Efficient Division Methods When the Divisor is known beforehand

DEA Stage
On the Direction of Jean-Michel Muller and Nicholas Brisebarre
Projet Arénaire - LIP - ENS Lyon

Saurabh Kumar Raina

2 July 2003
Contents

1 Introduction .............................................. 5
   1.1 Aim .............................................. 5

2 Definitions and Notations .................................. 6
   2.1 Preliminary Results .................................. 7

3 State-of-the-art ............................................ 9
   3.1 Classical Division Methods ........................... 9
       3.1.1 Subtractive Algorithms .......................... 10
       3.1.2 Multiplicative Algorithms ......................... 10
   3.2 Division methods in current architecture .............. 11

4 The naive method .......................................... 13
   4.1 Maximum error of the naive solution .................. 13
   4.2 Probability that naive solution always works ........ 14
   4.3 Some values of y for which the naive method always works 15

5 Division with one multiplication and two fused MACs 15

6 An efficient technique ..................................... 16
   6.1 Performing Division with one multiplication and one fused MAC 16
       6.1.1 The algorithm .................................... 17
   6.2 Can a larger precision help? .......................... 22

7 Proposed Vs Conventional ................................ 23

8 Conclusion .................................................. 23
Acknowledgement

I am very grateful to my Director of stage, Mr. Jean-Michel Muller for all his support by all means. I am specially thankful for all those meetings with him and Nicholas Brisebarre. Nicholas was there for all the things during my stage. He helped me to learn many things and particularly improving on my presentations. We had developed a good camaraderie and always talked about many things apart from our work. Despite having a busy schedule Jean-Michel always had time whenever I asked for it and always appreciated even my silly ideas.

I am very much indebted to my room partner who shared many things with me specially during these five months. From the rudimentary ideas to complex concepts he always helped me to understand them whether its day or night. He always rejuvenated me with his magnificent thoughts over disparate subjects.

Last but not the least, I am thankful to the whole Arénare team for conducting a group meeting every month followed by a luncheon.
Participation and achievements

This report is a compilation of all the facts and ideas that were collected in the five month internship. The major source of this report is an article “Accelerating Floating-Point Division When the Divisor is Known in Advance” by “Jean-Michel Muller” [31] which came out of embryo as an INRIA's Report of Research No.4532 in Aug'2002. After that some significant improvements have been done in the article. My contributions to it can be seen in the later half of the paper. I helped in the analysis of Algorithms 2 and 3. I wrote many programs for searching exhaustively the cases for which Algorithm 2 doesn’t work. In fact it is these results which show that our algorithm is very much feasible and gives good results in more than 98.7% cases. These results also helped in changing some conditions of Theorem 3. Now comes an interesting Algorithm 3, which was written as Maple program to find solutions to a set of equations which helps further in founding the bad cases. I observed a peculiar thing in these bad cases and then proved that there always exist a unique set of solution to these equations. This very fact helped my programs to run faster and search quickly the cases where Algorithm 2 doesn’t work and then I was able to do exhaustive searching for higher precision. After that something more came from this wonderful Maple program. It doesn’t only give the bad cases but it gives correct result from these cases (with a slight manipulation) where Algorithm 2 fails to deliver.

It was Jean-Michel’s idea to check the algorithm for larger precision. So, I again searched exhaustively the bad cases for Algorithm 2 but with larger precision for intermediate computations while keeping the target precision same as before. I found the algorithm always working so I slightly reduced the increased precision and tried to impose some lower bound on it. After that I and Jean-Michel both came up independently with a new Theorem 4 stating that Algorithm 2 always works if a precision of just one more bit is available for intermediate computations. In the last part of the report we have compared our method with the conventional one depending on their floating-point latencies but major comparisons have yet to be done. Although this report presents other methods also it mainly concentrates on the method I worked on.

After improving the article and when we had enough results we again compiled everything and changed the title slightly to “Accelerating Correctly Rounded Floating-Point Division When the Divisor is Known in Advance” [32] before sending it to the journal. Our article is under review at IEEE Transactions on Computers.
Abstract

In recent years, many algorithms for floating-point division have been developed among which some are suitable for hardware implementation. Various factors such as available hardware operations and complexity of algorithms influence the design of these algorithms. Currently, in some architectures division is being deferred to software for many reasons. This report considers the division algorithm designed to perform division in software for IA-64. Following this concept, we present a technique to perform division in a special case when the divisor is known in advance. We try to improve the conventional algorithm designed for architectures with an available fused-MAC unit. Our results reduce the floating-point latencies. This technique can be used by compilers to optimize some numerical programs to make them run faster without any loss in terms of accuracy.

1 Introduction

A major focus of computer arithmetic is the development of high-speed arithmetic algorithms and the design of application specific circuits to enhance the speed of numerical applications. Despite many advances, among the basic arithmetic operations division is still the most complex and consequently the most time consuming. The latency for double precision division ranges from less than 8 cycles to over 60 cycles in contrast with addition (2 to 4 cycles) and multiplication (2 to 8 cycles). Many feel that division is an infrequent operation which doesn’t need high priority but it has been shown that ignoring its implementation can result in significant system performance degradation for many applications [22]. Division algorithms can be divided into five classes: digit recurrence, functional iteration, very high radix, table look-up, and variable latency. These classes differ each other by the differences in hardware operations used in their implementation. Detailed presentation of all these classes can be found in [4, 10, 14, 15].

This report discusses about different algorithms to perform division in software and provides methods for accelerating floating-point divisions of the form \( x/y \) when \( y \) is known before \( x \), either at compile-time, or at run time. Integer division by a constant has already attracted many authors [29]. The IA-64 architecture defers floating-point and integer division to software. To ensure correctness and maximum efficiency, Intel provides a number of recommended algorithms which can be called as subroutines or inlined by compilers and assembly language programmers. In IA-64, the only instruction specifically intended to support division is the floating-point reciprocal approximation instruction, \( \text{frcpa} \). This instruction gives a tabulated approximation and after that the software uses this approximation and refines it towards a good reciprocal or quotient approximation.

1.1 Aim

Beside presenting different techniques including the conventional one the report aims to improve two important parameters, speed and accuracy. Whatever technique has been given we always try to get a correctly rounded value and that too more quickly than a mere division. Certainly, if \( y \) is a constant, much pre-computation can be done using \( y \) than if it is available dynamically. Moreover, if it is available dynamically this algorithm can be useful if several divisions are performed by the same \( y \) as in a case of numerical program given below. We have tried to improve the conventional algorithm and the results show that our algorithm almost always works with complete accuracy. We rely on the correctly rounded values provided by IEEE754 Standard [1, 6] to compare our results. In this paper, we focus on rounding to nearest only (see section 2). Consider for instance Gaussian elimination:
for j=1 to n-1 do  
  if a[j,j] = 0 then stop  
  else  
    p = 1 / a[j,j]  
    for i = j+1 to n do  
      c[i,j] = a[i,j] * p  
      for k = j+1 to n do a[i,k] = a[i,k] - c[i,j]*a[j,k]  
    end for  
    b[i] = b[i] - l[i,j]*b[j]  
  end for  
end for

It is highly likely that in computing reciprocal p and multiplying it with a[i,j], we are committing some error. This motivates us to search for new methods which can give better trade-off between speed and accuracy.

The remainder of this report is organized as follows. Section 2 presents some definitions and notations with some basic properties which will be used further in the report. Section 3 presents the State-of-the-art technique and section 4 explains naïve method. Section 5 introduces the conventional method. Section 6 presents improved method followed by its comparison with conventional one in section 7. Section 8 is the conclusion.

