# Exact Accumulation of Floating-Point Numbers

Michael Müller
Max-Planck-Institut für Informatik
D-6600 Saarbrücken

Christine Rüb
Max-Planck-Institut für Informatik
D-6600 Saarbrücken

Wolfgang Rülling
Fachhochschule Furtwangen
D-7743 Furtwangen

## Abstract

*We present a new idea for designing a chip which computes the exact sum of arbitrary many floating-point numbers, i.e. it can accumulate the floating-point numbers without cancellation. Such a chip is needed to provide a fast implementation of Kulisch-arithmetic. This is a new theory of floating-point arithmetic which makes it possible to compute least significant bit accurate solutions to even ill conditioned numerical problems. Our approach avoids the disadvantages of previously suggested designs which are too large, too slow or consume too much power. The crucial point is a technique for a fast carry resolution in a long accumulator. It can also be implemented in software.*

## 1 Introduction

Everybody who has implemented an algorithm using arithmetic on "real" numbers knows the problems arising from approximating reals by floating-point numbers. Even in very simple geometric algorithms, inaccurate calculations can lead to a crash of the program or to completely wrong results.

The major reasons for inexact calculations in computers are cancellation occurring in subtractions and truncation after multiplications. Also, during the huge computations performed on modern vector processors very small errors have enough time to accumulate until the result has nothing in common with the desired solution of the given problem. Hence, calculating exactly is much more important in those computers, but they still supply no remedy against this problem.

To avoid these problems, Kulisch and Miranker (cf. [14], [15]) derived a new theory of floating-point arithmetic. They propose to provide, besides the four standard basic operations $+$, $-$, $\cdot$, $/$, the inner product of two vectors (denoted as $*$) as a fifth basic operation. This operation should, as well as the standard basic operations, be available in the three rounding modes rounding to nearest ($\Box$), rounding towards $+\infty$ ($\triangle$) and rounding towards $-\infty$ ($\triangledown$). These three rounding modes, implemented in a way such that $x \circledast y = \bigcirc(x \star y)$ for $\bigcirc \in \{\Box, \triangledown, \triangle\}$ and $\star \in \{+, -, \cdot, /, *\}$, are necessary to derive reliable error estimations (cf. [13]). Here, $\circledast$ denotes the operation actually performed by the machine.

We will refer to the operation $\circledast$ as the **rounded exact inner product** or, for short, **exact inner product**.

This theory allowed the development of new algorithms for the numerical standard problems. These algorithms compute bounds for the (real) solution of the given problem. In general, the bounds are adjacent floating-point numbers, and in this case the result is called **least significant bit accurate**. Even for very ill conditioned problems where traditional numerical methods fail to give a reasonable answer, the new algorithms mostly compute least significant bit accurate results. Also the computed intervals are proven to contain the actual result, i.e. one can rely on them.

To implement these algorithms, one needs the exact inner product and there exist extensions of Pascal and Fortran (PASCAL-SC [4,9] and FORTRAN-SC [6]) and subroutine packages (ACRITH [10], ARITHMOS [20]) providing this operation in software. In [3] one can find a good overview over the state of the art in this field. But as far as we know there still exists no chip for this purpose and thus the acceptance of the new methods is quite small since there is no really fast implementation of the exact inner product. Especially for high performance vector computers a software realization seems not to be competitive.

There have already been several proposals how to design such a chip ([24], [16], [7], [12], [25]), but most of them seem to have no chance of being realized in the near future because they are either too large, consume too much power since in most cycles the whole circuit is active, or too slow since one has to wait a hundred or more cycles to obtain the result after the last input has been accepted. Thus, we present a new strategy to compute the exact inner product of two vectors which promises to avoid the essential disadvantages of the other concepts.

This paper is structured as follows. In section 2 we describe the algorithms which form the basis of our design. In section 2.1 we outline the basic ideas of our algorithms, and in section 2.2 we describe the algorithms in more detail. Section 2.2.1 is dedicated to accepting a new summand and section 2.2.2 treats the problem of rounding the sum to get a floating-point result. Section 3 briefly describes how we intend to realize the algorithms of section 2 in hardware and in section 4 we show how the algorithm can be implemented in software.

