Accelerating Correctly Rounded Floating-PointDivision when the Divisor is Known in Advance

Jean-Michel Muller, Nicolas Brisebarre, Saurabh Raina

Accelerating Correctly Rounded Floating-Point Division when the Divisor Is Known in Advance

Nicolas Brisebarre, Jean-Michel Muller, *Member*, *IEEE*, and Saurabh Kumar Raina

Abstract—We present techniques for accelerating the floating-point computation of x/y when y is known before x. The proposed algorithms are oriented toward architectures with available fused-mac operations. The goal is to get exactly the same result as with usual division with rounding to nearest. It is known that the advanced computation of $1/y$ allows performing correctly rounded division in one multiplication plus two fused-macs. We show algorithms that reduce this latency to one multiplication and one fused-mac. This is achieved if a precision of at least $n+1$ bits is available, where n is the number of mantissa bits in the target format, or if y satisfies some properties that can be easily checked at compile-time. This requires a double-word approximation of $1/y$ (we also show how to get it). These techniques can be used by compilers to accelerate some numerical programs without loss of accuracy.

Index Terms—Computer arithmetic, floating-point arithmetic, division by software, division with fused-mac, compilation optimization.

1 Motivation of This Research

We wish to provide methods for accelerating floating-point (FP for short) divisions of the form x/y, when y is known before x, either at compile-time (i.e., y is a constant; in such a case, much precomputation can be performed) or at runtime. We want to get the result more quickly than by just performing a division, yet with the same accuracy: We need a correctly rounded value, as required by the IEEE 754 Standard for FP arithmetic [1], [6].

Divisions by constants are a clear application of our work. There are other applications, for instance, when many divisions by the same y are performed (an example is Gaussian elimination).

We assume that a fused multiply-accumulator is available and that division is done in software (this happens, for instance, on RS6000, PowerPC, or Itanium architectures). In this paper, we focus on rounding to nearest only. Presentation of conventional division methods can be found in [4], [9], [12].

2 Introduction

For computing x/y when y is known in advance, a naive approach consists of computing the reciprocal of y (with rounding to nearest) and then, once x is available, multiplying the obtained result by x. It is well-known that such a "naive method" does not always produce a correctly rounded result. And yet, if the probability of getting an incorrect rounding was small enough, one could choose to use that method anyway, to check if the result is correctly rounded and to perform some correction step when this is not the case. Also, one could imagine that there might exist some values of y for which the naive method always work (for any x). For these reasons, we have decided to dedicate a short section to the analysis of the naive method.

Our main approach starts as previously: Once x is known, it is multiplied by the precomputed reciprocal of y. Then, a "remainder" is computed and used to correct the final result. This does not require testing. That approach looks like the final steps of a Newton-Raphson division. It is clear from the literature that the iterative algorithms for division require an initial approximation of the reciprocal of the divisor and that the number of iterations is reduced by having a more accurate initial approximation. Of course, this initial approximation can be computed in advance if the divisor is known, but some care is needed to get correctly rounded results at low cost.

In Section 5.1, we show that, under some conditions on y, that could be checked at compile-time, we can return a correctly rounded quotient using one multiplication and one fused-mac. In Section 5.2, we show that, if a larger internal precision than the target precision is available (one more bit suffices), then we can always return a correctly rounded quotient using one multiplication and one fused-mac.

To make this paper easier to read and to save space, we have put all proofs, tables, intermediate lemmas, and supplementary material in an appendix which can be found on the Computer Society Digital Library at http://computer.org/tc/archives.htm or on our own web site [2].

3 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, 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 $m(x) = M/2^{n-1}$.

We assume that the reader is familiar with the notions of rounding modes, ulps, floating-point successor, and predecessor. See [6] for definitions. In the following, $\circ_\nu(t)$ means t rounded to the nearest even, $\circ_d(t)$ means t rounded to $-\infty$, and $\circ_u(t)$ means t rounded to $+\infty$.