2 Definitions and Notations

Define \( M_n \) as the set of exponent-unbounded, \( n \)-bit mantissa, binary floating-point numbers (with \( n \geq 1 \)), that is:

\[
M_n = \{ M \times 2^E, 2^n-1 \leq M \leq 2^n - 1, M, E \in \mathbb{Z} \} \cup \{ 0 \}
\]

\( M_n \) is an “ideal” system, without any overflows or underflows as the exponent is unbounded. All the results will be shown using the elements of \( M_n \) and will be the same for any system implementing IEEE754 Standard provided that no overflow or underflow occurs. The mantissa of a nonzero element \( M \times 2^E \) of \( M_n \) is the number \( m(x) = M/2^n-1 \). The accuracy of the results obtained in a floating-point arithmetic unit is limited by the fact that the number of computed digits may exceed the total number of digits allowed and we have to chop off the extra digits. So the result of an arithmetic operation whose input values belong to \( M_n \) may not belong to \( M_n \) (in general it does not). Hence that result must be rounded. The standard defines 4 different rounding modes:

- rounding towards \( +\infty \), or upwards: \( o_u(x) \) is the smallest element of \( M_n \) that is greater than or equal to \( x \):
  \[
o_u(x) = \min \{ X \in M_n \mid X \geq x \}.
\]

- rounding towards \( -\infty \), or downwards: \( o_d(x) \) is the largest element of \( M_n \) that is less than or equal to \( x \):
  \[
o_d(x) = \max \{ X \in M_n \mid X \leq x \}.
\]

- rounding towards 0: \( o_z(x) \) is equal to \( o_u(x) \) if \( x < 0 \), and to \( o_d(x) \) otherwise;
If \( \frac{4}{7} \leq x < \frac{5}{7} \), \( \circ_d(x) = \circ_z(x) = \frac{4}{7} \)  
If \( \frac{9}{8} < x < \frac{11}{8} \), \( \circ_n(x) = \frac{5}{7} \)

If \( \frac{-5}{4} < x \leq \frac{-4}{7} \), \( \circ_u(x) = \circ_z(x) = \frac{-4}{7} \)  
If \( \frac{11}{8} \leq x \leq \frac{13}{8} \), \( \circ_n(x) = \frac{6}{7} \)

\[
\begin{array}{cccccccc}
\frac{-5}{4} & \frac{-4}{7} & 0 & \frac{6}{8} & \frac{7}{8} & \frac{9}{8} & \frac{11}{8} & \frac{13}{8} & \frac{15}{8} \\
\end{array}
\]

If \( \frac{4}{7} \leq x < \frac{8}{7} \), \( \text{ulp}(x) = \frac{1}{2^{3}} = \frac{1}{4} \)

If \( \frac{6}{8} \leq x < \frac{4}{7} \), \( \text{ulp}(x) = \frac{1}{2^{2}} = \frac{1}{8} \)

**Figure 1:** Illustration of rounding modes and definitions

- rounding to the nearest even: \( \circ_n(x) \) is the element of \( \mathbb{M}_n \) that is closest to \( x \). If \( x \) is the exact mean of two elements of \( \mathbb{M}_n \) i.e. in case of tie, the value \( \circ_n(x) \) is the one for which \( M \) is an even number.

The illustration below will make it clear how the rounding mode works.

The IEEE 754 standard requires that the user should be able to choose one rounding mode among these ones, called the **active rounding mode** and in this report it is **rounding to the nearest even**.

For \( a \in \mathbb{M}_n \), we define \( a^+ \) as its **successor** in \( \mathbb{M}_n \), that is, \( a^+ = \min\{b \in \mathbb{M}_n, b > a\} \), and \( a^- \) as its **predecessor** in \( \mathbb{M}_n \), that is, \( a^- = \max\{b \in \mathbb{M}_n, b < a\} \). If \( a \) is an element of \( \mathbb{M}_n \) then **unit in the last place** or \( \text{ulp}(a) \) is defined as \( |a^+| - |a| \) and as \( \circ_n(a) - \circ_d(a) \) otherwise. When \( x \in \mathbb{M}_n \), \( \text{ulp}(x) \) is the “weight” of the last mantissa bit of \( x \). **So ulp is the distance between two consecutive mantissas of floating-point numbers.**

A **Breakpoint** or BP is defined as a value \( z \) where the rounding changes, that is, if \( t_1 \) and \( t_2 \) are real numbers satisfying \( t_1 < z < t_2 \) and \( \circ_t \) is the rounding mode, then \( \circ_t(t_1) < \circ_t(t_2) \). For “directed” rounding modes (i.e., towards \( +\infty \), \( -\infty \) or \( 0 \)), the breakpoints are the floating-point numbers and for rounding to the nearest mode, they are the exact middle of two consecutive floating-point numbers.

**2.1 Preliminary Results**

The following results will be used throughout to prove theorems and other properties. We assume that both the divisor \( x \) and dividend \( y \) lie between 1 and 2, i.e. \( 1 \leq x, y < 2 \) as it suffices to focus
on their mantissas for division operation. The default radix is 2 for all the floating-point arithmetic operations.

Property 1 Let $y \in \mathbb{M}_n$, then the reciprocal of $y$ belongs to some $\mathbb{M}_q$ if and only if $y$ is a power of 2.

So, for particular precision $n$ there exist only one element belonging to $\mathbb{M}_n$ and having significand $M = 2^{n-1}$ for which the reciprocal is in $\mathbb{M}_n$.

Property 2 In a floating-point representation with a prime radix, the product of two $n$-bit floating-point numbers $1 \leq a, b < 2$ has a length of at least $2n - 1$.

Of course, if the length of a product exceeds the precision of the floating-point format, the product cannot be represented exactly in that format.

Property 3 In floating-point arithmetic the exact quotient of two $n$-bit numbers cannot have a length $m > n$.

The above property removes the need to consider rounding in cases a quotient lies exactly halfway between two representable floating-point numbers. Also, if the quotient is not exactly representable in the considered format, it must be nonterminating.

Property 4 Let $x, y \in \mathbb{M}_n$. If $x \neq y$, then the distance between $q = x/y$ and 1 is at least $2^{-n}$.

Proof of Property 4 For a given $y$ the values of $q$ closest to 1 are obtained when $x = y^+$ or $y^-$. In fact if we consider $x = 2^a(1 - 2^{-n})$ and $y = 2^b$ then this minimal distance is actually attained as:

$$\left|\frac{x}{y} - 1\right| = 2^{-n}.$$

From the illustration we can see that it is actually the distance between 1 and its predecessor ($1^-)$.

The next result gives a lower bound on the distance between a breakpoint (in round-to-nearest mode) and the quotient of two floating-point numbers. This is a very interesting property as it helps to reduce some unwanted conditions (see Theorem 3) in the main algorithm and dramatically increases the probability of the feasibility of the algorithm to more than $3/4$. 


Property 5 If \(x, y \in \mathbb{M_n}\) and \(1 \leq x, y < 2\), then the distance between \(q = x/y\) and the middle of two consecutive floating-point numbers (or BP) is lower-bounded by

- \(\frac{1}{y \times 2^{2n-1}} > \frac{1}{2^{2n}}\) if \(x \geq y\);
- \(\frac{1}{y \times 2^{2n}} > \frac{1}{2^{2n+1}}\) otherwise.

Moreover, if the last mantissa bit of \(y\) is a zero, then the lower bounds become twice these ones.

Proof of Property 5. The numbers \(X = x \times 2^{n-1}\) and \(Y = y \times 2^{n-1}\) are integers. Now there exist two cases:

1. If \(x \geq y\). A breakpoint (for the round-to-nearest mode) can be written as the sum of two consecutive floating-point numbers divided by 2. So it has the form \(P/2^n\) where \(P\) is odd. Now,

\[
\left| \frac{X}{Y} - BP \right| = \left| \frac{2^nX - PY}{Y2^n} \right|
\]

(1)

2. If \(x < y\). In this case breakpoint has the form \(P/2^{n+1}\) where \(P\) is odd. Now,

\[
\left| \frac{X}{Y} - BP \right| = \left| \frac{2^{n+1}X - PY}{Y2^{n+1}} \right|
\]

(2)

If the numerators of above equation are zero then, since \(P\) is odd, \(Y\) must be a multiple of \(2^n\) which is impossible. Hence, the absolute value of these numerators is at least one. So the distance between \(x/y\) and a breakpoint is:

\[
\frac{1}{y \times 2^{2n-1}}, \text{ if } x \geq y
\]

and

\[
\frac{1}{y \times 2^{2n}}, \text{ if } x < y.
\]

Moreover, if \(Y\) is even, the numerators are even numbers, so their absolute value is at least 2 and the above distances become:

\[
\frac{1}{y \times 2^{2n-2}}, \text{ if } x \geq y
\]

and

\[
\frac{1}{y \times 2^{2n-1}}, \text{ if } x < y.
\]

The illustration below shows the interval \((BP-2^{-2n}, BP+2^{2n})\) in which the quotient \(q = x/y\) lies.

