2011 20th IEEE Symposium on Computer Arithmetic

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

Claude-Pierre Jeannerod[†], Jingyan Jourdan-Lu[‡†], Christophe Monat[‡] and Guillaume Revy[§]
[†]*INRIA, Laboratoire LIP (CNRS, ENS de Lyon, INRIA, UCBL), Université de Lyon, France*
[‡]*STMicroelectronics Compilation Expertise Center, Grenoble, France*
[§]*DALI research team - Université de Perpignan Via Domitia / LIRMM (CNRS, UM2), Perpignan, France*

***Abstract*—We consider the problem of computing IEEE floating-point squares by means of integer arithmetic. We show how to exploit the specific properties of squaring in order to design and implement algorithms that have much lower latency than those for general multiplication, while still guaranteeing correct rounding. Our algorithms 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.**

I. Introduction

The STMicroelectronics ST231 is a 32-bit, 64-register, 4-issue, embedded integer VLIW processor that is mainly used for intensive media and signal processing. As this processor has integer-only register file and ALUs, the single precision floating-point support is available through software emulation, based on the highly optimized FLIP 1.0 library [1]. 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 operators $f(x, y)$ 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. [2].

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. First, we show how to exploit the specific properties of squaring 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 ST231, 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.

For squaring in integer or fixed-point arithmetic several optimized hardware designs have been proposed [3]–[5]. However, for squaring in IEEE floating-point arithmetic much less seems to be available: for example, the software implementations presented in [6], [7] do not mention that operator; furthermore, the implementation of squaring for the binary32 format and the ST231 processor outlined in [8] does not support subnormal numbers, is available only for rounding 'to nearest even,' and has a latency of 27 cycles.

This paper is organized as follows. After some reminders on the ST231 processor and IEEE binary floating point in §II, we show in §III 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 square operator. §§IV and V 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 ST231 are reported in §VI, and proofs of Properties 1 to 9 are available at `http://hal.archives-ouvertes.fr/ensl-00532829/`.

II. Background

A. STMicroelectronics' ST231 processor

The targeted processor is the ST231 from STMicroelectronics. Since it is a scoreboarded 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.

1063-6889/11 $26.00 © 2011 IEEE
DOI 10.1109/ARITH.2011.19

Instruction scheduling is performed after code selection, that must be aggressive to favor high ILP. Specifically, the if-conversion optimization that replaces conditional branches by conditional 'slct' instructions helps creating large straight-line code regions that are subject to efficient scheduling. The latency of this 'slct' instruction is 1 cycle.

Up to 4 independent instructions can be bundled into a *syllable*, achieving the maximum throughput of 4 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 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.

B. IEEE 754-2008 binary floating point

Binary floating-point data. Besides NaNs, signed zeros, and signed infinities, the IEEE 754-2008 standard [9] specifies finite nonzero floating-point numbers as follows: given a precision p and an exponent range $[e_{\min}, e_{\max}]$,

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

where s is either 0 or 1, and where

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

Such numbers must in fact be either *normal* ($m_0 = 1$) or *subnormal* ($m_0 = 0$ and $e = e_{\min}$). Thus, the smallest positive subnormal number is $\alpha = 2^{e_{\min}-p+1}$, while the largest normal number is $\Omega = (2 - 2^{1-p}) \cdot 2^{e_{\max}}$.

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

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

$$e_{\min} = 1 - e_{\max}, \qquad 2 \le p < e_{\max}, \tag{2b}$$

and we shall write $k = w + p$. All the binary$k$ formats of the IEEE 754-2008 standard satisfy these assumptions. 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: $w = 8$, $e_{\min} = -126$, $e_{\max} = 127$, $p = 24$.

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

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

with $E = e - e_{\min} + m_0$. Zeros, infinities, and NaNs are encoded by special values of X: with $|X| = X \bmod 2^{k-1}$,

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

Correct rounding. Besides floating-point data and their encoding into integers, the standard [9, §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), 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 $y \ge 0$.

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 [1] to which we compare, we have not implemented them for square either.

III. Specification and high-level algorithm

As a special case of multiplication, squaring is specified by the IEEE 754-2008 standard: given a rounding mode $\circ$ and with $x = y$, the result r prescribed by [9] for $x \times y$ is

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

Let $\min_\circ$ and $\max_\circ$ denote, respectively, the minimum value and maximum value of $\circ(x^2)$ for $|x|$ in $[\alpha, \Omega]$:

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

(6)

Let us also define the following two quantities:

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

Property 1: The values α' and Ω' 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, $\circ(x^2) = \max_\circ$ iff $|x| \ge \Omega'$.