## 2 The Algorithm

Suppose that we want to compute the exact inner product of two vectors $v = (v_1, \ldots, v_n)$ and $w = (w_1, \ldots, w_n)$, and that the components of the vectors are represented with $\overline{m}$ bits of mantissa and an exponent range of $[\bar{e}_1 \mathinner{..} \bar{e}_2]$, $\bar{e}_1 \leq \bar{e}_2$. To compute the inner product, we first have to compute the $n$ exact products $v_i \cdot w_i =: z_i$, $1 \leq i \leq n$, where the $n$ products $z_i$ are each represented with $m = 2 \cdot \overline{m}$ bits of mantissa and an exponent range of $[e_1 \mathinner{..} e_2] = [2 \cdot \bar{e}_1 \mathinner{..} 2 \cdot \bar{e}_2]$. Then we have to add the $n$ products $z_i$, $1 \leq i \leq n$, and round the exact sum according to the desired rounding mode.

The exact product of two floating-point numbers can be computed by standard techniques and we thus concentrate on the exact, i.e. cancellation-free, summation of floating-point numbers. We use the following technique which is also used by [24], [16], [7], [12] and [25]:

> Each summand that arrives at the chip is expanded to a fixed-point number with $m + e_2 - e_1 + 1$ digits. The expanded summands are then added using pipelining. After the last summand has been accepted, remaining carries are removed if necessary, and the computed sum is rounded to the required floating-point format.

The problems that arise when this technique is implemented in hardware originate from the length of the expanded summands. E.g. if the components of the input vectors are single-precision floating-point numbers (we refer to the IEEE standard 754, cf. [11]), the length of their mantissas is 24 and the length of their exponents is 8. Thus, the exact products of the components have mantissas of length 48 and exponents of length 9, and the expanded summands have a length of $48 + 2^9 = 560$. If the components of the input vectors are double-precision floating-point numbers, then their mantissas have a length of 53 and their exponents have a length of 11. Thus, the expanded summands have a length of $106 + 2^{12} = 4202$.

In our design we want to avoid the three essential disadvantages of the previous layouts mentioned in the introduction, i.e. the design should be small, only a small part of the circuit should be active in each cycle to reduce power consumption and the result should be available only a short time after the last summand has been accepted. How this can be achieved is shown in the rest of this section.

### 2.1 The Basic Ideas

As mentioned above, we reduce the computation of the inner product to the summation of products computed elsewhere. This is done by adding the summands into a long accumulator.

Intuition tells us that it is not necessary to consider all accumulator digits when we add a number covering only a small part of it. This is clearly true if the number is positive and can be added without producing a carry at the left end of the range covered. If this locality would hold for all additions, we could maintain the contents of the accumulator in an ordinary storage, read the part affected by the addition, use a moderately sized adder to do the main work and write the modified section back to the storage. To make this idea work in all cases, we have to solve two major problems:

1. What do we do with the carries?
2. How do we treat negative summands?

Let us first ignore negative summands and concentrate on carry handling. A first approach could be to accumulate the carries as suggested by Kirchner and Kulisch (cf. [12]). But this means that the accumulated carries have to be resolved at the end of the addition which takes a lot of time. Thus, we looked for a way to get immediately rid of the carry resulting from an addition.

Let us inspect carry handling during a conventional addition w.l.o.g. in the binary system. Suppose we add two numbers $a$ and $b$ where $b$ has many leading zeros. Consider what happens to a carry when we reach the leading zeros of $b$, cf. Figure 1. As long as the corresponding digits in $a$ are 1, the bits in the sum become 0 and the carry propagates. When we reach a 0 in the binary representation of $a$ this digit of the sum becomes 1 and the carry disappears. The following bits of the sum are the same as the leading bits of $a$. Hence, in order to resolve a carry we only need to find the next zero in $a$ and toggle all bits between this position and the position from which the carry originated.

```
 XXX...XX0111...11XXXX...XXX
+                1
----------------------------
=XXX...XX1000...00XXXX...XXX
```

Figure 1: Carry resolution.