4 Preliminary Results

4.1 The Naive Method

As said in the introduction, we have to evaluate x/y and y is known before x, where x and y belong to $\mathbb{M}_n$. An obvious solution consists of precomputing $z = 1/y$ (or, more precisely, z rounded-to-nearest, that is, $z_h = \circ_\nu(1/y)$) and then multiplying x by z_h. We will refer to this as "the naive method." We assume round-to-nearest mode.

4.1.1 Maximum Error of the Naive Solution

Property 1. *The naive solution returns a result that is at most at distance* 1.5 *ulps from the exact result if* $m(x) < m(y)$ *(reminder:* $m(u)$ *is the mantissa of* u*) and* 1 *ulp from the exact result if* $m(x) \geq m(y)$.

Property 1 gives tight bounds: There are values x and y for which the naive solution leads to an error very close to 1.5 ulps. More precisely,

Property 2. *We have shown [2] that the maximum error of the naive algorithm can be obtained through a reasonably fast algorithm. This maximum error converges to* 1.5 *ulps as* $n \to \infty$.

• *The authors are with the Laboratoire LIP, ENSL/CNRS/INRIA Arenaire Project, Ecole Normale Supérieure de Lyon, 46 Allée d'Italie, 69364 Lyon Cedex 07, France. N. Brisebarre is also with LArAl, Univesité Jean Monnet, Saint-Etienne.*
E-mail: {Nicolas.Brisebarre, Jean-Michel.Muller, Saurabh-Kumar.Raina}@ens-lyon.fr.

Manuscript received 27 Aug. 2002; revised 2 June 2003; accepted 11 Dec. 2003.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 117197.

For instance, in the IEEE-754 double precision format ($n = 53$), the division of $x = \frac{268,435,449}{134,217,728}$ by $y = \frac{9,007,199,120,523,265}{4,503,599,627,370,496}$ by the naive algorithm leads to an error equal to $1.4999999739\cdots$ ulps.

4.1.2 Probability of Getting a Correctly Rounded Result Using the Naive Solution

For the first few values of n, we have computed, through exhaustive testing, the proportion of couples (x, y) for which the naive method gives an incorrectly rounded result. The proportion seems to converge, as n grows, to a constant value that is around 27 percent. More precisely,

Conjecture 1. *Assuming a uniform distribution of the mantissas of FP numbers, rounding to nearest, and n bits of mantissa, the probability that the naive method returns a result different from $\circ_\nu(x/y)$ goes to $13/48 = 0.2708\cdots$ as n goes to $+\infty$.*

This conjecture is a "half-conjecture" only since we have a rough sketch of a proof [2]. This tends to show that, for any n, the naive method gives a proportion of incorrectly rounded results that is far too large to be neglected.

4.1.3 Values of y for which the Naive Method Always Works

Depending on n, there are a very few values of y for which the naive method always works (i.e., for all values of x). For instance, for $n = 13$, the four values of y between 1 and 2 for which the naive method always works are 1, $4,411/4,096$, $4,551/4,096$, and $4,915/4,096$. We are not able to compute them much faster than by exhaustive testing, which does not allow us to tackle the most interesting values of n, namely, 24, 53, and 113.

4.2 Division with One Multiplication and Two Fused-Macs

On some modern processors (such as the PowerPC, the IBM RISCSystem/6000 [11], and IA64-based architectures [3], [10]), a fused-multiply accumulate instruction (fused-mac) is available. This makes it possible to evaluate an expression $ax + b$ with one final rounding only, which facilitates software implementation of division and elementary functions. Let us now investigate how such an instruction can be used to solve our problem. The following result (see the work of Markstein [3], [10], [11] for this kind of algorithm) shows that one multiplication and two fused-macs allow us to get correctly rounded results.

Theorem 1 (Division with one multiplication and two fused-macs [10], [11]). *Algorithm 1, given below, always returns the correctly rounded (to nearest) quotient $\circ_\nu(x/y)$.*

Algorithm 1 (Division with one multiplication and two fused-macs.

- In advance, evaluate $z_h = \circ_\nu(1/y)$;
- As soon as x is known, compute $q = \circ_\nu(xz_h)$, $r = \circ_\nu(x - qy)$, and $q' = \circ_\nu(q + rz_h)$;

This method requires one division before x is known, one multiplication and two fused-macs once x is known. In the following section, we try to design a faster algorithm. Unfortunately, either there are a few (predictable) values of y for which it does not work or it requires the availability of an internal precision slightly larger than the target precision.

