Some Functions Computable with a Fused-mac

Sylvie Boldo, Jean-Michel Muller

Laboratoire de l'Informatique du Parallélisme

École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON-UCBL n° 5668

Some Functions Computable with a Fused-mac

Sylvie Boldo ,
Jean-Michel Muller

Septembre 2004

Research Report N° 2004-41

École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.37
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : `lip@ens-lyon.fr`

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE

INRIA

Some Functions Computable with a Fused-mac

Sylvie Boldo , Jean-Michel Muller

Septembre 2004

Abstract

The fused multiply accumulate instruction (fused-mac) that is available on some current processors such as the Power PC or the Itanium eases some calculations. We give examples of some floating-point functions (such as $\mathrm{ulp}(x)$ or $\mathrm{Nextafter}(x, y)$), or some useful tests, that are easily computable using a fused-mac. Then, we show that, with rounding to the nearest, the error of a fused-mac instruction is exactly representable as the sum of two floating-point numbers. We give an algorithm that computes that error.

Keywords: Floating-point arithmetic, fused multiply accumulate, computer arithmetic

Résumé

L'instruction "fused-mac" (multiplication-addition regroupées), qui est disponible sur certains processeurs récents comme le Power PC ou l'Itanium facilite certains calculs. Nous donnons ici quelques exemples de fonctions virgule flottante (comme $\mathrm{ulp}(x)$ ou $\mathrm{Nextafter}(x, y)$), ou de tests, qui sont facilement implantables avec un fused-mac. Nous montrons ensuite qu'en arrondi au plus proche, l'erreur d'une instruction fused-mac est exactement représentable comme somme de deux nombres virgule flottante. Nous donnons un algorithme calculant cette erreur.

Mots-clés: Arithmétique virgule flottante, multiplieur-additionneur, arithmétique des ordinateurs

1 Introduction

The fused multiply accumulate instruction (fused-mac) is available on some current processors such as the IBM Power PC or the Intel/HP Itanium. That instruction computes an expression $ax + b$ or more generally $\pm ax \pm b$ with one final rounding error only. This makes it possible to perform correctly rounded division using Newton-Raphson division [17, 7, 16] (the main idea behind that is that if q approximates x/y with enough accuracy, then the remainder $x - yq$ will be exactly computed with a fused-mac, allowing to correct the quotient estimation). Also, this makes evaluation of scalar products and polynomials faster and, generally, more accurate than with conventional (addition and multiplication) floating-point operations. This is important, since scalar products appear everywhere in linear algebra, and since polynomials are very often used for approximating functions.

It has been known for three decades [9] that (assuming rounding to nearest) the error of a floating point addition or a floating-point multiplication in a given format is exactly representable as a floating-point number of the same format. This is also true for the remainder of a division or a square root with any rounding mode [2, 3]. A natural question arises: is there a similar property for the fused-mac operation?

Also, expert floating-point programming sometimes requires the evaluation of functions such as $\mathrm{Nextafter}(x, y)$, or the successor of a given floating-point number, or (for error estimation), $\mathrm{ulp}(x)$. We may also, for some calculations, need to know if the last mantissa bit of a number is a zero [4]. These various functions can always be computed at a low level, using masks and integer arithmetic: this results in software that is not portable, and sometimes quite slow, since the corresponding calculations are not performed in the floating-point pipeline. With conventional arithmetic, designing portable software for these functions is feasible [5] but might be costly. We aim at showing that the availability of a fused-mac instruction facilitates portable yet efficient implementation of such functions.

2 Definitions and notations

Define $\mathbb{M}_n$ as the set of exponent-unbounded, n-bit mantissa, binary FP numbers (with $n \geq 1$), that is: $\mathbb{M}_n = \left\{M \times 2^E, 2^{n-1} \leq M \leq 2^n - 1, M, E \in \mathbb{Z}\right\} \cup \{0\}$. It is an "ideal" system, with no overflows or underflows. We will show results in $\mathbb{M}_n$. These results will remain true in actual systems that implement the IEEE-754 standard [6, 1], provided that no overflows or underflows do occur. The **mantissa** of a nonzero element $M \times 2^E$ of $\mathbb{M}_n$ is the number $m(x) = M/2^{n-1}$, its **integral mantissa**, noted M_x is M and its corresponding **exponent**, noted e_x is E.

