Some Functions Computable with a Fused-mac
Sylvie Boldo, Jean-Michel Muller

To cite this version:
Sylvie Boldo, Jean-Michel Muller. Some Functions Computable with a Fused-mac. 17th IEEE Symposium on Computer Arithmetic, Jun 2005, Cape Cod, Massachusetts, USA, pp.52-58. inria-00000895

HAL Id: inria-00000895
https://inria.hal.science/inria-00000895
Submitted on 1 Dec 2005

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Some Functions Computable with a Fused-mac

Sylvie Boldo and Jean-Michel Muller
Laboratoire LIP (CNRS/ENS Lyon/Inria/Univ. lyon 1),
Projet Arénaire, 46 allée d’Italie, 69364 Lyon Cedex 07,
FRANCE
Sylvie.Boldo@ens-lyon.fr, Jean-Michel.Muller@ens-lyon.fr

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 \( \text{ulp}(x) \) or \( \text{Nextafter}(x, y) \)), 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.

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 \( \pm ax \pm b \) with one final rounding error only. This allows one 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 is well known [9, 2, 3] that (assuming rounding to nearest) the error of a floating point addition or the remainder of a square root is exactly representable as a floating-point number of the same format. This is also true (for any rounding mode) for the error of a multiplication or the remainder of a division. 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 \( \text{Nextafter}(x, y) \), or the successor of a given floating-point number, or (for error estimation), \( \text{ulp}(x) \). We may also, for some calculations, need to know if the last bit of the significand 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 \( M_n \) as the set of exponent-unbounded, \( n \)-bit significand, binary floating-point (FP) numbers (with \( n \geq 1 \)), that is: \( M_n = \{ M \times 2^E, 2^{n-1} \leq M \leq 2^n - 1, M, E \in \mathbb{Z} \} \cup \{0\} \). It is an “ideal” system, with no overflows or underflows. We will show results in \( M_n \). 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 or significand of a nonzero element \( M \times 2^E \) of \( M_n \) is the number \( m(x) = M/2^{n-1} \), its integral significand, 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 arithmetic: rounding modes, ulps, . . . See [10] for definitions. In the following \( o(t) \) means \( t \) rounded to the nearest even.

3 Previous results and preliminary properties

We will use the 2sum and Fast2Sum algorithm, presented below. They do not require the availability of a fused-mac, and make it possible to compute the error of a
floating-point addition exactly, represented by an 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

\[
\begin{align*}
x &= \circ(a + b) \\
b' &= \circ(x - a) \\
a' &= \circ(x - b') \\
e_b &= \circ(b - b') \\
e_a &= \circ(a - a') \\
y &= \circ(e_a + e_b)
\end{align*}
\]

We have:

- \(|x + y| \leq \frac{1}{2} \text{ulp}(x)|\]
- \(|y| \leq \frac{1}{2} \text{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

\[
\begin{align*}
x &= \circ(a + b) \\
b' &= \circ(x - a) \\
y &= \circ(b - b')
\end{align*}
\]

We have:

- \(|x + y| \leq \frac{1}{2} \text{ulp}(x)|\]
- \(|y| \leq \frac{1}{2} \text{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 \(\geq 2\), 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 an FP number.

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

**Property 3** If \((x, y) = 2\text{Sum}(a, b)\) 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

\[
\begin{align*}
x &= \circ(ab) \\
y &= \circ(ab - x)
\end{align*}
\]

we have:

- \(|x + y| = ab|\]
- \(|y| \leq \frac{1}{2} \text{ulp}(x)|

Without a fused-mac, computing \(x\) and \(y\) is possible, but requires much more computation [9] (the significands 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 bit of the significand 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 significand 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 significand bit of \(x\) is a zero.

\[
\begin{align*}
\alpha &= \circ(3x) \\
\beta &= \circ(\alpha - 2x) \\
\text{IsEven} &= (\beta = x)
\end{align*}
\]

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.

\[
\begin{align*}
y_h &= \circ(xC) \\
y_e &= \circ(xC - y_h) \\
\text{IsAPowerOf2} &= (y_e = 0).
\end{align*}
\]

**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 \( P2^n \), where \( P \) is odd and larger than \( 2^n \). Hence \( P \) cannot be exactly representable with \( n \) bits, hence \( y_h \neq xC \), hence \( y_e \neq 0 \).