To make use of this observation, we use the following strategy. We divide the accumulator into several blocks. To add a new summand, we perform ordinary additions for the blocks intersected by the mantissa of the summand. The less significant blocks remain unchanged. If the most significant addition yields a carry, this carry has to be added to the more significant blocks as sketched above. To avoid stepping through all bits of the more significant blocks, we maintain for every block some information about its contents which describes whether the block consists only of ones or whether there is a zero in it. Thus we know whether the carry passes through a block or whether it is accepted in it. The first block that contains a zero has to be incremented by 1. The blocks consisting only of ones that we have skipped should be inverted, but this is too time and power consuming. Hence, we invert only the contents information and register in a second information bit that the contents of this section of the accumulator are not the contents of the storage but only zeros. This information always has to be considered when we want to read something from the storage and has to be updated when we write a block to the storage. The storage is organized such that a row of the storage corresponds to a block of the accu.

To avoid running sequentially through all information bits, we use the adder for carry propagation as follows. We define a binary number which represents the states of the blocks of the accu. The least significant bit of the number corresponds to the least significant block of the accu. A bit is one iff the corresponding block of the accu contains only ones. To this number we add the carry, i.e. we add a number having a 1 only at the position corresponding to the block first entered by the carry. Figure 2 illustrates this procedure. In the result of this addition the bits different from the bits in the input number are exactly the bits corresponding to the rows passed by the carry (marked with a bar in Fig. 2) and the row which absorbs the carry (marked with a cross). Hence, it is easy to update the information bits and to address the row which absorbs the carry.

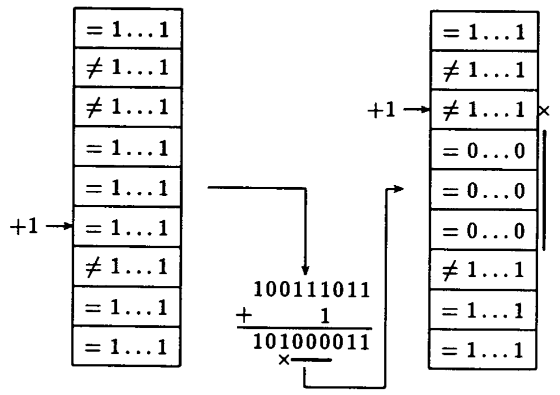

Figure 2: The use of the adder to propagate the carry over the blocks.

Now we have shown how carries can be treated if all summands are positive. Next we show how we will handle negative summands. When expanded to the fixed-point format, the negative numbers are represented in two's complement and thus require all the leading digits being filled up with ones. At first sight this seems to be incompatible with our effort of working only locally, but this is not true. Again we consider the way we perform arithmetic manually. Suppose we want to subtract $b$ from $a$. If we use binary representation, we compute $a + (-b)$ representing $-b$ in two's complement which we get by inverting all digits of $b$ and adding 1. If $a$ corresponds to a section of the accumulator and $a \geq b$, then this method of subtracting two numbers can also be applied locally to our accumulator since only local changes are necessary. If $a < b$, then the result of the subtraction is a negative number and we cannot replace some section of the accumulator by a negative number.

But consider the way a negative number is represented in two's complement. The sign bit has value $-2^n$ where $n$ is the number of digits used to represent the number, not counting the sign bit. Hence, if we manipulate a section of the accumulator in the usual way, a negative sign bit of the result can be interpreted as a negative carry, i.e. from the more significant bits we have to subtract this carry. This negative carry can be handled nearly the same way as a positive carry. The only difference is that it is accepted by a 1 and propagated by a 0. Hence, the meaning of the two information bits associated with a row has to be extended in the following way. One bit indicates whether all bits in the row are equal and if they are, the other bit indicates whether the row has to be interpreted as constant 1 or constant 0. The propagation of a negative carry over the blocks is similar to the propagation of a positive carry: a block containing only zeros is represented as a zero, other blocks are represented as one and we subtract the carry from the number constructed in this way. In the following section we give a more detailed description of the addition algorithm and the rounding process.

