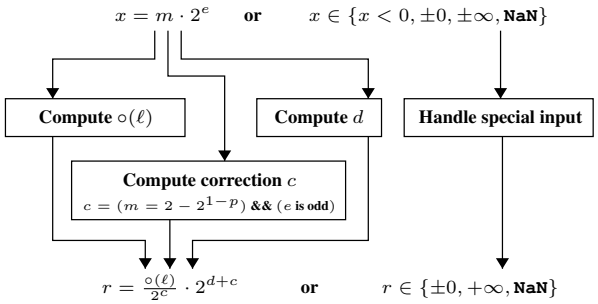

Fig. 2. Square root algorithm for $\circ = \mathsf{RU}$.

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, for each rounding mode, the meaning of $\circ(\ell)$. Since $\ell \in [1, 2)$, we have the following, where n denotes a *normal* floating-point number:

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

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

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

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

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

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

Both inequalities in (9) are strict because of the fact that a square root cannot be the exact midpoint between two consecutive floating-point numbers (see for example [4, Theorem 9.4], [5, p. 242], or [6, p. 463]). Note also that both (9) and (10) imply $n \in [1, 2)$, whereas (11) implies $n \in [1, 2]$.

Given p, $\circ$, m', and σ, there are many ways of producing 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 that approximates ℓ from above with absolute error less than 2^{-p}, that is,

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

- deduce w by truncating v after p fraction bits:

$$0 \leq v - w < 2^{-p}; \tag{13}$$

- obtain n by adding, if necessary, a small correction to w 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 k bits, the number of bits of the encoding of the underlying binary format (for example, $p = 24$ and $k = 32$; see Section 4). On the other hand, using $\ell \leq \sqrt{2} \cdot \sqrt{2 - 2^{1-p}}$ one may check that if v satisfies (12) then necessarily

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

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 gives

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

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

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

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 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 either *iterative refinement* (Newton-Raphson or Goldschmidt method [6, § 7.3]), or *polynomial evaluation* [7], or

a combination of both [8], [9]. 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; this makes the approach simpler, more flexible and eventually faster.

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

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

First, using (6) and defining

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

the real number $\ell + 2^{-p-1}$ is seen as the value at $(s,t) = (\sigma,\tau)$ of the 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, respectively, σ and τ, 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, in particular, 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

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

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). 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,...). In the two paragraphs below we illustrate how to choose a and $\mathcal{P}$ in the case where $(k,p) = (32, 24)$.

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 ring of univariate real polynomials of degree $d \in \mathbb{N}$ and recall (for example from [10, §3.2]) that the *minimax polynomial of degree* d with respect to $\sqrt{1+t}$ 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),

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

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 (computed using e.g. Remez' algorithm (available for example in *Sollya*[1]; see also [10, §3.5] and [11])) in Table 1 indicates that a should be of degree at least 8.

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

1. http://sollya.gforge.inria.fr/

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. Continuing the previous example where $k = 32$, it turns out that, in this case, truncating the Remez coefficients is enough and gives $a(Y) = \sum_{i=0}^{8} a_i Y^i$, where

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

The values found for each A_i are displayed in Listing 2. Each of those integers can be stored using only 32 bits. In addition, notice that

$$A_8 \le \cdots \le A_2 \le A_1 \le A_0. \tag{26}$$

A certified infinite-norm computation (implemented for example in *Sollya*; see also [12]) 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

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 throuput, 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 the input (s,t) and the output $P(s,t)$ both have a fairly small range.

Lemma 2: $s \in \{1, \sqrt{2}\}$, $t \in [0, 1^-]$, and $P(s,t) \in (1,2)$.

