

How to square floats accurately and efficiently on the ST231 integer processor

Claude-Pierre Jeannerod^{1,3} Jingyan Jourdan-Lu^{2,3} Christophe Monat² Guillaume Revy⁴

¹INRIA, ²STMicroelectronics, ³Université de Lyon, ⁴Université de Perpignan

Laboratoire LIP (CNRS, ENS de Lyon, INRIA, UCBL), Université de Lyon — 46, allée d'Italie, 69364 Lyon cedex 07, France

Compilation Expertise Center, STMicroelectronics — 12, rue Jules Horowitz BP217, 38019 Grenoble cedex, France

Laboratoire ELIAUS-DALI, Université de Perpignan Via Domitia — 52 avenue Paul Alduy, 66860 Perpignan cedex 9, France

Abstract

We consider the problem of computing IEEE floating-point squares by means of integer arithmetic. We show how the specific properties of squaring can be exploited in order to design and implement algorithms that have much lower latency than those for general multiplication, while still guaranteeing correct rounding. Our algorithm descriptions are parameterized by the floating-point format, aim at high instruction-level parallelism (ILP) exposure, and cover all rounding modes. We show further that their C implementation for the binary32 format yields efficient codes for targets like the ST231 VLIW integer processor from STMicroelectronics, with a latency at least 1.75x smaller than that of general multiplication in the same context.

Keywords: *squaring, binary floating-point arithmetic, correct rounding, IEEE 754, instruction level parallelism, C software implementation, VLIW integer processor*

1. Introduction

The STMicroelectronics ST231 is a 32-bit, 64-register, 4-issue, embedded integer VLIW processor. It is currently used in consumer electronics devices (set-top-boxes, sound processors, camera systems...) mainly performing intensive media and signal processing. As this processor has integer-only register file and ALUs, all the single precision floating-point support is available through software emulation, based on the highly optimized FLIP 1.0 library [9]. This comprehensive support is efficient enough to allow application developers to use out-of-the box floating-point code, instead of converting it to fixed-point representations.

However, the implementation of floating-point operations on integer-only processors can lead to sub-optimal use of the computational resources: for example, all binary op-

erators $f(x, y)$ called with the same argument $x = y$ lead to redundant unpacking of the floating-point format, and to useless checks for special cases, and fail to exploit some properties of $f(x, x)$. Since the compiler can detect all static cases of such occurrences, this leads to the idea of specializing operators to gain additional performance.

In this paper we focus on the specialization of the multiplication operator $f : (x, y) \mapsto x \times y$ into a square operator $x \mapsto x^2$. Squares of floating-point values are ubiquitous in scientific computing and signal processing, since they are intensively used in any algorithm requiring the computation of Euclidean norms, powers, sample variances, etc. [6, 11].

We give a thorough study of how to design efficient software for IEEE floating-point squaring on targets like the ST231, that is, by means of integer arithmetic and logic only, and with high ILP exposure.

Our first contribution is to show how the specific properties of squaring can be exploited in order to refine the IEEE specification of multiplication, to deduce definitions of special input and generic input that are suitable for implementation, and to optimize the generic path and the special path for latency. This analysis is done for all rounding modes and presented in a parameterized fashion, in terms of the precision and the exponent range of the input/output floating-point format.

Second, this analysis allows us to produce a complete portable C code for the binary32 format and each rounding mode. On the ST231 processor, the result is a latency of 12 cycles for rounding 'to nearest even,' which is 1.75x faster than the 21 cycle latency of the multiply operator of FLIP 1.0; for the other rounding modes, the speedups are even higher and range from 1.9 to 2.3. Also, the average number of instructions issued every cycle lies between 3.4 and 3.5, thus indicating heavy use of the 4 issues available.

Third, we report on some experiments involving non-IEEE variants and square-intensive applications. We show that relaxing the IEEE requirements (finite math only, no

support of subnormals) saves at most 1 cycle in the context of the ST231. We also show that for applications like the Euclidean norm, the sample variance, and binary powering the practical impact of our fast squarer reflects well the best that can theoretically be achieved.

For squaring in integer / fixed-point arithmetic several optimized hardware designs have been proposed; see for example [3, §4.9] as well as [14, 4] and the references therein. However, for squaring in IEEE floating-point arithmetic much less seems to be available and to the best of our knowledge, no optimized design has been presented and analyzed in the details as we do here, be it in hardware or in software. Furthermore, the implementation of squaring for the binary32 format and the ST231 processor outlined in [12] does not support subnormal numbers, is available only for rounding 'to nearest even,' and has a latency of 27 cycles, which is 2.25x more than our 12 cycle latency.

Notation. For a real number r we write $\lfloor r \rfloor$ for the largest integer not greater than r , and $\lceil r \rceil$ for the smallest integer not less than r . If r satisfies further $0 \leq r < 2$ then its binary expansion $\sum_{i \geq 0} r_i 2^{-i}$ with $r_i \in \{0, 1\}$ for all i will be simply written $(r_0.r_1r_2\dots)_2$. For a floating-point datum x , we write $|x|$ for the absolute value of x , assuming implicitly that x is not NaN. Finally, we shall often use the following square bracket notation: for any true-or-false statement S , let $[S]$ be 1 if S is true, 0 otherwise.

Outline. This paper is organized as follows. After some reminders on the ST231 processor and IEEE binary floating point in Section 2, we show in Section 3 how to specialize to squaring the IEEE specification of multiplication, deduce suitable definitions of generic and special input, and give a high-level algorithmic view of the squaring operator. Sections 4 and 5 then detail our algorithms for handling, respectively, generic and special input by means of integer arithmetic, and give the corresponding C implementation for the binary32 format. The performances of this implementation on the ST231 processor are reported in Section 6, and we conclude in Section 7. The detailed proofs of Properties 1 to 9 have been postponed to Appendix A for more legibility.

2. Background

2.1. STMicroelectronics' ST231 processor

The targeted processor is the ST231 from STMicroelectronics. Since it is a scoreboardd VLIW, there is no dynamic instruction dispatch contrary to what could achieve an equivalent 4-way superscalar architecture: reaching the best performance relies heavily on compiler efficiency.

Instruction scheduling is done after code selection, that must be aggressive to favor high ILP. Specifically, the if-conversion optimization that replaces conditional branches by conditional 'slet' instructions helps creating

large straight-line code regions that are subject to efficient scheduling. The latency of this 'slet' instruction is 1 cycle.

Up to four independent instructions can be bundled into a 'syllable,' achieving the maximum throughput of four instructions per cycle. One constraint is that the immediate value that can be encoded as part of an instruction is limited to 9-bit signed: a larger 32-bit constant, named extended immediate, can be built by consuming an additional instruction syllable in the bundle. This reduces the actual parallelism available locally, but this compares favorably with other mechanisms such as loading the constant from memory, that incurs a performance impact on the data cache side of the machine.

Only a very small subset of the ST231 instruction set is needed by our squaring algorithms: logical and bitwise operators (&&, ||, &, |), relational operators (<=, >=, >), bitwise shift operators (<<, >>), addition, subtraction, maximum, and minimum of (un)signed integers (+, -, max, maxu, minu), and finally a multiply operator mul giving the higher half $\lfloor AB/2^{32} \rfloor$ of the exact product of two 32-bit unsigned integers A and B . Except for mul whose latency is 3 cycles, the latency of each of these operators is 1 cycle.