5 Proposed Techniques

5.1 Division with One Multiplication and One Fused-Mac

Using the method presented in Section 3.2, we could compute x/y using one multiplication and two fused-macs, once x is known. Let us show that, in many cases, one multiplication and one fused-mac (once x is known) do suffice. To do this, we need a double-word approximation to $1/y$. Let us first see how can such an approximation be computed.

5.1.1 Preliminary Result: Getting a Double-Word Approximation to $1/y$

Kahan [7] explains that the fused-mac allows us to compute remainders exactly. This is done as follows.

Property 3. *Let $x, y, q \in \mathbb{M}_n$ such that $q \in \{\circ_d(x/y), \circ_u(x/y)\}$. The remainder $r = x - qy$ is computed exactly with a fused-mac. That is, $\circ_\nu(x - qy) = x - qy$.*

The algorithms we are going to examine require a double-word approximation to $1/y$, that is, two FP values z_h and z_ℓ such that $z_h = \circ_\nu(1/y)$ and $z_\ell = \circ_\nu(1/y - z_h)$. The only reasonably fast algorithm we know for getting these values requires a fused-mac. Using Property 3, z_h and z_ℓ can be computed as follows.

Property 4. *Assume $y \in \mathbb{M}_n$, $y \neq 0$. The following sequence of three operations computes z_h and z_ℓ such that $z_h = \circ_\nu(1/y)$ and $z_\ell = \circ_\nu(1/y - z_h)$:*

$$z_h = \circ_\nu(1/y),\ \rho = \circ_\nu(1 - yz_h),\ z_\ell = \circ_\nu(\rho/y).$$

5.1.2 The Algorithm

We assume that, from y, we have computed $z_h = \circ_\nu(z)$, and $z_\ell = \circ_\nu(z - z_h)$, where $z = 1/y$ (for instance, using Property 4). We suggest the following 2-step method:

Algorithm 2 (Division with one multiplication and one fused-mac) Compute: $q_1 = \circ_\nu(xz_\ell)$ and $q_2 = \circ_\nu(xz_h + q_1)$.

This algorithm almost always works and, for $n \leq 7$, it always works. Exhaustive searching [2] shows that, for $n \leq 29$, there are more than 98.7 percent of values of y for which the algorithm returns a correctly rounded quotient for all values of x (these figures have been obtained through exhaustive checking). Moreover, in the other cases (see the proof of Theorem 2 in [2]), for a given y, there is *at most one value of the mantissa of x* (that can be computed in advance) for which the algorithm may return an incorrectly rounded quotient.

Theorem 2. *Algorithm 2 gives a correct result (that is, $q_2 = \circ_\nu(x/y)$) as soon as at least one of the following conditions is satisfied:*

1. *The last mantissa bit of y is a zero;*
2. *$|z_\ell| < 2^{-n-2-e}$, where e is the exponent of y (i.e., $2^e \leq |y| < 2^{e+1}$);*
3. *Algorithm 3, given below, returns **true** when the input value is the integer $Y = y \times 2^{n-1-e_y}$, where e_y is the exponent of y (Y is the mantissa of y, interpreted as an integer).*

Algorithm 3. *We give the algorithm as a Maple program (to make it more didactic). If it returns "true," then Algorithm 2 always returns a correctly rounded result when dividing by y. It requires the availability of $2n + 1$-bit integer arithmetic.*

```
TestY := proc(Y,n)
  local Pminus, Qminus, Xminus, OK, Pplus, Qplus,
    Xplus;
  Pminus := (1/Y) mod 2^(n+1);
  # requires computation of a modular inverse
  Qminus := (Pminus-1) / 2; Xminus := (Pminus * Y - 1) /
    2^(n+1);
  if (Qminus >= 2^(n-1)) and (Xminus >= 2^(n-1))
      then OK := false
else
  OK := true;
```

```
   Pplus := 2^(n+1)-Pminus; Qplus := (Pplus-1) / 2;
   Xplus := (Pplus * Y + 1) / 2^(n+1);
   if (Qplus >= 2^(n-1)) and (Xplus >= 2^(n-1))
        then OK := false end if; end if;
   print(OK)
end proc;
```