**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\(^1\) recommends (but does not require) the availability of function \texttt{Nextafter}. \texttt{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 \texttt{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 an 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 \( \text{succ}(x) \) is \( x^- \) if \( x < 0 \), and \( x^+ \) if \( x > 0 \). The symmetrical predecessor \( \text{pred}(x) \) 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 significand bits is less than the absolute value of the smallest exponent. This is required by the IEEE-854 Standard for Floating-Point \( \text{arithmetic} [12] \), that says that \((E_{\text{max}} - E_{\text{min}})/n\) shall exceed 5 and should exceed 10, and that \( bE_{\text{max}} + E_{\text{min}} + 1\) should be the smallest integral power of \( b \), that is greater than or equal to 4, where \( b \) is the radix.

**Property 7** Computation of \( \text{succ}(x) \) If \( n \geq 2 \) and \( x \neq 0 \), then
\[ \text{succ}(x) = \circ(x + sx) \]

**Proof** Assume \( x > 0 \) and \( 2^e \leq x < 2^{e+1} \) (i.e., the exponent of \( x \) is \( e \)). Since, in that case, \( \text{succ}(x) = x + 2^{e-n+1} \) and \( \text{ulp}(x) = 2^{e-n+1} \), to show that \( \circ(x + sx) \) is equal to \( \text{succ}(x) \) it suffices to show that
\[ x + 2^{e-n} < x + sx < x + 3 \times 2^{e-n} \]
(i.e., that \( x + sx \) is within \( 1/2\text{ulp from} \text{succ}(x) \)).

Thus, it suffices to show that
\[ 2^{e-n} < sx < 3 \times 2^{e-n}. \]
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 \( 3 \times 2^{e-n} \) as soon as \( n \geq 2 \).

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

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

**Property 8** Computation of \( \text{pred}(x) \) If \( n \geq 2 \) and \( x \neq 0 \), then
\[ \text{pred}(x) = \circ(x - sx) \]
Now, from functions `succ` and `pred`, one can very easily compute functions `Nextafter`, \( x^+ \) and \( x^- \):

### Property 9

If \( x \neq 0 \) then

\[
x^+ = \circ(x + s|x|)\\
x^- = \circ(x - s|x|)\\
\text{Nextafter}(x, y) = \begin{cases} x^+ & \text{if } y > x \\ x & \text{if } y = x \\ x^- & \text{if } y < x \end{cases}
\]

This may be different with other rounding modes. For instance, if rounding towards zero \( Z(x) \) is used, then \( Z(x) \) returns \( \text{pred}(x) \) for any nonzero \( x \in \mathbb{M}_n \), with \( \sigma = 1 - 2^{-n} \).

One can also implement \( x^+ \) as \((x+\epsilon) \) rounded towards \(+\infty\), where \( \epsilon \) is the smallest positive nonzero subnormal number. 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 \( \text{ulp}(x) \)

Function \( \text{ulp} \) (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 an FP number:

\[
\text{ulp}(x) = |x|^+ - |x|
\]

then (if \( x \in \mathbb{M}_n \) is nonzero) one can compute function \( \text{ulp} \) through the following sequence

\[
y = \circ(x + s)\\
\text{ulp} = |y - x|
\]

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

\[
\text{ulp}(x) = |x| - |x^-|
\]

then function \( \text{ulp} \) is computed through

\[
y = \circ(x - s)\\
\text{ulp} = |y - x|
\]

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).

### 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.

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

\(^3\)Kahan’s definition is: \( \text{ulp}(x) \) is the gap between the two finite floating-point numbers nearest \( x \), even if \( x \) is one of them (But \( \text{ulp}(\text{NaN}) \) is NaN.)
5.1 The algorithm \textit{ErrFmac}

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

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

we have:

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

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 $\alpha_2$. Then the other terms $u_1$ and $\alpha_1$ are bigger than this value and can be combined with $r_1$ into a single value $\gamma$.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure1.png}
\caption{Intermediate values of the \textit{ErrFmac} algorithm.}
\end{figure}

5.2 Proof of the correctness of the \textit{ErrFmac} algorithm

If $\gamma = o(\beta_1 - r_1) + \beta_2$ 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 $|r_2 + r_3| \leq \frac{1}{2}\ulp(r_1)$ and $|r_3| \leq \frac{1}{2}\ulp(r_2)$, from previous properties.

