Advanced Arithmetic for the Digital Computer Design of Arithmetic Units

Ulrich W. Kulisch [1]

Universität Karlsruhe, D-76128 Karlsruhe, Germany

Abstract

Advances in computer technology are now so profound that the arithmetic capability and repertoire of computers can and should be expanded. Of the elementary floating-point operations $+, -, \times, /$ nowadays is required that for any choice of operands, the computed result coincides with the rounded exact result of the operation. Advanced computer arithmetic extends this accuracy requirement to all operations in the usual product spaces of computation: the real and complex vector spaces as well as their interval correspondents. This enhances the mathematical power of the digital computer considerably. A new computer operation, the scalar product, is fundamental to the development of advanced computer arithmetic.

This paper studies the design of arithmetic units for advanced computer arithmetic. Scalar product units are developed for different kinds of computers like Personal Computers, Workstations, Mainframes, Super Computers or Digital Signal Processors. The new expanded computational capability is gained at modest costs. It is obtained by putting a methodology into modern computer hardware which was already available on old calculators before the electronic computer entered the scene. In general the new arithmetic units increase both, the speed of computation as well as the accuracy of the computed result. The circuits developed in this paper show that there is no way to compute an approximation of a scalar product faster than the correct result.

A collection of constructs in terms of which a source language may accommodate advanced computer arithmetic is described in the paper. The development of programming languages in the context of advanced computer arithmetic is reviewed. The simulation of the accurate scalar product on existing, conventional processors is discussed. Finally the theoretical foundation of advanced computer arithmetic is reviewed and a comparison with other approaches to achieve higher accuracy in computation is given. Shortcomings of existing processors and standards are discussed.

Keywords: floating-point arithmetic, fixed-point arithmetic, semimorphism, accurate scalar product, computer arithmetic, computation with guarantees, design of arithmetic units, computation with automatic result verification.

Contents

[1] This article was prepared during a stay of the author at the Electrotechnical Laboratory, Agency of Industrial Science and Technology, MITI, at Tsukuba, Ibaraki 305-8568, Japan in March-May, 1998

1 Introduction and Historic Remarks

1.1 Introduction

Advances in computer technology are now so profound that the arithmetic capability and repertoire of computers can and should be expanded. At a time where more than 100 million transistors can be placed on a single chip, computing speed is measured in GIGA- and TERAFLOPS and memory space in GIGA-words there is no need anymore to perform all computer calculations by the four elementary floating-point operations with all the shortcomings of this methodology (see, for instance, the three examples listed in section 1.2).

Of the elementary floating-point operations $+, -, \times, /$ nowadays is required that for any choice of operands the computed result coincides with the rounded exact result of the operation. See, for instance, the IEEE-Arithmetic Standards 754 and 854, [IEEE85,IEEE87]. Advanced computer arithmetic extends this accuracy requirement to all operations in the usual product spaces of computation: the complex numbers, real and complex vectors, real and complex matrices, real and complex intervals as well as real and complex interval vectors and interval matrices. This enhances the mathematical power of the digital computer considerably. A major number of computer operations then can be performed with but a single rounding error.

If, for instance, the scalar product of two vectors with 1000 components is to be computed about 2000 roundings are executed in conventional floating-point arithmetic. This number now is reduced to a single rounding. The computed result is within a single rounding error of the correct result.

The new operations are distinctly different from the customary ones which are based on elementary floating-point arithmetic. A careful analysis and a general theory of computer arithmetic show that the new operations can be built up on the computer by a modular technique as soon as on a low level, possibly in hardware, a new fundamental operation, the scalar product, is provided with full accuracy.

The computer realization of the scalar product of two floating-point vectors with full accuracy can be achieved in several ways. A most natural way is to add the products of corresponding vector components into a long fixed-point register (accumulator) which covers twice the exponent range of the floating-point format in which the vector components are given. This method has the advantage of being rather simple, straightforward and fast. Since fixed-point accumulation of numbers is error free it always provides the desired accurate answer. The methodology was already used on old mechanical calculators long before the electronic computer entered the scene.

In a floating-point system the number of mantissa digits as well as the exponent range are finite. Therefore, the fixed-point register is finite as well, and it is relatively small, consisting of about 1 to 4 thousand bits depending on the data format in use. So we have the seemingly paradox and striking situation that scalar products of floating-point vectors with even millions of components can be computed to a fully accurate result in infinite precision arithmetic using a relatively small finite local register memory on the arithmetic unit.

In Numerical Analysis the scalar or dot product is ubiquitous. Not just that it is a fundamental operation in all the product spaces mentioned above.

The process of residual or defect correction or iterative refinement is composed of scalar products. There are well known limitations to these processes in floating-point arithmetic. The question how many digits of a defect can be guaranteed with single, double or extended precision arithmetic has been carefully investigated. With the optimal scalar product the defect can always be computed to full accuracy. It is the accurate scalar product which makes residual correction effective.

With the accurate scalar product quadruple or multiple precision arithmetic can easily be provided on the computer. This enables the user to use higher precision operations in numerically critical parts of his computation. It helps to increase software reliability. A multiple precision number is represented as an array of floating-point numbers. The value of this number is the sum of its components. It can be represented in the long accumulator. Addition and subtraction of multiple precision variables or numbers can easily be performed in the long accumulator. Multiplication of two such numbers is simply a sum of products. It can be computed by means of the accurate scalar product. For instance in case of a fourfold precision the product of two such numbers

$a = (a_1 + a_2 + a_3 + a_4)$ and $b = (b_1 + b_2 + b_3 + b_4)$ is obtained by

$$
\begin{aligned}
a \times b &= (a_1 + a_2 + a_3 + a_4) \times (b_1 + b_2 + b_3 + b_4) \\
&= a_1 b_1 + a_1 b_2 + a_1 b_3 + a_1 b_4 + a_2 b_1 + \cdots + a_4 b_3 + a_4 b_4 \\
&= \textstyle\sum_{i=1}^{4} \sum_{j=1}^{4} a_i b_j.
\end{aligned}
$$

The result is independent of the sequence in which the summands are added.

With increasing speed of computers, problems to be dealt with become larger. Instead of two dimensional problems users would like to solve three dimensional problems. Gauss elimination for a linear system of equations requires the magnitude of $\mathcal{O}(n^3)$ operations. Large, sparse or structured linear or non linear systems, therefore, can only be solved iteratively. The basic operation of iterative methods (Jacobi method, Gauss-Seidel method, overrelaxation method, conjugate gradient method, Krylow space methods, multigrid methods and others like the QR method for the computation of eigenvalues) is the matrix-vector multiplication which consists of a number of scalar products. It is well known that finite precision arithmetic often deteriorates the convergence of these methods. An iterative method which converges to the solution in infinite precision arithmetic often converges much slower or even diverges in finite precision arithmetic. The optimal scalar product is faster than a computation in conventional floating-point arithmetic. In addition to that it can speed up the rate of convergence of iterative methods essentially in many cases.

Many other applications require that rigorous mathematics can be done with the computer using floating-point arithmetic. As an example, this is essential in simulation runs (fusion reactor, eigenfrequencies of large generators) or mathematical modelling where the user has to distinguish between computational artifacts and genuine reactions of the model. The model can only be developed systematically if errors entering into the computation can be essentially excluded.

Nowadays computer applications are of immense variety. Any discussion of the question where a dot product computed in quadruple or extended precision arithmetic can be used as substitutes of the accurate scalar product is useless and superfluous. Since the former can fail to produce a correct answer an error analysis is due for all applications. This can be left to the computer. An operation as the scalar product which always can be executed correctly with moderate technical effort should indeed always be executed correctly. An error analysis thus becomes superfluous once and for all. Furthermore, always the same result is obtained on different computer platforms. A fully accurate scalar product eliminates many rounding errors in numerical computations. It stabilizes these computations and speeds them up as well. It is the necessary completion of floating-point arithmetic.

This paper studies the design of arithmetic units for advanced computer arithmetic. Scalar product units are developed for different kinds of computers like Personal Computers, Workstations, Mainframes, Super Computers or even Digital Signal Processors. The differences in the circuits for these diverse processors are more or less caused by the way respectively the speed with which the processor delivers the data to the arithmetic or scalar product unit. The data are the vector components. In all cases the new expanded computational capability is gained at modest costs. The cost increase is comparable to the one from a simple to a fast multiplier, for instance, by a Wallace tree. This step has already been accepted years ago. It is a main result of our study that for all processors mentioned above circuits can be given for the computation of the accurate scalar product where virtually no computing time is needed for the execution of the arithmetic. In a pipeline, the arithmetic can be executed within the time the processor needs to read the data into the arithmetic unit. This means, among others, that no other method to compute a scalar product can be faster. This includes, in particular, a conventional approximate computation of the scalar product in floating-point arithmetic which can lead to an incorrect result.

In the pipeline a multiplication and the accumulation of a product to the intermediate sum are performed simultaneously. This doubles the speed of the optimal scalar product in comparison with a conventional computation in floating-point arithmetic where these operations are performed sequentially. Furthermore, fixed-point accumulation of the products is simpler than accumulation in floating-point. Many intermediate steps that are executed in a floating-point accumulation such as normalization and rounding of the products and the intermediate sum, composition into a floating-point number and decomposition into mantissa and exponent for the next operation do not occur in the fixed-point accumulation of the accurate scalar product.