Property 3: The value α' 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):

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

This brings a natural distinction between two kinds of input:

Definition 1: Input x is called *generic* if $\alpha' \le |x| < \Omega'$, 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.

Setting $C_{\text{spec}} = [x \text{ is special}]$ (with $[\mathcal{S}] = 1$ if $\mathcal{S}$ is true, 0 otherwise) allows us to translate (8) into a high-level algorithmic description based on 3 independent tasks T_i:

evaluate the condition C_{spec}	(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 proceed in 2 steps, as in [10]: assuming unbounded parallelism we optimize the *a priori* most expensive task first, namely task T_3 (see §IV), and then only T_1 and T_2, by trying to reuse as much as possible what was computed for T_3 (see §V). The latency of 'slct' is of 1 cycle, so the lowest latency we can expect for squaring is 1 cycle more than that of T_3.

IV. 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

$$\alpha' \le |x| < \Omega'. \tag{9}$$

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

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

Normalized representation of the exact square. Let $c = [m \ge \sqrt{2}]$, $m' = m^2 \cdot 2^{-c}$ and $e' = c + 2e$. Then $1 \le m' < 2$, and $x^2 = m' \cdot 2^{e'}$ is the so-called *normalized representation* of the exact square. Tight bounds for e' are as follows.

Property 4: The normalized exponent e' of x^2 satisfies $2\lfloor (e_{\min} - p)/2 \rfloor \le e' \le e_{\max}$.

Note that the lower bound is less than $e_{\min}$ since $p \ge 2$.

Correctly-rounded value of the exact square. When $e' \ge e_{\min}$ we have $x^2 \in [2^{e_{\min}}, 2^{e_{\max}+1})$ and, since $m' \in [1, 2)$,

$$\circ(x^2) = \circ(m^2 \cdot 2^{-c}) \cdot 2^{c+2e}. \tag{10a}$$

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:

$$\circ(x^2) = \tilde{\circ}\big(m^2 \cdot 2^{-(e_{\min}-2e)}\big) \cdot 2^{e_{\min}}, \tag{10b}$$

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

To handle (10a) and (10b) simultaneously, let us define

$$\mu = \max(c, e_{\min} - 2e). \tag{11}$$

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

$$\circ(x^2) = \tilde{\circ}(\ell) \cdot 2^d, \tag{12a}$$

where $\ell = (\ell_0.\ell_1 \ldots \ell_{p-1}\ell_p \ldots)_2$ and d satisfy

$$\ell = m^2 \cdot 2^{-\mu} \qquad \text{and} \qquad d = \mu + 2e. \tag{12b}$$

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

Property 5: One has $c \le \mu \le p + \epsilon$ with $\epsilon = \big[p \text{ is odd}\big]$.

An explicit formula for $\tilde{\circ}(\ell)$. Using $\vee$ for logical OR, the *guard bit* and *sticky bit* of ℓ are, respectively,

$$g = \ell_p \qquad \text{and} \qquad t = \ell_{p+1} \vee \ell_{p+2} \vee \cdots. \tag{13}$$

The correctly-rounded value of ℓ is then given by

$$\tilde{\circ}(\ell) = (\ell_0.\ell_1 \ldots \ell_{p-1})_2 + b \cdot 2^{1-p}, \tag{14}$$

where the *round bit* b satisfies (see [4, pp. 436-437])

$$b = \begin{cases} g \wedge (\ell_{p-1} \vee t) & \text{if } \circ = \text{RN}, \\ 0 & \text{if } \circ = \text{RD}, \\ g \vee t & \text{if } \circ = \text{RU}. \end{cases} \tag{15}$$

B. Implementation for the binary$k$ floating-point format

We show here how to implement the computation of $r = \circ(x^2)$ using k-bit integer arithmetic and logic. We assume x given by its standard encoding into an unsigned k-bit integer X. Since r satisfies (12a), it is known (see e.g. [11, §2.3.1]) that its standard integer encoding is $R = D \cdot 2^{p-1} + \widetilde{L}$, where

$$D = d + e_{\max} - 1 \tag{16}$$

and $\widetilde{L} = \tilde{\circ}(\ell) \cdot 2^{p-1}$. Using (14) we get $\widetilde{L} = L + b$ with

$$L = \big\lfloor \ell \cdot 2^{p-1} \big\rfloor \tag{17}$$

and b the rounding bit defined in (15), so that eventually

$$R = D \cdot 2^{p-1} + L + b. \tag{18}$$

Thus, computing R amounts to deducing D, L, b from X, which we detail now for the binary$k$ format. For binary32 and $\circ = \text{RN}$ the resulting C code is displayed in Listing 1.