We assume that the reader is familiar with the basic notions of floating-point (FP, for short) arithmetic: rounding modes, ulps, ... See [10] for definitions. In the following $\circ(t)$ means t rounded to the nearest even.

3 Previous results and preliminary properties

We will use the 2sum and Fast2Sum algorithm, presented below. These algorithms do not require the availability of a fused-mac. They make it possible to compute the error of a floating-point addition exactly, represented by a FP number. The first one [14, 18] only assumes a and b are normalized FP numbers (i.e., elements of $\mathbb{M}_n$).

Property 1 (2Sum Algorithm) *Let $a, b \in \mathbb{M}_n$. Define x and y as*

$$\begin{array}{rcl} x & = & \circ(a+b) \\ b' & = & \circ(x-a) \\ a' & = & \circ(x-b') \\ \epsilon_b & = & \circ(b-b') \\ \epsilon_a & = & \circ(a-a') \\ y & = & \circ(\epsilon_a+\epsilon_b) \end{array}$$

We have:

- $x + y = a + b$ *exactly;*
- $|y| \leq \frac{1}{2}\mathrm{ulp}(x)$.

If we know in advance that $|a| \geq |b|$ (as a matter of fact, it suffices to have $e_a \geq e_b$), a much faster algorithm can be used [9, 14]:

Property 2 (Fast2Sum Algorithm) *Let $a, b \in \mathbb{M}_n$, with $|a| \geq |b|$. Define x and y as*

$$\begin{array}{rcl} x & = & \circ(a+b) \\ b' & = & \circ(x-a) \\ y & = & \circ(b-b') \end{array}$$

We have:

- $x + y = a + b$ *exactly;*
- $|y| \leq \frac{1}{2}\mathrm{ulp}(x)$.

Although we have presented these properties assuming a radix-2 number system, it is worth being noticed that the 2Sum algorithm (property 1) works in any radix ≥ 2, and that the Fast2Sum algorithm (property 2) works in radices 2 and 3. And yet, rounding to nearest is mandatory: with "directed" roundings it is possible [14] to exhibit cases where the difference between the computed value of $a + b$ and the exact value cannot be exactly expressed as a FP number.

The 2Sum algorithm satisfies the following property, that will be needed in Section 5.

Property 3 *If $(x, y) = 2\mathrm{Sum}(a, b)$ then $|y| \leq |b|$.*

Proof. x is the FP number that is closest to $(a + b)$. This implies that x is closer to $(a + b)$ than a. Hence, $|(a + b) - x| = |y|$ is smaller than $|(a + b) - a| = |b|$. □

A well known and useful property of the fused-mac instruction, noticed by Karp and Markstein [13], is that it allows to very quickly compute the product of two FP numbers x and y exactly, expressed as the sum of two FP numbers u and v. More precisely,

Property 4 (Fast2Mult Algorithm) *Let $a, b \in \mathbb{M}_n$. Define x and y as*

$$\begin{array}{rcl} x & = & \circ(ab) \\ y & = & \circ(ab-x) \end{array}$$

we have:

- $x + y = ab$ *exactly;*
- $|y| \leq \frac{1}{2}\mathrm{ulp}(x)$.

Without a fused-mac, computing x and y is possible, but requires much more computation [9] (the mantissas of x and y are splitted, then partial products are computed and summed up).

4 Basic functions computable with a fused-mac

4.1 Checking if the last mantissa bit of some number is a zero

Brisebarre, Muller and Raina [4] have suggested an algorithm for division by a constant that works when the last bit of the divisor mantissa is a zero. Checking that condition is easily done with a fused-mac.

Property 5 (Algorithm IsEven) *The following algorithm on x checks if the last mantissa bit of x is a zero.*

$$\begin{array}{lcl} \alpha & = & \circ(3x) \\ \beta & = & \circ(\alpha - 2x) \\ \text{IsEven} & = & (\beta = x) \end{array}$$

One may notice that the same algorithm also works with the usual (addition and multiplication) floating-point instructions. The availability of a fused-mac, here, only saves one operation.

4.2 Checking if a number is a power of 2.

The following algorithm requires storage of the constant

$$C = 2^n - 1.$$

Of course, $C \in \mathbb{M}_n$: it is exactly representable as a floating-point number.

