Laboratoire de l'Informatique du Parallélisme

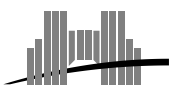
École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON
n° 5668

Software Carry-Save for Fast Multiple-Precision Algorithms

David Defour,
Florent de Dinechin

Février 2002

Research Report N° 2002-08



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

Software Carry-Save for Fast Multiple-Precision Algorithms

David Defour,
Florent de Dinechin

Février 2002

Abstract

This paper introduces a new machine representations of multiple-precision (MP) numbers, geared toward simple and fast implementations. We observe, in the usual high-radix representations, that carry management accounts for a lot of the complexity of the core multiple-precision algorithms (addition and multiplication). Our representation therefore trades space for simplicity : the digits of the multiple-precision numbers are coded on less bits than the machine numbers can offer. The reserved bits are used during MP addition and multiplication to guarantee that no overflow or rounding can occur in any internal computations. This leads to simple and therefore fast algorithms. In other words, all the carries generated in these internal computations are saved in these reserved bits, to be managed as late as possible. In addition to simplicity, this *software carry-save representation* allows to expose parallelism just like its hardware counterpart. An initial implementation of this representation is shown to compare favorably to other multiple-precision libraries.

Keywords: Multiple precision, representation, addition, multiplication.

Résumé

Cet article présente une nouvelle représentation machine des nombres multiprécision, permettant des implémentations simples et rapides. Nous observons que, dans les représentations usuelles en grande base, la propagation des retenues constitue une part importante de la complexité des algorithmes d'addition et multiplication. Nous proposons donc de coder les chiffres en grande base sur moins de précision que ce que la machine peut traiter. Les bits mis de côté sont utilisés, lors des additions et multiplications multiprécision, comme *retenue conservée* pour garantir qu'aucun dépassement de capacité n'a lieu pendant un calcul interne. Ceci conduit à des algorithmes plus simples, donc plus rapides même s'ils ont plus de chiffres à traiter pour une même précision. En outre, cette retenue conservée logicielle permet comme sa version matérielle des algorithmes parallèles au niveau du chiffre. Les tests réalisés montrent que les processeurs superscalaires récents peuvent en tirer parti.

Mots-clés: Précision multiple, représentation, addition, multiplication.

1 Introduction

Most modern computers obey the IEEE-754 standard for floating-point (FP) arithmetic, which defines the well-known *single* and *double precision* FP formats, along with the behavior of the usual arithmetic operators in several rounding modes. For some applications, however, the precision provided by this standard (53 bits in double precision) is not enough. These applications include large-scale numerical simulations, but also very small computations where precision is paramount. For instance, previous research has shown that an internal precision of around 200 bits is needed to provide correctly rounded elementary functions in double precision [9].

1.1 An overview of multiple-precision formats

In addition to scientific computing packages such as Maple or Mathematica, many multiple-precision libraries exist, such as Briggs' *double-double*[2], Bailey's *quad-double*[8], the *floating-point expansions* of Daumas et al [7], or the GNU Multi Precision (GMP) library [3] – on top of which several other libraries are built (see MPFR [4] or Arithmos [6]). These libraries are widely used for numerical analysis, cryptography or computational geometry. Some offer arbitrary precision with static or dynamic precision control, other simply offer a fixed precision which is higher than that of the IEEE-754 standard.

All of these libraries internally represent MP numbers as arrays of *digits*, which are themselves *machine numbers*, i.e numbers in one of the native format on which the microprocessor can directly compute: integer, or IEEE-754 floating-point numbers. According to the type of machine numbers used, there are thus two classes of multiple-precision libraries :

Integer-based libraries This is the case of commercial packages like Maple or Mathematica. It is also the case of the GNU Multi Precision (GMP) library, in which the digits are integers of the native machine word size (currently 32 or 64 bits). In GMP, the core loops of both addition and multiplication are written in assembly language for most existing architectures, and are thus very efficient (all algorithms for the division use additions and/or multiplications, and we will usually not mention them in this paper). Conversions between integer-based MP format and IEEE-754 FP format, however, are sometimes fairly expensive.