2.2. IEEE 754-2008 binary floating point

Binary floating-point data. As specified in the IEEE 754-2008 standard [7], binary floating-point data consist of quiet or signaling not-a-numbers (qNaN, sNaN), signed zeros (+0, -0), signed infinities (+∞, -∞), and finite nonzero binary floating-point numbers having the following form: given a precision p and an exponent range $[e_{\min}, e_{\max}]$,

$$x = (-1)^s \cdot m \cdot 2^e, \quad (1a)$$

where s is either 0 or 1, and where

$$m = (m_0.m_1\dots m_{p-1})_2 \quad \text{and} \quad e_{\min} \leq e \leq e_{\max}. \quad (1b)$$

Any such number must in fact be either *normal* ($m_0 = 1$) or *subnormal* ($m_0 = 0$ and $e = e_{\min}$). Thus, writing α for the smallest positive subnormal number and Ω for the largest normal number, we have

$$\alpha = 2^{e_{\min}-p+1} \quad \text{and} \quad \Omega = (2 - 2^{1-p}) \cdot 2^{e_{\max}}.$$

Assumptions on e_{\max} , e_{\min} , and p . We shall assume that

$$e_{\max} = 2^w - 1 \quad \text{for some positive integer } w, \quad (2a)$$

$$e_{\min} = 1 - e_{\max}, \quad 2 \leq p < e_{\max}, \quad (2b)$$

and we shall write

$$k = w + p.$$

All the binary k formats of the IEEE 754-2008 standard satisfy these assumptions; see [7, Table 3.5] and, for a proof

that $2 \leq p < e_{\max}$, see Appendix A. This allows us to carry out all the analysis in a parameterized way and specialize later to a given format, for example the binary32 format:

$$w = 8, \quad e_{\min} = -126, \quad e_{\max} = 127, \quad p = 24.$$

Standard encoding into k -bit integers. For the binary k format the standard encoding of x in (1) is via a k -bit unsigned integer X whose bit string satisfies

$$X = [s|E_{w-1} \cdots E_0|m_1 \cdots m_{p-1}], \quad E = \sum_{i=0}^{w-1} E_i 2^i, \quad (3)$$

with $E = e - e_{\min} + m_0$. Zeros, infinities, and NaNs are encoded via special values of X . In particular, writing $|X| = X \bmod 2^{k-1}$, we have

$$x = \begin{cases} +0 & \text{iff } X = 0, \\ +\infty & \text{iff } X = 2^{k-1} - 2^{p-1}, \\ \text{sNaN} & \text{iff } 2^{k-1} - 2^{p-1} < |X| < 2^{k-1} - 2^{p-2}, \\ \text{qNaN} & \text{iff } |X| \geq 2^{k-1} - 2^{p-2}. \end{cases} \quad (4)$$

Correct rounding. Besides floating-point data and their encoding into integers, the standard [7, §4.3] defines rounding modes \circ to map any real number y to a unique floating-point datum $x = \circ(y)$. In radix 2 four rounding modes are required: to nearest even (RN), down (RD), up (RU), and to zero (RZ). In the case of the square operator we shall restrict to RN, RD, and RU, since $\text{RZ}(y) = \text{RD}(y)$ when $y \geq 0$.

Exceptions and flags. Finally, the standard defines five exceptional situations (invalid operation, division by zero, overflow, underflow, inexact) and requires that they shall be signaled by raising some status flags. For square, all exceptions but 'division by zero' can occur, just like for multiply. However, since the status flags are currently not set in the multiply operator of the FLIP library [9] to which we compare, we have not implemented them for square either.

3. Specification and high-level algorithm

Squaring being a special case of general multiplication $x \times y$, it is fully specified by the IEEE 754-2008 standard: given a rounding mode \circ and assuming $x = y$, the result r prescribed by [7] for $x \times y$ is as follows:

$$r = \begin{cases} |x| & \text{if } x \in \{\pm 0, \pm \infty\}, \\ \text{qNaN} & \text{if } x \text{ is NaN}, \\ \circ(x^2) & \text{otherwise.} \end{cases} \quad (5)$$

This specification shows that r is essentially known in advance when x is zero, infinity, or NaN. However, in the particular case of squaring, if $|x|$ is nonzero but "small enough" then the rounded value $\circ(x^2)$ will always be equal to a tiny

constant. Similarly, if $|x|$ is finite but "large enough" then $\circ(x^2)$ will always be equal to a huge constant. The rest of this section studies such properties of $\circ(x^2)$ in order to refine the specification (5) and deduce a high-level algorithm.

Let \min_{\circ} and \max_{\circ} denote, respectively, the minimum value and maximum value of $\circ(x^2)$ for $|x|$ in $[\alpha, \Omega]$. Using the monotonicity, for $x \geq 0$, of the map $x \mapsto x^2$ together with the definitions of rounding and overflow in [7], one may check that these values are as follows:

\circ	RN	RD	RU
\min_{\circ}	+0	+0	α
\max_{\circ}	$+\infty$	Ω	$+\infty$

(6)

For which values of x are such extremal values attained? To answer this, let us define the following two quantities

$$\alpha' = 2^{\lfloor (e_{\min} - p)/2 \rfloor} \quad \text{and} \quad \Omega' = 2^{(e_{\max} + 1)/2}. \quad (7)$$

We give below three properties regarding these quantities.

Property 1. *The values α' and Ω' defined in (7) are normal floating-point numbers such that $\alpha' < \Omega'$.*

Property 2. *For $\circ \in \{\text{RN}, \text{RD}, \text{RU}\}$ and x a finite nonzero floating-point number, one has $\circ(x^2) = \max_{\circ}$ iff $|x| \geq \Omega'$.*

The interval $[\Omega', \Omega]$ thus defines the widest input range on which \max_{\circ} is achieved. Interestingly, this range is the same for all our rounding modes, which makes things simpler from the implementer point of view. Note also that Ω' is an integer power of two because of (2a). For \min_{\circ} the situation is slightly more complex, as this minimum value is achieved on an input range $[\alpha, \alpha']$ whose upper bound now depends on \circ . However, the property below shows that one can suppress this dependency by restricting to input ranges whose upper bound has the form 2^i for some integer i .

Property 3. *The value α' in (7) is the largest integer power of two such that, for every finite nonzero floating-point number x in $[\alpha, \alpha']$, $\circ(x^2) = \min_{\circ}$ for $\circ \in \{\text{RN}, \text{RD}, \text{RU}\}$.*

The main outcome of Properties 2 and 3 is the following specification of floating-point squaring, which refines (5):

$$r = \begin{cases} +0 & \text{if } x = \pm 0, \\ \min_{\circ} & \text{if } \alpha \leq |x| < \alpha', \\ \circ(x^2) & \text{if } \alpha' \leq |x| < \Omega', \\ \max_{\circ} & \text{if } \Omega' \leq |x| \leq \Omega, \\ +\infty & \text{if } x = \pm \infty, \\ \text{qNaN} & \text{if } x \text{ is NaN.} \end{cases} \quad (8)$$

This brings a natural distinction between two kinds of input:

Definition 1. *Input x is called generic if $\alpha' \leq |x| < \Omega'$, and special otherwise.*

By Property 1 every subnormal input is special, so that generic input consist of normal numbers only. The corresponding output $\circ(x^2)$ must be finite because of the “only if” part in Property 2, but it can be (sub)normal or zero.