## 2.2 Some Details of the Algorithms

In describing the algorithms for adding and rounding, we will widely abstract from a concrete hardware realization, but some considerations are influenced by the floating-point system we use. To keep things simple, we describe the algorithm for the binary system but the principles can also be applied to any other number system, e.g. the decimal system. Since we want to construct a chip that fits together with other hardware, we have to consider existing standards. A commonly accepted standard for the representation of floating-point numbers is the IEEE floating-point standard 754 (cf. [11]). This standard uses sign magnitude representation for the mantissas. Hence, we choose the following **floating-point format** for our abstract description: The summands are represented as floating-point numbers consisting of $m$ mantissa digits for the absolute value of the mantissa, a sign bit and an integer exponent in the range $[e_1 .. e_2]$. We do not require the summands to be normalized.

Let the accumulator be realized by $L$ digits (including the sign bit) representing the accumulator contents in two's complement. $L$ must be at least $e_2 - e_1 + m + 1$. To be able to add $k$-times the largest representable summand, there must be $\lceil \log k \rceil$ additional overflow bits. Since it is not hard to make enough overflow bits available, we will always assume that $L$ is chosen large enough such that a new summand can be added without producing an overflow. As we already stated in section 2.1, the accumulator is subdivided into several blocks. Here we assume that the accumulator is divided into blocks of size $s$ and that we are able to add $s$-bit numbers. We also assume that $L = n \cdot s$ for some $n \in \mathbb{N}$.

**2.2.1 The Algorithm for Adding a New Summand:** The algorithm for adding a new summand works as follows. First we "convert" the summand into fixed-point representation. This can be done by adjusting the mantissa to the corresponding position at the accumulator. Since we can address each block of the accumulator individually, we can perform the positioning in two steps. One step is a coarse positioning done by addressing the blocks of the accu that are intersected by the mantissa digits (cf. Figure 3). The other step, the fine positioning, is done by a shifting circuit. The output of this circuit is stored in a register and we know (from the exponent of the summand) which part of the register corresponds to which block of the accu. For each of these blocks of the accu we perform a usual addition stepping through them from right to left as indicated in Figure 3. Each rectangle represents a block of $s$ bits and each addition is an $s$-bit addition. If in the last addition a carry occurs, we have to pass all blocks that cannot accept the carry and add it to the first accepting block.

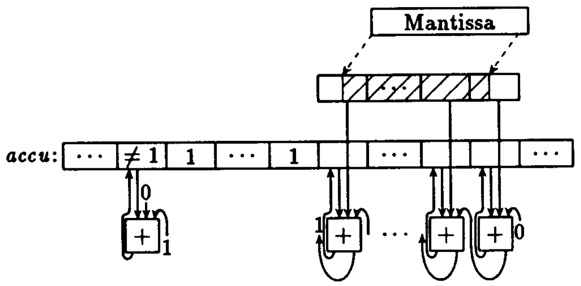

Figure 3: Illustration of the addition algorithm if we add a positive summand and when after the local additions a carry remains to be resolved.

The address of the accu block containing the least significant mantissa digit is given by the more significant bits of the exponent, provided that $s$ is a power of 2. The shift distance is $s-1$ at most. Then the shifted mantissa digits cover at most $r = \lceil (m+s-1)/s \rceil$ blocks of the accu.

If the summand is negative, then the mantissa has to be subtracted. This is done by adding the two's complement, i.e. by adding the inverted output of the shifter plus one. Here it is not necessary to represent the sign bit explicitly since subtracting a positive $r \cdot s$-bit number from a positive $r \cdot s$-bit number always yields a number that can be represented in two's complement using $r \cdot s$ bits plus a sign bit. Hence, no overflow or underflow occurs. Furthermore, the sign bit of the result is 1 iff there is no carry entering the column of the sign bit. This means that we have to resolve a negative carry iff the summand is negative and no carry resulted from the last one of the $r$ addition. The procedure for carry resolution is already described in section 2.1.