FP-based libraries Briggs'[2] *double double*, Bailey's *quad doubles* [8] and Daumas' *floating-point expansions* [7] consist of the unevaluated sum of several double-precision FP numbers. Arithmetic operations on this format use only FP operations. The motivation here is that the floating-point units (FPUs) in modern processors are highly optimized: for example, in recent 64-bit architectures like UltraSparc and Itanium, floating-point multiplications are faster than their integer counterpart. Besides, superscalar processors have several such units, and the trend is towards having more FPUs than integer units.

Another advantage of FP based multiple precision is that conversion from IEEE-754 floating point to such a format is trivial, and conversion back is relatively easy. Addition and multiplication, however, are more complicated: When adding or multiplying two FP number, some precision can

be lost to rounding, and has to be recovered. This requires sophisticated algorithms which have to be carefully proven, sometimes even requiring the assistance of automatic theorem provers [5].

The purpose of this article is to propose an alternative internal format for multiple precision which will address this last problem.

1.2 A new multiple-precision format for faster algorithms

Our format codes a MP number as a sum of integers or FP numbers, which we will also call *digits*. However, to ensure that no precision is lost in the FP multiplication of two digits, these digits don't use all the precision available in the machine format. They use sufficiently little precision to ensure that all the internal computation performed in a MP addition or multiplication are exact. A similar idea seems to be used by Abraham Ziv et al in their IBM mathematical library with correct rounding [1], although we couldn't find any article on the subject.

Knowing that all the internal operations will be exact greatly simplifies the algorithms for MP addition and multiplication. The format of course wastes space, as it stores many zeroes instead of useful information. It also wastes time, as it uses the machine operators to less than their full power. In the sequel, we discuss this tradeoff, and demonstrate some tests which show that the advantage of simpler algorithms prevails.

1.3 A carry-save representation expresses parallelism

Another very important aspect of the proposed format is that it allows to add several MP numbers with only one carry propagation in the end. In this respect, it is very similar to the *carry-save* representation commonly used in hardware multipliers: Our format can be thought of as a software, high radix carry-save representation, the carry being saved in the unused bits of each digit.

A parallel algorithm may seem a luxury in a software context. Indeed, as our tests will show, recent super-scalar processors, with their many functional units, can take advantage of it.

1.4 Outline of this paper

Section 2 presents in details the proposed representation, along with its main motivation in a MP multiplication. Section 3 details the algorithms used to perform MP additions/subtractions and multiplications. Section 5 presents some results of our initial implementation, comments and compares them to other available multiple-precision libraries. Section 6 concludes the paper.

2 A format for fast multiple-precision multiplication

2.1 Overview of the standard MP multiplication algorithm

There are several multiple-precision multiplication algorithms. The simplest one is similar to the multiplication algorithm learnt in elementary school, and

is depicted on Fig. 1. It should be pointed out that algorithms exist with a better asymptotic complexity (for instance the Karatsuba algorithm), however they become interesting for precisions much larger than those considered here.

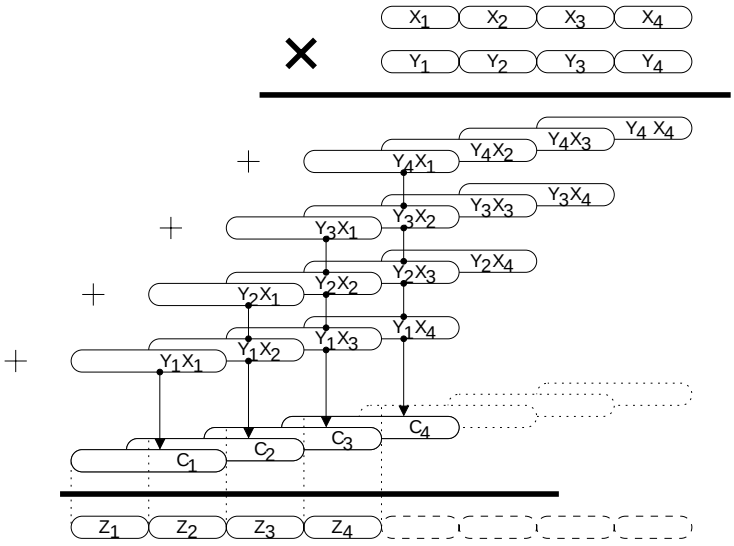


Figure 1: Multiple-Precision multiplication

The figure represents the two input numbers X and Y, decomposed into their n digits x_i and y_i (with $n = 4$ on the figure). Each digit is itself coded in m bits of precision: These digits will be machine integers in GMP ($m = 32$ or 64), or IEEE-754 double-precision numbers in Bailey's library ($m = 53$).