Let us now define

$$C_{\text{spec}} = [x \text{ is special}]. \quad (9)$$

This condition allows us to translate (8) into the following high-level algorithmic description, which shows that squaring essentially reduces to three independent tasks T_i :

evaluate the condition C_{spec} in (9)	$[T_1]$
if (C_{spec}) then	
handle special input as in (8)	$[T_2]$
else	
compute $\circ(x^2)$	$[T_3]$

Thanks to if-conversion, the generated assembly for the above algorithm will consist of a straight-line piece of code computing the result R_i of each task T_i and ending with a ‘slct’ instruction that selects R_2 if R_1 is true, R_3 otherwise.

For the design and implementation of each task we shall proceed in two steps as in [1]: assuming unbounded parallelism we optimize the *a priori* most expensive task first, namely task T_3 (see Section 4), and then only T_1 and T_2 , by trying to reuse as much as possible what was computed for T_3 (see Section 5). The latency of ‘slct’ being of 1 cycle, the lowest latency we can expect for squaring thus is 1 cycle more than that of T_3 .

4. Computing correctly-rounded squares

In this section we consider the computation of $\circ(x^2)$ for x generic, that is, x as in (1) and such that

$$\alpha' \leq |x| < \Omega'. \quad (10)$$

By Property 1 such an x is normal, and thus $1 \leq m < 2$.

4.1. Parameterized formula for $\circ(x^2)$

Normalized representation of the exact square. A first step towards the computation of $\circ(x^2)$ consists in normalizing the representation $m^2 \cdot 2^{2e}$ of x^2 implied by (1a). Let

$$c = \left[m \geq \sqrt{2} \right]. \quad (11)$$

Defining $m' = m^2 \cdot 2^{-c}$ and $e' = c + 2e$ then yields the unique pair $(m', e') \in \mathbb{R} \times \mathbb{Z}$ such that $1 \leq m' < 2$ and

$$x^2 = m' \cdot 2^{e'}. \quad (12)$$

This is the so-called *normalized representation* of the exact square. Tight bounds for e' are given by the next result.

Property 4. *The normalized exponent e' of x^2 satisfies*

$$2 \lfloor (e_{\min} - p) / 2 \rfloor \leq e' \leq e_{\max}.$$

These bounds for e' are the best possible ones:

- The lower bound is attained when $|x| = \alpha'$. It is equal to $e_{\min} - p - \epsilon$ with $\epsilon = \lceil p \text{ is odd} \rceil$. One has $\epsilon = 0$ for the standard *binary32* format, and $\epsilon = 1$ for the standard *binary16*, *binary64*, and *binary128* formats [7, §3.6].
- The upper bound e_{\max} is attained for example when $x = (1.1)_2 \cdot 2^{(e_{\max}-1)/2}$, which satisfies (10) and whose square is $(1.001)_2 \cdot 2^{e_{\max}}$.

Also, the tight lower bound on e' is less than e_{\min} for $p \geq 2$, so that both situations $e' \geq e_{\min}$ and $e' < e_{\min}$ do occur.

Correctly-rounded value of the exact square. When $e' \geq e_{\min}$ the normalized representation (12) already allows to express the correctly-rounded value $\circ(x^2)$. In this case x^2 lies in the range $[2^{e_{\min}}, 2^{e_{\max}+1})$ and, since $m' \in [1, 2)$,

$$\begin{aligned} \circ(x^2) &= \circ(m') \cdot 2^{e'} \\ &= \circ(m^2 \cdot 2^{-c}) \cdot 2^{c+2e}. \end{aligned} \quad (13a)$$

When $e' < e_{\min}$ the exact square ranges in $(0, 2^{e_{\min}})$. In this case, we first set the exponent to e_{\min} and only then round the resulting scaled significand in fixed point:

$$\begin{aligned} \circ(x^2) &= \tilde{\circ}(m' \cdot 2^{-(e_{\min}-e')}) \cdot 2^{e_{\min}} \\ &= \tilde{\circ}(m^2 \cdot 2^{-(e_{\min}-2e)}) \cdot 2^{e_{\min}}, \end{aligned} \quad (13b)$$

where $\tilde{\circ}$ denotes the function that rounds the reals from $[0, 2)$ in the same direction as \circ but on the regular grid $\{i \cdot 2^{1-p} : i = 0, 1, \dots, 2^p\}$.

In order to handle the cases (13a) and (13b) simultaneously, let us define

$$\mu = \max(c, e_{\min} - 2e). \quad (14)$$

Since $\tilde{\circ}$ coincides with \circ on $[1, 2)$, the correctly-rounded value of the exact square is given in both cases by

$$\circ(x^2) = \tilde{\circ}(\ell) \cdot 2^d, \quad (15a)$$

where

$$\ell = m^2 \cdot 2^{-\mu} \quad \text{and} \quad d = \mu + 2e. \quad (15b)$$

By construction one has $0 < \ell < 2$ and $e_{\min} \leq d \leq e_{\max}$. The property below further gives bounds for the range of μ .

Property 5. *One has $c \leq \mu \leq p + \epsilon$ with $\epsilon = \lceil p \text{ is odd} \rceil$.*

Again, these bounds are tight: $x = 1$ gives $\mu = 0$, while $x = \pm\alpha'$ gives $c = 0$, $e' = e_{\min} - p - \epsilon$, and then $\mu = p + \epsilon$.

An explicit formula for $\tilde{\circ}(\ell)$. The above analysis has reduced, for precision p , floating-point rounding of the exact result x^2 to fixed-point rounding of the scaled significand ℓ

in (15). Using classical rounding formulae (see for example [3, §8.4.3]), we now give an explicit expression for $\tilde{\circ}(\ell)$. Writing $\ell = (\ell_0.\ell_1 \dots \ell_{p-1}\ell_p \dots)_2$ and using \vee for logical OR, the *guard bit* and *sticky bit* of ℓ are, respectively,

$$g = \ell_p \quad \text{and} \quad t = \ell_{p+1} \vee \ell_{p+2} \vee \dots \quad (16)$$

Its correctly-rounded value is then given by the formula

$$\tilde{\circ}(\ell) = (\ell_0.\ell_1 \dots \ell_{p-1})_2 + b \cdot 2^{1-p}, \quad (17)$$

where the *round bit* b satisfies (see [3, pp. 436-437])

$$b = \begin{cases} g \wedge (\ell_{p-1} \vee t) & \text{if } \circ = \text{RN}, \\ 0 & \text{if } \circ = \text{RD}, \\ g \vee t & \text{if } \circ = \text{RU}. \end{cases} \quad (18)$$

4.2. Implementation for the binary k format

We detail here how to implement, for x generic and the binary k floating-point format, the computation of $r = \circ(x^2)$ using k -bit integer arithmetic and logic. We assume x is given by its standard encoding into an unsigned k -bit integer X . Since the result r satisfies (15a), it is known (see for example [13, §2.3.1]) that its standard integer encoding R is given by $R = D \cdot 2^{p-1} + \tilde{L}$, where

$$D = d + e_{\max} - 1 \quad (19)$$

and $\tilde{L} = \tilde{\circ}(\ell) \cdot 2^{p-1}$. Using (17) we obtain $\tilde{L} = L + b$ with

$$L = \lfloor \ell \cdot 2^{p-1} \rfloor \quad (20)$$

and b the sticky bit defined in (18), so that eventually

$$R = D \cdot 2^{p-1} + L + b. \quad (21)$$

Consequently, computing R amounts to deducing D , L , b from X , which we detail now in a parameterized fashion, that is, for the binary k format. We illustrate our analysis for the binary32 format, and the corresponding C code (for rounding 'to nearest even') appears in Listing 1. In order to give an idea of the ILP exposed by our approach, this C code has been set out in such a way that line i displays only the expressions that can be evaluated in i cycles with the ST231 latencies and assuming unbounded parallelism.