Property 6 (Algorithm IsAPowerOf2) *The following algorithm on x returns "true" if x is a power of* 2.

$$\begin{array}{lcl} y_h & = & \circ(xC) \\ y_\ell & = & \circ(xC - y_h) \\ \text{IsAPowerOf2} & = & (y_\ell = 0). \end{array}$$

Proof if x is not a power of 2 then M_x has at least a prime factor different from 2, thus M_xC is of the form $P2^\alpha$, where P is odd and larger than 2^n. Hence P cannot be exactly representable with n bits, hence $y_h \neq xC$, hence $y_\ell \neq 0$. □

Important remark The above given algorithm works in the "ideal" set $\mathbb{M}_n$, which means that with "real world" floating-point arithmetic it will work provided that no overflow or underflow occur. To minimize the risk of overflow/underflow, one should choose

$$C = (2^n - 1)/(2^n),$$

instead of the previously given constant. The proof will be the same, overflow will never occur, and underflow will occur only where x is a subnormal FP number.

4.3 Floating-point successors

There are several notions of "floating-point successor" that can be defined. The IEEE-754 standard for FP arithmetic[1] [1] *recommends* (but does not *require*) the availability of the function `Nextafter`. $\text{Nextafter}(x, y)$ returns the next representable neighbor of x in the direction toward y. If $x = y$, then the result is x without any exception being signaled. If either x or y is a NaN, then the result is a NaN. Overflow is signaled when x is finite but $\text{Nextafter}(x, y)$ is infinite; underflow is signaled when the result is subnormal or zero. Cody and Coonen [5] provide a portable C version of that function.

Let us show how such a function can be implemented using fused-mac instructions. First, define the following four functions.

[1] See `http://754r.ucbtest.org/standards/754.txt`

Definition 1 *The* successor *of a FP number x, denoted x^+ is the smallest FP number larger than x. The* predecessor *x^- of x is the largest FP number less than x. The* symmetrical successor *of x, denoted $\mathrm{succ}(x)$ is x^- if $x < 0$, and x^+ if $x > 0$. The* symmetrical predecessor *$\mathrm{pred}(x)$ of x is x^+ if $x < 0$ and x^- if $x > 0$.*

The following algorithm will use the constant

$$s = 2^{-n} + 2^{-2n+1}.$$

Notice that $s \in \mathbb{M}_n$. Even on "real life" floating-point systems, s will be representable: on all floating-point systems of current use, the number of mantissa bits is less than the absolute value of the smallest exponent. This is required by the IEEE-854 Standard for Floating-Point arithmetic [12], that says that $(E_{max} - E_{min})/n$ shall exceed 5 and should exceed 10, and that $b^{E_{max}+E_{min}+1}$ should be the smallest integral power of b, where b is the radix.

Property 7 *Computation of $\mathrm{succ}(x)$ If $n \geq 2$, then*

$$\mathrm{succ}(x) = \circ(x + sx)$$

Proof Assume $2^e \leq x < 2^{e+1}$ (i.e., the exponent of x is e). Since, in that case, $\mathrm{succ}(x) = x + 2^{e-n+1}$ and $\mathrm{ulp}(x) = 2^{e-n+1}$, to show that $\circ(x + sx)$ is equal to $\mathrm{succ}(x)$ it suffices to show that

$$x + 2^{e-n} < x + sx < x + 3 \times 2^{e-n}$$

(i.e., that $x + sx$ is within $1/2\mathrm{ulp}$ from $\mathrm{succ}(x)$).

Thus, it suffices to show that

$$2^{e-n} < sx < 3 \times 2^{e-n}. \tag{1}$$

Since $x \geq 2^e$, $sx > 2^{e-n}$. Since $x < 2^{e+1}$, $sx < 2(1 + 2^{-n+1})2^{e-n}$, which is less than 3.2^{e-n} as soon as $n \geq 2$. □

Property 7 shows that $\mathrm{succ}(x)$ can be computed with one fused-mac only.

Function $\mathrm{pred}(x)$ is also computable with one fused-mac only. The proof is very similar to that of Property 7.