3 State-of-the-art

3.1 Classical Division Methods

Many classes of algorithms exist for implementing division. These include the subtractive method, the multiplicative method, various approximations methods, and special methods such as the CORDIC and continued product methods [14]. The most commonly used algorithms in modern FPU's are the subtractive and multiplicative methods.
3.1.1 Subtractive Algorithms

Digit Recurrence algorithms use subtractive methods to calculate quotients one digit per iteration. SRT (after the name of their inventors, Sweeney, Robertson and Tocher) [4, 10] division is the most common digit recurrence division algorithm. SRT is a non-restoring division algorithm that is basically a trial and error process. It utilizes the following recurrence:

\[ p_{j+1} = r p_j - q_{j+1} D \]

where, \( P_b \) is the dividend, \( D \) is the divisor, \( r \) is the radix and \( q_{j+1} \) is the \( j + 1 \)st digit of the quotient represented in radix \( r \). To calculate a next partial remainder, the divisor is multiplied by the next quotient digit, and the result is subtracted from the product of the last partial remainder or residual, or dividend for the first iteration, and a radix \( r \). In each iteration, one digit of the quotient is determined by the quotient-digit selection function. The final quotient after \( k \) iterations is then obtained as:

\[ q = \sum_{j=1}^{k} q_j r^{-j} \]

The remainder is computed from the final residual by:

\[
\text{remainder} = \begin{cases} 
P_n & \text{if } P_n \geq 0 \\ 
P_n + D & \text{if } P_n < 0 
\end{cases}
\]

Furthermore, the quotient has to be adjusted when \( P_n < 0 \) by subtracting 1 ulp. In such algorithms, some range of digits is decided upon for the allowed values of the quotient in each iteration like for radix \( r \) there are exactly \( r \) allowed values but to increase the performance some redundancy is added to the digit set. Generally, the radix is increased in order to have a low overall latency.

3.1.2 Multiplicative Algorithms

These algorithms utilize multiplication as the fundamental operation. The primary difficulty with subtractive division is the linear convergence to the quotient. Multiplicative division algorithms [20, 21] are able to take advantage of high-speed multipliers to converge to a result quadratically. Rather than retiring a fixed number of quotient bits in every cycle, Multiplication-based algorithms are able to double the number of correct quotient bits in every iteration. Division can be written as the product of the dividend and the reciprocal of the divisor, or

\[ Q = a/b = a \times (1/b) \]
where \(Q\) is the quotient, \(a\) is the dividend, and \(b\) is the divisor. In the Newton-Raphson (NR) algorithm, a priming function is chosen, which has its root equal to the reciprocal:

\[
f(X) = \frac{1}{X} - b = 0
\]

The NR equation is then applied to this function to find an approximation to the reciprocal:

\[
X_{i+1} = X_i \times (2 - b \times X_i)
\]

The corresponding error term is given by

\[
\epsilon_{i+1} = \epsilon_i^2(b)
\]

and thus the error in the reciprocal decreases quadratically.

A different method of deriving a division iteration is based on a series expansion and this method is known as Goldschmidt’s iteration. We will see later that they are two different ways of computing the same recurrence which exhibit quadratic convergence. The difference between these two approaches is that Newton-Raphson involves dependent multiplications while series expansion involves independent multiplications so it can take advantage of a pipelined multiplier to obtain higher performance in the form of lower latency per operation. The main disadvantage of multiplicative method is the difficulty in obtaining a correctly rounded result. There exist some techniques for obtaining correctly rounded results like computing 10 extra bits of precision in the quotient as in IBM 360/91 or use of fused multiply accumulate as in IBM RS/6000 to ensure accuracy greater than \(2n\) bits throughout the iterations.

### 3.2 Division methods in current architecture

IA-64 is a new 64-bit computer architecture jointly developed by Hewlett-Packard and Intel, and the Intel Itanium \(\text{TM}\) processor is its first silicon implementation. The centerpiece of the architecture is the \texttt{fma} (fused multiply-accumulate) instruction.

The idea behind a Multiply-Add (or “MAC” for Multiply-Accumulate”) instruction is that an expression like \((\pm a \times b \pm c)\) be evaluated in one instruction. Clearly, such an instruction can be used for performing multiplication only, by setting \(c = 0\), and add(or subtract) only by setting, for example, \(b = 1\). Many machines have a MAC. Beyond that, a Fused MAC evaluates \((\pm a \times b \pm c)\) with just one rounding error at the end rather than twice (for the multiply and then for add).

The IA-64 floating-point architecture was designed with three objectives in mind. First, it was meant to allow high-performance computations. Second, the architecture aims to provide high floating-point accuracy. Third, compliance with the IEEE Standard for Binary Floating-Point Arithmetic was sought. The IA-64 architecture defers floating-point and integer division to software. The only instruction specifically intended to support division is the \texttt{floating point reciprocal approximation} instruction, \texttt{frcpa}. This merely provides an approximate reciprocal which software can use to generate a correctly rounded quotient. There are several reasons for relegating division to software.

- By implementing division in software it automatically inherits the high degree of pipelining in the basic \texttt{fma} operations. So, many division operations can proceed in parallel, leading to much higher throughput in comparison with other typical hardware implementations.

- Greater flexibility is afforded because alternative algorithms can be substituted where it is advantageous. It is often the case that in a particular context a simpler algorithm suffices, e.g. because the IEEE rounding mode is known at compile time or even because only a moderately accurate result is required (e.g. in some graphics application).
In typical applications, division is not an extremely frequent operation, and so it may be that
die area on chip would be better devoted to something else.

Intel provides a number of recommended division algorithms, in the form of short straight-line
sequences of fma operations written in IA-64 assembly language [2, 23, 24, 28]. This approach to
division was pioneered by Markstein on the IBM RS/6000 family. Most algorithms have two sepa-
rate variants, one of which is designed to minimize latency (i.e. the number of clock cycles between
starting the operation and having the result available), and the other to maximize throughput (the
number of operations executed per cycle, averaged over a large number of independent instances).
Which variant is best to use depends on the kind of program within which it is being invoked. The
later variant allows the best utilization of the parallel resources of the IA-64, yielding the minimum
time per operation when performing the operation on multiple sets of operands. All algorithms
perform an initial approximation step (frcpa) and then refine the resulting approximation to give
a correctly rounded result, using power series or iterative methods such as the Newton-Raphson
or Goldschmidt iteration. Now we will see how fma operations can be used to refine an initial
reciprocal approximation towards a better reciprocal or quotient approximation.

Consider determining the reciprocal of some floating point value y. Starting with a reciprocal
approximation z with a relative error ε:

\[
z = \frac{1}{y}(1 + \epsilon)
\]

we can just perform one fma operation:

\[
e = 1 - yz
\]

and get:

\[
e = 1 - yz = 1 - y \frac{1}{y}(1 + \epsilon) = -\epsilon
\]

We can observe that:

\[
\frac{1}{y} = \frac{z}{(1 + \epsilon)} = z(1 - \epsilon + \epsilon^2 - \epsilon^3 + \cdots) = z(1 + \epsilon + \epsilon^2 + \epsilon^3 + \cdots)
\]

So we can improve our reciprocal approximation by multiplying z by some truncation of the series
\[1 + \epsilon + \epsilon^2 + \epsilon^3 + \cdots\]. The simplest case using a linear polynomial in \(\epsilon\) can be done with just one more
fma operation.

\[
z' = z + \epsilon z
\]

Now we have, \(z' = \frac{1}{y}(1 - \epsilon^2)\). The magnitude of the relative error has thus been squared, or we can say,
the number of significant bits has been approximately doubled. If we closely look, it is in fact
a step of the traditional Newton-Raphson iteration for reciprocals. In order to get a still better
approximation, one can either use a longer polynomial in \(\epsilon\), or repeat the Newton-Raphson linear
correction several times. But, repeating Newton-Raphson iteration \(n\) times is equivalent to using
a polynomial \(1 + \epsilon + \cdots + \epsilon^{2^n-1}\), e.g. since \(\epsilon' = \epsilon^2 = \epsilon^2\), two iterations yield:

\[
z'' = z(1 + \epsilon)(1 + \epsilon^2) = z(1 + \epsilon + \epsilon^2 + \epsilon^3)
\]

However which approach is better depends on a careful analysis of efficiency and the impact of
rounding error. The Intel algorithms use both, as appropriate. We have presented state-of-the-art
technique for performing division in software. As we know that our aim is to know if there exist
a better possible solution if we know the divisor in advance. We present three main methods to
achieve our goal with different speeds and accuracies but it depends on the requirements of an
application to implement a particular method.
4 The naive method

This method comprises of two steps. As we have to calculate $x/y$, and $y$ is known before $x$ (either at run-time, or at compile time) so the first step is to pre-compute $z = 1/y$ (or more precisely $z$ rounded-to-nearest, that is, $z_h = o_n(1/y)$), and then to multiply $x$ by $z_h$ followed by rounding of $xz_h$ in the second step once $x$ is known.