The algorithm (depicted on the figure) proceeds as follows:

- The first step is to produce an array of partial products $x_i y_j$. Each partial product is a $2m$-bit number.
- These partial products have then to be summed. There are at most n of them in a column, which means that the sum c_i in each column is a number of up to $2m + \lceil \log_2 n \rceil$ bits.
- Finally the column sums c_i are decomposed into numbers of only m bits. This is done in an iterative manner by summing from right to left the numbers of equivalent weight, storing the lower m bits as a digit of the result, and keeping the higher bits for the next iteration.

The exact product of two n-digit numbers is a $2n$-digit number. We consider here that we want the result as an n-digit number, therefore some rounding has to be done. In the sequel, we will simply not compute all the lower partial products, and give a bound on the error made. This choice has no influence on the validity of what follows. For example it is well known that correct rounding needs the computation of all the partial products, and an implementation of our format with correct rounding is possible.

With the previous description in mind, it should be clear that any implementation using, as digits, full machine numbers (be it integer or floating-point)

can not represent in one machine number the partial products, neither the column sums. Algorithmic tricks are therefore needed to get these values coded on two or more digits. In Bailey's FP based library, these tricks involve elaborate considerations on the rounding of the operations. In the case of GMP, specific assembly language instructions are used, exploiting at their best the facilities offered by each specific processor.

These various tricks imply in turn strong dependencies on the operations. In GMP, for instance, computing each line of partial product involves a carry propagation (each line is converted to a standard MP number). For a small n, this may lead to poorer performance than the processor can offer. In Bailey's library, the dependencies are not even regular.

2.2 The proposed format

The idea behind our format is therefore to sacrifice enough machine precision in coding the digits of X and Y to ensure that each column sum c_i can fit on a machine number. If we note m_I the precision offered by the machine format, then each digit in our representation will only use m_r bits of this precision, and this condition can be expressed as:

$$m_I > log_2(n_r) + 2m_r \tag{1}$$

This ensures that all the intermediate computations in the multiplication algorithms are exact, in the sense that their results fits in a machine number without any rounding error. This guarantee that the same holds for additions and subtractions.

Of course, we will need to compute a little bit more than four times the number of partial products computed in the dense implementations. But we observe that in these dense representation, computing and summing exactly these partial products ends up being more than four times more expensive than our simple, exact operations.

Equation (1) is a relation between the number m_r of bits stored on each machine number and the number of digits manipulated n_r. The overall precision is the product $n_r \times m_r$. Table 2 gives some values of these parameters for various machine word formats, using sensible values of m_r.

Machine word	m_I	m_r	n_r	max. precision
IEEE double precision	52+1 bits	26 Bits	15 terms	390 Bits
IEEE double precision	52+1 bits	25 Bits	63 terms	1575 Bits
Double-extended (x86)	64 bits	30 Bits	31 terms	930 Bits
Double-extended (x86)	64 bits	29 Bits	127 terms	3683 Bits
64-bit integer	64 bits	30 Bits	15 terms	450 Bits
64-bit integer	64 bits	29 Bits	63 terms	1827 Bits

Figure 2: Some examples of precisions that can be reached with the proposed format.

We believe that our format is interesting only for moderate precisions (a few hundred bits), because the waste of space and time increases as m_r decreases for a given m_I. However a detailed study of this tradeoff remains to be done.

2.3 Machine implementation

A MP number is represented in the proposed format as a structure R, as shown on Figure 3. The Software Carry Save Structure (SCSS) R is composed of the following fields:

R.digits[n_r] A table of n_r digits with m_r bits of precision, using either integer or FP machine numbers (we will chose the most efficient for a given machine);

R.index An integer storing the index of the first digit in the range of representable numbers, as depicted on Figure 3;

R.sign A sign information.