Proof: The ranges for s and t are clearly $\mathcal{S}$ and $\mathcal{T}$. Let us now find the range of $P(s,t)$. It follows from the definition of $\alpha(a)$ and the bound in (23) that

$$\sqrt{1+t} - 2^{-p-3/2} < a(t) < \sqrt{1+t} + 2^{-p-3/2}. \tag{27}$$

Thus, using $t \ge 0$ gives $a(t) > 1 - 2^{-p-3/2}$. Using $t \le 1^- = 1 - 2^{1-p}$ gives $a(t) < \sqrt{2 - 2^{1-p}} + 2^{-p-3/2} \le \sqrt{2} - 2^{-p-3/2}$. It the follows from $1 \le s \le \sqrt{2}$ and

$$1 - 2^{-p-3/2} < a(t) < \sqrt{2} - 2^{-p-3/2}$$

that $P(s,t) = 2^{-p-1} + s \cdot a(t)$ lies in the range $(1,2)$. □

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

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

The lemma below gives an upper bound on the loss of accuracy that occurs when rounding the input (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

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

Proof: Let $(s,t) \in \widehat{\mathcal{S}} \times \mathcal{T}$. Writing $\widehat{s} = \mathrm{RN}_k(s)$, we deduce from (21) that

$$|P(s,t) - P(\widehat{s},t)| = |s - \widehat{s}| \cdot |a(t)|.$$

On the one hand, by definition of rounding-to-nearest in precision k, we have $|s - \widehat{s}| \le 2^{-k}$. On the other hand, it follows from (27) that $0 < a(t) < \sqrt{2}$. Therefore, $|P(s,t) - P(\widehat{s},t)| < 2^{-k+1/2}$ and, by applying the triangle inequality to the definition of $\rho(\mathcal{P})$,

$$\rho(\mathcal{P}) < 2^{-k+1/2} + \max_{(s,t) \in \mathcal{S} \times \mathcal{T}} |P(\widehat{s},t) - \mathcal{P}(s,t)|.$$

Now, $\mathcal{P}(s,t) = \mathcal{P}(\widehat{s},t)$ and the conclusion follows. □

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

$$\widehat{\rho}(\mathcal{P}) \le 2^{-26.840\cdots} - 2^{-31.5} = 2^{-26.899\cdots}. \tag{29}$$

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 thus parenthesize the expression $P(s,t)$ in order to expose as much ILP as we can. An example of such parenthesization 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 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).

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:

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

3.2 Correction to ensure correct rounding

For each rounding mode $\circ$ we will now obtain $n = \circ(\ell)$ by correcting w in (16) and (17). How to correct w depends on whether w is above or below ℓ. Thus the rounding algorithms below rely on either $w \geq \ell$ or $w > \ell$, 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 $w \geq \ell$ then
  $n :=$ truncate w after $p-1$ fraction bits
else
  $n :=$ truncate $w + 2^{-p}$ after $p-1$ fraction bits

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

3.2.2 Rounding down

An algorithm producing n as in (10) is:

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

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

3.2.3 Rounding up

An algorithm producing n as in (11) is:

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

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

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.

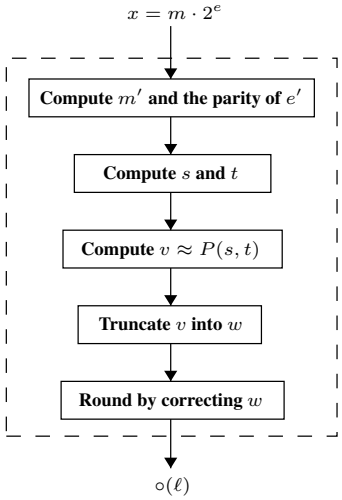

Fig. 3. Computation of $\circ(\ell)$ for $\circ \in \{\mathsf{RN}, \mathsf{RD}, \mathsf{RU}\}$.

4 Implementation details

The above square root method, which is summarized in Figures 1–3, can be implemented by operating exclusively on integers. We will now detail such an implementation, in C and for single precision floating-point numbers; by "single precision" we mean the basic format *binary32* of [2], for which storage bit-width, precision, and maximum exponent are, respectively,

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

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 binary format encoding of floating-point data (see [2, §3.4]). Our implementation of `sqrt` will thus be a C fonction as in line 1 of Listing 1, 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 as $x = (-1)^{X_{31}}|x|$; 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 so-called trailing significand field. Finally, 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 we have 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. (How to implement `max`, `nlz`, and `mul` 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 by means of (33).