It is well known that this approach doesn’t always give correctly-rounded result so it may or may not be useful to examine it further depending on the requirement. As this method is quite fast and gives straightway the result without much computation, so if the accuracy is not of much concern or the IEEE754 standard has not to be followed strictly (as in some graphics application) then one can investigate it further. Moreover, if the probability of getting an incorrect result is small enough to neglect then the following strategy can be considered:

- We assume that naive method always works and so we perform all the divisions using this method.
- The remainder is computed simultaneously without disturbing the normal computation, to check whether the division was correctly rounded. We will later see how the remainders can be computed exactly with a fused-MAC.
- If the division was not correctly rounded, the result of the division is corrected using the computed remainder, and the computation is started again at that point.

As a matter of fact, it turns out that the probability of getting an incorrectly rounded result is far too large to use this strategy, as we will soon see. Now we will focus on the properties of the naive method that will be used further by better algorithms.

4.1 Maximum error of the naive solution

The following property gives the maximum error committed in terms of “ulps” in two different cases.

Property 6 The naive solution returns a result that is at most at a distance of:

- 1 ulp from the exact result if $m(x) \geq m(y)$;
- 1.5 ulps from the exact result if $m(x) < m(y)$

where $m(u)$ is the mantissa of $u$.

Proof of Property 6. Let $x, y \in M_n$. Since the cases $x, y = 1$ or 2 are straightforward, we assume that $x$ and $y$ belong to $(1, 2)$. Thus $1/y \notin M_n$. Since $z_h = o_n(z)$ and $z \in (1/2, 1)$, we have,

$$\left| \frac{1}{y} - z_h \right| < 2^{-n-1}.$$ 

This come from the fact that rounding error committed is half of ulp in case of rounding to the nearest-even. Here the ulp is $1/2^n$.

Therefore,

$$\left| \frac{x}{y} - xz_h \right| < 2^{-n}. \quad (3)$$

13
Property 4 and (3) show that we cannot have \( x/y > 1 \) and \( xz_h < 1 \) at the same time or the converse so both of these quantities lie on either side of 1 or in other words they both have same ulp. So \( xz \) and \( xz_h \) belong to the same “binade” (i.e., \( \text{ulp}(xz_h) = \text{ulp}(xz) \)).

Now, there are two possible cases:

- if \( x \geq y \), then
  \[
  |xz_h - o_n(xz_h)| \leq 2^{-n}
  \]
  Since, \( x \geq y \) so \( xz_h \geq 1 \) and \( \text{ulp}(xz_h) = 1/2^{n-1} \) so
  \[
  \left| \frac{x}{y} - o_n(xz_h) \right| < 2^{-n+1} = \text{ulp}(x/y).
  \]

- if \( x < y \), then
  \[
  |xz_h - o_n(xz_h)| \leq 2^{-n-1}
  \]
  Since \( xz_h < 1 \) so \( \text{ulp}(xz_h) = 1/2^n \) so
  \[
  \left| \frac{x}{y} - o_n(xz_h) \right| < 3 \times 2^{-n-1} = 1.5 \times \text{ulp}(x/y).
  \]

The above results give an upper bound on the error encountered in the naive method but the next property tries to find whether the bound can be reduced in case of \( x < y \).

More precisely, if \( x < y \) and \( 1 \leq x, y < 2 \), the following property holds. This property will allow us to analyze the behavior of another algorithm (Algorithm 1).

**Property 7** If \( x < y \) and \( 1 \leq x, y < 2 \), then the naive solution returns a result \( q \) that satisfies:

- either \( q \) is within 1 ulp from \( x/y \);
- or \( x/y \) is at least at a distance
  \[
  \frac{2^{-2n+1}}{y} + 2^{-2n+1} - \frac{2^{-3n+2}}{y}
  \]
  from a breakpoint of the round-to-nearest mode.

The upper bound given by Property 6 on the maximum error of the naive method is not an overestimation, as shown by the following property.

**Property 8** The maximum error of the naive algorithm converges to 1.5 ulps as \( n \to \infty \).

Table 1 gives couples \((x, y)\) for which the naive quotient \( q = o_n(x o_n (\frac{1}{y})) \) gives an error quite close to 1.5ulps. For the proof of this property readers can see [32].

### 4.2 Probability that naive solution always works

For the first few values of \( n \) (up to \( n = 13 \)), Table 3 shows that the proportion of couples \((x, y)\) for which the naive method gives an incorrectly rounded result seems to converge, as \( n \) grows, to a constant value that is around 27% which is by far too large to be neglected. We have computed these results through exhaustive testing.
4.3 Some values of $y$ for which the naive method always works

Depending on $n$, there are a very few values of $y$ (including, of course, the powers of 2) for which the naive method always works (i.e., for all values of $x$). These values for $n$ less than 13 are given in Table 4. It is an open question to have a fast algorithm for predicting such $y$'s.

We have presented the Naive method of performing the division. It is fast and straightway computes the quotient but lacks in accuracy. Although, we didn’t explore more about this technique but it can be judged by its properties that the application which requires fast division algorithm and doesn’t care much for accuracy can be benefited from this method.

5 Division with one multiplication and two fused MACs

This approach is not far from the naive method. After performing the first step of naive method i.e. pre-compute the reciprocal of $y$, a remainder is computed to correct the final result.

This method is oriented towards the architecture having fused MAC instruction. The IBM PowerPC and Apple Power Macintosh, both derived from the IBM RISCSystem/6000 [13] architecture claim to conform to IEEE754 but use a "Fused" Multiply-Add instruction oftenly.

The theorem below shows the use of fused MAC instruction to get a correctly rounded result.

**Theorem 1 (Markstein, 1990 [3, 13])** Assume $x, y \in \mathbb{M}_n$. If $z_h$ is within $1/2$ ulp of $1/y$ and $q \in \mathbb{M}_n$, within $1$ ulp of $x/y$ then one application of

$$
\begin{align*}
    r &= o_n(x - qy) \\
    q' &= o_n(q + rz_h)
\end{align*}
$$

yields $q' = o_n(x/y)$.

This work originally belongs to Peter Markstein but several variations came up after that as can be seen in the literature[]. It computes remainder and requires no testing as it always gives correctly rounded result provided that $z_h$ and $q$ are available as required. This result was originally developed to get a correctly rounded quotient from an initial quotient approximated by Newton-Raphson or Goldschmidt iterations.

Obviously, instead of using Newton-Raphson iterations for an approximation of a quotient, this method can be benefited by using directly the $q$ generated by na"ive method if $q$ satisfies the condition of being within one ulp of $x/y$. Unfortunately, $q$ will not always be within one ulp from $x/y$ (see Property 6) so we could get a better initial approximation to $x/y$ by performing one step of Newton-Raphson iteration from $q$ generated by na"ive division. But theorem 2 below obviates the necessity of such iteration by directly using the quotient generated by naive method.

**Theorem 2 (Division with one multiplication and two Macs [12, 13])** Algorithm 1, given below, always returns the correctly rounded (to nearest) quotient $o_n(x/y)$.

**Algorithm 1 (Division with one multiplication and two Macs)**

- in advance, evaluate $z_h = o_n(1/y)$;
- as soon as $x$ is known, compute $q = o_n(x \times z_h)$;
- compute $r = o_n(x - qy)$;
- compute $q' = o_n(q + rz_h)$.
This method requires one division before \( x \) is known, and three consecutive (and dependent) MACs once \( x \) is known. In this section we studied the conventional approach for performing floating-point division and discussed about their speed and accuracy. This algorithm always works so on the same concept we try to do better and present a faster algorithm in the next section. Unfortunately, this algorithm doesn’t work for some divisors so in gaining speed we had to sacrifice on accuracy for them or it demands the internal precision larger than the target.