Property 8 *Computation of $\mathrm{pred}(x)$ If $n \geq 2$, then*

$$\mathrm{pred}(x) = \circ(x - sx)$$

Now, from functions succ and pred, one can very easily compute functions `Nextafter`, x^+ and x^-:

Property 9

$$\begin{array}{lll} x^+ & = & \circ(x + s|x|) \\ x^- & = & \circ(x - s|x|) \\ \mathrm{Nextafter}(x, y) & = & \begin{cases} x^+ & \text{if} \quad y > x \\ x & \text{if} \quad y = x \\ x^- & \text{if} \quad y < x \end{cases} \end{array}$$

Important remark: although we have proven these algorithms assuming an ideal FP arithmetic with unbounded exponents, they work well with "real life" arithmetic. From the definition of $\mathrm{succ}(x)$, underflow is impossible. Also, if $|x|$ is equal to the largest representable FP number, then on a machine compliant with the IEEE 754 standard, $\pm\infty$ (depending on the sign of x) will be returned[2], which is the right answer. If x is a NaN, then the fused-mac operation will return

[2]This is due to the definition of rounding to the nearest: the standard specifies that *An infinitely precise result with magnitude at least $2^{Emax}(2 - 2^{-n})$ shall round to ∞ with no change in sign.*

a NaN. Hence, our algorithm for $\mathrm{succ}(x)$ is always correct, unless x is a subnormal number. Function $\mathrm{pred}(x)$ cannot generate an overflow, correctly propagates NaNs, and correctly signal underflows, however, it does not work correctly if x is a subnormal number: that (rare) case should be handled separately.

If we use rounding to nearest, then the availability of a fused-mac instruction is mandatory for designing such algorithms. For example:

Property 10 *Apart from the "toy case" $n \leq 2$, there is no constant $C \in \mathbb{M}_n$ such that $\circ(xC)$ always equals* $\mathrm{succ}(x)$.

Proof: Suppose that there exists $C \in \mathbb{M}_n$ such that $\circ(xC)$ always equals $\mathrm{succ}(x)$. Assume $1 \leq x < 2$ (the other cases are easily deduced from this one). This implies

$$x + 2^{-n} \leq Cx \leq x + 3.2^{-n}.$$

Hence,

$$2^{-n} \leq (C-1)x \leq 3.2^{-n}$$

for any $x \in \mathbb{M}_n$, $1 \leq x < 2$. For $x = 1$, this implies $C \geq 1 + 2^{-n}$. Since the smallest element of $\mathbb{M}_n$ larger than or equal to $1 + 2^{-n}$ is $1 + 2^{-n+1}$, we then have $C \geq 1 + 2^{-n+1}$. And yet, for x equal to the largest element of $\mathbb{M}_n$ less than 2 (i.e., $2 - 2^{-n+1}$), $C \geq 1 + 2^{-n+1}$ implies $(C-1)x \geq 2^{-n+1}(2 - 2^{-n+1}) = 4.2^{-n} - 2^{-2n+2}$. Therefore, in that case, $(C-1)x > 3.2^{-n}$, unless $n \leq 2$. □

This may be different with other rounding modes. For instance, if rounding towards zero $\mathcal{Z}(x)$ is used, then $\mathcal{Z}(x\sigma)$ returns $\mathrm{pred}(x)$ for any $x \in \mathbb{M}_n$, with $\sigma = 1 - 2^{-n}$. And yet, in practice, changing the rounding mode may be quite time consuming: this is why an algorithm that works in the default mode (i.e., round-to-nearest) is preferable.

4.4 Function $\mathrm{ulp}(x)$

Function ulp (unit in the last place) is very frequently used for expressing the accuracy of a floating-point result. Several definitions have been given(see [11] for a discussion on that topic), they differ near the powers of 2. If we use as a definition, when x is a FP number:

$$\mathrm{ulp}(x) = |x|^{+} - |x|$$

then one can compute function ulp through the following sequence

$$\begin{array}{rcl} y & = & \circ(x + sx) \\ \mathrm{ulp} & = & |y - x| \end{array}$$

where s is the same constant as in Section 4.3. If we define $\mathrm{ulp}(x)$ as

$$\mathrm{ulp}(x) = |x| - |x|^{-}$$

then function ulp is computed through

$$\begin{array}{rcl} y & = & \circ(x - sx) \\ \mathrm{ulp} & = & |y - x| \end{array}$$

The two functions differ only when x is a power of 2. The first one is compatible with Goldberg's definition [10] (which is given for *real* numbers, not only for floating-point ones), the second is compatible with Kahan's one[3] and Harrison's one [11] (they differ for real numbers but coincide on FP numbers).

[3] Kahan's definition is: $\mathrm{ulp}(x)$ is the gap between the two finite floating-point numbers nearest x , even if x is one of them (But ulp(NaN) is NaN .)