In recent years there has been a significant shift of numerical computation from general purpose computers towards vector and parallel computers – so-called super computers. Along with the four elementary floating-point operations these computers usually offer compound operations as additional computer operations. A particular such compound operation *multiply and accumulate* is available for the computation of the scalar product of two vectors. These compound operations are heavily pipelined and make the computation really fast. They are automatically inserted in a user's program by a vectorizing compiler. If these operations are not carefully implemented the user loses complete control of his computation.

In 1987 GAMM [2] and IMACS [3] published a *Resolution on Computer Arithmetic* which criticized the mathematically not adequate execution of matrix and vector operations on all existing vector processors. An amendment was demanded. The user should not be obliged to perform an error analysis every time an elementary compound operation, predefined by the manufacturer, is employed. In 1993 the two organizations approved and published a *Proposal for Accurate Floating-Point Vector Arithmetic* [IMACS93]. It requires a mathematically correct implementation of matrix and vector operations, in particular, of the accurate scalar product on **all** computers. In 1995 the IFIP-Working Group 2.5 on Numerical Software endorsed this proposal. Meanwhile it became an EU Guideline.

We finish this introduction with a hint to the reader. This paper does not consist of independent chapters and sections. The later sections are built upon the former ones. On the other hand material that is presented later can be helpful and contribute to a full understanding of circuits that are discussed earlier.

1.2 Historic Remarks

Floating-point arithmetic is used since the early forties and fifties (Zuse Z3, 1941). Technology in those days was poor (electromechanical relays, electron tubes). It was complex and expensive. The word size of the Z3 consisted of 24 bits. The storage provided 64 words. The four elementary floating-point operations were all that could be provided. For more complicated calculations an error analysis was left to and put on the shoulder of the user.

Before that time, highly sophisticated mechanical computing devices were used. Several very interesting techniques provided the four elementary operations addition, subtraction, multiplication and division. Many of these calculators were able to perform an additional *fifth operation* which was called *Auflaufenlassen* or the *running total*. The input register of such a machine had perhaps 10 or 12 decimal digits. Compared with that the result register was much wider and had perhaps 30 digits. It was a fixed-point register which could be shifted back and forth relatively to the input register. This allowed a continuous accumulation of numbers and of products of numbers into different positions of the result register. Fixed-point accumulation is error free. *This* fifth arithmetic operation *was the fastest way to use the computer. It was applied as often as possible. No intermediate results needed to be written*

[2] GAMM = Gesellschaft für Angewandte Mathematik und Mechanik
[3] IMACS = International Association for Mathematics and Computers in Simulation

down and typed in again for the next operation. No intermediate roundings or normalizations had to be performed. No error analysis was necessary. As long as no under- or overflow occurred which was obvious and visible, the result was always correct. It was independent of the order in which the summands were added. If desired, only one final rounding was executed at the very end of the accumulation.

This extremely useful and fast fifth arithmetic operation was not built into the early floating-point computers. It was too expensive for the technologies of these days. Later its superior properties had been forgotten.

The early electronic computers in the late forties and early fifties then represented their data as fixed-point numbers. Fixed-point addition and subtraction are error free. Fixed-point arithmetic with a rather limited word size, however, imposed a scaling requirement. Problems had to be preprocessed by the user so that they could be accommodated by this fixed-point number representation. With increasing speed of computers, problems that could be solved, became larger and larger. The necessary preprocessing soon became an enormous burden.

Thus floating-point arithmetic finally was generally accepted. It largely eliminated this burden. A scaling factor is appended to each number in floating-point representation. The arithmetic itself takes care of the scaling. An exponent addition resp. subtraction is executed during multiplication resp. division. It may result in a big change in the value of the exponent. But multiplication and division are relatively stable operations in floating-point arithmetic. Addition and subtraction, in contrary, are troublesome in floating-point.

The quality of floating-point arithmetic has been improved over the years. The data format was extended from less to 64 and even more bits and the IEEE-arithmetic standard has finally taken the bugs out of particular realizations. Floating-point arithmetic has been used very successfully in the past. Very sophisticated and intelligent algorithms and libraries have been developed for particular problems. However, in a general application the result of a floating-point computation is often hard to judge. It can be satisfactory, inaccurate or even completely wrong. The computation itself as well as the computed data do not indicate which one of the three cases has occurred. We illustrate the typical shortcomings by three very simple examples. For these and other examples see [Rum83]:

1. Compute the following, theoretically equivalent expressions:

$$
\begin{array}{l}
10^{20} + 17 - 10 + 130 - 10^{20} \\
10^{20} - 10 + 130 - 10^{20} + 17 \\
10^{20} + 17 - 10^{20} - 10 + 130 \\
10^{20} - 10 - 10^{20} + 130 + 17 \\
10^{20} - 10^{20} + 17 - 10 + 130 \\
10^{20} + 17 + 130 - 10^{20} - 10
\end{array}
$$

A conventional computer using the data format double-precision of the IEEE floating-point arithmetic standard returns the values 0, 17, 120, 147, 137, −10. These errors come about because the floating-point arithmetic is unable to cope with the digit range required with this calculation. Notice that the data cover less than 4% of the digit range of the data format double precision!

2. Compute the solution of a system of two linear equations $Ax = b$, with

$$A = \begin{pmatrix} 64919121 & -159018721 \\ 41869520.5 & -102558961 \end{pmatrix}, \quad b = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

The solution can be expressed by the formulas:

$$x_1 = a_{22}/(a_{11}a_{22} - a_{12}a_{21}) \text{ and } x_2 = -a_{21}/(a_{11}a_{22} - a_{12}a_{21}) .$$

A workstation using IEEE double precision floating-point arithmetic returns the *approximate solution*:

$$\tilde{x}_1 = 102558961 \text{ and } \tilde{x}_2 = 41869520.5 ,$$

while the correct solution is

$$x_1 = 205117922 \text{ and } x_2 = 83739041 .$$

After only 4 floating-point operations all digits of the computed solution are wrong. A closer look into the problem reveals that the accident happens during the computation of the denominator. This is just the kind of expression which always can be computed error free by the missing fifth operation.

3. Compute the scalar product of the two vectors a and b with five components each:

$$
\begin{array}{ll}
a_1 = 2.718281828 * 10^{10} & b_1 = 1486.2497 * 10^9 \\
a_2 = -3.141592654 * 10^{10} & b_2 = 878366.9879 * 10^9 \\
a_3 = 1.414213562 * 10^{10} & b_3 = -22.37492 * 10^9 \\
a_4 = 0.5772156649 * 10^{10} & b_4 = 4773714.647 * 10^9 \\
a_5 = 0.3010299957 * 10^{10} & b_5 = 0.000185049 * 10^9
\end{array}
$$

The correct value of the scalar product is $-1.00657107 * 10^8$. IEEE-double precision arithmetic delivers $+4.328386285 * 10^9$ so even the sign is incorrect. Note that no vector element has more than 10 decimal digits.

Problems that can be dealt with computers, become larger and larger. Today fast computers are able to execute several billion floating-point operations in each second. This number exceeds the imagination of any user. Traditional error analysis of numerical algorithms is based on estimates of the error of each individual arithmetic operation and on the propagation of these errors through a complicated algorithm. It is simply no longer possible to expect that the error of such computations can be controlled by the user. There remains no alternative to further develop the computer and to furnish it with the capability of control and validation of the computational process.

Computer technology is extremely powerful today. It allows solutions which even an experienced computer user is totally unaware of. Floating-point arithmetic which fails already in simple calculations, as illustrated above, is no longer adequate to be used exclusively in computers of such gigantic speed for huge problems. The reintroduction of the fifth arithmetic operation, the accurate scalar product, into computers is a step which is long overdue. A central and fundamental operation of Numerical Analysis which can be executed correctly with only modest technical effort should indeed always be executed correctly and no longer only *approximately*. With the accurate scalar product all the nice properties which have been listed in connection with the old mechanical calculators return to the modern digital computer. *The accurate scalar product is the fastest way to use the computer. It should be applied as often as possible. No intermediate results need to be stored and read in again for the next operation. No intermediate roundings and normalizations have to be performed. No intermediate over- or underflow can occur. No error analysis is necessary. The result is always correct. It is independent of the order in which the summands are added. If desired, only one final rounding is executed at the very end of the accumulation.*

The accurate scalar product reintegrates the advantages of fixed-point arithmetic — error free accumulation of numbers and of single products of numbers even for very long sums — into digital computing and floating-point arithmetic. It is obtained by putting a methodology into modern computer hardware which was already available on old calculators before the electronic computer entered the scene.