In the following section we present a faster algorithm by slightly improving this one and show that one MAC is sufficient.

6 An efficient technique

6.1 Performing Division with one multiplication and one fused MAC

In the last section we presented an algorithm using one multiplication and two fused-MAC instructions. It is a conventional algorithm which was designed keeping accuracy as the foremost objective. In this section we try to find an answer to the question “Can we have a better (fast) algorithm without sacrificing accuracy?” Following the trail of previous algorithm we try to show that we can save one fused-MAC instruction. A major difference between both techniques is the amount of pre-computation being done with the divisor. Of course if we do much pre-computation as in this algorithm then we are consuming more compile time.

The division method we are going to present requires a double-word approximation to \( 1/y \), that is, two floating-point values \( z_h \) and \( z_t \) such that \( z_h = o_n(1/y) \) and \( z_t = o_n(1/y - z_h) \). Fused-MAC is necessary to compute these values quickly. The following two properties help to compute these values.

**Property 9** Let \( x, y, q \in \mathbb{M}_n \), such that

\[
q \in \{o_d(x/y), o_n(x/y)\}.
\]

Then the remainder \( r = x - qy \) is always computed exactly with a fused MAC. So there is no need to round the result: \( o_n(x - qy) = x - qy \).

**Proof of Property 9.** Assuming \( 1 \leq x, y < 2 \). Define

\[
K = \begin{cases}  
n + 1 & \text{if } q \leq 1 \\
n & \text{if } q > 1 \end{cases}
\]

The result \( r = x - qy \) must be less than \( y\text{ulp}(x/y) \) in order that \( |q - \frac{x}{y}| < \text{ulp}(x/y) \). We can see that \( r \) is a multiple of \( 2^{-K} \) that is less than \( 2^{-K+1}y \) which suffices to show that \( r \in \mathbb{M}_n \). Hence, it is always computed exactly regardless of any rounding mode.

The quantity \( r \) can be computed exactly if the full, double length product \( qy \) can be preserved, or if the hardware is capable of an “accumulate” instruction and allows all the bits of the product \( qy \) to participate in the sum. William Kahan also explains in [8] that to round \( q \) correctly its remainder must be obtained exactly. This is what the "Fused" in the fused-MAC is for, it fuses the rounding.

**Property 10** Assume \( y \in \mathbb{M}_n \), \( y \neq 0 \). The following sequence of 3 operations computes \( z_h \) and \( z_t \) such that \( z_h = o_n(1/y) \) and \( z_t = o_n(1/y - z_h) \).
\[ \bullet z_h = o_n(1/y); \]
\[ \bullet \rho = o_n(1 - yz_h) = (1 - yz_h); \]
\[ \bullet z_\ell = o_n(\rho/y) = o_n(1/y - z_h); \]

**Proof of Property 10.** From Property 9, \( \rho \) is computed exactly. Therefore it is exactly equal to \( 1 - yz_h \). Hence, \( \rho/y \) is equal to \( 1/y - z_h \) and \( z_\ell \) is equal to \( o_n(1/y - z_h) \). ■

The first step computes an approximation to \( 1/y \) straightaway and then rounds it to the nearest even. \( \rho \) is a remainder which can always be computed exactly with a fused-MAC so the next step is just a **fma** operation. The last step is an important one. It evaluates another approximation of \( 1/y \) but we can see that \( z_\ell \) is nothing but a rounding of the error committed in approximating \( 1/y \) to \( z_h \). **The idea is to have a very good approximation of the reciprocal of the divisor using the elements of \( M_n \) in order to be very close to it.** Now we will show in the following algorithm how these two values help to compute correctly rounded quotient almost always once the dividend is known.

### 6.1.1 The algorithm

We assume that from \( y \), we have computed \( z = 1/y \), \( z_h = o_n(z) \) and \( z_\ell = o_n(z - z_h) \). We suggest the following 2-step method:

**Algorithm 2 (Division with one multiplication and one MAC)** *Compute:*

\[ \bullet q_1 = o_n(xz_\ell); \]
\[ \bullet q_2 = o_n(xz_h + q_1). \]

Unlike the previous algorithm it requires only one multiplication and one MAC operation but is it as accurate as the previous one? Well, let us see with the help of some examples given below.

Now, we give some examples and try to compare the quotient \( q = x/y \) of the above algorithm with the correctly rounded values provided by IEEE Standard.

**Example 1:** For precision \( n = 7 \), \( x = \frac{73}{64} \) and \( y = \frac{127}{64} \):

\[ z_h = \frac{65}{128}, \quad z_\ell = -\frac{127}{32768}, \quad q_1 = -\frac{9}{2048}, \quad q_2 = \frac{37}{64} = o_n(x/y) \]

**Example 2:** For precision \( n = 8 \), \( x = \frac{226}{128} \) and \( y = \frac{251}{128} \):

\[ z_h = \frac{131}{256}, \quad z_\ell = -\frac{231}{131072}, \quad q_1 = -\frac{51}{16384}, \quad q_2 = \frac{115}{128} \neq o_n(x/y) = \frac{231}{256} \]

The quotients show that the algorithm doesn’t always work so it is not as accurate as the previous one. But, when we tried to investigate more about this algorithm we found some really interesting information which makes it worth exploring. We have captured all those properties in Theorem 3 followed by its detailed proof.

**Theorem 3** *Algorithm 2 gives a correct result (that is, \( q_2 = o_n(x/y) \)), as soon as at least one of the following conditions is satisfied:*

1. \( n \) is less than or equal to 7;
2. the last mantissa bit of $y$ is a zero;
3. $|z| < 2^{-n-2-e}$, where $e$ is the exponent of $y$ (i.e., $2^e \leq |y| < 2^{e+1}$);
4. for some reason, we know in advance that the mantissa of $x$ will be larger than that of $y$;
5. 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 $y$ (that is, $Y$ is the mantissa of $y$, interpreted as an integer).

**Proof of Theorem 3.** We have done exhaustive searching for the precision $n \leq 8$ and found that algorithm always works for $n \leq 7$ and for $n=8$ it doesn’t work for only one couple given in Example 2 above.

Let us deal with other cases. Without loss of generality, we can assume $x \in (1, 2)$ and $y \in (1, 2)$ as the cases $x = 1$ and $y = 1$ are straightforward. For $x = 1$, the quotient is just equal to $z_h$ as $q_2 = c_n(z_h + z_\ell) = z_h$. We will see later about this case. For $y=1$ the quotient is always exact. So, $\frac{z}{y} \in (1/2, 1)$. Hence, $z_h \in [1/2, 1]$. Since $y > 1$ and $y \in M_n$ so $y \in [1 + 2^{-n+1}, 2)$.

Therefore, $1/y \leq 1 - 2^{n+p-1} + 2^{n+p-1} - 2^{3n+p-3} + \cdots < 1 - 2^{-n} \in M_n$ and so $z_h = c_n(1/y) \in [1/2, 1 - 2^{-n}]$.

We want an estimate of the error we are committing in the algorithm so we calculate every rounding error starting from the first step of the pre-computation and proceeding towards the computation of final quotient. Since $z_h$ is obtained by rounding $z$ to the nearest, we have:

$$|z - z_h| \leq \frac{1}{2} \text{ulp}(z) = 2^{-n-1}$$

. Property 1 shows that the case $|z - z_h| = 2^{-n-1}$ is impossible unless until $z$ lies at the exact middle of two consecutive floating-point which is only possible when $y$ is a power of 2. Therefore,

$$|z - z_h| < 2^{-n-1}.$$
Figure 5: Illustration of the inequality

From this, we deduce: \(|z_{\ell}| = |\alpha_n (z - z_h)| \leq 2^{-n-1}\). Again, the case \(|z_{\ell}| = 2^{-n-1}\) is impossible. This implies:

\[
|z - (z_h + 2^{-n-1})| < 2^{-2n-2} < 2^{-2n-1}
\]
or

\[
|z - (z_h - 2^{-n-1})| < 2^{-2n-2} < 2^{-2n-1}
\]

But it contradicts the fact that the binary representation of the reciprocal of an \(n\)-bit number cannot contain more than \(n - 1\) consecutive zeros or ones [7, 11]. Therefore:

\[
|z_{\ell}| < 2^{-n-1}.
\]

Now,

\[
|z_h + z_{\ell}| < |z_h + \frac{1}{2} \text{ulp}(z_h)|
\]

Therefore,

\[
\alpha_n (z_h + z_{\ell}) = z_h
\]

So for \(x = 1\) the quotient is just equal to \(z_h\).