5 Computing the error term of a fused-mac

We require here that $n \geq 3$. The correcting term cannot be a single FP number, even in rounding to the nearest. We will therefore compute two FP numbers such that their sum is the exact correcting term of the fused-mac.

5.1 The algorithm ErrFmac

Property 11 (Algorithm ErrFmac) *Let a, x, $y \in \mathbb{M}_n$. Define r_1, r_2 and r_3 as*

$$\begin{array}{lcl} r_1 & = & \circ(ax+y) \\ (u_1,u_2) & = & \text{Fast2Mult}(a,x) \\ (\alpha_1,\alpha_2) & = & \text{2Sum}(y,u_2) \\ (\beta_1,\beta_2) & = & \text{2Sum}(u_1,\alpha_1) \\ \gamma & = & \circ(\circ(\beta_1-r_1)+\beta_2) \\ (r_2,r_3) & = & \text{Fast2Sum}(\gamma,\alpha_2) \end{array}$$

we have:

- $ax + y = r_1 + r_2 + r_3$ *exactly;*
- $|r_2 + r_3| \leq \frac{1}{2}\text{ulp}(r_1)$*;*
- $|r_3| \leq \frac{1}{2}\text{ulp}(r_2)$*.*

Figure 1 gives the idea behind the algorithm: we want to exactly add the 3 FP numbers y, u_1 and u_2. This is usually difficult, but as we know the correct answer (r_1) thanks to the fused-mac computation, we just have to get the two error terms. We first compute the "small" error, namely α_2. Then the other terms u_1 and α_1 are bigger than this value and can be combined with r_1 into a single value γ.

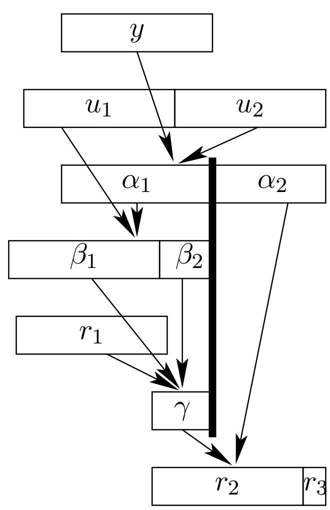

Figure 1: Intermediate values of the ErrFmac algorithm.

5.2 Proof of the correctness of the ErrFmac algorithm

If $\gamma = \circ(\circ(\beta_1 - r_1) + \beta_2)$ is equal to $(\beta_1 - r_1) + \beta_2$, then $r_1 + r_2 + r_3 = r_1 + \gamma + \alpha_2 = r_1 + \beta_1 - r_1 + \beta_2 + \alpha_2 = u_1 + \alpha_1 + \alpha_2 = u_1 + u_2 + y = y + ax$. If this equality holds, we easily also have that $|r_2 + r_3| \leq \frac{1}{2}\text{ulp}(r_1)$ and $|r_3| \leq \frac{1}{2}\text{ulp}(r_2)$.

There is left to prove that $\beta_1 - r_1$ and $(\beta_1 - r_1) + \beta_2$ are in $\mathbb{M}_n$. If they are, then they are exactly computed and the algorithm is correct. To guarantee that a value v is in $\mathbb{M}_n$, we just have to find an exponent e such that $v2^{-e}$ is an integer and $|v2^{-e}| < 2^n$. There may exist more than one suitable e, but the existence of one is enough. We split the proof into two subcases.
If we have $\beta_2 = 0$,

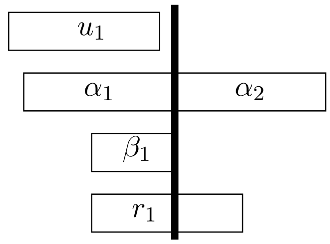

Figure 2: Intermediate values of of the ErrFmac algorithm when $\beta_2 = 0$.

Figure 2 reminds the compared positions of the FP numbers involved. As $\beta_2 = 0$, we have left to prove that $\beta_1 - r_1$ is in $\mathbb{M}_n$. If $\beta_1 = 0$, then this is correct. Let us assume that $\beta_1 \neq 0$. We then know that $r_1 = \circ(\beta_1 + \alpha_2)$ as $\beta_2 = 0$.