2 Implementation Principles

A normalized floating-point number x (in sign-magnitude representation) is a real number of the form $x = *m\, b^e$. Here $* \in \{+, -\}$ is the sign of the number, b is the base of the number system in use and e is the exponent. b is an integer greater than unity. The exponent is an integer between two fixed integer bounds $e1$ and $e2$, and in general $e1 \leq 0 \leq e2$. The mantissa m is of the form

$$m = \sum_{i=1}^{l} d[i]\; b^{-i}$$

The $d[i]$ are the digits of the mantissa. They have the property $d[i] \in \{0, 1, \ldots, b-1\}$ for all $i = 1(1)l$ and $d[1] \neq 0$. Without this condition floating-point numbers are said to be not normalized. The set of normalized floating-point numbers does not contain zero. For a unique representation of zero we assume the mantissa and the exponent to be zero. Thus a floating-point system depends on the four constants $b, l, e1$ and $e2$. We denote it by $R = R(b, l, e1, e2)$. Occasionally we shall use the abbreviations $sign(x)$, $mant(x)$ and $exp(x)$ to denote the sign, mantissa and exponent of x respectively.

Of the elementary floating-point operations $+, -, \times, /$ nowadays is required that for any choice of operands the computed result coincides with the rounded exact result of the operation. See, for instance, the IEEE Arithmetic Standards 754 and 854, [IEEE85,IEEE87]. Advanced computer arithmetic extends this accuracy requirement to all operations in the usual product spaces of computation: the complex numbers, the real and complex vectors, real and complex matrices, real and complex intervals as well as the real and complex interval vectors and interval matrices.

A careful analysis and a general theory of computer arithmetic [Kul76,Kul81] show that all arithmetic operations in the computer representable subsets of these spaces can be realized on the computer by a modular technique as soon as on a low level fifteen fundamental operations are made available possibly by fast hardware routines. These fifteen operations are

$$
\begin{array}{ccccc}
\boxplus & \boxminus & \boxtimes & \boxslash & \square \\
\bigtriangledown\hspace{-0.6em}{\scriptstyle +} & \bigtriangledown\hspace{-0.6em}{\scriptstyle -} & \bigtriangledown\hspace{-0.6em}{\scriptstyle \times} & \bigtriangledown\hspace{-0.6em}{\scriptstyle /} & \bigtriangledown \\
\bigtriangleup\hspace{-0.6em}{\scriptstyle +} & \bigtriangleup\hspace{-0.6em}{\scriptstyle -} & \bigtriangleup\hspace{-0.6em}{\scriptstyle \times} & \bigtriangleup\hspace{-0.6em}{\scriptstyle /} & \bigtriangleup
\end{array}
$$

Here $\boxast, * \in \{+, -, \times, /\}$ denotes (semimorphic[4]) operations using some particular monotone and antisymmetric rounding $\square\colon I\!\!R \to R$ such as rounding to the nearest floating-point number or rounding towards zero. Likewise $\bigtriangledown\hspace{-0.6em}{\scriptstyle *}$ respectively $\bigtriangleup\hspace{-0.6em}{\scriptstyle *}$, $* \in \{+, -, \times, /\}$ denote the operations using the optimal (monotone[4]) rounding downwards $\bigtriangledown\colon I\!\!R \to R$ respectively the optimal (monotone[4]) rounding upwards $\bigtriangleup\colon I\!\!R \to R$. $\square$, $\bigtriangledown$ and $\bigtriangleup$ denote scalar products with high accuracy. That is, if $a = (a_i)$ and $b = (b_i)$ are vectors with floating-point components, $a_i, b_i \in R$, then $a \bigcirc b := \bigcirc(a_1 \times b_1 + a_2 \times b_2 + \ldots + a_n \times b_n)$. The multiplication and addition signs on the right hand side of the assignment denote exact multiplication and summation in the sense of real numbers.

These 15 operations are sufficient for the computer implementation of all arithmetic operations that are to be defined for all numerical data types listed above in the second paragraph of this section. Of the 15 fundamental operations, traditional numerical methods use only the four operations $\boxplus$, $\boxminus$, $\boxtimes$ and $\boxslash$. Interval arithmetic employs the eight operations $\bigtriangledown\hspace{-0.6em}{\scriptstyle +}$, $\bigtriangledown\hspace{-0.6em}{\scriptstyle -}$, $\bigtriangledown\hspace{-0.6em}{\scriptstyle \times}$, $\bigtriangledown\hspace{-0.6em}{\scriptstyle /}$ and $\bigtriangleup\hspace{-0.6em}{\scriptstyle +}$, $\bigtriangleup\hspace{-0.6em}{\scriptstyle -}$, $\bigtriangleup\hspace{-0.6em}{\scriptstyle \times}$, $\bigtriangleup\hspace{-0.6em}{\scriptstyle /}$. These eight operations are computer equivalents of the operations for real floating-point intervals, i. e. of interval arithmetic. Processors which provide the IEEE arithmetic standard, for instance offer 12 of these 15 operations: $\boxast$, $\bigtriangledown\hspace{-0.6em}{\scriptstyle *}$, $\bigtriangleup\hspace{-0.6em}{\scriptstyle *}$, $* \in \{+, -, \times, /\}$. The latter 8 operations $\bigtriangledown\hspace{-0.6em}{\scriptstyle *}$, $\bigtriangleup\hspace{-0.6em}{\scriptstyle *}$, $* \in \{+, -, \times, /\}$ are not yet provided by the usual high level programming languages. They are available and can be used in PASCAL-XSC, [Kul87,Kul87a,Kla91,Kla92,Kla93a], a PASCAL extension for high accurate scientific computing which was developed at the authors institute. Roughly speaking, interval arithmetic brings guarantees into computation while the three scalar or dot products deliver high accuracy. These two features should not be confused.

The implementation of the 12 operations $\boxast$, $\bigtriangledown\hspace{-0.6em}{\scriptstyle *}$, $\bigtriangleup\hspace{-0.6em}{\scriptstyle *}$, $* \in \{+, -, \times, /\}$ on computers is routine and standard nowadays. Fast techniques are largely discussed in the literature. So we now turn to the implementation of the three optimal scalar products $\square$, $\bigtriangledown$ and $\bigtriangleup$ on computers. We shall discuss circuits for the hardware realization of these operations for different kinds of processors like Personal Computers, Workstations, Mainframes, Super Computers and Digital Signal Processors, for instance. The differences in the circuits for these diverse processors are more or less caused by the way respectively the speed

[4] For a precise mathematical definition see section 7.

with which the processor delivers the data to the arithmetic or scalar product unit. The data are the vector components a_i and b_i, $i = 1, 2...., n$.

After a brief discussion of the basic idea for the implementation of the accurate scalar product on computers we shall sketch two principle solutions for the problem. Solution A is using a long adder and a long shift. Solution B can be characterized by a short adder and some local memory on the arithmetic unit. At first sight both of these principle solutions seem to lead to relatively slow hardware circuits. However later, more refined studies will show that very fast circuits can be given for both methods and the diverse processors mentioned above. A first step in this direction is the provision of a very fast carry resolution scheme in section 2.3.

Actually it is a main result of this study that for all processors under consideration circuits for the computation of the optimal scalar product are available where virtually no computing time for the execution of the arithmetic is needed. In a pipeline, the arithmetic can be done within the time the processor needs to read the data into the arithmetic unit. This means, among others, that no other method to compute the scalar product can be faster. This includes, in particular, a conventional computation of scalar products in floating-point arithmetic which may lead to an incorrect answer. Once more we emphasize the fact that the methods to be discussed here compute the scalar product of two floating-point vectors of arbitrary finite length without loss of information to full accuracy or with only one final rounding at the very end of the computation.

Now we turn to our task. Let $a = (a_i)$ and $b = (b_i)$, $i = 1(1)n$, be two vectors with n components which are floating-point numbers, i. e.

$$a_i, b_i \in R(b, l, e1, e2), \quad \text{for } i = 1(1)n$$

We are going to compute the two results (scalar products):

$$s \;:=\; \sum_{i=1}^{n} a_i \times b_i \;=\; a_1 \times b_1 + a_2 \times b_2 + \ldots + a_n \times b_n$$

and

$$c \;:=\; \bigcirc \sum_{i=1}^{n} a_i \times b_i \;=\; \bigcirc(a_1 \times b_1 + a_2 \times b_2 + \ldots + a_n \times b_n) \;=\; \bigcirc s$$

where all additions and multiplications are the operations for real numbers and $\bigcirc$ is a rounding symbol representing, for instance, rounding to nearest, rounding towards zero, rounding upwards or downwards.