Thus, from the definition of \(z_{\ell}\):

\[
|(z - z_h) - z_{\ell}| < \frac{1}{2} \text{ulp}(z - z_h) = 2^{-2n-2},
\]

thus,

\[
|x(z - z_h) - xz_{\ell}| < 2^{-2n-1},
\]

thus,

\[
|x(z - z_h) - \alpha_n (xz_{\ell})| < 2^{-2n-1} + \frac{1}{2} \text{ulp}(xz_{\ell}) = 2^{-2n-1} + 2^{-2n-1} < 2^{-2n}.
\]

Therefore,

\[
|xz - \alpha_n [xz_h + \alpha_n(xz_{\ell})]| < 2^{-2n} + \frac{1}{2} \text{ulp} (xz_h + \alpha_n(xz_{\ell})). \tag{4}
\]

The above equation gives the difference between the original division \(xz_h\) and the quotient produced by the Algorithm 2 or in other words we can say that the maximum error committed by this algorithm can be represented by the inequality above.

For condition 2 & 4 If for a given \(y\) there does not exist any \(x\) such that \(x/y = xz\) is at a distance less than \(2^{-2n}\) from the middle of two consecutive floating-point numbers i.e. a Break-Point, then \(\alpha_n [xz_h + \alpha_n(xz_{\ell})]\) will always be equal to \(\alpha_n(xz)\), and so Algorithm 2 will always give correct result.
But this is not always the case as shown by Example 2 so we have to segregate the couples \((x, y)\) for which \(q = x/y\) lies in the interval \((BP - 2^{-2n}, BP + 2^{-2n})\) where \(BP\) is the Break-point. Now, if we go back to section 2 we see that Property 5 drastically reduces the number couples to be checked against above condition. It says that if the last mantissa bit of \(y\) is zero then there doesn’t exist any such couple which lies in the above said interval\(^1\) and so the probability that algorithm works due to that is \(\geq 1/2\). It also says that if \(x \geq y\) then Algorithm 2 will return a correctly rounded quotient and so the probability is \(\geq 3/4\). Of course, this is not an interesting condition as the dividend is not known so we have to wait till the dividend is available but it is very expensive to do comparison at run-time. This property is only interesting if somehow we know in advance that mantissa of \(x\) is larger than that of \(y\).

For the condition 3 If \(|z| < 2^{-n-2}\) then following the above proof we get a tighter bound:

\[
|xz - o_n[xz_h + o_n(xz_t)]| < 2^{-2n-1} + \frac{1}{2}\text{ulp}(xz_h + o_n(xz_t))
\]

and Property 5 implies that we get a correctly rounded quotient. For a particular precision \(n\), among the \(2^{n-2}\) odd divisors, the table ?? shows that more than 48% divisors satisfy the condition \(|z| < 2^{-n-2}\) for a precision \(10 \leq n \leq 21\).

Let us now focus on the case \(x < y\). Let \(q \in \mathbb{M}_n\), \(1/2 \leq q < 1\), and define integers \(X, Y\) and \(Q\) as

\[
\begin{align*}
X &= x \times 2^{n-1} \\
Y &= y \times 2^{n-1} \\
Q &= q \times 2^n
\end{align*}
\]

If we have

\[
\frac{x}{y} = q + 2^{-n-1} + \epsilon, \text{ with } |\epsilon| < 2^{-2n},
\]

then

\[
2^{n+1}X = 2QY + Y + 2^{n+1}Y, \text{ with } |\epsilon| < 2^{-2n}.
\]

But:

- Equation (6) implies that \(R' = 2^{n+1}Y\) should be an integer.
- The bounds \(Y < 2^n\) and \(\epsilon < 2^{-2n}\) imply \(|R'| < 2\);
- Property 1 implies \(R' \neq 0\).

Hence, the only possibility is \(R' = \pm 1\). Therefore, to find values \(y\) for which for any \(x\) Algorithm 2 gives a correct result, we have to examine the possible integer solutions to

\[
\begin{align*}
2^{n+1}X &= (2Q + 1)Y \pm 1 \\
2^{n-1} &\leq X \leq 2^n - 1 \\
2^{n-1} &\leq Y \leq 2^n - 1 \\
2^{n-1} &\leq Q \leq 2^n - 1
\end{align*}
\]  

(7)

There are no solutions to (7) for which \(Y\) is even so Algorithm 2 always returns a correctly rounded result as was shown by Property 5 also. Now, if \(Y\) is odd then it has a reciprocal modulo \(2^{n+1}\). If there exist any solution to above set of equations then for that couple \((x, y)\) our algorithm may or may not round correctly the last bit of quotient but in many cases there exist no solution. The following algorithm 3 finds the solution to above equations as a Maple program.

\[1(BP - 2^{-2n}, BP + 2^{-2n})\]
Algorithm 3 (Tries to find solutions to Eqn. (7).) We give the algorithm as a Maple program. If it returns “true” then Algorithm 2 returns a correctly rounded result. 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
else
Pplus := $2^n$ - Pminus;
Qplus := (Pplus-1) / 2;
Xplus := Y - Xminus;
if (Qplus >= $2^{n-1}$ and (Xplus >= $2^{n-1}$))
then OK := false else OK := true end if; end if;
print(OK)
end proc;

This algorithm is one of the cornerstone of this work. It gives information about the uncorrectly rounded couples and as a matter of fact it computes in advance correctly rounded quotients for these couples.

Define

$$X^- = \frac{P^-Y-1}{2^{n+1}}$$

$$X^+ = \frac{P^+Y+1}{2^{n+1}}$$

We know that $P^- + P^+ = 2^{n+1}$ so we can easily show that $X^- + X^+ = Y$. Now either $X^-$ or $X^+$ lies in the interval $[2^{n-1}, 2^n - 1]$ as $2^{n-1} \leq Y \leq 2^n - 1$. Hence, either $Y, X^+, Q^+$ or $Y, X^-, Q^-$ can be solution to Eq. (7), but both are impossible. Algorithm 3 checks these two possible solutions. This explains the last condition of the theorem.

Let us recapitulate the conditions of Theorem 3 and discuss their consequences.

- The probability that the last mantissa bit of of $y$ is a zero is 1/2. This allows to accelerate half divisions. Also, this condition can be easily checked on most systems so it is worth when $y$ is known at run-time, soon enough before $x$.

- Although there is no fixed reason for having mantissa of $x$ larger than that of $y$ but we can compare them at compile-time and if $x \geq y$ then the algorithm can be applied straight away.

- Assuming a uniform distribution of $z_\ell$ in $(-2^{-n-1} - \epsilon, +2^{-n-1} - \epsilon)$, which seems reasonable (see [5]), Condition $|z_\ell| < 2^{-n-1-\epsilon}$ allows to accelerate half remaining cases;

- The condition “Algorithm 3 returns true” allows to accelerate around 39% of the last remaining cases which can’t be checked by other conditions. So these these cases are all the odd divisors greater than the dividend. It is worth being noticed that this algorithm also computes the only possible $X = 2^{n-1}x$ 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 $x$ (that is, to try to divide $x$ by $y$ at compile time) to know if the algorithm will always work, or if it will work for all X’s but this one. Interestingly, this algorithm computes the correctly
rounded quotient also for the un-correctly rounded couple. If either of $Q^-$ or $Q^+$ lies in a good range then we can give the following equation:

$$Q^- = a_n \left( \frac{X}{Y} \right)$$

$$Q^+ = a_n \left( \frac{X}{Y} \right) - 1$$

- Exhaustive searching convinces that algorithm almost always work. Table 5 shows that for $n \leq 29$, there are more than 98.7% of values of $y$ for which the algorithm returns a correctly rounded quotient for all possible values of $x$.

And yet, all this requires much more computation: it is probably not interesting if $y$ is not known at compile-time.