A SCSS number R is said to be *n*ormal when each of its digit store information only on the least m_r significant bits. The other bits from m_i to m_r of $R.digits$ field correspond to the carry field. If R have a digit with a carry field different from 0 then R is said to be unnormal.

In other words, the value x of a representation R is:

$$x = R.sign \times \sum_{j=1}^{n_r} R.digits[j] \times 2^{m_r*(R.index-j)} \tag{2}$$

A MP number is define as

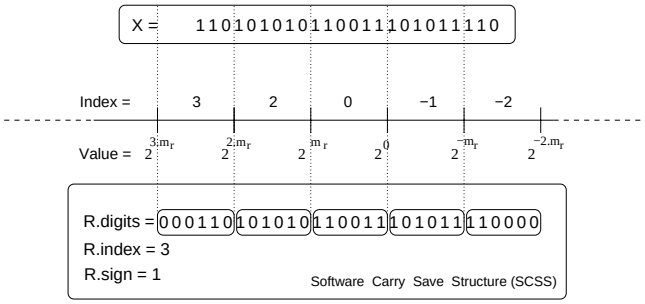


Figure 3: The proposed format

In the case when FP machine numbers are used, their exponent field is unused. However, a trick allows to exploit their sign bit , along with the well-defined rounding properties of the IEEE-754 standard, to increase the precision of each word by one bit, which explains some values of Table 2. Due to lack of space we do not detail this trick here.

3 Arithmetic operations

In this section we present algorithms used to perform basic arithmetic operations. For easier comprehension, we adopt a syntax very similar to the C language syntax.

3.1 Conversion from double to SCSS

There are two ways to convert a double precision floating point number d into an SCSS representation. The first solution is to extract the exponent d_{exp} from d and then determine the corresponding $R.index$. This can be done by taking the integer part of $\frac{d_{exp}}{2^{m_r}}$.

The second solution, described below, just uses some multiplications by 2^{-m_r}. As we don't have to extract the exponent from d, it is faster than the first when the exponent of the input number is close to 0.

Algorithm 1 (double2SCSS)

Input: *A double precision floating point number d.*

Output: *A multi-precision number R.*

Method:
$R.index = 0;$
if d **is a** $Nan, +/-\infty$ **then**
  $R = d;$ **return**;
if $d > 0$ $R.sign = 1;$
else $R.sign = -1;$ $d = -d;$
while $d > 2^{m_r}$ **do** $R.index++;$ $d = d \times 2^{-m_r};$
while $d < 1$ **do** $R.index--;$ $d = d \times 2^{m_r};$
for$(i = 1 ; i \leq n_r ; i++)$
  $R.digits[i] = \lfloor d \rfloor;$
  $d = (d - R.digits[i]) \times 2^{m_r};$

3.2 Renormalisation (carry propagation)

This step is used at the end of additions or multiplications to propagate the saved carries. It may be executed only once after several consecutives SCSS additions.

Working from low order to high order digits, each digit is split into two parts. The low m_r bits are a digit of the normalized result, and the upper part is the carry that has to be added to the next digit. To take high part from a floating point digit, we use a well-known trick consisting of adding and then subtracting a constant. When working with integer, the split can be done by using an integer mask.

Algorithm 2 (Carry propagation)

Input: *A multi-precision number A where each digit are less than 2^{m_i-1}.*

Output: *A multi-precision number R with carry information set to 0.*

Method:
$C[n_k + 1] = 0;$
$R.index = A.index;$
for$(k = n_r ; k \geq 0 ; k--)$
  $High = (A[k] + (2^{m_i+m_r} + 2^{m_i+m_r-1}))$
    $-(2^{m_i+m_r} + 2^{m_i+m_r-1});$
  $C[k+1] += A[k] - High;$
  $C[k] = High \times 2^{-m_r};$

$k = 1$;
While $C[k] = 0$ **do** $k++$; $R.index--$;
for$(j = 1$; $j \leq n_r$; $j++$, $k++$ **)** $R.digits[j] = C[k]$;