Translation of Algorithm 3 into a C or Fortran program is easily done since computing a modular reciprocal modulo a power of two requires a few operations only, using the extended Euclidean GCD algorithm [8]. Algorithm 3 also computes the only possible mantissa X for which, for the considered value of Y, Algorithm 2 might not work. *Hence, if the algorithm returns* ***false****, it suffices to check this very value of* X to know if the algorithm will always work or if it will work for all Xs but this one.

Let us discuss the consequences of Theorem 2.

- Condition "the last mantissa bit of y is a zero" is easily checked on most systems. Hence, that condition can be used for accelerating divisions when y is known at runtime, soon enough[1] before x. That condition allows us to accelerate half divisions;
- Assuming a uniform distribution of z_ℓ in $(-2^{-n-1-e}, +2^{-n-1-e})$, which is reasonable (see [5]), Condition "$|z_\ell| < 2^{-n-2-e}$" allows us to accelerate half of the remaining cases;
- Our experimental testings up to $n = 24$ show that Condition "Algorithm 3 returns **true**" allows us to accelerate around 39 percent of the remaining cases (i.e., the cases for which the last bit of y is a 1 and $|z_\ell| \geq 2^{-n-2-e}$). If Algorithm 3 returns **false**, then checking the only value of x for which the division algorithm might not work suffices to deal with all remaining cases. This requires much more computation: It is probably not interesting if y is not known at compile-time.

5.2 If a Larger Precision than Target Precision Is Available

A larger precision than the target precision is frequently available. A typical example is the double extended precision that is available on Intel microprocessors. We now show that, if an internal format is available, with at least $n+1$-bit mantissas (which is only one bit more than the target format), then an algorithm very similar to Algorithm 2 always works. In the following, $\circ_{t:+p}(x)$ means x rounded to $n+p$ bits, with rounding mode t. Define $z = 1/y$. We assume that, from y, we have computed $z_h = \circ_\nu(z)$ and $z_\ell = \circ_{\nu:+1}(z - z_h)$. They can be computed through:

$$z_h = \circ_\nu(1/y),\ \rho = \circ_\nu(1 - yz_h),\ z_\ell = \circ_{\nu:+1}(\rho/y).$$

We suggest the following 2-step method:

Algorithm 4 (Division with one multiplication and one fused-mac). *Compute:*

$$q_1 = \circ_{\nu:+1}(xz_\ell),\ q_2 = \circ_\nu(xz_h + q_1).$$

Theorem 3. *Algorithm 4 always returns a correctly rounded quotient.*

If the first operation returns a result with more than $n+1$ bits, the algorithm still works. We can, for instance, perform the first operation in double extended precision if the target precision is double precision.

1. The order of magnitude behind this "soon enough" highly depends on the architecture and operating system.

6 Comparisons

Let us give an example of a division algorithm used on an architecture with an available fused-mac. In [10], Markstein suggests the following sequence of instructions for double-precision division on IA-64. The intermediate calculations are performed using the internal double-extended format. The first instruction, `frcpa`, returns a tabulated approximation to the reciprocal of the operand, with at least 8.886 valid bits. When two instructions are put on the same line, they can be performed "in parallel." The returned result is the correctly rounded quotient with rounding mode $\circ_t$.

Algorithm 5 (Double precision division. This is Algorithm 8.10 of [10])

1. $z_1 = \texttt{frcpa}(y)$;
2. $e = \circ_\nu(1 - yz_1)$;
3. $z_2 = \circ_\nu(z_1 + z_1 e)$; $e_1 = \circ_\nu(e \times e)$;
4. $z_3 = \circ_\nu(z_2 + z_2 e_1)$; $e_2 = \circ_\nu(e_1 \times e_1)$;
5. $z_4 = \circ_\nu(z_3 + z_3 e_2)$;
6. $q_1 = \circ_\nu(xz_4)$;
7. $r = \circ_\nu(x - yq_1)$;
8. $q = \circ_t(q_1 + rz_4)$.