Since a_i and b_i are floating-point numbers with a mantissa of l digits, the products $a_i \times b_i$ in the sums for s and c are floating-point numbers with

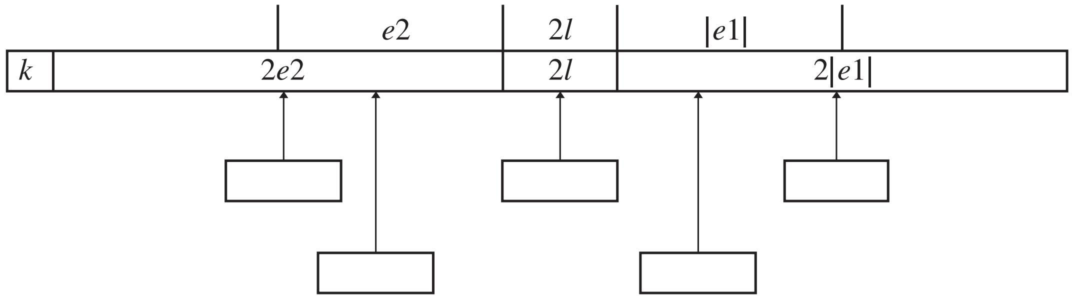


Fig. 1. Long accumulator with long shift for accurate scalar product accumulation

a mantissa of $2l$ digits. The exponent range of these numbers doubles also, i. e. $a_i \times b_i \in R(b, 2l, 2e1, 2e2)$. All these summands can be expressed in a fixed-point register of length $2e2 + 2l + 2|e1|$ without loss of information, see Fig. 1: If one of the summands has an exponent 0, its mantissa can be expressed in a register of length $2l$. If another summand has exponent 1, it can be expressed with exponent 0, if the register provides further digits on the left and the mantissa is shifted one place to the left. An exponent -1 in one of the summands requires a corresponding shift to the right. The largest exponents in magnitude that may occur in the summands are $2e2$ and $2|e1|$. So all summands can be expressed with exponent 0 in a fixed-point register of length $2e2 + 2l + 2|e1|$ without loss of information.

2.1 Solution A: Long Adder and Long Shift

If the register is built as an accumulator with an adder, all summands could even be added without loss of information. In order to accommodate possible overflows, it is convenient to provide a few, say k more digits of base b on the left. In such an accumulator, every such sum or scalar product can be added without loss of information. As many as b^k overflows may occur and be accommodated for without loss of information. In the worst case, presuming every sum causes an overflow, we can accommodate sums with $n \leq b^k$ summands.

A GIGAFLOPS computer would perform about 10^{17} operations in 10 years. So 17 decimal or about 57 binary digits certainly are a reasonable upper bound for k. Thus, the long accumulator and the long adder consist of $L = k + 2e2 + 2l + 2|e1|$ digits of base b. The summands are shifted to the proper position and added. See Fig. 1. Fast carry resolution techniques will be discussed later. The final sums s and c are supposed to be in the single exponent range $e1 \leq e \leq e2$, otherwise c is not representable as a floating-point number und the problem has to be scaled.

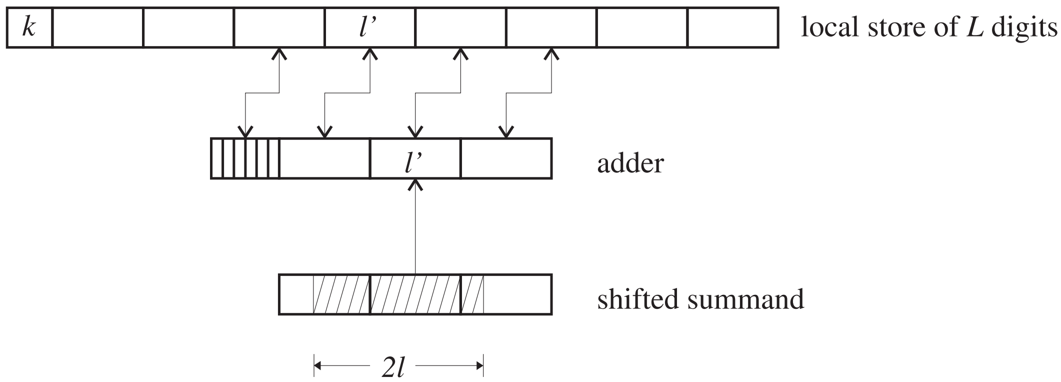


Fig. 2. Short adder and local store on the arithmetic unit for accurate scalar product accumulation

2.2 Solution B: Short Adder with Local Memory on the Arithmetic Unit

In a scalar product computation the summands are all of length $2l$. So actually the long adder and long accumulator may be replaced by a short adder and a local store of size L on the arithmetic unit. The local store is organized in words of length l or l', where l' is a power of 2 and slightly larger than l. (For instance $l = 53$ bits and $l' = 64$ bits). Since the summands are of length $2l$, they fit into a part of the local store of length $3l'$. This part of the store is determined by the exponent of the summand. We load this part of the store into an accumulator of length $3l'$. The summand mantissa is placed in a shift register and is shifted to the correct position as determined by the exponent. Then the shift register contents are added to the contents of the accumulator. Fig. 2.

An addition into the accumulator may produce a carry. As a simple method to accommodate carries, we enlarge the accumulator on its left end by a few more digit positions. These positions are filled with the corresponding digits of the local store. If not all of these digits equal $b - 1$ in case of addition (or zero in case of subtraction), they will accommodate a possible carry of the addition (or borrow in case of subtraction). Of course, it is possible that all these additional digits are $b - 1$ (or zero). In this case, a loop can be provided that takes care of the carry and adds it to the next digits of the local store. This loop may need to be traversed several times. Other carry handling processes are possible and will be dealt with later. This completes our sketch of the second method for an accurate computation of scalar products using a short adder and some local store on the arithmetic unit. See Fig. 2.

Remark 1: The purpose of the k digits on the left end of the register in Fig. 1 and Fig. 2 is to accommodate possible overflows. The only numbers that are added to this part of the register are plus or minus unity. So this part of the register just can be treated as a counter by an incrementer/decrementer.

Remark 2: The final result of a scalar product computation is assumed to be a floating-point number with an exponent in the range $e1 \leq e \leq e2$. During the computation, however, summands with an exponent outside of this range may very well occur. The remaining computation then has to cancel all these digits. This shows that normally in a scalar product computation, the register space outside the range $e1 \leq e \leq e2$ will be used less frequently. The conclusion should not be drawn from this consideration that the register size can be restricted to the single exponent range in order to save some silicon area. This would require the instalment of complicated exception handling routines which finally require as much silicon but do not solve the problem in principle. The scalar product is a most frequent operation in scientific computing. The two solutions A and B are both simple, straightforward and mature.

Remark 3: We emphasize once more that the number L of digits needed for the register to compute scalar products of two vectors to full accuracy only depends on the floating-point data format. In particular it is independent of the number n of components of the two vectors to be multiplied.

As samples we calculate the register width L for a few typical and frequently used floating-point data formats:

a) IEEE-arithmetic single precision:
$b = 2$; word length: 32 bits; sign: 1 bit; exponent: 8 bits; mantissa: $l = 24$ bits; exponent range: $e1 = -126$, $e2 = 127$, binary.
$L = k + 2e2 + 2l + 2|e1| = k + 554$ bits.
With $k = 86$ bits we obtain $L = 640$ bits. This register can be represented by 10 words of 64 bits.

b) /370 architecture, data format long:
$b = 16$; word length: 64 bits; sign: 1 bit; mantissa: $l = 14$ hex digits; exponent range: $e1 = -64$, $e2 = 63$, hexadecimal.
$L = k + 2e2 + 2l + 2|e1| = k + 282$ bits.
With $k = 88$ bits we obtain $L = 88 + 4 * 282 = 1216$ bits. This register can be represented by 16 words of 64 bits.

c) IEEE-arithmetic double precision:
$b = 2$; word length: 64 bits; sign: 1 bit; exponent: 11 bits; mantissa: $l = 53$ bits; exponent range: $e1 = -1022$, $e2 = 1023$, binary.
$L = k + 2e2 + 2l + 2|e1| = k + 4196$ bits.
With $k = 92$ bits we obtain $L = 4288$ bits. This register can be represented by 67 words of 64 bits.

These samples show that the register size (at a time where memory space is measured in gigabits and gigabytes) is modest in all cases. It grows with the exponent range of the data format. If this range should be extremely large, as for instance in case of an extended precision floating-point format, only an inner part of the register would be supported by hardware. The outer parts which then appear very rarely could be simulated in software. The data format long of the /370 architecture covers in decimal a range from about 10^{-75} to 10^{75} which is very modest. This architecture dominated the market for more than 20 years and most problems could conveniently be solved with machines of this architecture within this range of numbers.

Remark 4: In theoretical considerations, multiplication often is considered to be more complex than addition. In modern computer technology this is no longer the case. Very fast circuits for multiplication using carry-save-adders (Wallace tree) are available and common practise. They nearly equalize the time to compute a sum and a product of two floating-point numbers. In a scalar product computation usually a large number of products are to be computed. The multiplier is able to produce these products very quickly. In a balanced scalar product unit the accumulator should be able to absorb a product in about the same time the multiplier needs to produce it. Therefore, measures have to be taken to equalize the speed of both operations. Because of a possible long carry propagation the accumulation seems to be the more complicated process.

Remark 5: Techniques to implement the optimal scalar product on machines which do not provide enough register space on the arithmetic logical unit will be discussed in section 6 later in this paper.

2.3 Fast Carry Resolution

Both solutions A and B for our problem which we sketched above seem to be slow. Solution A requires a long shift which is necessarily slow. The addition over perhaps 4000 bits is slow also, in particular if a long carry propagation is necessary. For solution B, five steps have to be carried out: 1. read from the local store, 2. perform the shift, 3. add the summand, 4. resolve the carry, possibly by loops, and 5. write the result back into the local store. Again the carry resolution may be very time consuming.

As a first step to speed up solutions A and B, we discuss a technique which allows a very fast carry resolution. Actually a possible carry can already be accommodated while the product, the addition of which might produce a carry, is still being computed.