Theorem 1 *This algorithm normalizes a SCSS number with a relative error less than* $2^{m_r \times (n_r - 1) + 1}$

Proof 1 *The worst case happens when there is a carry-out from* $C.digits[1]$. *In this case we have to perform a translation of digits that leads to the lost of the last digit. Then the result is composed of* $(n_r - 1)$ *digits with* m_r *bits of precision. The first digit has at least one bit of precision. Therefore the final relative error is then less than* $2^{m_r \times (n_r - 1) + 1}$

3.3 Addition and subtraction

The addition of two numbers SCSS is done by adding in parallel the digits of same order. These additions will be exact as long as each digit is less than $2^{m_i - 2}$. The advantage of this representation is therefore that we can sum-up a huge amount of SCSS numbers avoiding the time-expensive carry propagate step.

Example 1 *Let us consider integer arithmetic with* $m_i = 64$ *and* $m_r = 30$ *(see Table 2). Then we can add approximately* 8×10^9 *normal SCSS numbers.*

If the user wants to perform another operation different than addition, as for example a multiplication it is recommended to perform a "renormalisation" that set all the carry field to 0. This necessity of the renormalisation depends on the maximum value for each digits and on the value chosen for m_r, m_i and n_r.

The subtraction is very similar to the addition algorithm, although more depending on the choice of integer or FP format. It may also classically lead to a cancellation, which may need an update of the index of the result.

3.4 Multiplication

In the multiplication of two multiple precision numbers, we only perform the operations significant to the result. Contrary to the addition algorithm no dramatic cancellation of digits can appears. The SCSS multiplication is done in three steps, depicted on the figure (1):

1. Compute the partial products. These partial products correspond to digit per digit multiplications, and can be done in parallel.
2. Sum up the partial products of same order. Each column can be summed in parallel, as for the addition algorithm.
3. Call the renormalisation/carry-propagation procedure.

The last step may even be skipped in some contexts (typically, a multiplication followed by one or several additions as in polynomial evaluations, if the values of m_r, m_i, n_r allow). This may allow for further optimization.

4 Conversion from SCSS to floating-point

In applications where conversion from the multi-precision format to the working precision are done very often, this step needs to be done very fast. With SCSS representation, this conversion is composed of a few multiplications and additions. Note that no renormalisation of the input SCSS number is needed for this algorithm.

Algorithm 3 (Conversion to floating-point)

Input: *A SCSS number A.*

Output: *A double precision floating-point number d.*

Method: $d = A.sign \times 2^{A.index \times m_r} \times ($
$A.digits[0]+$
$2^{-m_r} \times A.digits[1]+$
$2^{-2.m_r} \times A.digits[2]+$
$2^{-3.m_r} \times A.digits[3]);$

The number of digits of a SCSS number to add in this algorithm depends on the parameters of the representation. In this algorithm we used 4 digits corresponding to the worst case for $m_i = 53$ and $m_r = 26$.

5 Results and timing

It is quite difficult to compare the performances of MP libraries. The cost of a multiplication or addition is very different from one machine to another one, and also depends on the context of this operation in today's sophisticated processors with deep pipelines and out-of-order execution. Even the definition of an operation is machine-dependent (32- or 64-bit, availability or not of a fused multiply-and-add).

We have written (in C) and refined several implementations of this library, with various machine formats and instruction ordering. In our tests, we use the same code, compiled with the same compiler (gcc.2.95.4), the only change being the machine format which can be floating-point or integer (we chose the best). In both cases we ensure 200 bits of precision.

We compare our code with the reference multiple-precision library GMP[3] (more precisely it floating representation MPF), with Bailey's quad-double library [8], and also with Ziv's library[1]. Each result is obtain by measuring the execution times on 10^6 random values (the same values are used for all the libraries). For clarity we normalized results to SCSS library.

Each additions and multiplications performed with SCSS algorithms include a call to the renormalisation procedure (carry propagate). So the following results don't show all the potentiality of our methods.