But we also have that $|\alpha_2| \leq \frac{1}{2}\text{ulp}(\alpha_1)$ from Property 1 and that $|\alpha_2| \leq |u_2| \leq \frac{1}{2}\text{ulp}(u_1)$ from Property 3 and by definition $\beta_1 = \circ(u_1 + \alpha_1)$. This means that $|\alpha_2| \ll |\beta_1|$. More precisely, we either have:

- the general case: $|\beta_1| \geq 4\,|\alpha_2|$;
- the special case where β_1 is a result of a near-total cancellation: $\beta_1 = 2^{\min(e_{u_1}, e_{\alpha_1})}$ and $|\beta_1| \geq 2\,|\alpha_2|$.

In the general case, we are in the conditions of Sterbenz's theorem [19]: r_1 and β_1 share the same sign and

$$
\begin{aligned}
|r_1| &\leq \frac{|\beta_1 + \alpha_2|}{1 - 2^{-n}} \leq \frac{5}{4}\frac{1}{1-2^{-n}}|\beta_1| \leq 2\,|\beta_1| \\
|r_1| &\geq \frac{|\beta_1 + \alpha_2|}{1 + 2^{-n}} \geq \frac{3}{4}\frac{1}{1+2^{-n}}|\beta_1| \geq \frac{1}{2}|\beta_1|
\end{aligned}
$$

In the special case, we have $4\,|\alpha_2| > |\beta_1| \geq 2\,|\alpha_2|$. As β_1 is a power of 2, we know that $e_{\beta_1} - 1 \leq e_{r_1} \leq e_{\beta_1}$, so e_{r_1} is a suitable exponent for $\beta_1 - r_1$ and

$$
\begin{aligned}
|\beta_1 - r_1|2^{-e_{r_1}} &= |\beta_1 - \circ(\beta_1 + \alpha_2)|2^{-e_{r_1}} \\
&\leq \left(\frac{1}{2}\text{ulp}(r_1) + |\alpha_2|\right) 2^{-e_{r_1}} \\
&\leq \frac{1}{2} + |\beta_1|2^{-e_{r_1}-1} \\
&\leq \frac{1}{2} + (2^n - 1)2^{e_{r_1}+1-e_{r_1}-1} < 2^n.
\end{aligned}
$$

If we have $\beta_2 \neq 0$,

Figure 3 reminds the compared positions of the FP numbers involved. In the general case, we have here that $\beta_1 = r_1$, then of course $\beta_1 - r_1 = 0$ and $(\beta_1 - r_1) + \beta_2 = \beta_2$ are in $\mathbb{M}_n$. If not, as $\beta_2 \neq 0$, the only possibility for $\beta_1 = \circ(\beta_1 + \beta_2)$ not to be equal to $\circ(\beta_1 + \beta_2 + \alpha_2) = r_1$ is that either $|\beta_2| = \frac{1}{2}\text{ulp}(\beta_1)$ or $\beta_2 = -\frac{1}{4}\text{ulp}(\beta_1)$ if β_1 is a power of 2.

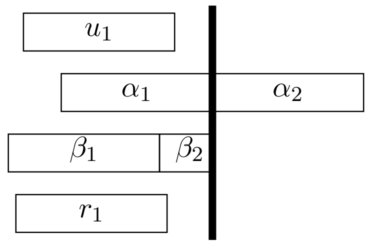

Figure 3: Intermediate values of of the $\mathrm{ErrFmac}$ algorithm when $\beta_2 \neq 0$.

We also deduce that the exponent of r_1 and of β_1 differ from at most 1. Lastly, we know that $|\alpha_2| \leq |\beta_2| \leq 2^{e_{\beta_1}-1}$. The value $\min(e_{r_1}, e_{\beta_1})$ is a suitable exponent for $\beta_1 - r_1$ and

$$\begin{aligned} |\beta_1 - r_1| 2^{-\min(e_{r_1}, e_{\beta_1})} &= |\beta_1 - \circ(\beta_1 + \beta_2 + \alpha_2)| 2^{-\min(e_{r_1}, e_{\beta_1})} \\ &\leq \left(\frac{1}{2}\mathrm{ulp}(r_1) + |\beta_2| + |\alpha_2|\right) 2^{-\min(e_{r_1}, e_{\beta_1})} \\ &\leq \left(2^{e_{r_1}-1} + 2^{e_{\beta_1}-1} + 2^{e_{\beta_1}-1}\right) 2^{-\min(e_{r_1}, e_{\beta_1})} \leq 4 \end{aligned}$$