Both solutions A and B require a long register in which the final sum in a scalar product computation is built up. Henceforth we shall call this register *Long Accumulator* and abbreviate it by LA. It carries L bits. The long accumulator LA is a fixed-point register wherein any sum of floating-point numbers and of simple products of floating-point numbers can be represented without error.

To be more specific we now assume that we are using the data format double of the IEEE-arithmetic standard 754. See case c) of remark 3. As soon as the principles are clear, a transfer of the technique to other data formats is easy. Thus, in particular, the mantissa consists of $l = 53$ bits. We assume additionally that the long accumulator LA that appears in solutions A and B is subdivided into words of $l' = 64$ bits. The mantissa of the product $a_i \times b_i$ then is 106 bits wide. It touches at most three consecutive words of the long accumulator LA. These words are determined by the exponent of the product. A shifter then aligns the 106 bit product into the correct position for the subsequent addition into the three consecutive words of the long accumulator LA. This addition may produce a carry (or a borrow in case of subtraction). The carry is absorbed by that next more significant 64 bit word of the LA in which not all digits are 1 (or 0 in case of subtraction). Fig. 3, a). For a fast detection of this word two information bits or flags are appended to each long accumulator word. Fig. 3, b). One of these bits, the *all bits 1* flag, is set 1 if all 64 bits of the register word are 1. This means that a carry will propagate through the entire word. The other bit, the *all bits 0* flag, is set 0, if all 64 bits of the register word are 0. This means that in case of subtraction a borrow will propagate through the entire word.

During the addition of a product into the three consecutive words of LA, a search is started for the next more significant word of LA, where the *all bits 1* flag is not set. This is the word which will absorb a possible carry. If the addition generates a carry, this word must be incremented by one and all intermediate words must be changed from *all bits 1* to *all bits 0*. The easiest way to do this is simply to switch the flag bits from *all bits 1* to *all bits 0* with the additional semantics that if a flag bit is set, the appropriate constant (*all bits 0* or *all bits 1*) must be generated instead of reading the LA word contents when reading a LA word, Fig. 3, b). Borrows are handled in an analogous way.

This carry handling scheme allows a very fast carry resolution. The generation of the carry resolution address is independent of the addition of the product, so it can be performed in parallel. At the same time, a second set of flags is set up for the case that a carry is generated. If the latter is the case, the carry is added into the appropriate word and the second set of flags is copied into the former flag word.

Simultaneously with the multiplication of the mantissa of a_i and b_i their exponents are added. This is just an eleven bit addition. The result is available very fast. It delivers the exponent of the product and the address for its addition. By looking at the flags, the carry resolution address can be determined and the carry word can already be incremented/decremented as soon as the exponent of the product is available. This may already be the case before the multiplication of the mantissas is finished. If the accumulation of the product then produces a carry, the incremented/decremented carry word is written

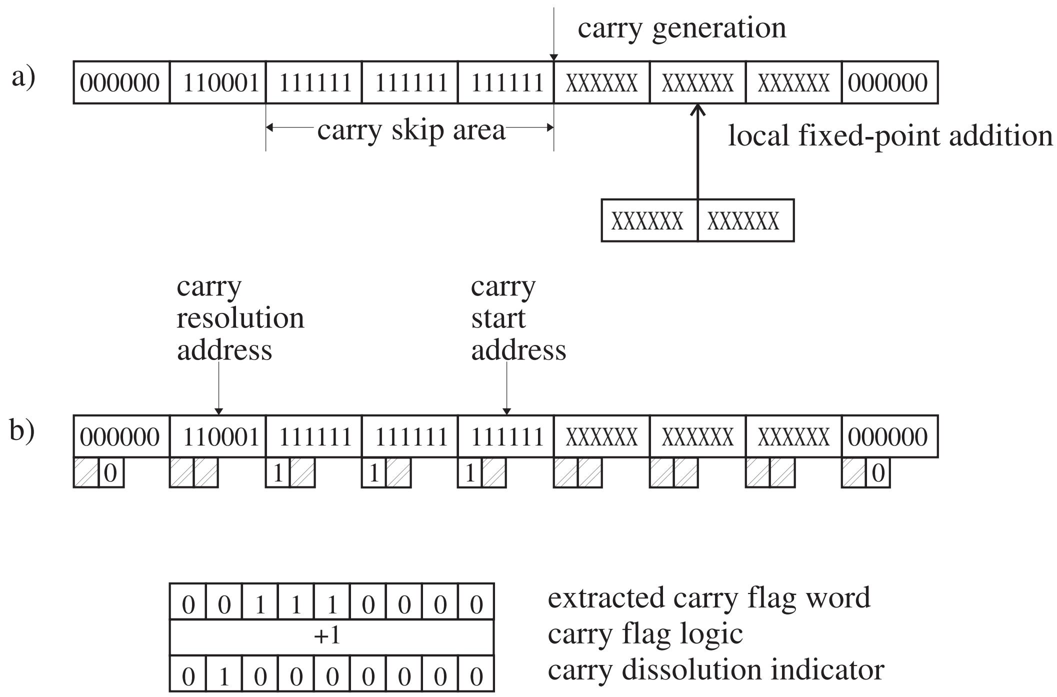


Fig. 3. Fast carry resolution

back into the LA otherwise nothing is changed.

This very fast carry resolution technique could be used in particular for the computation of short scalar products which occur, for instance, in the computation of the real and imaginary part of a product of two complex floating-point numbers. A long scalar product, however, is usually performed in a pipeline. Then, during the execution of a product, the former product is added. It seems to be reasonable then, to wait with the carry resolution until the former addition is really finished.

3 High-Performance Scalar Product Units (SPU)

After having discussed the two principle Solutions A and B for exact scalar product computation as well as a very fast carry handling scheme, we now turn to a more detailed design of scalar product computation units for diverse processors. These units will shortly be called SPU, which stands for Scalar Product Unit. If not otherwise mentioned we assume throughout this section that the data are stored in the format double precision of the IEEE-arithmetic standard 754. There the floating-point word has 64 bits and the mantissa consists of 53 bits. A central building block for the SPU is the long accumulator LA. It is a fixed-point register wherein any sum of floating-point numbers and of simple products of floating-point numbers can be represented without error. The unit allows the computation of scalar products of two vectors with any finite number of floating-point components to full accuracy or with one single rounding at the very end of the computation. As shown in Remark 3c), the LA consists of 4288 bits. It can be represented by 67 words of 64 bits.

The scalar product is a most frequent operation in scientific computation. So its execution should be fast. All circuits to be discussed in this section perform the scalar product in a pipeline which simultaneously executes the following steps:

a) read the two factors a_i and b_i to perform a product,
b) compute the product $a_i \times b_i$ to the full double length and
c) add the product $a_i \times b_i$ to the LA.

Step a) turns out to be the bottle neck of this pipeline. Therefore, we shall develop different circuits for computers which are able to read the two factors a_i and b_i into the SPU in 4, 2 or 1 portion. The latter case will be discussed in section 5. Step b) produces a product of 106 bits. It touches at most three consecutive words of the LA. The address of these words is determined by the products exponent. In step c) the 106 bit product is added to the three consecutive words of the LA. First we consider a

3.1 SPU for Computers with 32 Bit Data Bus (PC)

Here we consider a computer which is capable to read the data into the arithmetic logical unit and/or the SPU in portions of 32 bits. The Personal Computer is a typical representative of this kind of computers.

Solution A with an adder and a shifter for the full LA of 4288 bits would be too expensive. So the SPU for these computers is built upon solution B. For the computation of the product $a_i \times b_i$ the two factors a_i and b_i are to be read. Both consist of 64 bits. Since the data only can be read in 32 bit portions, the unit has to read 4 times. This is rather slow and turns out to be the bottle neck for the whole pipeline. In a balanced SPU the multiplier should be able to produce a product and the adder should be able to accumulate the product in about the same time the unit needs to read the data. Therefore, it suffices to provide a 27×27 bit multiplier. It computes the 106 bit product of the two 53 bit mantissas of a_i and b_i by 4 partial products. The following addition of the product into the three consecutive words of the LA is performed by an adder of 64 bits. The three words of the LA are loaded into the adder one after the other and the appropriate portion of the product is added. The sum is written back into the same word of the LA where the portion has been read from. A 64 out of 106 bit shifter must be used to align the product onto the relevant word boundaries. See Fig. 4. The addition of the three portions of the product into the LA may cause a carry. The carry is absorbed by incrementing