Computing L . From (15b) and (20) it follows that

$$L = \lfloor m^2 / 2^{1-p+\mu} \rfloor \quad (22)$$

and, therefore, we first need to deduce from X an integer encoding of m , say M , as well as the integer μ .

To produce M , recall that $m = (1.m_1 \dots m_{p-1})_2$ as x is normal. Since $p \leq k$ a possible choice is to set up $m \cdot 2^{k-1}$, which can be obtained from (3) by shifting and masking:

$$M = m \cdot 2^{k-1} = (X \ll w) \mid 2^{k-1}.$$

For the binary32 format, where $w = 8$ and $k = 32$, this corresponds to line 2 in Listing 1; on ST231, this takes 2 cycles and, due to the extended immediate value $2^{31} = (80000000)_{16}$, 3 instruction syllables.

To get μ , recall first that x normal implies $e = E - e_{\max}$. Then, recalling that $e_{\min} = 1 - e_{\max}$ and applying (14),

$$\mu = \max(c, F), \quad F = e_{\max} + 1 - 2E. \quad (23)$$

To get the boolean value c , it suffices to remark that (11) and $M = m \cdot 2^{k-1}$ imply

$$c = [M > M_0], \quad \text{with} \quad M_0 = \lfloor \sqrt{2} \cdot 2^{k-1} \rfloor.$$

To get the (possibly negative) integer F , first we extract $2E$ from X in (3) by using the identity

$$2E = (X \gg (p-2)) \& (2^{w+1} - 2), \quad (24)$$

and then we subtract $2E$ from the constant $e_{\max} + 1$. For the binary32 format the computation of c and F appears at line 3 of Listing 1, where $(b504f333)_{16}$ is M_0 for $k = 32$; on ST231 each of c and F takes 3 cycles, so that μ is eventually obtained in 4 cycles.

Let us now see how to deduce L from M and μ . The property below shows that the k most significant bits of the $2k$ -bit integer M^2 are enough for that purpose.

Property 6. $L = \lfloor H / 2^{\mu+w-1} \rfloor$ with $H = \lfloor M^2 / 2^k \rfloor$.

Given M the `mul` instruction computes the higher half H of M^2 , which then, by Property 6, simply has to be shifted right by $\mu + w - 1$ in order to yield L . For the binary32 format, this appears at lines 5 and 6 of Listing 1. Since here $p = 24$ is even, we deduce from Property 5 that $\mu + 7$ is at most 31, which thus agrees with the C99 specification of the bitwise shift operator [8, p. 84]. With the ST231 latency constraints, both H and $\mu + 7$ are computed from X in 5 cycles, so that L is obtained in 6 cycles.

Computing b . We focus here on the most interesting case, rounding to nearest even, for which $b = g \wedge (\ell_{p-1} \vee t)$ with g and t as in (16).

Note first that g is the least significant bit of the integer $G = \lfloor \ell \cdot 2^p \rfloor = \sum_{0 \leq i \leq p} \ell_i 2^{p-i}$ and, using a proof similar to that of Property 6, we arrive at

$$g = G \bmod 2, \quad G = \lfloor H / 2^{\mu+w-2} \rfloor.$$

For the binary32 format, the corresponding C code appears at lines 6 and 9 of Listing 1. On ST231, G will have the same latency as L (6 cycles), and we thus get g in 7 cycles.

Since ℓ_{p-1} is the least significant bit of L we have

$$\ell_{p-1} = L \bmod 2,$$

so that we are left with computing the sticky bit t . The next result shows how to recover this bit simply by checking that some lower parts of H and X are nonzero, that is, without computing the lower half of the exact square M^2 .

Listing 1. Computing $\circ(x^2)$ for the binary32 format, $\circ = \text{RN}$, and x generic.

```

1 T2 = X & 0xff;
2 M = (X << 8) | 0x80000000; E2 = (X >> 22) & 0x1ffe;
3 F = 128 - E2; c = M > 0xb504f333;
4 mu = max(c, F);
5 H = mul(M, M);
6 L = H >> (mu + 7); G = H >> (mu + 6); T1 = H << (26 - mu);
7
8
9 b = (G & 1) && ((L & 1) | (T1 | T2));
10 return ((mu - F) << 23) + L + b;

```

Property 7. One has $t = [T_1 \neq 0] \vee [T_2 \neq 0]$ with T_1 and T_2 the two k -bit integers given by

$$T_1 = H \ll (p + 2 - \mu) \quad \text{and} \quad T_2 = X \bmod 2^{p - \lfloor k/2 \rfloor}.$$

For the binary32 format, Property 7 gives $T_2 = X \bmod 2^8$, which can be implemented by masking X as shown at line 1 of Listing 1. The computation of T_1 is a mere left shift of H by $26 - \mu$, the latter value ranging in $[0, 31]$ thanks to Property 5. Then notice that the bit $\ell_{p-1} \vee t$ is zero if and only if the integer U obtained by bitwise-ORing the integers $L \& 1$ and T_1 and T_2 is zero. The logical AND of $g = G \& 1 \in \{0, 1\}$ and U is thus enough to yield b , which allows us to avoid testing explicitly if T_1 or T_2 is nonzero. This is shown at line 9 of Listing 1. The parenthesization chosen there aims to reduce the overall latency for b on ST231: both L and T_1 can be obtained in 6 cycles, while T_2 costs 1 cycle; therefore, both $L \& 1$ and $T_1 | T_2$ follow in 7 cycles, which yields b in 9 cycles.

Computing D . From the definitions of d and D in (15b) and (19) we deduce that $D = \mu + 2e + e_{\max} - 1$. Hence, recalling that $e = E - e_{\max}$ and the definition of F in (23),

$$D = \mu - F.$$

For the binary32 format, this subtraction appears at line 10 of Listing 1. Recalling that on ST231 we get F and μ in, respectively, 3 and 4 cycles, we will thus get D in 5 cycles.

Packing the result. From (21) the integer encoding R of the result satisfies $R = D \cdot 2^{p-1} + L + b$ and we have just detailed how to get from X the integers D , L , and b . Moreover, assuming the latency model of the ST231, their respective cost has been shown to be of 5, 6, and 9 cycles. This implies a latency of 6 cycles for $D \cdot 2^{p-1}$, and using the parenthesization shown at the last line of Listing 1 we eventually get R in 10 cycles. Thus, when \circ is RN the overall cost is larger than that of the round bit b by only one cycle.

Some simplifications when \circ is not RN. When the rounding mode \circ is RD the round bit b in (18) is zero. Consequently, the instructions involving G , T_1 , T_2 , and b can be suppressed and the last line of Listing 1 replaced with:

```
return ((mu - F) << 23) + L;
```

When \circ is RU the bit ℓ_{p-1} is not needed and b is the logical OR of $g = G \& 1$ and $T_1 | T_2$. In this case, we thus replace line 9 of Listing 1 by:

```
b = (G & 1) | (T1 | T2);
```

With these new codes the expected latency of R on ST231 drops from 10 to 7 cycles for $\circ = \text{RD}$, and from 10 to 9 cycles for $\circ = \text{RU}$.