The multiplication times on Celeron and Itanium seem contradictory. An interpretation could be that we get the best results on the Itanium architecture, whose instruction set was designed with parallelism in mind, and therefore makes better use of our more parallel algorithms. However this hypothesis has to be taken with caution. A deeper investigation is needed to see what is the

Description	SCSS lib.	Ziv lib.	Bailey lib.	GMP lib.
Itanium 733 Mhz	1	13.05	1.73	1.72
x86 Celeron 566 Mhz	1	5.85	0.33	0.74
Sun Ultra-5 400 Mhz	1	1.22	0.32	0.37

Figure 4: Comparisons of conversion times from double-precision to multiple-precision

Description	SCSS lib.	Ziv lib.	Bailey lib.	GMP lib.
Itanium 733 Mhz	1	8.74	25.1	1.90
x86 Celeron 566 Mhz	1	2.35	9.0	1.41
Sun Ultra-5 400 Mhz	1	2.97	26.33	1.39

Figure 5: Comparisons of MP addition times

influence of the compiler and coding style on these results. It should also be mentionned that the quad-double library is written in C++ language, which may affect its performances.

6 Conclusion and future work

We have presented the software carry-save representation for multiple-precision numbers that allows simpler, more parallel, and therefore faster MP additions and multiplications. This representation can provide very high precision, however is probably most suited for limited precisions of up to a few hundred bits.

The main idea developed in this method is to avoid overflows/rounding/carries in the machine operations internal to a MP operation. Being parallel, the code is suited to modern highly super-scalar processors. Being simple, it is also fairly compact.

After our first tests, the most obvious limitation of this method is its high dependence to the architecture. We will try to investigate the interaction between code, compiler, and processor architecture to write code which performs well on most architectures. Using IEEE-754 normalized arithmetic should allow us to avoid GMP-like machine-specific assembly optimizations.

Future work also include real-world tests of this library, in particular to compute transcendental functions such as cosine, sine, exponential.

References

[1] Accurate portable mathematical library.
`http://oss.software.ibm.com/mathlib/`.

[2] Doubledouble floating point arithmetic.
`http://www-epidem.plantsci.cam.ac.uk/~kbriggs/doubledouble.html`.

[3] GMP, the GNU multi-precision library. `http://swox.com/gmp/`.

Description	SCSS lib.	Ziv lib.	Bailey lib.	GMP lib.
Itanium 733 Mhz	1	7.84	37.71	1.23
x86 Celeron 566 Mhz	1	0.81	8.45	0.32
Sun Ultra-5 400 Mhz	1	3.28	38.41	1.47

Figure 6: Comparisons of MP multiplication times

Description	SCSS lib.	Ziv lib.	Bailey lib.	GMP lib.
Itanium 733 Mhz	1	7.15	0.66	1.57
x86 Celeron 566 Mhz	1	6.0	0.19	2.37
Sun Ultra-5 400 Mhz	1	8.64	0.46	4.36

Figure 7: Comparisons of conversion times from MP to double-precision

[4] MPFR,. `http://www.loria.fr/equipes/polka/software.html`.

[5] Sylvie Boldo and Marc Daumas. A mechanically validated technique for extending the available precision. In *35th Asilomar Conference on Signals, Systems, and Computers*, Pacific Grove, California, 2001.

[6] A. Cuyt, P. Kuterna, B. Verdonk, and J. Vervloet. *Arithmos: a reliable integrated computational environment.* Available at http://win-www.uia.ac.be/u/cant/arithmos/index.html, 2001.

[7] Marc Daumas. Expansions: lightweight multiple precison arithmetic. In *Architecture and Arithmetic Support for Multimedia*, page 14, Dagstuhl, Germany, 1998.

[8] Yozo Hida, Xiaoye S. Li, and David H. Bailey. Algorithms for quad-double precision floating-point arithmetic. In Neil Burgess and Luigi Ciminiera, editors, *15th IEEE Symposium on Computer Arithmetic*, pages 155–162, Vail, Colorado, June 2001.

[9] V. Lefèvre, J.M. Muller, and A. Tisserand. Towards correctly rounded transcendentals. *IEEE Transactions on Computers*, 47(11):1235–1243, November 1998.

Acknowledgements

The support of Intel and HP through the donation of an Itanium based machine is gratefully acknowledged.