So $\beta_1 - r_1 \in \mathbb{M}_n$ as $n \geq 3$. There is left to prove that $(\beta_1 - r_1) + \beta_2 = u_1 + \alpha_1 - r_1$ is in $\mathbb{M}_n$. We know that $e_{\beta_1} + 1 \geq e_{r_1} \geq e_{\beta_1} - 1$ and that β_2 is either $2^{e_{\beta_1}-1}$ or $2^{e_{\beta_1}-2}$, so $e_{\beta_1} - 2$ is a suitable exponent for $(\beta_1 - r_1) + \beta_2$ and

$$\begin{aligned} |(\beta_1 - r_1) + \beta_2| 2^{-e_{\beta_1}+2} &= |u_1 + \alpha_1 - \circ(u_1 + \alpha_1 + \alpha_2)| 2^{-e_{\beta_1}+2} \\ &\leq \left(\frac{1}{2}\mathrm{ulp}(r_1) + |\alpha_2|\right) 2^{-e_{\beta_1}+2} \\ &\leq \left(2^{e_{r_1}-1} + 2^{e_{\beta_1}-1}\right) 2^{-e_{\beta_1}+2} \leq 6 \end{aligned}$$

So $(\beta_1 - r_1) + \beta_2 \in \mathbb{M}_n$ as $n \geq 3$. □

5.3 With other rounding modes

Such correcting terms for the fused-mac are only representable when the rounding is to the nearest. For example, when rounding up, if $a = x = 2^n - 1$ and $y = 2^{4n}$ then $ax + y = 2^{4n} + 2^{2n} - 2^{n+1} + 1$ and therefore r_1 must be strictly greater than 2^{4n} so $r_1 = \triangle(ax + y) = 2^{4n} + 2^{3n+1}$. So $r_2 + r_3$ should be exactly equal to $-2^{3n+1} + 2^{2n} - 2^{n+1} + 1$ that cannot be represented as the sum of two FP numbers in $\mathbb{M}_n$.

5.4 Cost of the algorithm

The basic cost of the algorithm is 20 cycles, but this can be tremendously reduced.

The first enhancement is when we know that $|y| \geq |ax|$ or that $|y| \geq |u_1|$. Then, the first $\mathrm{2Sum}$ is useless as $\alpha_1 = y$ and $\alpha_2 = u_2$. This is typically the case in range reduction [8, 15].

The second enhancement is to get rid of the final $\mathrm{Fast2Sum}$: this means that the result will not be compressed. It means that we only have:

- $ax + y = r_1 + r_2 + r_3$ exactly;
- $|r_2 + r_3| \leq \frac{1}{2}\mathrm{ulp}(r_1)$;
- $r_2 = 0$ or $|r_2| > |r_3|$.

The last enhancement is if the processor can use several floating-point units (FPUs) in parallel. There are indeed several computations that can be done either at the same time or at consecutive steps in a pipe-line, as there is no dependence between them. For example, the computations of a' and ϵ_b in the $\mathrm{2Sum}$ algorithm (Property 1) can be done in parallel.

If 3 FPUs are available, the algorithm only costs 12 cycles. The tasks given to each processor are given in Figure 4. More FPUs are useless to speed up the algorithm.

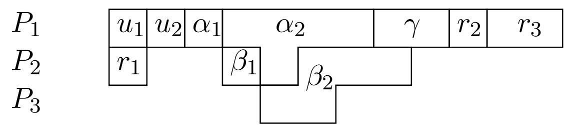

Figure 4: Task repartition when 3 FPUs are available.

If only 2 FPUs are available, the algorithm costs 14 cycles. The tasks given to each processor are shown in Figure 5.

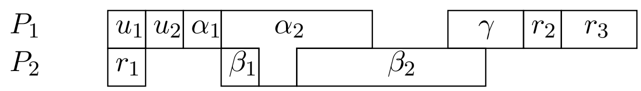

Figure 5: Task repartition when 2 FPUs are available.