5. Detecting and handling special input

We first have to decide whether input x is special or not, that is, to compute from X the value of C_{spec} in (9). By Definition 1 this condition satisfies

$$C_{\text{spec}} = C_{\text{small}} \vee C_{\text{large}} \vee C_{\text{nan}} \quad (25)$$

with $C_{\text{small}} = [|x| < \alpha']$, $C_{\text{large}} = [|x| \geq \Omega']$, and $C_{\text{nan}} = [x \text{ is NaN}]$. The next two properties show how to obtain C_{small} and $C_{\text{large}} \vee C_{\text{nan}}$ by reusing the value $2E$ computed for the generic case (see (24) and line 2 of Listing 1).

Property 8. $C_{\text{small}} = [2E \leq e_{\max} - p - 1]$.

Property 9. $C_{\text{large}} \vee C_{\text{nan}} = [2E \geq 3e_{\max} + 1]$.

For the binary32 format, a C fragment implementing C_{spec} by means of $2E$ and the two previous properties is shown at lines 2 to 4 of Listing 2. On ST231 the cost will be of 4 cycles and, as C_{spec} is independent of \circ , this fragment holds not only for $\circ = \text{RN}$ but also for $\circ \in \{\text{RD}, \text{RU}\}$.

Listing 2. Detecting and handling special input for the binary32 format and $\circ = \text{RN}$.

```

absX = X & 0x7fffffff;
1 E2 = (X >> 22) & 0x1ffe; Cnan = absX > 0x7f800000;
2 Csmall = E2 <= 102; Clarge_or_nan = E2 >= 382;
3 Cspec = Csmall || Clarge_or_nan;
4
5 if (Cspec) {
6   if (Csmall) return 0; // r = +0
7   else {
8     if (Cnan) return 0x7fc00000; // r = qNaN
9     else return 0x7f800000; // r = +oo
10  } else {
11    // generic case (Listing 1).
12  }

```

Once special input have been filtered out, it remains to return, for the given rounding mode \circ , the standard integer encoding R of the associated result r prescribed by (8):

When \circ is RN. We deduce from (6) and (8) that for x special, r must be $+0$ if $|x| < \alpha'$, qNaN if x is NaN, and $+\infty$ otherwise. Implementing this is then straightforward as it suffices to recall from (4) that, on the one hand 0, $2^{k-1} - 2^{p-1}$, and $2^{k-1} - 2^{p-2}$ are standard encodings of $+0$, $+\infty$, and qNaNs, and that, on the other hand,

$$C_{\text{nan}} = [X \& (2^{k-1} - 1) > 2^{k-1} - 2^{p-1}].$$

For the binary32 format, the computation of C_{nan} is shown at lines 1 and 2 of Listing 2, while lines 6 to 9 display the computation of R . On ST231, lines 6 to 9 will be if-converted as shown by the following pseudo-code:

```
Rlarge_or_nan = slct(Cnan, 0x7fc00000, 0x7f800000)
R = slct(Csmall, 0, Rlarge_or_nan)
```

With a latency of 1 cycle for the 'slct' instruction and since C_{nan} costs 2 cycles, we thus get R for x special in 4 cycles.

When \circ is not RN. For $\circ = \text{RD}$ the only difference with the previous case is when $|x| \geq \Omega'$ and, by(6) and (8), we now have $r = \max(|x|, \Omega)$. The standard integer encoding of Ω is $2^{k-1} - 2^{p-1} - 1$, which equals $(7f7fffff)_{16}$ for the binary32 format. Consequently, it suffices to replace line 9 of Listing 2 with:

```
else return maxu(absX, 0x7f7fffff);
```

For $\circ = \text{RU}$, the specification differs from that for $\circ = \text{RN}$ only in the case where $|x| < \alpha'$, for which we have $r = \min(|x|, \alpha)$. Since the standard integer encoding of α is 1, an implementation for the binary32 format follows by simply replacing line 6 in Listing 2 by

```
if (Csmall) return minu(absX, 1);
```

On ST231, R still costs 4 cycles as both $\max(|x|, \Omega)$ and $\min(|x|, \alpha)$ have a latency of 2 cycles, like C_{nan} .

6. Experimental results obtained on ST231

The C codes detailed in Sections 4 and 5 yield a full implementation of squaring, for the binary32 format and each rounding mode. To check correctness we compiled them with gcc (using a C emulation of the mul, max, maxu, and minu operators as given for example in [13, Appendix B]), and compared with the results of multiplication $x \times x$ obtained on an Intel[®] Xeon[®] workstation. For each rounding mode this exhaustive test took about five minutes.

We also compiled our C codes with the ST200 compiler, in -O3 for the ST231 processor. The remainder of this section details the performances obtained in this context.

6.1. Operator performances

Latency and comparison with general multiplication.

The latency on ST231 of our binary32 square implementation is shown in the second column of Table 1. Due to if-conversion, it gives for each rounding mode a number of clock cycles independent of the input value x . The values within square brackets indicate the lowest latencies we can theoretically achieve with the ST231 latency constraints and assuming unbounded parallelism; these best latencies follow from our analysis in Sections 4 and 5 and have the form $1 + \mathcal{L}$ with \mathcal{L} the best latency for the generic case. This first experiment shows that the latencies achieved in practice are at most 1 cycle from the best possible ones.

For comparison, the third column of Table 1 displays the latencies of the multiply operator $x \times y$ available in the FLIP 1.0 library and optimized for the ST231 [9]. As shown in the fourth column, our specialization of this multiply operator into a square operator yields a speedup between 1.75 and 2.3, depending on the rounding mode.

\circ	square	FLIP 1.0 multiply	speedup
RN	12 [11]	21	1.75
RD	9 [8]	21	2.3
RU	11 [10]	21	1.9
RZ	9 [8]	18	2

Table 1. Latency comparison for square and multiply.

Instruction-level parallelism. When designing our algorithms in Sections 4 and 5 we strived to expose as much ILP as we could. To evaluate ILP in practice we use instructions-per-cycle (IPC), which is the parallelism really exposed on the target. As shown in Table 2 it is deduced from the assembly code by dividing the number of instructions by the latency. The IPC achieved is close to the highest ILP reachable within the architectural constraints of the machine, demonstrating a very efficient usage of its resources.

\circ	Latency L	Number N of instructions	IPC = N/L
RN	12	42	3.5
RD	9	31	3.4
RU	11	37	3.4

Table 2. Latency, code size, and IPC for square.

Comparison with two non-IEEE variants. To study the impact on latency of relaxing the IEEE 754 specification used so far, we have implemented for each rounding mode a finite-math-only variant and a variant without subnormals.

Finite math only. We assume here that input and output are neither infinity nor NaN, and that overflow does not occur. Hence x now satisfies $|x| < \Omega'$. On the one hand, this leaves the generic path unchanged, so that the best possible

latencies are the same as for our IEEE version. On the other hand, (25) becomes $C_{\text{spec}} = C_{\text{small}}$ and the C codes of Section 5 can be simplified accordingly. As the third column of Table 3 shows, in practice this simplification has no impact on latency for RD, while it saves 1 cycle for RN and RU.