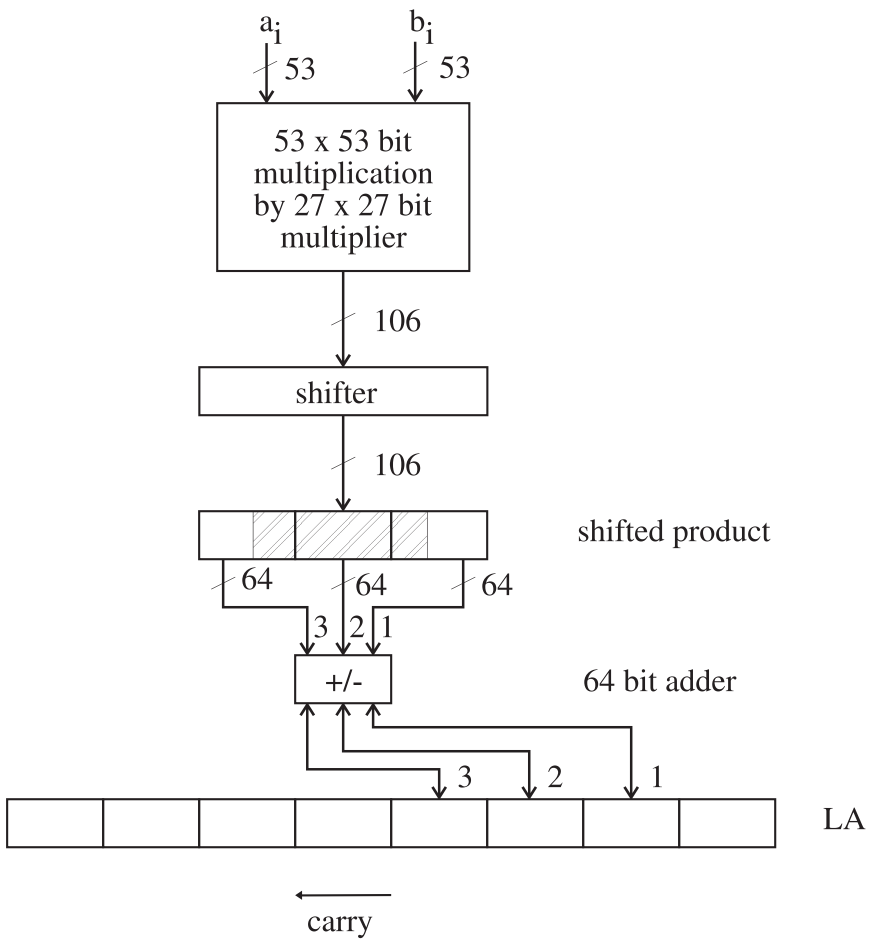


Fig. 4. Accumulation of a product to the LA by a 64 bit adder

(or decrementing in case of subtraction) a more significant word of the LA as determined by the carry handling scheme.

A brief sketch of the pipeline is shown in Fig. 5. There, we assume that a dual port RAM is available on the SPU to store the LA. This is usual for register memory. It allows to read from the LA and to write into the LA simultaneously. To read the two 64 bit factors a_i and b_i for a product, 8 machine cycles are needed, including the necessary decoding of the data. This also is about the time in which the multiplication and the shift can be performed in the second step of the pipeline. The three successive additions and the carry resolution in the third step of the pipeline again can be done in about the same time. See Fig. 5. Fig. 6 shows a block diagram for a SPU with 32 bit data supply.

The sum of the exponents of a_i and b_i delivers the exponent of the product $a_i \times b_i$. It consists of 12 bits. The 6 low order (less significant) bits of this sum are used to perform the shift. The more significant bits of the sum deliver the LA address to which the product $a_i \times b_i$ has to be added. So the originally very long shift is split into two fast shifts. The shifter performs a relatively

cycle	read	mult/shift	accumulate			
	read a_{i-1}^1					
	read a_{i-1}^2					
	read b_{i-1}^1					
	read b_{i-1}^2					
	read a_i^1					
	read a_i^2	$c_{i-1} := a_{i-1} * b_{i-1}$				
	read b_i^1	$c_{i-1} :=$ shift (c_{i-1})				
	read b_i^2					
	read a_{i+1}^1		load1			
			add/sub	load2		
	read a_{i+1}^2	$c_i := a_i * b_i$	store1	add/sub	load3	
				store2	add/sub	load carry
	read b_{i+1}^1	$c_i :=$ shift (c_i)			store3	inc/dec
						store carry
	read b_{i+1}^2					store flags
	read a_{i+2}^1		load1			
			add/sub	load2		
	read a_{i+2}^2	$c_{i+1} := a_{i+1} * b_{i+1}$	store1	add/sub	load3	
				store2	add/sub	load carry
	read b_{i+2}^1	$c_{i+1} :=$ shift (c_{i+1})			store3	inc/dec
						store carry
	read b_{i+2}^2					store flags
	read a_{i+3}^1		load1			
			add/sub	load2		
	read a_{i+3}^2	$c_{i+2} := a_{i+2} * b_{i+2}$	store1	add/sub	load3	
				store2	add/sub	load carry
	read b_{i+3}^1	$c_{i+2} :=$ shift (c_{i+2})			store3	inc/dec
						store carry
	read b_{i+3}^2					store flags

Fig. 5. Pipeline for the accumulation of scalar products on computers with 32 bit data supply

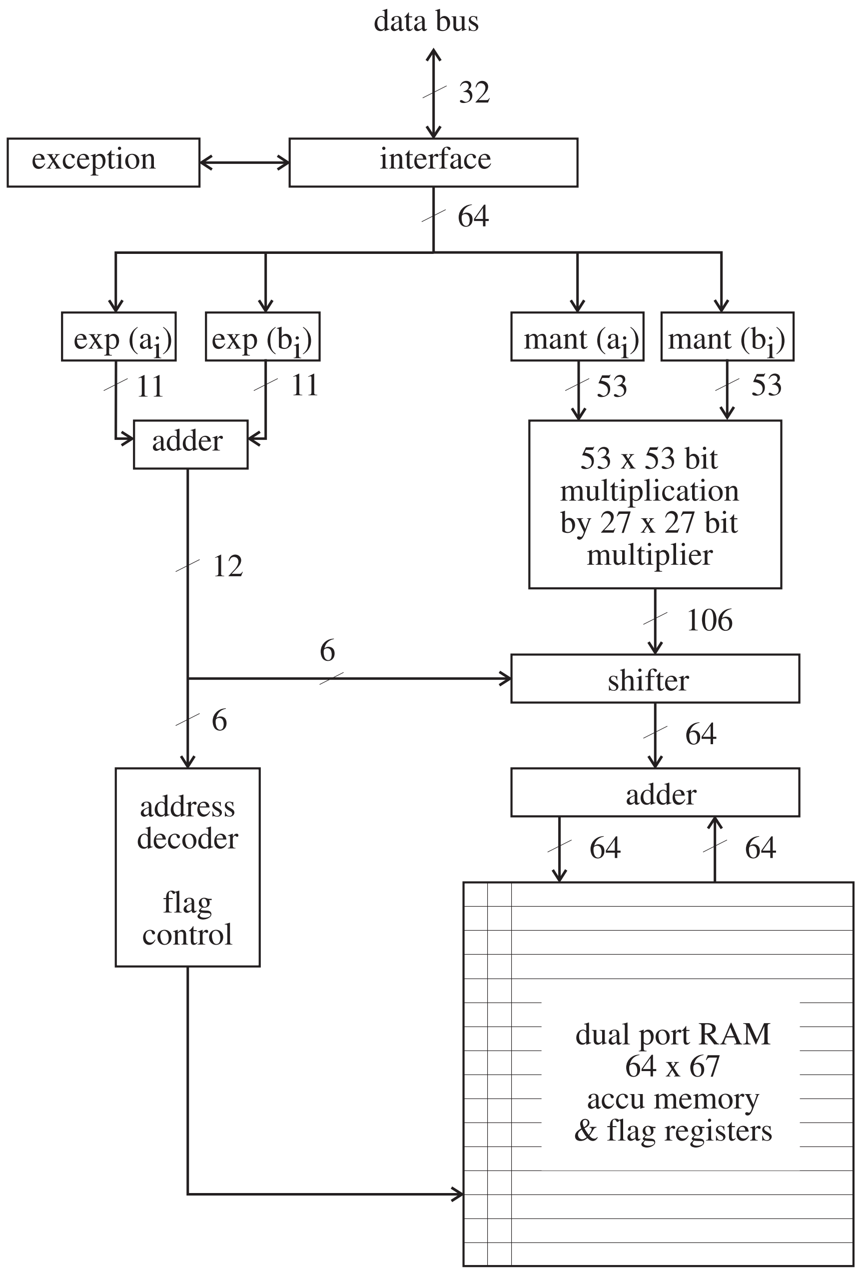


Fig. 6. Block diagram for a SPU with 32 bit data supply and sequential addition into SPU

short shift operation. The longer shift happens automatically while the three words of the LA are selected for the addition of the product.

The LA RAM needs only one address decoder to find the start address for an addition. The two more significant parts of the product are added to the contents of the two LA words with the two subsequent addresses. The carry logic determines the word which absorbs the carry. All these address decodings can be hard wired. The result of each one of the four additions is written back into the same LA word to which the addition has been executed. Symbolically the two flags appended to each accumulator word are indicated in Fig. 6 also. Practically the flags are kept in separate registers.

We stress the fact that in the circuit just discussed virtually no computing time is needed for the execution of the arithmetic. In the pipeline the arithmetic is performed in the time which is needed to read the data into the SPU. Here, we assumed that this requires 8 cycles. This allows to perform both the multiplication and the accumulation very economically and sequentially by a 27×27 bit multiplier and a 64 bit adder. Both, the multiplication and the addition are themselves performed in a pipeline.