**2.2.2 Rounding and Normalization:** Given the contents of the accumulator, we have to perform the specified rounding at the end of the summation. Rounding towards $-\infty$ can clearly be done by truncation; this corresponds to omitting a nonnegative part of the sum. Rounding towards $+\infty$ can be performed by truncating and computing the next larger floating-point number if the truncated part was $\neq 0$. Rounding towards zero is now trivial. Rounding to nearest is a little bit harder. Again we truncate the part of the contents of the accu that won't fit into the mantissa. If what we truncated was less than $100\ldots00$, then we are done. If it was more than $100\ldots00$, we compute the next larger floating-point number. If it was equal to $100\ldots00$ and the mantissa is even, we are done, too, but if the mantissa is not even, we also have to compute its successor. Computing the successor of a floating-point number can usually be achieved by adding 1 to its mantissa. If the resulting number is not representable in the given floating-point format a, suitable exception has to be signaled.

In order to determine the significant part of the accu we need to know the position of the most significant bit (m.s.b.), i.e. the first bit different from the sign bit. To find out whether the part of the accu that will be truncated is greater, equal or less than $100\ldots00$, it is also helpful to know the position of the least significant bit (l.s.b.), i.e. the rightmost bit that is 1.

To determine the m.s.b. and the l.s.b. we use the same principle that we used to find the position where a carry is accepted. A negative carry is accepted at the rightmost 1 which is to the left of the position from which the carry originated. Hence, the position of the l.s.b. of a binary number is the position where a negative carry, which enters from the right, is accepted. It can be found by subtracting 1 from the number. If the number is not zero this subtraction causes exactly one digit, namely the l.s.b., to change from 1 to 0. Hence, we can determine the l.s.b. in two steps. To determine the block, we define a binary number which represents the states of the blocks of the accu as we already did for the carry resolution. A bit in this number is 0 iff the corresponding block contains only zeros. To this number we add $1\ldots1$ which is equivalent to subtracting 1. To determine the position within the block we apply the same procedure to the contents of the block. Determining the m.s.b. is similar; we have to find the left-most bit which is different from the sign bit.

Let $\Sigma$ denote the computed exact sum, i.e. the value of the contents of the accumulator. Having computed the m.s.b. and the l.s.b., we extract the significant part of the accu and compute the appropriate exponent to get $\bigtriangledown(\Sigma)$ with the mantissa represented in two's complement. If the sign is negative, we mainly have to invert the mantissa and to add 1 to obtain the sign magnitude representation of $\bigtriangledown(\Sigma)$.

Now we have treated rounding towards $-\infty$ and next we will explain what modifications are necessary to support the other rounding modes. For any floating-point number $x$ let $succ(x)$ denote the next larger floating-point number. Since $\bigtriangledown(\Sigma) \leq \Sigma \leq succ(\bigtriangledown(\Sigma))$, the rounded contents $\bigcirc(\Sigma)$ of the accu are contained in $\{\bigtriangledown(\Sigma), succ(\bigtriangledown(\Sigma))\}$ for any rounding mode $\bigcirc$. Hence, it is sufficient to compute $\bigtriangledown(\Sigma)$ and $succ(\bigtriangledown(\Sigma))$ and decide which of both values is the right one. Considering the remarks at the beginning of this section this decision can easily be made.

## 3 Realization in Hardware

In this section we turn to the hardware implementation of the algorithms presented in section 2. As one can see from the previous discussion, the circuit consists only of well known components: an adder for $s$-bit nubmers, a shifter, a storage and some decoders.

The sizes of the modules (i.e. the number of bits they must be designed for) are fixed by the choice of the floating-point system and the choice of the size $s$ of a block of the accumulator. The choice of $s$ influences the performance and space requirement of the circuit significantly. If we choose $s$ small, the adder and the shifter are small, but on the other hand we have to use the adder and the shifter many times to process a summand. If we choose $s$ large, things are the other way round: the adder and the shifter are large, but we need only few additions and shifts to process a summand. It is also important to choose $s$ as a power of 2 since then the computation of the shift distance and the blocks affected by the addition of a new summand are trivial; they merely require selecting the more, less resp. significant bits of the exponent.