6.2 Can a larger precision help?

It is frequent that a larger precision than the target precision is 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, $o_{t+p}(x)$ means $x$ rounded to $n + p$ bits, according to rounding mode $t$.

Define $z = 1/y$. We assume that from $y$, we have computed $z_h = o_n(z)$ and $z_t = o_{n+1}(z - z_h)$. They can be computed through (the proof is similar to the one of Property 10):

- $z_h = o_n(1/y)$;
- $\rho = o_n(1 - yz_h)$;
- $z_t = o_{n+1}(\rho/y)$;

We suggest the following 2-step method:

**Algorithm 4 (Division with one multiplication and one MAC)** Compute:

- $q_1 = o_{n+1}(xz_t)$;
- $q_2 = o_n(xz_h + q_1)$.

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

**Proof of Theorem 4.**

As previously, we can assume $x \in (1/2, 1)$. The proof of Theorem 3 is immediately adapted if $x \geq y$, so that we focus on the case $x < y$. Using exactly the same computations as in the proof of Theorem 3, we can show that

$$|xz - o_n(xz_h + o_{n+1}(xz_t))| < 2^{-2n-1} + \frac{1}{2} \text{ulp}(xz_h + o_{n+1}(xz_t)),$$

and Property 5 implies that we get a correctly rounded quotient.

It is worth being noticed that 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 format, if the target precision is double precision.
7 Proposed Vs Conventional

We will compare our division method with the conventional one. Of course the proposed method is faster but not always accurate, yet we can compare the floating-point latencies and throughput of different methods presented in order to see our gain. We will give a following sequence of instructions from [12] for performing double-precision division.

Algorithm 5 (Double precision division. This is Algorithm 8.10 of [12])

\[
\begin{align*}
z_1 &= \text{frcpa}(y); \\
e &= o_n(1 - yz_1); \\
z_2 &= o_n(z_1 + z_1 e); \quad e_1 = o_n(e \times e); \\
z_3 &= o_n(z_2 + z_2 e_1); \quad e_2 = o_n(e_1 \times e_1); \\
z_4 &= o_n(z_3 + z_3 e_2); \\
q_1 &= o_n(xz_4); \\
r &= o_n(x - yq_1); \\
q &= o_t(q_1 + rz_4).
\end{align*}
\]

All the intermediate computations are being done in double-extended precision provided by IA-64. The final result is obtained in a target precision using the rounding mode \(o_t\). This result is the correctly rounded quotient. It is worth to notice that the last three steps of this algorithm resemble to the instructions of Algorithm 1 except that the above algorithm assumes the availability of extended precision.

The above algorithm requires 8 floating-point latencies, and uses 10 instructions. Another algorithm suggested by Markstein for extended precision requires 8 floating-point latencies and uses 14 floating-point instructions but we cannot always assume the availability of extended precision. We can replace 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. Algorithm 1 replaces it by 3 floating-point latencies and Algorithm 2 by 2 if it satisfies the conditions of Theorem 3. Whenever an even slightly larger precision is available (one more bit suffices), Algorithm 4 is of interest, since it requires 2 floating-point latencies instead of 3. Algorithm 2 is certainly interesting if it is known or if it can be checked that \(y\) is even or \(|x| < 2^{-n-2-e}\). In the other cases we must run Algorithm 3 at compile-time to check whether algorithm can be used or not and so it is only suitable for applications having enough compile-time.

8 Conclusion

We have proposed various methods for accelerating division of the form \(x/y\) when the divisor \(y\) is known before \(x\). We recommend the following points to implement them accordingly:

- A direct and fast way is to simply follow the naive method. It requires only to pre-compute \(1/y\) and gives result directly. It is not always accurate so can be used by some applications which prefer speed to accuracy.
• If there is not enough compile time for much pre-computation then Algorithm 1 can be very useful as it always works. So if $y$ is known only few tens of cycles before $x$ it gives a good combination of speed and accuracy.

• Algorithm 2 is faster but requires much pre-computation for computing $z_h, z_l$ and to execute Algorithm 3 in order to ensure that algorithm always works. So it can be worth using it if division is done by a constant.

• Algorithm 4 closely resembles Algorithm 2 in number of instructions but it always works if a slightly larger precision is available.
Table 1: Some attained error (in ulps) of the naïve solution for various values of \( n \).

<table>
<thead>
<tr>
<th>( n )</th>
<th>( x )</th>
<th>( y )</th>
<th>Error &gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>4294868995 ( \frac{2147483648}{2} )</td>
<td>65535 ( \frac{32768}{2} )</td>
<td>1.4999618524452582589456</td>
</tr>
<tr>
<td>53</td>
<td>268435449 ( \frac{154217728}{2} )</td>
<td>9007199120523395 ( \frac{4503599627370464}{2} )</td>
<td>1.499999739229677997443</td>
</tr>
<tr>
<td>64</td>
<td>18446744066117050369 ( \frac{9223472036854775808}{2} )</td>
<td>184467440661735550617 ( \frac{9223472036854775808}{2} )</td>
<td>1.499999994316597271551</td>
</tr>
<tr>
<td>113</td>
<td>288230376151711737 ( \frac{14411518807585555872}{2} )</td>
<td>103845937170696551129045804583584321 ( \frac{519229685854827628590496929220696}{2} )</td>
<td>1.49999999999999757138</td>
</tr>
</tbody>
</table>

Table 2: Number \( \delta(n) \) and percentage \( 100\delta(n)/2^{n-2} \) of values of \( y \) such that \( |z_\ell| < 2^{-n-2} \) among all the odd \( y \)'s which is a condition of Theorem 3.

<table>
<thead>
<tr>
<th>( n )</th>
<th>( \delta(n) )</th>
<th>percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>125</td>
<td>&gt; 48.82</td>
</tr>
<tr>
<td>11</td>
<td>259</td>
<td>&gt; 50.58</td>
</tr>
<tr>
<td>12</td>
<td>510</td>
<td>&gt; 49.80</td>
</tr>
<tr>
<td>13</td>
<td>998</td>
<td>&gt; 48.73</td>
</tr>
<tr>
<td>14</td>
<td>1990</td>
<td>&gt; 48.58</td>
</tr>
<tr>
<td>15</td>
<td>4106</td>
<td>&gt; 50.12</td>
</tr>
<tr>
<td>16</td>
<td>8255</td>
<td>&gt; 50.38</td>
</tr>
<tr>
<td>17</td>
<td>16396</td>
<td>&gt; 50.03</td>
</tr>
<tr>
<td>18</td>
<td>32637</td>
<td>&gt; 49.80</td>
</tr>
<tr>
<td>19</td>
<td>65424</td>
<td>&gt; 49.91</td>
</tr>
<tr>
<td>20</td>
<td>130699</td>
<td>&gt; 49.85</td>
</tr>
<tr>
<td>21</td>
<td>262303</td>
<td>&gt; 50.03</td>
</tr>
</tbody>
</table>

25
Table 3: Actual probability of getting an incorrect result for small values of $n$.

<table>
<thead>
<tr>
<th>$n$</th>
<th>probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0.2485 ...</td>
</tr>
<tr>
<td>8</td>
<td>0.2559 ...</td>
</tr>
<tr>
<td>9</td>
<td>0.2662 ...</td>
</tr>
<tr>
<td>10</td>
<td>0.2711 ...</td>
</tr>
<tr>
<td>11</td>
<td>0.2741 ...</td>
</tr>
</tbody>
</table>

Table 4: The $n$-bit numbers $y$ between 1 and 2 for which, for any $n$-bit number $x$, $o_n(x \times o_n(1/y))$ equals $o_n(x/y)$.