There are processors on the market, where the data supply to the arithmetic unit or the SPU is much faster. We discuss the design of an SPU for such processors in the next section and in section 5.

3.2 SPU for Computers with 64 Bit Data Bus (Workstation, Mainframe)

Now we consider a computer which is capable to read the data into the arithmetic logical unit and/or the SPU in portions of 64 bits. Fast workstations or mainframes are typical representatives of this kind of computer.

Now the time to perform the multiplication and the accumulation in pipelines themselves as before is no longer available. In order to keep the execution time for the arithmetic within the time the SPU needs to read the data, we have to invest more hardware. For the multiplication now a 53×53 bit multiplier is to be used. The result is 106 bits wide again. It touches three 64 bit words of the LA. But the addition of the product as well as the carry resolution now have to be performed in parallel.

The 106 bit summand may fit into two instead of three consecutive 64 bit words of the LA. A closer look at the details shows that the 22 least significant bits of the three consecutive LA words are never changed by an addition of the 106 bit product. Thus the adder needs to be 170 bits wide only. Fig. 7 shows a sketch for the parallel accumulation of a product.

In the circuit a 106 to 170 bit shifter is used. The four additions are to be performed in parallel. So four read/write ports are to be provided for the LA RAM. A sophisticated logic must be used for the generation of the carry resolution address, since this address must be generated very fast. Again the LA RAM needs only one address decoder to find the start address for an addition. The more significant parts of the product are added to the contents of the two LA words with the two subsequent addresses. A tree structured carry logic now determines the LA word which absorbs the carry. A very fast hardwired multi-port driver can be designed which allows to read all 4 LA

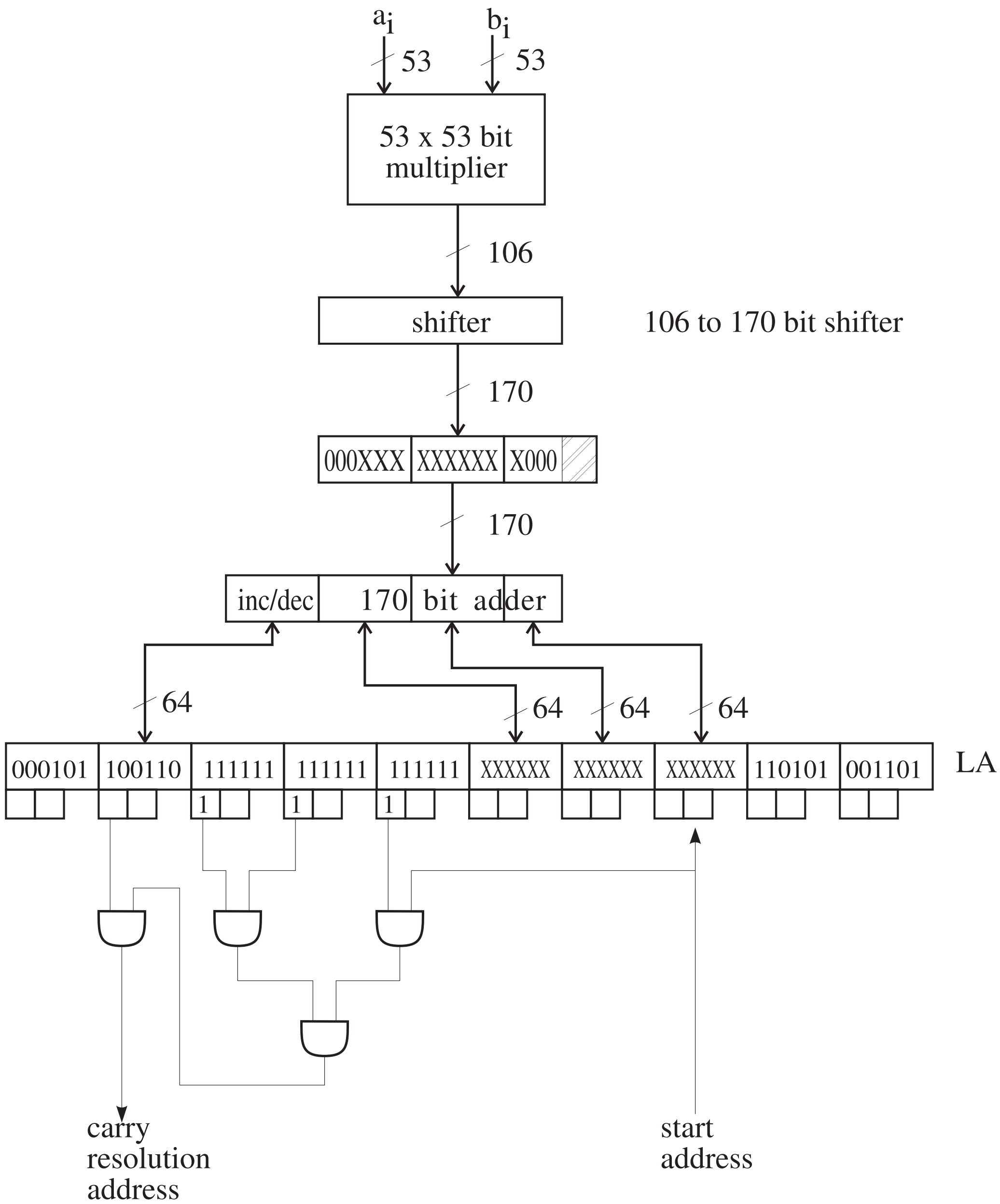


Fig. 7. Parallel accumulation of a product into the LA

words into the adder in one cycle.

Fig. 8 shows the pipeline for this kind of addition. In the figure we assume that 2 machine cycles are needed to decode and read one 64 bit word into the SPU.

Fig. 9 shows a block diagram for a SPU with 64 bit data supply and parallel addition.

We emphasize again that virtually no computing time is needed for the execution of the arithmetic. In a pipeline the arithmetic is performed in the time

cycle	read	mult/shift	accumulate
	read a_{i-1}		
	read b_{i-1}		
	read a_i	$c_{i-1} := a_{i-1} * b_{i-1}$	
	read b_i	$c_{i-1} :=$ shift (c_{i-1})	
	read a_{i+1}	$c_i := a_i * b_i$	address decoding load
	read b_{i+1}	$c_i :=$ shift (c_i)	add/sub c_{i-1} store & store flags
	read a_{i+2}	$c_{i+1} := a_{i+1} * b_{i+1}$	address decoding load
	read b_{i+2}	$c_{i+1} :=$ shift (c_{i+1})	add/sub c_i store & store flags
	read a_{i+3}	$c_{i+2} := a_{i+2} * b_{i+2}$	address decoding load
	read b_{i+3}	$c_{i+2} :=$ shift (c_{i+2})	add/sub c_{i+1} store & store flags

Fig. 8. Pipeline for the accumulation of scalar products

which is needed to read the data into the SPU. Here, we assumed that this requires 4 cycles for the two 64 bit factors a_i and b_i for a product. To keep up with the shorter time required to read the data, more hardware has to be invested for the multiplier and the adder.

If the technology is fast enough it may be reasonable to provide a 256 bit adder instead of the 170 bit adder. An adder width of a power of 2 may simplify the shift operation as well as the address decoding. The lower bits of the exponent of the product control the shift operation while the higher bits are directly used as the start address for the accumulation of the product into the LA.

The two flag registers appended to each accumulator word are indicated in Fig. 9 again. Practically the flags are kept in separate registers.