It seems that $s$ should be chosen as large as possible to achieve a high performance. This is because the reduction of the number of additions seems to gain much more than the loss of speed resulting from the longer delay time of the larger modules makes up. To get a better feeling for a good choice of $s$ we consider the following example.

**Example:** The input numbers shall be (cf. the beginning of section 2) exact products of double precision floating-point numbers corresponding to the IEEE standard 754, cf. [11]. This requires an accumulator of about 4200 bits plus a sufficient number (e.g. 60) of additional bits to avoid overflow (cf. section 2.1). The mantissas of the products have a length of $m = 106$.

To calculate the number of additions needed to process a summand (depending on $s$) we recall from section 2.2.1 that the number of blocks intersected by a mantissa of length $m$ is $r = \lceil (m + s - 1)/s \rceil$ at most. We need one addition for each of those blocks plus $\lceil (n-r)/s \rceil$ to find which block accepts a carry plus one for removing the carry, i.e. we need $r + \lceil (n - r)/s \rceil + 1$ additions. The expression $\lceil (n-r)/s \rceil$ results from the fact that a carry can run over $n-r$ blocks at most, where $n$ is the number of blocks of the accumulator. Table 1 shows the number of additions required and the number $n = \lceil 4260/s \rceil$ of blocks of the accumulator if we choose $s = 2^k$ for some $k$.

| $k$ | $s = 2^k$ | $r$ | # additions | $n$ |
|---|---|---|---|---|
| 4 | 16 | 8 | 26 | 267 |
| 5 | 32 | 5 | 11 | 134 |
| 6 | 64 | 3 | 5 | 67 |
| 7 | 128 | 2 | 4 | 34 |

Table 1: The effect of choosing $s = 2^k$ for some $k$.

For $s = 128$ we need the least possible number of additions and hence this seems to be the best value to achieve a high speed. Choosing $s$ larger than 128 is not useful since we need at least 4 additions in the worst case: it can always happen that the mantissa intersects two blocks of the accu and that we need two additions to remove a carry. But $s = 64$ might also be a good choice if there is not enough space to realize a 128-bit adder and shifter. ∎

## 4 Implementation in Software

The algorithm presented above can also be used to obtain a fast software implementation of the long accumulator. Almost all operations we use are available in an ordinary computer. The only problem is to find in a binary number $x$ the w.l.o.g. left-most bit which equals 1. This corresponds to compute $\lfloor \lg x \rfloor$, and not all processors provide this operation. But as shown in [8] $\lfloor \lg x \rfloor$ can be computed in constant time using ordinary arithmetic and bitwise Boolean operations as follows. Let $b$ be the wordlength and consider a partitioning of the word $x$ into blocks of $s = \lceil \sqrt{b+1} \rceil$ bits. Define $C$ to have 1's precisely in the left-most bit position of each block. Evaluating the expression $(C - [(C - x \wedge \overline{C}) \wedge C]) \vee (x \wedge C)$, where $\overline{C}$ is the bitwise complement of $C$, yields a number in which the left-most bit of a block is 1 iff the corresponding block of $x$ contains a 1, and all other bits in the number are 0. Multiplying this number with a suitable constant compresses all the leading bits of the blocks into the rightmost $\lceil b/s \rceil$ bits. Now, by table look-up we can find the number of the block of $x$, which contains the left-most 1 and by a second look-up we can find the position within this block. Hence $\lfloor \lg x \rfloor$ can be computed using about 4 bitwise Boolean operations, 2 subtractions, 1 multiplication and 2 table look-ups.

## References

[1] B. Becker, "Efficient Testing of Optimal Time Adders", TR 04/1985, SFB 124, Universität des Saarlandes

[2] B. Becker, R. Kolla, "On the Construction of Optimal Time Adders", TR 07/1987, SFB 124, Universität des Saarlandes

[3] G. Bohlender, "What Do We Need Beyond IEEE Arithmetic?", in [22]

[4] G. Bohlender, L. Rall, Ch. Ullrich, J. Wolff von Guldenberg, "PASCAL-SC: A Computer Language for Scientific Computation", Academic Press, New York, 1987