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

TABLE 3
Square root results for special operands

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

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 (also, if x is a qNaN then we return x, as recommended).

Using the fact that the addition of two `unsigned int` is done modulo 2^{32} and taking the hexadecimal values of the constants in (33) and (34), we finally get the following C code for handling special operands:

```
if ( (X + 0xFFFFFFFF) >= 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 Listing 1.
}
```

In the above C code, notice that the four operations `+`, `<=`, `==`, and `|` on `X` are independent of each other.

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 D of the exponent d of r satisfies

$$D = d + 127.$$

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	$1111111111X_{21} \dots X_0$

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

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

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

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

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,

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

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

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

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

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

(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 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 $r = (1.0 \dots 0)_2 \cdot 2^{d+1}$.

Consequently, we have implemented the computation of $D - 1$ using the formula below, which is a direct consequence of (38):

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

This corresponds to the computation of variable `Dm1` at line 7 of Listing 1. The only difference with the implementation of D given below (38) occurs at line 6, where we perform `B + 133` instead of `B + 135`.

The parity of e' in (5) will be needed in Sections 4.3 and 4.5. Using (31), (36) and (37), we deduce that e' is even if 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, 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);
```

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

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

Listing 1
Square root implementation for the binary32 format, assuming a non-special operand and rounding to nearest.

```
1  unsigned int binary32sqrt( unsigned int X )
2  {
3    unsigned int B, C, Dm1, E, even, M, S, T, Z, P, Q, V, W;
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 = poly_eval(S, T); // Bivariate polynomial evaluation: S [1.31], T [0.32], V [2.30]
11
12   W = V & 0xFFFFFFC0;  // Truncation after 24 fraction bits: W [2.24]
13
14   P = mul(W, W);                             Q = (( T >> 1) | 0x80000000) >> (even + 2);
15   if( P >= Q )
16     return (Dm1 << 23) + (W >> 7);
17   else
18     return (Dm1 << 23) + ((W + 0x00000040) >> 7);
19 }
```

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 1.

The number τ in (19) satisfies $\tau = 0.m_{\lambda+1}\ldots m_{23}$. 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 instruction-level parallelism can be exposed by implementing this shift as in line 7 of Listing 1.

4.4 Computing the approximate polynomial value v

An implementation of the evaluation scheme (30) using 32-bit unsigned integers and the relation (25), is described in Listing 2. Notice that the multiplications by coefficients equal to a power of two (like A_1, A_2, and A_8) are implemented as simple shifts.

Listing 2
Bivariate polynomial evaluation code.