Listing 1
COMPUTATION OF $\circ(x^2)$ FOR THE BINARY32 FORMAT, $\circ = \mathrm{RN}$, AND x GENERIC.

```
1                                                          T2 = X & 0xff;
2  M = (X << 8) | 0x80000000;      E2 = (X >> 22) & 0x1fe;
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;
```

Computing L. From (12b) and (17) it follows that $L = \lfloor m^2/2^{1-p+\mu} \rfloor$ and thus 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 $p \le k$ we may choose $M = m \cdot 2^{k-1}$, which is obtained from (3) as $M = (X \ll w) \mid 2^{k-1}$. For the binary32 format, this appears at line 2 of Listing 1.

To get μ, recall first that x normal implies $e = E - e_{\text{max}}$. Then, recalling that $e_{\text{min}} = 1 - e_{\text{max}}$ and applying (11),

$$\mu = \max(c, F), \qquad F = e_{\text{max}} + 1 - 2E. \tag{19}$$

To get c, it suffices to note that $c = [m \ge \sqrt{2}]$ and $M = m \cdot 2^{k-1}$ imply $c = [M > M_0]$ with $M_0 = \lfloor \sqrt{2} \cdot 2^{k-1} \rfloor$.

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

$$2E = \big(X \gg (p-2)\big) \,\&\, (2^{w+1} - 2), \tag{20}$$

and then we subtract $2E$ from the constant $e_{\text{max}} + 1$. For the binary32 format the computation of c and F appears at line 3 of Listing 1, where $(b504f333)_{16}$ is M_0 for $k = 32$.

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 $p = 24$ is even, Property 5 implies $\mu + 7 \le 31$, which agrees with the C99 specification of the bitwise shift operator [12, p. 84].

Computing b. We focus here on rounding to nearest even, for which $b = g \wedge (\ell_{p-1} \vee t)$ with g and t as in (13). Note first that g is the least significant bit of $G = \lfloor \ell \cdot 2^p \rfloor = \sum_{0 \le i \le p} \ell_i 2^{p-i}$ and that, similarly to Property 6, we have $g = G \bmod 2$ and $G = \lfloor H/2^{\mu+w-2} \rfloor$. For binary32, the corresponding C code appears at lines 6 and 9 of Listing 1. Second, since ℓ_{p-1} is the least significant bit of L we have $\ell_{p-1} = L \bmod 2$. Third, the sticky bit t shall be obtained without computing the lower half of the exact square M^2:

Property 7: One has $t = [T_1 \ne 0] \vee [T_2 \ne 0]$, where $T_1 = H \ll (p + 2 - \mu)$ and $T_2 = X \bmod 2^{p - \lfloor k/2 \rfloor}$.

For the binary32 format, Property 7 gives $T_2 = X \bmod 2^8$, which is implemented 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.

Computing D. We have $D = \mu + 2e + e_{\text{max}} - 1$ because of (12b) and (16). Using $e = E - e_{\text{max}}$ and (19) then gives $D = \mu - F$. For binary32, this appears at line 10 of Listing 1.

Packing the result. From (18) 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 can be 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 (15) is zero. Thus, 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 $T1 \,|\, T2$. In this case, we thus replace line 9 of Listing 1 by:

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

With these codes the expected latency of R on ST231 drops from 10 to 7 (resp. 9) cycles for $\circ = \mathrm{RD}$ (resp. RU).

V. DETECTING AND HANDLING SPECIAL INPUT

We first have to decide whether input x is special or not, that is, to get from X the value of $C_{\text{spec}} = [x \text{ is special}]$. By Definition 1 we have $C_{\text{spec}} = C_{\text{small}} \vee C_{\text{large}} \vee C_{\text{nan}}$ with $C_{\text{small}} = \big[|x| < \alpha'\big]$, $C_{\text{large}} = \big[|x| \ge \Omega'\big]$, and $C_{\text{nan}} = \big[x \text{ is NaN}\big]$. 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 (20) 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. As C_{spec} is independent of $\circ$, this code holds not only for $\circ = \text{RN}$ but also for $\circ \in \{\text{RD}, \text{RU}\}$.