No subnormals. Here we assume that x is not subnormal, which means, writing $\lambda = 2^{e_{\text{min}}}$ for the smallest positive normal number, that x is either NaN, zero, or such that $\lambda \leq |x|$. We also assume that if the exact result x^2 lies in the subnormal range $(0, \lambda)$ then $r = +0$ for RN and RD, and $r = \lambda$ for RU. Since $x^2 < \lambda$ is equivalent to $|x| < \lambda'$ with $\lambda' = \sqrt{\lambda}$, the specification of this non-IEEE variant can thus be deduced from (6) and (8) simply by replacing α and α' with, respectively, λ and λ' .

For the special path, this relaxed specification implies $C_{\text{small}} = [|x| < \lambda']$ and the proof of Property 8 can be adapted to show that we now have $C_{\text{small}} = [2E \leq e_{\text{max}} - 1]$. Thus, it suffices to replace 102 by 126 at line 3 of Listing 2 and, for RU, to replace 1 by 2^{23} in the `minu` operation. These updates clearly have no impact on the latency.

Concerning the generic path, the case (13b) need not be considered anymore, so that μ in (14) is now equal to c . Hence the `max` operation at line 4 of Listing 1 can be removed and we can replace μ by c at line 6. However, as H still has a latency of 5 cycles, this simplification does not shorten the critical path. This means that the best latencies that we can theoretically achieve with this second non-IEEE variant are the same as for our IEEE version. Furthermore, Table 3 shows that this is true in practice as well.

o	IEEE	finite math only	no subnormals
RN	12	11	12
RD	9	9	9
RU	11	10	11

Table 3. Latency comparison with two non-IEEE variants.

6.2. Application examples

We now apply our fast operator to some square-intensive algorithms in order to study its effect on real applications. After giving a theoretical model of the speedup achievable on ST231, we show practical speedups and how they match that model. All experiments have been done on the ST231 cycle-accurate simulator, and while we focus on rounding 'to nearest even' (RN) similar results hold for RU and RD.

Speedup model for loop nests involving squares. While Table 1 gives speedups for a single replacement of multiply by square, we now evaluate the theoretical expectation of speedup for loops. When the square operator ap-

pears in a loop of n iterations, we introduce the definition

$$\text{theoretical speedup} = \frac{(\mathcal{L} + \Delta \cdot \sigma) \cdot n + \mathcal{C}}{\mathcal{L} \cdot n + \mathcal{C}}$$

with \mathcal{L} , Δ , σ , and \mathcal{C} given as follows:

- \mathcal{L} is the latency of the loop body where squares are used, which includes the application-related operations inside the loop and the cost to maintain the loop.
- Δ is the latency gap between multiply and square.
- σ is the number of squares in a single iteration.
- \mathcal{C} is the cost of the straight-line code outside the loop.

Note that when n tends to infinity, the theoretical speedup tends to $1 + \Delta \cdot \sigma / \mathcal{L}$, which does not depend on \mathcal{C} .

On ST231 the values Δ , \mathcal{L} , and \mathcal{C} have the following features. First, Table 1 gives $\Delta = 21 - 12 = 9$ cycles. Second, \mathcal{L} and \mathcal{C} can be modelled as

$$\mathcal{L} = a \cdot \text{Br} + \text{Idx} + \text{Ld} + \mathcal{C}_{\text{in}}, \quad \mathcal{C} = \text{St} + \mathcal{C}_{\text{out}}.$$

Here, `Br` is the cost of a taken branch, which is 3 cycles. The unrolling transformation done by the compiler has the effect of dividing the number of branches taken to jump back to the head of the loop by the unrolling factor; a in $(0, 1]$ is used to denote this effect. The value of a depends on the application and on the compiler optimization level; to estimate the trend of the speedup, we take $a = 0.5$, meaning that the loop is always unrolled by a factor of 2. `Idx` is the cost to test loop index, which is 1 cycle; `Ld` is the cost to load input values, which is 3 cycles. \mathcal{C}_{in} (resp. \mathcal{C}_{out}) is the latency of the application-related code within (resp. outside) the loop; for each application, the values of \mathcal{C}_{in} and \mathcal{C}_{out} can be deduced from the latencies of the 5 basic operators of FLIP 1.0 [13, Table 1] and from the latency of 12 cycles of our square operator. Finally, `St` is the cost of stack handling of the function call; it ranges from 10 to 12 cycles depending on the achievable ILP of each application.

The above model has been applied to three application examples, which we review now.

Example 1: Euclidean norm. Given a vector v of n floating-point data v_i we consider the computation of its Euclidean norm $\|v\|_2 = (\sum_{i=1}^n v_i^2)^{1/2}$ by means of three different algorithms.

Naive algorithm. Here $\|v\|_2$ is produced by a for loop whose i th iteration simply squares v_i and adds it to the current partial sum of squares. Figure 1 shows that the theoretical speedup tends to about 1.185 and that in practice we are close to this value as soon as $n \geq 20$, assuming n is known at runtime. If n is known at compile time then an even higher speedup is observed, since the compiler achieves more efficient loop unrolling optimization. The nine black bullets in Figure 1 illustrate this fact for $n = 2, \dots, 10$. The greatest speedups are for $n \leq 4$, since in this case all the

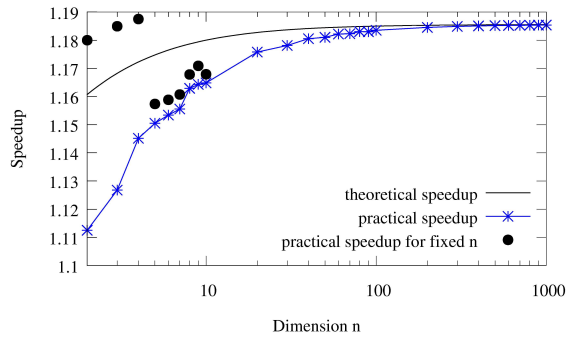


Figure 1. Impact of square on naive Euclidean norm.

parameters are directly passed to the function by registers instead of by stack and the loop is fully unrolled.

Two-pass algorithm. This algorithm, which is already mentioned in [2], aims to avoid overflow by first computing $\|v\|_\infty = \max_i |v_i|$ and then applying the naive algorithm to the scaled vector $[v_i/\|v\|_\infty]_i$. The input is scanned twice and n divisions are used. Thus, the speedup we can expect is lower than for the naive algorithm, as shown in Figure 2. However, we see that the practical speedup still matches well the theoretical model as soon as $n \geq 20$.

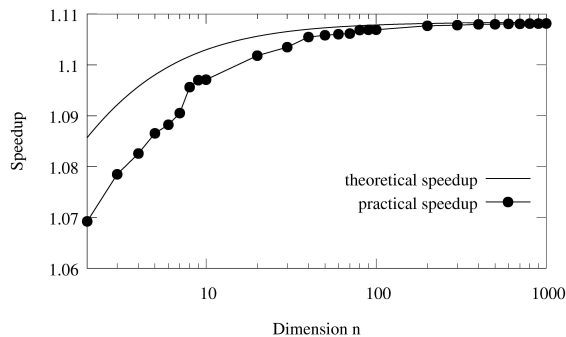


Figure 2. Impact of square on two-pass Euclidean norm.

One-pass algorithm. This algorithm also intends to avoid overflow but requires only one pass over the data, the scaling factor being now computed on the fly. It is an adaptation of Blue's algorithm [2] attributed to Hammarling and implemented in LAPACK [6, p. 507]. Each of the n iterations performs exactly 1 square as well as 2 or 4 additional operations, depending on the data v_i . The total number of operations thus varies dynamically and turns out to be maximum when $|v_1| < |v_2| < \dots < |v_n|$, and minimum when $|v_1| \geq |v_2| \geq \dots \geq |v_n|$. Each of these two extreme cases yields a behavior similar to the one displayed in Figure 2, the limiting value when $n \rightarrow \infty$ being about 1.077 in the first case, and about 1.096 in the second case.