<table>
<thead>
<tr>
<th>$n$</th>
<th>y</th>
<th>$1/2$</th>
<th>$1/3$</th>
<th>$1/5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>1</td>
<td>0.99</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>1.01</td>
<td>1.28</td>
<td>1.28</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>1.03</td>
<td>1.23</td>
<td>1.23</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1.00</td>
<td>1.09</td>
<td>1.09</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>1.09</td>
<td>1.09</td>
<td>1.09</td>
</tr>
</tbody>
</table>
Table 5: Number $\gamma(n)$ and percentage $100\gamma(n)/2^{n-1}$ of values of $y$ for which Algorithm 2 returns a correctly rounded quotient for all values of $x$. For $n \leq 7$, the algorithm always works.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$\gamma(n)$</th>
<th>percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>64</td>
<td>100</td>
</tr>
<tr>
<td>8</td>
<td>127</td>
<td>&gt; 99.218</td>
</tr>
<tr>
<td>9</td>
<td>254</td>
<td>&gt; 99.218</td>
</tr>
<tr>
<td>10</td>
<td>510</td>
<td>&gt; 99.609</td>
</tr>
<tr>
<td>11</td>
<td>1011</td>
<td>&gt; 98.730</td>
</tr>
<tr>
<td>12</td>
<td>2022</td>
<td>&gt; 98.730</td>
</tr>
<tr>
<td>13</td>
<td>4045</td>
<td>&gt; 98.754</td>
</tr>
<tr>
<td>14</td>
<td>8097</td>
<td>&gt; 98.840</td>
</tr>
<tr>
<td>15</td>
<td>16175</td>
<td>&gt; 98.724</td>
</tr>
<tr>
<td>16</td>
<td>32360</td>
<td>&gt; 98.754</td>
</tr>
<tr>
<td>17</td>
<td>64686</td>
<td>&gt; 98.703</td>
</tr>
<tr>
<td>18</td>
<td>129419</td>
<td>&gt; 98.738</td>
</tr>
<tr>
<td>19</td>
<td>258953</td>
<td>&gt; 98.782</td>
</tr>
<tr>
<td>20</td>
<td>517591</td>
<td>&gt; 98.722</td>
</tr>
<tr>
<td>21</td>
<td>1035255</td>
<td>&gt; 98.729</td>
</tr>
<tr>
<td>22</td>
<td>2070463</td>
<td>&gt; 98.727</td>
</tr>
<tr>
<td>23</td>
<td>4140543</td>
<td>&gt; 98.718</td>
</tr>
<tr>
<td>24</td>
<td>8281846</td>
<td>&gt; 98.727</td>
</tr>
<tr>
<td>25</td>
<td>16563692</td>
<td>&gt; 98.727</td>
</tr>
<tr>
<td>26</td>
<td>33126395</td>
<td>&gt; 98.724</td>
</tr>
<tr>
<td>27</td>
<td>66254485</td>
<td>&gt; 98.726</td>
</tr>
<tr>
<td>28</td>
<td>132509483</td>
<td>&gt; 98.727</td>
</tr>
<tr>
<td>29</td>
<td>265016794</td>
<td>&gt; 98.726</td>
</tr>
</tbody>
</table>
Appendix: proof of the properties and theorems

Proof of Property 1 It obviously suffices to assume that $1 \leq y < 2$. Define $z = 1/y$ and assume $z \in M_q$. This gives:

- $Y = 2^{n-1}y$ is an integer;
- $Z = 2^qz$ is an integer (since $1/2 < z \leq 1$).

$YZ$ is equal to $2^{n+q-1}$. Therefore, in the prime number decomposition of $Y$ and $Z$, 2 is the only prime number that can appear.

Proof of Property 2 Let the significands of the floating-point numbers be $\sum_{i=0}^{n-1} a_i 2^{-i}$ and $\sum_{i=0}^{n-1} b_i 2^{-i}$ with $a_{n-1} \neq 0$ and $b_{n-1} \neq 0$. In the product, the coefficient of $2^{-(2n-2)}$, $a_{n-1}b_{n-1} \neq 0 \mod 2$ and hence the coefficient of $2^{-(2n-2)}$ is nonzero. $a_0b_0$ is nonzero as both the numbers lie between 1 and 2. So, the length of the product is at least $2n-1$.

Proof of Property 3 Let $q = x/y$, where $x$ and $y$ are the two $n$-bit numbers. Now if the quotient is exact and the length of $q$ is $m, m > n$ then, $x = yq$. By Property 2, the length of $x$ must be at least $m$, contradicting the assumption that $x$ is representable in precision $n$.

Proof of Property 7. The proof is very similar to that of Property 6. We are looking for tighter bounds:

1. From Property 5 we know that $1/y$ is at least at a distance of $2^{-2n}/y$ from a breakpoint so:

$$|1/y - z_h| < 2^{-n-1} - 2^{-2n}/y$$

2. $x \leq 2 - 2^{-n+2}$ (Since $x < y < 2$, which implies $x \leq (2^{-})^+$).

Combining these bounds gives

$$\left| \frac{x}{y} - xz_h \right| \leq 2^{-n} - \frac{2^{-2n+1}}{y} - 2^{-2n+1} + \frac{2^{-3n+2}}{y}.$$  

The final bound $\ell_{mn}$ is obtained by adding the 1/2 ulp bound on $[xz_h - o_n(xz_h)]$:

$$\left| \frac{x}{y} - o_n(xz_h) \right| \leq \ell_{min} = 3 \times 2^{-n-1} - \frac{2^{-2n+1}}{y} - 2^{-2n+1} + \frac{2^{-3n+2}}{y}.$$  

Now, if $o_n(xz_h)$ is not within 1 ulp from $x/y$, it means that $x/y$ is at a distance at least 1/2 ulp from the breakpoints that are immediately above or below $q = o_n(xz_h)$. And since the breakpoints that are immediately above $o_n(xz_h)^+$ or below $o_n(xz_h)^-$ are at a distance 1.5 ulps = $3 \times 2^{-n-1}$ from $o_n(xz_h)$, $x/y$ is at least at a distance $3 \times 2^{-n-1} - \ell_{min}$ from these breakpoints.

28
**Proof of Theorem 2.** We assume \(1 \leq x, y < 2\). First, let us notice that if \(x \geq y\), then (from Property 6), \(q\) is within one ulp from \(x/y\), therefore Theorem 1 applies, hence \(q' = o_n(x/y)\). Let us now focus on the case \(x < y\). Define

\[
\begin{align*}
\epsilon_1 &= \frac{x}{y} - q \\
\epsilon_2 &= \frac{1}{y} - z_h
\end{align*}
\]

From Property 6 and the definition of rounding to nearest, we have,

\[
\begin{align*}
|\epsilon_1| &< 3 \times 2^{-n-1} \\
|\epsilon_2| &< 2^{-n-1}
\end{align*}
\]

The number \(\rho = x - qy = \epsilon_1 y\) is less than \(3 \times 2^{-n}\) and is a multiple of \(2^{-2n+1}\). This shows that it can be represented exactly with \(n + 1\) bits of mantissa. Hence, the difference between that number and \(r = o_n(x - qy)\) (i.e., \(\rho\) rounded to \(n\) bits of mantissa) is zero or \(\pm 2^{-2n+1}\). Therefore,

\[
r = \epsilon_1 y + \epsilon_3, \text{ with } \epsilon_3 \in \{0, \pm 2^{-2n+1}\}.
\]

Let us now compute \(q + rz_h\). We have

\[
q + rz_h = \left(\frac{x}{y} - \epsilon_1\right) + \left(\epsilon_1 y + \epsilon_3\right) \left(\frac{1}{y} - \epsilon_2\right) = \frac{x}{y} + \frac{\epsilon_3}{y} - \epsilon_1 \epsilon_2 y - \epsilon_2 \epsilon_3
\]

Hence,

\[
\left|\frac{x}{y} - (q + rz_h)\right| \leq \frac{2^{-2n+1}}{y} + 3 \times 2^{-2n-2}y + 2^{-3n}
\]

Define \(\epsilon = \frac{2^{-2n+1}}{y} + 3 \times 2^{-2n-2}y + 2^{-3n}\). Now, From Property 7:

- either \(q\) was at a distance less than one ulp from \(x/y\) (but in such a case, \(q' = o_n(x/y)\) from Theorem 1);
- or \(q\) is at a distance larger than one ulp from \(x/y\). In such a case, \(x/y\) is at least at a distance

\[
\delta = \frac{2^{-2n+1}}{y} + 2^{-2n+1} - \frac{2^{-3n+2}}{y}.
\]

A straightforward calculation shows that, if \(n \geq 4\), then \(\epsilon < \delta\). It makes it possible to deduce that there is no breakpoint between \(x/y\) and \(q + rz_h\). Hence \(o_n(q + rz_h) = o_n(x/y)\). The cases \(n < 4\) are easily checked through exhaustive testing.

\[\blacksquare\]
References