```
//--------------------------------------------
// A0 = 0x80000000;
// A1 = 0x40000000;
// A2 = 0x10000000;
// A3 = 0x07fe93e4;
// A4 = 0x04eef694;
// A5 = 0x032d6643;
// A6 = 0x01c6cebd;
// A7 = 0x00aebe7d;
// A8 = 0x00200000;
//--------------------------------------------
static inline
unsigned int poly_eval( unsigned int S,
                        unsigned int T )
{
  unsigned int r0  = T >> 2;
  unsigned int r1  = 0x80000000 + r0;
  unsigned int r2  = mul(S,r1);
  unsigned int r3  = 0x00000020 + r2;
  unsigned int r4  = mul(T,T);
  unsigned int r5  = mul(S,r4);
  unsigned int r6  = r5 >> 4;
  unsigned int r7  = r3 - r6;
  unsigned int r8  = mul(T,0x07fe93e4);
  unsigned int r9  = mul(r5,r8);
  unsigned int r10 = r7 + r9;
  unsigned int r11 = mul(r4,r5);
  unsigned int r12 = mul(T,0x032d6643);
  unsigned int r13 = 0x04eef694 - r12;
  unsigned int r14 = mul(T,0x00aebe7d);
  unsigned int r15 = 0x01c6cebd - r14;
  unsigned int r16 = r4 >> 11;
  unsigned int r17 = r15 + r16;
  unsigned int r18 = mul(r4,r17);
  unsigned int r19 = r13 + r18;
  unsigned int r20 = mul(r11,r19);
  unsigned int r21 = r10 - r20;
  return r21;
}
```

Assuming a latency of 1 for additions, subtractions and shifts, a latency of 3 for 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 below. Notice that three issues are enough for that particular polynomial evaluation code.

TABLE 4
Feasible scheduling on ST231.

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

The numerical quality of the code in Listing 2 has been verified using the *Gappa* software (see http://lipforge.ens-lyon.fr/www/gappa/ and [13]; see also [14, §4] for some guidelines on how to translate a C code into *Gappa* syntax). With this software, we first checked that all variables $r_0, \dots r_{21}$ 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).

4.5 Implementing the rounding tests

There the only non-trivial part is to evaluate the expressions $w \geq \ell$ (used when rounding to nearest and rounding up; see §3.2.1 and §3.2.3), and $w > \ell$ (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 will define below by considering w^2 and ℓ^2 instead of w and ℓ.

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

$$w = W \cdot 2^{-30}, \tag{44}$$

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

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

On the one hand let P be the integer in $[0, 2^{32})$ given by

$$P = \mathtt{mul}(W, W).$$

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

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

On the other hand, $\ell^2 = \sigma^2 m'$ is equal to either

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

or

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

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 w^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 from T in (42-43) and from the parity of e' can be found at line 14 of Listing 1.

4.5.1 Rounding to nearest and rounding up

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

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

Proof: Since w and ℓ are non-negative, $w \geq \ell$ is equivalent to $w^2 \geq \ell^2$. If $w^2 \geq \ell^2$ then, by (46) and the left inequality in (45), we deduce that $P + 1 > Q$. Since both P and Q are integers, this latter condition is equivalent to $P \geq Q$. Conversly, if $P \geq Q$ then, multiplying both sides by 2^{-28} gives $P \cdot 2^{-28} \geq \ell^2$ and thus, using the right inequality in (45), $w^2 \geq \ell^2$. Therefore $w \geq \ell$ if and only if $P \geq Q$ or, equivalently, if and only if the C expression `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 1.

Since rounding up depends also on the condition $w \geq \ell$ (see Section 3.2.3), this rounding mode can be implemented by simply replacing lines 15 to 18 in Listing 1 by the following code fragment:

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

4.5.2 Rounding down

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

In order to implement the condition $w > \ell$, let $Q' \in \{0, 1\}$ be such that $Q' = 1$ if and only if equality $P \cdot 2^{-28} = w^2$ occurs in (45), that is, if and only if $P = W^2 \cdot 2^{-32}$. The latter equality means that W has at least 16 trailing zeros; since the bit string of W 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 & 0xFFC0 ) == 0x0;
```

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

Proof: Since w and ℓ are non-negative, $w \geq \ell$ is equivalent to $w^2 \geq \ell^2$. 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 $w > \ell$ 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 $P \cdot 2^{-28} < w^2$ and thus $P \geq Q$ implies $w^2 > Q \cdot 2^{-28} = \ell^2$; conversely, if $w^2 > Q \cdot 2^{-28}$ then, using the left inequality in (45), we find $P + 1 > Q$ and thus $P \geq Q$. Therefore, $w > \ell$ 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 into an unsigned 32-bit 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 follwing implementation of rounding down:

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

5 Experiments with the ST231 core

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 [15]. The ST231 is the most recently designed core of this family, and is widely used in STMicroelectronics SOCs for multimedia acceleration.

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 1-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 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.

To enable the reduction of conditional branches, the architecture provides a 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 1-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 psi-SSA representation is used in the Open64 compiler to generate partially predicated code based on the `slct` instruction [16].

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 this article, requiring only the usage of one specific intrinsic to select the `nlz` instruction (number of leading zeros, see Section 4).

5.2 Performances on ST231

This section presents some numerical results obtained with the complete C implementation of this paper compiled with the ST200 VLIW compiler, using the optimization level -O3. The Table 5 shows the latency, the number of integer instructions, and the number of instructions per cycle (IPC), for all rounding modes.