The following table gives the cost of the $\mathrm{ErrFmac}$ algorithm depending on the conditions (number of FPUs, final compression and knowledge that the inequality $|y| \geq |ax|$ holds):

Cost (in cycles)	**1 FPU**	**2 FPUs**	**3 FPUs**
Given algorithm	20	14	12
Without the final compression	17	11	9
When $\lvert y\rvert \geq \lvert ax\rvert$	14	10	10
When $\lvert y\rvert \geq \lvert ax\rvert$ and without compression	11	7	7

6 Conclusion

We have shown that the fused-mac instruction makes it possible to implement efficiently and in a portable way many functions that are useful for expert floating-point programming. We also have shown that the error of a fused-mac operation in a given format is exactly representable as a sum of two floating-point numbers of the same format. We have given a fast and portable algorithm that returns that error. We can take advantage of this algorithm for implementing a very accurate range reduction.

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] G. Bohlender, P. Kornerup, D. W. Matula, and W. Walter. Semantics for exact floating-point operations. In P. Kornerup and D. W. Matula, editors, *Proceedings of the 10th IEEE Symposium on Computer Arithmetic*, pages 22–26, Grenoble, France, June 1991. IEEE Computer Society Press, Los Alamitos, CA.

[3] Sylvie Boldo and Marc Daumas. Representable correcting terms for possibly underflowing floating point operations. In Jean-Claude Bajard and Michael Schulte, editors, *Proceedings*

of the 16th Symposium on Computer Arithmetic, pages 79–86, Santiago de Compostela, Spain, 2003.

[4] N. Brisebarre, J.-M. Muller, and S. Raina. Accelerating correctly rounded floating-point division when the divisor is known in advance. *IEEE Transactions on Computers*, 53(8):1069–1072, August 2004.

[5] W. J. Cody and J. T. Coonen. Algorithm 722: Functions to support the IEEE standard for binary floating-point arithmetic. *ACM Transactions on Mathematical Software*, 19(4):443–451, December 1993.

[6] W. J. Cody, J. T. Coonen, D. M. Gay, K. Hanson, D. Hough, W. Kahan, R. Karpinski, J. Palmer, F. N. Ris, and D. Stevenson. A proposed radix-and-word-length-independent standard for floating-point arithmetic. *IEEE MICRO*, 4(4):86–100, August 1984.

[7] M. A. Cornea-Hasegan, R. A. Golliver, and P. Markstein. Correctness proofs outline for newton-raphson based floating-point divide and square root algorithms. In Koren and Kornerup, editors, *Proceedings of the 14th IEEE Symposium on Computer Arithmetic (Adelaide, Australia)*, pages 96–105, Los Alamitos, CA, April 1999. IEEE Computer Society Press.

[8] D. Defour, P. Kornerup, J.-M. Muller, and N. Revol. A new range reduction algorithm. In *Proc. 35th Asilomar Conference on Signals, Systems, and*, Pacific Grove, California, November 2001.

[9] T. J. Dekker. A floating point technique for extending the available precision. *Numerische Mathematik*, 18:224–242, 3 1971.

[10] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–47, March 1991.

[11] J. Harrison. A machine-checked theory of floating-point arithmetic. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *Theorem Proving in Higher Order Logics: 12th International Conference, TPHOLs'99*, volume 1690 of *Lecture Notes in Computer Science*, pages 113–130, Nice, France, September 1999. Springer-Verlag.

[12] American National Standards Institute, Institute of Electrical, and Electronic Engineers. Ieee standard for radix independent floating-point arithmetic. *ANSI/IEEE Standard, Std 854-1987, New York*, 1987.

[13] Alan H. Karp and Peter Markstein. High-precision division and square root. *ACM Transactions on Mathematical Software*, 23(4):561–589, December 1997.

[14] D. Knuth. *The Art of Computer Programming, 3rd edition*, volume 2. Addison Wesley, Reading, MA, 1998.

[15] R.-C. Li, S. Boldo, and M. Daumas. Theorems on efficient argument reduction. In Bajard and Schulte, editors, *Proceedings of the 16th IEEE Symposium on Computer Arithmetic (ARITH16)*, pages 129–136. IEEE Computer Society Press, June 2003.

[16] P. Markstein. *Ia-64 and Elementary Functions : Speed and Precision*. Hewlett-Packard Professional Books. Prentice Hall, 2000. ISBN: 0130183482.

[17] P. W. Markstein. Computation of elementary functions on the IBM risc system/6000 processor. *IBM Journal of Research and Development*, 34(1):111–119, January 1990.

[18] 0. Møller. Quasi double-precision in floating-point addition. *BIT*, 5:37–50, 1965.

[19] P. H. Sterbenz. *Floating point computation*. Prentice-Hall, Englewood Cliffs, NJ, 1974.