Example 2: sample variance. The sample variance of v_1, \dots, v_n is $\frac{1}{n-1} \sum_{i=1}^n (v_i - \bar{v})^2$, where $\bar{v} = \frac{1}{n} \sum_{i=1}^n v_i$. It is known [6, p. 11] that it can be evaluated accurately

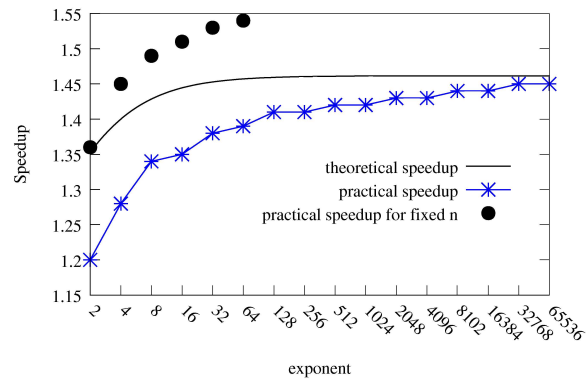


Figure 3. Impact of square on binary powering.

by the naive way, which requires two passes over the data: get \bar{v} first and then iteratively square and add the $v_i - \bar{v}$'s. This method has the same structure as the two-pass algorithm in the previous example (with the maximum replaced by a sum, and the division replaced by a subtraction). Consequently, the theoretical and practical speedups brought by our fast squarer are similar to those in Figure 2.

Example 3: binary powering. For a floating-point datum x and for n an integer power of two such that $n \geq 4$, we consider here the evaluation of x^n by means of $\log_2 n$ successive squares. The results obtained for this application are shown in Figure 3. The qualitative analysis done for the naive algorithm of Example 1 still applies but, since binary powering does not involve any operation other than squaring, higher speedups are observed.

7. Conclusion and perspectives

This work has presented optimized C codes for IEEE binary32 squaring which are, on ST231, significantly faster than the corresponding implementation of general multiplication, thus demonstrating the practical benefit of operator specialization in this context.

To achieve this, we strived to exploit the specific features of the square operator: it is univariate, always nonnegative, and has predictable overflow ranges; subnormal input can always be handled by the special path; and, in the generic path, the lower half of the exact square of the input significant is not needed, and this even for computing the sticky bit used to provide correct rounding. We also heavily relied on some features of our target processor and its compiler, like VLIW 4-issue parallelism, if-conversion and fast 'sct' instruction, fast min and max instructions, and the availability of extended immediates and of an integer multiplier of type '32 × 32 → 32 high.' However, as our C codes show, the subset of the ST231 instruction set needed by our fast squarer is extremely modest and, unlike for multiplication,

the leading-zero count instruction is useless in this context.

Furthermore, all our algorithmic descriptions are parameterized by the binary k format, thus offering increased confidence and flexibility: the correctness analysis is done once for all, and one can deduce portable C codes for various formats in a direct way as soon as the corresponding k -bit integer arithmetic and logic are available. In fact, we already have such a code for the binary64 format and it relies on a 64-bit integer layer highly optimized for ST231. This code is certainly suboptimal, though, as some parts of the algorithm for binary64 squaring could directly use 32-bit integers. Therefore, it would be interesting to implement an optimized 32-bit version of binary64 squaring and to evaluate the latency improvement compared to the 64-bit version produced with our approach.

Three other extensions would also be worth considering. First, although our approach assumes radix 2 we should investigate to which extent it carries over other radices, and especially radix 10. Second, the overhead (in terms of latency and code size) of setting status flags remains to be studied. Third, squaring is not only a specialization of multiplication but also of integer powering, that is, of $\text{pown}(x, n) = x^n$ with n an integer. Since the 2008 revision of the IEEE 754 standard recommends correct rounding for this operator [7, Table 9.1], its optimized implementation for VLIW integer processors will be a natural extension of the work we have presented here.

References

- [1] C. Bertin, C.-P. Jeannerod, J. Jourdan-Lu, H. Knochel, C. Monat, C. Moulleron, J.-M. Muller, and G. Revy. Techniques and tools for implementing IEEE 754 floating-point arithmetic on VLIW integer processors. In *Proceedings of PASCO'10*, pages 1–9, New York, NY, USA, 2010. ACM.
- [2] J. L. Blue. A portable Fortran program to find the Euclidean norm of a vector. *ACM Trans. Math. Software*, 4(1):15–23, 1978.
- [3] M. D. Ercegovic and T. Lang. *Digital Arithmetic*. Morgan Kaufmann, 2004.
- [4] M. Gök. Integer squarers with overflow detection. *Computers & Electrical Engineering*, 34(5):378–391, 2008.
- [5] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley, Reading, MA, USA, second edition, 1994.
- [6] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition, 2002.
- [7] IEEE Computer Society. *IEEE Standard for Floating-Point Arithmetic*. IEEE Standard 754-2008, Aug. 2008.
- [8] International Organization for Standardization. *Programming Languages – C*. ISO/IEC Standard 9899:1999, Geneva, Switzerland, Dec. 1999.
- [9] C.-P. Jeannerod and G. Revy. FLIP 1.0: a fast floating-point library for integer processors, February 2009. Available at <http://flip.gforge.inria.fr/>.

- [10] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010.
- [11] W. Press, B. Flannery, S. Teukolsky, and W. Vetterling. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, Oct. 1992.
- [12] S.-K. Raina. *FLIP: a Floating-point Library for Integer Processors*. PhD thesis, ÉNS Lyon, France, 2006.
- [13] G. Revy. *Implementation of binary floating-point arithmetic on integer processors: polynomial evaluation-based algorithms and certified code generation*. PhD thesis, Université de Lyon - ÉNS de Lyon, France, Dec. 2009.
- [14] E. G. Walters, J. Schlessman, and M. J. Schulte. Combined unsigned and two's complement hybrid squarers. In *Proceedings of the thirty-fifth Conference on Signals, Systems, and Computers (Asilomar 2001)*, volume 1, pages 861–866, Asilomar, Pacific Grove, CA, USA, 2001. IEEE.

A. Proofs

Proof that $2 \leq p < e_{\max}$ for all standard binary formats.