For comparison, a version of our code that does *not* support subnormal numbers has also been implemented. Removing the requirement of supporting subnormal numbers allows to simplify our code in Section 4 and to introduce further optimizations.

Thanks to the if-conversion technique mentioned in Section 5.1, the generated assembly code is a fully if-converted straigh-line program. In each case, the latency is the same regardless of the nature of the input (generic numbers or special values). And in both cases (with or without support of subnormal numbers) we can also observe that the latency is the same for all rounding modes.

Concerning the code we have detailed in Section 4, which supports subnormal numbers, the critical path is composed by:

- the computation of τ (second coordinate of the evaluation point): 3 cycles;
- the computation of the value v (polynomial evaluation): 13 cycles;
- the rounding algorithm (truncation into w, product, test and selection): 1 + 3 + 1 + 1 = 6 cycles;
- and the final result selection: 1 cycle.

Consequently, the critical path gives a theoretical latency of 23 cycles, which corresponds exactly to the latency observed in practice. In other words, our C implementation of Section 4 has been optimally scheduled by the ST200 VLIW compiler.

Now assuming that subnormal numbers are *not* supported, the computation of τ becomes:

```
T = X << 9;
```

This line costs 1 cycle instead of the 3 cycles needed before. In this case the critical path gives a theoretical latency of 21 cycles which, again, is achieved in practice.

	Subnormal numbers supported			Subnormal numbers *not* supported		
	Latency	Number of integer instructions	IPC	Latency	Number of integer instructions	IPC
RN	23	62	2.70	21	56	2.67
RU	23	63	2.74	21	57	2.71
RD/RZ	23	65	2.83	21	59	2.81

TABLE 5
Performances obtained for the ST231 VLIW integer processor.

Therefore, an interesting conclusion is that with our approach and on targets like the ST231, the overhead due to the support of subnormal numbers can be kept fairly small.

APPENDIX A
SOFTWARE IMPLEMENTATION OF THE `max`, `nlz`, AND `mul` OPERATORS

Maximum of two unsigned integers

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

Number of leading zeros of an unsigned integer

```
static inline unsigned int nlz(unsigned int X)
{
 unsigned int Z = 0;
 if (X == 0) return(32);
 if (X <= 0x0000FFFF) {Z = Z +16; X = X <<16;}
 if (X <= 0x00FFFFFF) {Z = Z + 8; X = X << 8;}
 if (X <= 0x0FFFFFFF) {Z = Z + 4; X = X << 4;}
 if (X <= 0x3FFFFFFF) {Z = Z + 2; X = X << 2;}
 if (X <= 0x7FFFFFFF) {Z = Z + 1;}
 return Z;
}
```

Since in the generic case x cannot be equal to zero, the first line of the previous listing

```
if (X == 0) return(32);
```

can be skipped.

Higher half of a 32-bit integer product

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

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, 2002.

[6] M. D. Ercegovac and T. Lang, *Digital Arithmetic*. Morgan Kaufmann, 2003.

[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] 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.

[9] 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

[10] J.-M. Muller, *Elementary functions: algorithms and implementation*, 2nd ed. Birkhäuser, 2006.

[11] C. Q. Lauter, "Arrondi correct de fonctions mathématiques - fonctions univariées et bivariées, certification et automatisation," Ph.D. dissertation, ÉNS Lyon, France, 2008.

[12] S. Chevillard and C. Lauter, "A certified infinite norm for the implementation of elementary functions," in *QSIC '07: Proceedings of the Seventh International Conference on Quality Software*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 153–160.

[13] G. Melquiond, "De l'arithmétique d'intervalles à la certification de programmes," Ph.D. dissertation, ÉNS Lyon, France, 2006. [Online]. Available: http://www.msr-inria.inria.fr/~gmelquio/doc/06-these.pdf

[14] 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

[15] J. A. Fisher, P. Faraboschi, and C. Young, *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*. Morgan Kaufmann, 2005.

[16] 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.