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.

\begin{enumerate}
    \item[If we have $\beta_2 = 0$,]
\end{enumerate}

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure2.png}
\caption{Intermediate values when $\beta_2 = 0$.}
\end{figure}

\begin{itemize}
    \item the general case: $|\beta_1| \geq 4|\alpha_2|$
    \item the special case where $\beta_1$ is a result of a near-total cancellation: $\beta_1 = 2^{\min(e_{\beta_1}, e_{\alpha_1})}$ and $|\beta_1| \geq 2|\alpha_2|$. In the general case, we are in the conditions of Sterbenz’s theorem [19]: $r_1$ and $\beta_1$ share the same sign and

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

In the special case, we have $4|\alpha_2| > |\beta_1| \geq 2|\alpha_2|$. As $\beta_1$ 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

$$
|\beta_1 - r_1|2^{-r_1} = |\beta_1 - o(\beta_1 + \alpha_2)|2^{-r_1} \leq \left( \frac{1}{2} \ulp(r_1) + |\alpha_2| \right)2^{-r_1} \leq \frac{1}{2} + |\beta_1|2^{-r_1 - 1} \leq \frac{1}{2} + (2^n - 1)2^{e_{r_1} + 1 - e_{r_1} - 1} < 2^n.
$$
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 $\beta_1 = \alpha (\beta_1 + \beta_2 + \alpha_2)$ not to be equal to $\alpha (\beta_1 + \beta_2 + \alpha_2) + r_1$ is that either $|\beta_2| = \frac{1}{2} \ulp (\beta_1)$ or $\beta_2 = -\frac{1}{2} \ulp (\beta_1)$ if $\beta_1$ is a power of 2.

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

$$|\beta_1 - r_1| 2^{-\min (e_{\beta_1}, e_{\beta_2})} = |\beta_1 - \alpha (\beta_1 + \beta_2 + \alpha_2)| 2^{-\min (e_{\beta_1}, e_{\beta_2})} \leq \left( \frac{1}{2} \ulp (r_1) + |\beta_2| + |\alpha_2| \right) 2^{-\min (e_{\beta_1}, e_{\beta_2})} \leq \left( 2^{e_{\beta_1} - 1} + 2^{s_{\beta_1} - 1} + 2^{s_{\beta_1} - 1} \right) 2^{-\min (e_{\beta_1}, e_{\beta_2})} \leq 4$$

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 $e_{\beta_1} + 1 \geq e_{\beta_1} \geq e_{\beta_1} - 1$ and that $\beta_2$ is either $2^{s_{\beta_1} - 1}$ or $2^{s_{\beta_1} - 2}$, so $e_{\beta_1} - 2$ is a suitable exponent for $(\beta_1 - r_1) + \beta_2$ and

$$|(\beta_1 - r_1) + \beta_2| 2^{-e_{\beta_1} + 2} = |u_1 + \alpha_1 - \alpha (u_1 + \alpha_1 + \alpha_2)| 2^{-e_{\beta_1} + 2} \leq \left( \frac{1}{2} \ulp (r_1) + |\alpha_2| \right) 2^{-e_{\beta_1} + 2} \leq \left( 2^{e_{\beta_1} - 1} + 2^{s_{\beta_1} - 1} \right) 2^{-e_{\beta_1} + 2} \leq 6$$

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 $r_1 = \Delta (ax + y) = 2^{4n} + 2^{3n+1}$. 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 fused-mac delays (FMD), 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 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 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;
- $|r_2 + r_3| \leq \frac{1}{2} \ulp (r_1)$;
- $r_2 = 0$ 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 $e_k$ in the 2Sum algorithm (Property 1) can be performed in parallel.

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

![Figure 4](image4.png)

**Figure 4. Task repartition when 3 FPUs are available.**

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

![Figure 5](image5.png)

**Figure 5. Task repartition when 2 FPUs are available.**

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

<table>
<thead>
<tr>
<th>Cost (in FMDs)</th>
<th>1 FPU</th>
<th>2 FPUs</th>
<th>3 FPUs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Given algorithm</td>
<td>20</td>
<td>14</td>
<td>12</td>
</tr>
<tr>
<td>Without the final compression</td>
<td>17</td>
<td>11</td>
<td>9</td>
</tr>
<tr>
<td>When $</td>
<td>y</td>
<td>\geq</td>
<td>ax</td>
</tr>
</tbody>
</table>

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, assuming rounding to nearest, 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