This algorithm requires eight FP latencies, and uses 10 instructions. The last three lines of this algorithm are Algorithm 1 of this paper (with a slightly different context since Algorithm 5 uses extended precision). Another algorithm also given by Markstein (Algorithm 8.11 of [10]) requires seven FP latencies only, but uses 11 instructions. The algorithm suggested by Markstein for extended precision is Algorithm 8.18 of [10]. It requires eight FP latencies and uses 14 FP instructions.

These figures show that replacing conventional division x/y by specific algorithms whenever y is a constant or division by the same y is performed many times in a loop is worth being done. For double-precision calculations, this replaces seven FP latencies by three (using Algorithm 1) or two (using Algorithm 2 if y satisfies the conditions of Theorem 2 or Algorithm 4 if a larger internal precision—e.g., double-extended precision—is available). Therefore, whenever an even slightly larger precision is available (one more bit suffices), Algorithm 4 is of interest. Algorithm 2 is certainly interesting when the last bit of y is a zero and, possibly, when $|z_\ell| < 2^{-n-2-e}$. In the other cases, the rather large amount of computation required by checking whether that algorithm can be used (we must run Algorithm 3 at compile-time) limits its use to divisions by constants in applications for which compile time can be large and running time must be as small as possible.

7 Conclusion

We have presented several ways of accelerating a division x/y, where y is known before x. Our methods could be used in optimizing compilers, to make some numerical programs run faster, without any loss of accuracy. Algorithm 1 always works and does not require much precomputation (so it can be used even if y is known a few tens of cycles only before x). Algorithm 2 is faster and yet it requires much precomputation (for computing z_h and z_ℓ and making sure that the algorithm works), so it is more suited for division by a constant. Algorithm 4 always works and requires two operations only once x is known, but it requires the availability of a slightly larger precision.

References

[1] ANSI and IEEE, "IEEE Standard for Binary Floating-Point Arithmetic," ANSI/IEEE Standard, Std 754-1985, 1985.

[2] N. Brisebarre, J.-M. Muller, and S.K. Raina, "Supplementary Material to 'Accelerating Correctly Rounded Floating-Point Division when the Divisor Is Known in Advance'," http://computer.org/tc/archives.htm or http://perso.ens-lyon.fr/jean-michel.muller/fpdiv.html, 2004.
[3] M. Cornea-Hasegan and B. Norin, "IA-64 Floating-Point Operations and the IEEE Standard for Binary Floating-Point Arithmetic," *Intel Technology J.*, Q4, 1999.
[4] M.D. Ercegovac and T. Lang, *Division and Square Root: Digit-Recurrence Algorithms and Implementations.* Boston: Kluwer Academic, 1994.
[5] A. Feldstein and R. Goodman, "Convergence Estimates for the Distribution of Trailing Digits," *J. ACM*, vol. 23, pp. 287-297, 1976.
[6] D. Goldberg, "What Every Computer Scientist Should Know about Floating-Point Arithmetic," *ACM Computing Surveys.* vol. 23, no. 1, pp. 5-47, Mar. 1991.
[7] W. Kahan, "Lecture Notes on the Status of IEEE-754," http://http.cs.berkeley.edu/~wkahan/ieee754status/ieee754.ps, 1996.
[8] D. Knuth, *The Art of Computer Programming*, vol. 2. Reading, Mass.: Addison Wesley, 1973.
[9] I. Koren, *Computer Arithmetic Algorithms*, Englewood Cliffs, N.J.: Prentice Hall, 1993.
[10] P.W. Markstein, *IA-64 and Elementary Functions: Speed and Precision.* Hewlett-Packard Professional Books, Prentice Hall, 2000.
[11] P.W. Markstein, "Computation of Elementary Functions on the IBM Risc System/6000 Processor," *IBM J. Research and Development*, vol. 34, no. 1, pp. 111-119, Jan. 1990.
[12] S.F. Oberman and M.J. Flynn, "Division Algorithms and Implementations," *IEEE Trans. Computers*, vol. 46, no. 8, pp. 833-854, Aug. 1997.