[5] R. P. Brent, H. T. Kung, "A Regular Layout for Parallel Adders", *IEEE Trans. on Comp.*, C-31, pp. 260–264, March 1982

[6] J. H. Behler, S. M. Rump, U. W. Kulisch, M. Metzger, Ch. Ullrich, W. Walter, "FORTRAN-SC: A Study of a FORTRAN Extension for Engineering/Scientific Computation with Access to ACRITH", *Computing 39*, pp. 93–110, 1987

[7] P. R. Capello, W. L. Miranker, "Systolic Super Summation", *IEEE Transactions on Computers*, Vol. 37, No. 6, pp. 675–676, June 1988

[8] M.L. Fredman, D.E. Willard, "BLASTING Through The Information Theoretic Barrier With FUSION TREES", *Proc. of the 22nd Annual ACM Symp. on Theory of Computing*, pp. 1–7, 1990

[9] R. Hammer, M. Neaga, D. Ratz, "PASCAL-SC: New Concepts for Scientific Computation and Numerical Data Processing", Institute for Applied Mathematics, University of Karlsruhe, Federal Republic of Germany

[10] IBM, "High-Accuracy Arithmetic Subroutine Library (ACRITH)", General Information Manual, 3rd ed., GC 33-6163-02, IBM Corporation, 1986

[11] American National Standards Institute / Institute of Electrical & Electronic Engineers, "A Standard for Binary Floating-Point Arithmetic", ANSI/IEEE Std. 754-1985, New York, August 1985

[12] R. Kirchner, U. W. Kulisch, "Accurate Arithmetic for Vector Processors", *Journal of Parallel and Distributed Computing*, Vol. 5, pp. 250–270, 1988

[13] U. W. Kulisch, "Grundlagen des Numerischen Rechnens — Mathematische Begründung der Rechnerarithmetik", Mannheim, Bibliographisches Institut, 1976

[14] U. W. Kulisch, W. L. Miranker, "Computer Arithmetic in Theory and Practice", New York, Academic Press, 1981

[15] U. W. Kulisch, W. L. Miranker (eds.), "A New Approach to Scientific Computation", New York, Academic Press, 1983

[16] P. Lichter, "Realisierung eines VLSI-Chips für das Gleitkomma-Skalarprodukt der Kulisch-Arithmetik", Diplomarbeit, Fachbereich 10, Angewandte Mathematik und Informatik, Universität des Saarlandes, March 1988

[17] M. Müller, Ch. Rüb, W. Rülling, "Exact Addition of Floating-Point Numbers", TR 05/1990, SFB 124, Universität des Saarlandes

[18] Th. Ottmann, G. Thiemt, Ch. Ullrich, "Numerical Stability of Geometric Algorithms (Extended Abstract)", *Proceedings of the 3rd Annual ACM Symp. on Computational Geometry*, pp. 119–125, 1987

[19] S. M. Rump, H. Böhm, "Least Significant Bit Evaluation of Arithmetic Expressions in Single-Precision", *Computing 30*, pp. 189–199, 1983

[20] SIEMENS, "ARITHMOS, (BS2000) Benutzerhandbuch", SIEMENS AG, U2900-J-Z87-1, 1986

[21] J. Sklansky, "Conditional Sum Addition Logic", *IRE-EC 9*, pp. 226–231, June 1960

[22] Ch. Ullrich (ed.), "Computer Arithmetic and Self-Validating Numerical Methods", Academic Press, June 1990

[23] J. Vuillemin, L. Guibas, "On Fast Binary Addition in MOS Technologies", *ICCC 82*, pp. 147–150

[24] Th. Winter, "Ein VLSI-Chip für Gleitkomma-Skalarprodukt mit maximaler Genauigkeit", Diplomarbeit, Fachbereich 10, Angewandte Mathematik und Informatik, Universität des Saarlandes, April 1985

[25] T. Yilmaz, J.F.M. Theeuwen, R.J.W.T. Tangelder, J.A.G. Jess, "The Design of a Chip for Scientific Computation", Eindhoven University of Technology, 1989 and Euro Asic, Grenoble, Jan. 25–27, 1989