Listing 2. SPECIAL CASES FOR THE BINARY32 FORMAT AND $\circ = \text{RN}$.

```
1                          absX = X & 0x7fffffff;
2  E2 = (X >> 22) & 0x1fe; Cnan = absX > 0x7f800000;
3  Csmall = E2 <= 102;    Clarge_or_nan = E2 >= 382;
4  Cspec = Csmall || Clarge_or_nan;
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 { ... }         // generic case (Listing 1)
```

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 (4) implies that 0, $2^{k-1} - 2^{p-1}$, and $2^{k-1} - 2^{p-2}$ are standard encodings of $+0$, $+\infty$, and qNaNs, and that $C_{\text{nan}} = \left[X \,\&\, (2^{k-1} - 1) > 2^{k-1} - 2^{p-1}\right]$. For the binary32 format, the computation of C_{nan} appears at lines 1 and 2 of Listing 2, while lines 6 to 9 display the computation of R.

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 $2^{k-1} - 2^{p-1} - 1$, that is $(7f7fffff)_{16}$ for the binary32 format. Hence 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);
```

VI. EXPERIMENTAL RESULTS OBTAINED ON ST231

The C codes detailed in §§IV and V 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 `mul`, `max`, `maxu`, and `minu` as in [11, 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. The table below displays the performances obtained in this context.

In the second column, leftmost numbers are the latencies L (in clock cycles) of our codes. The values within square brackets are the lowest latencies theoretically achievable with the ST231 latency constraints and assuming unbounded parallelism; these best latencies follow from our analysis in §§IV and V and have the form $\mathcal{L}+1$ with $\mathcal{L}$ the best latency for the generic case: the latencies achieved in practice are thus at most 1 cycle from the best possible ones. The values in parentheses are instruction numbers N and IPC values N/L. The IPC achieved is close to the highest ILP reachable with the architectural constraints of the machine.

$\circ$	square	FLIP 1.0 multiply	speedup
RN	12 [11] (42, 3.5)	21	1.75
RD	9 [8] (31, 3.4)	21	2.3
RU	11 [10] (37, 3.4)	21	1.9
RZ	9 [8] (31, 3.4)	18	2

For comparison, the third column gives the latencies of the multiply operator of FLIP 1.0, optimized for the ST231 [1].

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.

REFERENCES

[1] C.-P. Jeannerod and G. Revy, "FLIP 1.0," Feb. 2009, available at http://flip.gforge.inria.fr/.

[2] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery, *Numerical Recipes (3rd edition)*. Cambridge U. Press, 2007.

[3] E. G. Walters, J. Schlessman, and M. J. Schulte, "Combined unsigned and two's complement hybrid squarers," in *IEEE Proc. of the 35th Conf. on Signals, Systems, and Computers*, Asilomar, Pacific Grove, CA, USA, 2001, pp. 861–866.

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

[5] M. Gök, "Integer squarers with overflow detection," *Comp. & Electrical Engineering*, vol. 34, no. 5, pp. 378–391, 2008.

[6] C. Iordache and P. T. P. Tang, "An overview of floating-point support and math library on the Intel XScale™ architecture." in *IEEE Proc. of ARITH-16*, 2003, pp. 122–128.

[7] N. Sidwell and J. Myers, "Improving Software Floating Point Support," in *Proc. of the GCC Developers' Summit 2006*, Ottawa, Canada, 2006, pp. 211–218.

[8] S.-K. Raina, "FLIP: a floating-point library for integer processors," Ph.D. dissertation, ENS Lyon, France, Sep. 2006.

[9] IEEE Computer Society, *IEEE Standard for Floating-Point Arithmetic*. IEEE Standard 754-2008, Aug. 2008.

[10] C. Bertin, C.-P. Jeannerod, J. Jourdan-Lu, H. Knochel, C. Monat, C. Mouilleron, J.-M. Muller, and G. Revy, "Techniques and tools for implementing IEEE 754 floating-point arithmetic on VLIW integer processors," in *ACM Proc. of PASCO'10*. Grenoble, France, 2010, pp. 1–9.

[11] G. Revy, "Implementation of binary floating-point arithmetic on embedded integer processors: polynomial evaluation-based algorithms and certified code generation," Ph.D. dissertation, Université de Lyon - ÉNS de Lyon, France, Dec. 2009.

[12] International Organization for Standardization, *Programming Languages – C*. ISO/IEC Standard 9899:1999, Dec. 1999.