The standard binary formats are defined in [7, Table 3.5]. There we see that, on one hand, this is true when $k \leq 128$. On the other hand, when $k > 128$ we have

$$p = k - j_k + 13 \quad \text{and} \quad e_{\max} = 2^{j_k - 14} - 1,$$

with $j_k = \text{round}(4 \log_2 k)$, the integer closest to $4 \log_2 k$. (If k is an integer then $4 \log_2 k$ cannot be exactly halfway two consecutive integers, so that no tie can occur.) By definition of round, $4 \log_2 k - 1/2 < j_k < 4 \log_2 k + 1/2$. This implies first that, for $k > 2^7$, $2 \leq p \leq k$. Second, $2^{-14.5} k^4 - 1 < e_{\max}$. Third, $k^3 > 2^{15.5}$ and then $2^{-14.5} k^4 > 2k > k + 1$, so that $k < e_{\max}$. \square

Proof of Property 1. Since α' and Ω' in (7) are integer powers of two it suffices to verify that

$$e_{\min} \leq \lfloor (e_{\min} - p)/2 \rfloor < (e_{\max} + 1)/2 \leq e_{\max}.$$

The leftmost inequality is equivalent to $e_{\min} \leq (e_{\min} - p)/2$ because e_{\min} is an integer; since $e_{\min} = 1 - e_{\max}$, the latter inequality is itself equivalent to $p < e_{\max}$, which is true by (2b). From (2b) it also follows in particular that $e_{\max} \geq 1$, which implies the rightmost inequality. The remaining inequality follows from the fact that (2b) implies that e_{\min} is negative while p and e_{\max} are positive. \square

Proof of Property 2. If $|x| \geq \Omega'$ then $x^2 \geq 2^{e_{\max}+1}$, so that $\circ(x^2) = \max_{\circ}$ for each \circ . Conversely, assume that $|x| < \Omega'$. Since x is a finite floating-point number in precision p , we deduce that $|x| \leq (2 - 2^{1-p}) \cdot 2^{(e_{\max}-1)/2}$ and then $x^2 \leq C \cdot 2^{e_{\max}}$, with $C = 2(1 - 2^{-p})^2$. Since $C < 2 - 2^{1-p}$ one has further $x^2 < \Omega$. This implies $\circ(x^2) < \max_{\circ}$ for each \circ and the conclusion follows. \square

Proof of Property 3. If $\alpha \leq x < \alpha'$ then $0 < x^2 < \alpha/2$, so that $\text{RN}(x^2) = \text{RD}(x^2) = +0$ and $\text{RU}(x^2) = \alpha$. This shows that $x \in [\alpha, \alpha')$ implies $\circ(x^2) = \min_\circ$ for all \circ . To prove the maximality of α' it suffices to check that $\text{RN}(y^2) \neq \min_{\text{RN}}$ for some floating-point number y in $[\alpha', 2\alpha')$, say $y = \frac{3}{2}\alpha'$. We have $y^2 = \frac{9}{4}2^{2\lfloor(e_{\min}-p)/2\rfloor}$ and, using the fact that $\lfloor i/2 \rfloor \geq (i-1)/2$ when $i \in \mathbb{Z}$, we deduce that $y^2 \geq \frac{9}{8}\alpha/2 > \alpha/2$. It follows that $\text{RN}(y^2) = \alpha$, which differs from $\min_{\text{RN}} = +0$. \square

Proof of Property 4. Recalling (7) and taking squares in (10), we obtain $2^{2\lfloor(e_{\min}-p)/2\rfloor} \leq m' \cdot 2^{e'} < 2^{e_{\max}+1}$. On the one hand, $m' < 2$ implies $2\lfloor(e_{\min}-p)/2\rfloor < e' + 1$ and, since both sides are integers, the announced lower bound follows. On the other hand, $1 \leq m'$ implies $e' < e_{\max} + 1$ and, similarly, we deduce the announced upper bound. \square

Proof of Property 5. The lower bound is an immediate consequence of the definition of μ in (14). To establish the claimed upper bound we consider two cases:

- If $\mu = c$ then μ is at most 1 and cannot be larger than $p + \epsilon$, since $p \geq 2$ and $\epsilon \geq 0$.
- If $\mu = e_{\min} - 2e$ then, recalling that $e' = c + 2e$ and noticing that the lower bound in Property 4 equals $e_{\min} - p + \epsilon$ (because e_{\min} is even), we obtain $\mu \leq p + \epsilon + c$. Since both μ and $p + \epsilon$ are even integers, and since c is either 0 or 1, it follows that $\mu \leq p + \epsilon$.

Hence $\mu \leq p + \epsilon$ in both cases, and the proof follows. \square

Proof of Property 6. From (22), $M = m \cdot 2^{k-1}$, and $k = w + p$ it follows that $L = \lfloor y/n \rfloor$ with $y = M^2/2^k$ and $n = 2^{\mu+w-1}$. Since $w \geq 3$ and since, by Property 5, $\mu \geq 0$, n is a positive integer. To conclude it suffices to apply the fact that $\lfloor y/n \rfloor = \lfloor \lfloor y \rfloor / n \rfloor$ for $n > 0$ (see [5, p. 72]). \square

Proof of Property 7. Let q be the number of trailing zeros of $m = (1.m_1 \dots m_{p-1})_2$. Then m^2 can be written

$$m^2 = (s_{-1}s_0.s_1 \dots s_{2p-2q-2})_2 \quad \text{with} \quad s_{2p-2q-2} = 1.$$

Thus, $H = (s_{-1}s_0 \dots s_{k-2})_2$ and, using (15b) and (16), we can also decompose the sticky bit as $t = t_1 \vee t_2$ with

$$t_1 = s_{p-\mu+1} \vee \dots \vee s_{k-2} \quad \text{and} \quad t_2 = s_{k-1} \vee \dots \vee s_{2p-2q-2}.$$

By Property 5 and since $w \geq 3$, we have $k - 2 - (p - \mu + 1) + 1 = \mu + w - 2 \in \{1, \dots, k - 1\}$. Hence $t_1 = 1$ if and only if the last $\mu + w - 2$ bits of H are not all zero, that is, if and only if the integer T_1 obtained by shifting H left by $k - (\mu + w - 2) = p + 2 - \mu$ is nonzero. Since $s_{2p-2q-2} = 1$ we have $t_2 = 0$ if and only if $k - 1 > 2p - 2q - 2$, that is, if and only if the number q of trailing zeros of m is at least $p - \lfloor k/2 \rfloor$. The latter condition is equivalent to $X \bmod 2^{p-\lfloor k/2 \rfloor} = 0$, which concludes the proof. \square

Proof of Property 8. Let $|X|$ and A' denote the standard integer encodings of $|x|$ and α' , respectively. It is known [10, p. 58] that $|x| < \alpha'$ if and only if $|X| < A'$, that is,

$$|X| \leq A' - 1. \quad (26)$$

By (3), $|X| = (E + \epsilon) \cdot 2^{p-1}$ with $\epsilon \in [0, 1 - 2^{1-p}]$ and, by Property 1, $A' = E' \cdot 2^{p-1}$ with $E' = \lfloor (e_{\min} - p)/2 \rfloor + e_{\max}$. Thus, (26) is equivalent to $E \leq E' - (\epsilon + 2^{1-p})$. Since E, E' are integers and $\epsilon + 2^{1-p} \in (0, 1]$, the latter inequality is equivalent to $E \leq E' - 1$. Now, $e_{\min} = 1 - e_{\max}$ gives $E' - 1 = \lfloor (e_{\max} - p - 1)/2 \rfloor$ and we conclude using the fact that $i \leq \lfloor j/2 \rfloor$ is equivalent, for integers i, j , to $2i \leq j$. \square

Proof of Property 9. We use the same notation as in the proof of Property 8 and write O' for the standard integer encoding of Ω' . The special integer encoding used for NaNs gives $C_{\text{large}} \vee C_{\text{nan}} = [|X| \geq O']$. By Property 1 we have $O' = (3e_{\max} + 1)/2 \cdot 2^{p-1}$, so that $|X| \geq O'$ is equivalent to $E + \epsilon \geq (3e_{\max} + 1)/2$. Since $\epsilon \in [0, 1)$, this is equivalent to $E \geq (3e_{\max} + 1)/2$ and the conclusion follows. \square