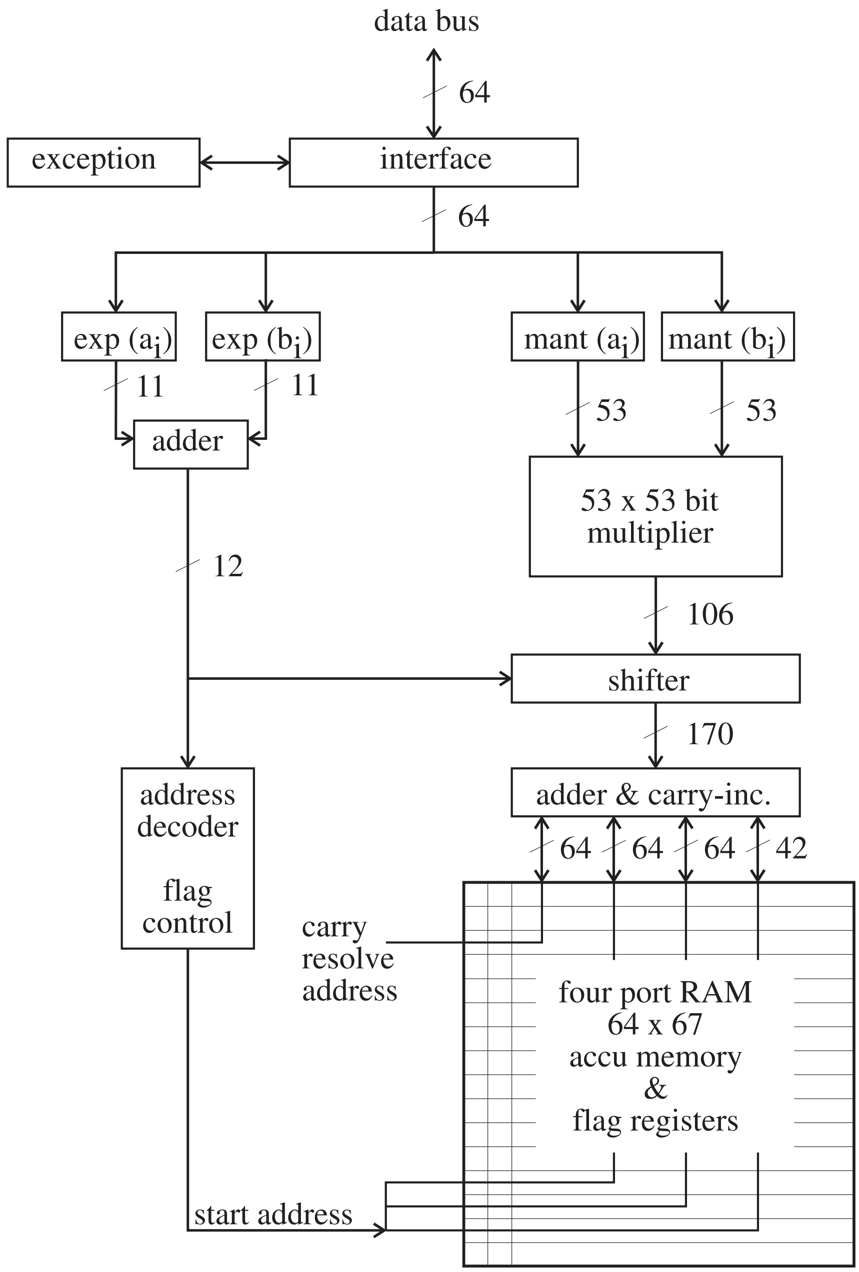


Fig. 9. Block diagram for a SPU with 64 bit data supply and parallel addition into SPU

4 Comments on the Scalar Product Units

4.1 Rounding

If the result of an exact scalar product computation is needed later in a program, the contents of the LA are to be put into the user memory. How this can be done will be discussed later in this section.

If not processed any further the correct result of a scalar product computation usually has to be rounded into a floating-point number or a floating-point interval, for instance. The flag bits that are used for the fast carry resolution can be used for the rounding of the LA contents also. By looking at the flag bits, the leading accumulator word can easily be identified. This and the next LA word are needed to compose the mantissa of the result. This 128 bit quantity must then be shifted to form a normalized mantissa of an IEEE-arithmetic double number. The shift length can be extracted by looking at the leading accumulator word practically with the same procedure which identified it by looking at the flag bit word.

For the correct execution of the rounding downwards (or upwards) it is necessary to check whether any one of the discarded bits is different from zero. This is done by testing the remaining bits of the 128 bit quantity in the shifter and by looking at the *all bits 0* flags of the following LA words. This information then is used to control the rounding.

Only one rounding appears at the very end of a scalar product computation. If a large number of products has been accumulated the contribution of the rounding to the computing time is not substantial. However, if many short scalar products or single floating-point operations addition and subtraction are to be computed with the SPU, a very fast rounding procedure is essential for the speed of the entire operation.

The rounding heavily depends on the speed with which the leading non zero digit of the LA can be detected. A pointer to this digit, which is carried along with the computation, would immediately identify this digit. The pointer logic requires additional hardware and its usefulness somehow decreases with the length of the scalar product to be computed.

For short scalar products or single floating-point operations a leading zero anticipation (LZA) would be more useful. The final result of a scalar product computation is supposed to lie in the exponent range between $e1$ and $e2$ of the LA. So hardware support for the LZA is only needed for this part of the LA. A comparison of the exponents of the summands identifies the LA word for which the LZA should be activated. The LZA consists of a fast computation of a provisional sum which differs from the correct sum by at most one leading zero. With this information the leading zeros and the shift width for the two LA words in question can be detected easily and fast. [Suz96].

4.2 How much Local Memory should be Provided on a SPU?

There are applications which make it desirable to provide more than one long accumulator on the SPU. If, for instance, the components of the two vectors

$a = (a_i)$ and $b = (b_i)$ are complex floating-point numbers, the scalar product $a \cdot b$ is a complex floating-point number again. It is obtained by accumulating the real and imaginary parts of the product of two complex floating-point numbers. The formula for the product of two complex floating-point numbers

$$(x = x_1 + ix_2,\ y = y_1 + iy_2 \ \Rightarrow$$
$$x \times y = (x_1 \times y_1 - x_2 \times y_2) + \ i\,(x_1 \times y_2 + x_2 \times y_1))$$

shows that the real and imaginary part of a_i and b_i are needed for the computation of both, the real part of the product $a_i \times b_i$ as well as the imaginary part.

Access to user memory is usually slower than access to register memory. To obtain high computing speed it is reasonable, therefore, to read the real and imaginary parts of the vector components only once and to compute the real and imaginary parts of the products simultaneously in two long accumulators on the SPU instead of reading the data twice and perform the two accumulations sequentially.

Very similar considerations show that a high speed computation of the scalar product of two vectors with interval components makes two long accumulators desirable as well.

There are still other good reasons to provide local memory space for more than one LA on the SPU. A program with higher priority may interrupt the computation of a scalar product and require a LA. The easiest way to solve this problem is to open a new LA for the program with higher priority. Of course, this can happen several times which raises the question how much local memory for how many long accumulators should be provided on a SPU. Three might be a good number to solve this problem. If a further interrupt requires another LA, the LA with the lowest priority could be mapped into the main memory by some kind of stack mechanism and so on. This technique would not limit the number of interrupts that may occur during a scalar product computation. These problems and questions must be solved in connection with the operating system.

For a time sharing environment memory space for more than one LA on the SPU may also be useful.

4.3 A SPU Instruction Set

For the SPU the following 10 instructions for the LA are useful and recommended:

(i) clear the LA,
(ii) add a product to the LA,
(iii) add a floating-point number to the LA,
(iv) subtract a product from the LA,
(v) subtract a floating-point number from the LA,
(vi) read LA and round to the destination format,
(vii) store LA contents in memory,
(viii) load LA contents from memory,
(ix) add LA to LA,
(x) Subtract LA from LA.

The clear instruction can be performed by setting all *all bits 0* flags to 0. The load and store instructions are performed by using the load/store instructions of the processor. For the add, subtract and round instructions the following denotations could be used. There the prefix *sp* identifies SPU instructions. *ln* denotes the floating-point format that is used and will be *db* for IEEE double. In all SPU instructions, the LA is an implicit source and destination operand. The number of the operation above is repeated at the end in parenthesis.

- spadd *ln* src1, src2
  multiply the numbers in the given registers and add the product to the LA, (2.).
- spadd *ln* src
  add the number in the given register to the LA, (3.).
- spsub *ln* src1, src2
  multiply the numbers in the given registers and subtract the product from the LA, (4.).
- spsub *ln* src
  subtract the number in the given register from the LA, (5.).
- spstore *ln.rd* dest
  get LA contents and put the rounded value into the destination register, (6.).
  In the instruction *rd* controls the rounding mode that is used when the LA contents is stored in a floating-point register. It is one of the following:
  - *rn* round to nearest
  - *rz* round towards zero
  - *rp* round upwards, i. e. towards plus infinity
  - *rm* round downwards, i. e. towards minus infinity

- spstore dest
  get LA contents and put its value into the destination memory operand, (7.).
- spload src
  load accumulator contents from the given memory operand into the LA, (8.).
- spadd src
  the contents of the accumulator at the location src are added to the contents of the accumulator in the processor, (9.).
- spsub src
  the contents of the accumulator at the location src are subtracted from the contents of the accumulator in the processor, (10.).

4.4 Interaction with High Level Programming Languages

This paper is motivated by the tremendous advances in computer technology that have been made in recent years. 100 million transistors can be placed on a single chip. This allows to extend the quality and high accuracy of the basic floating-point operations of addition, subtraction, multiplication and division to the arithmetic operations in the linear spaces and their interval extensions which are most commonly used in computation. A new fundamental operation, the scalar product is needed to provide this advanced computer arithmetic. The scalar product can be produced by an instruction *multiply and accumulate* and placed in the LA which has enough digit positions to contain the exact sum without rounding. Only a single rounding error of at most one unit in the last place is introduced when the completed scalar product (often also called dot product) is returned to one of the floating-point registers.

By operator overloading in modern programming languages matrix and vector operations can be provided with highest accuracy and in a simple notation, if the optimal scalar product is available. However, many scalar products that occur in a computation do not appear as vector or matrix operations in the program. A vectorizing compiler certainly is a good tool and method to detect such additional scalar products in a program. Since the hardware supported optimal scalar product is faster than a conventional computation in floating-point arithmetic this would increase both, the accuracy and the speed of the computation.

In the computer, the scalar product is produced by several, more elementary computer instructions as shown in the last section. Programming and the detection of scalar products in a program can be simplified by a great deal if several of these computer instructions are given into the hands of the user and incorporated into high level programming languages. This has been done with great success in the so called XSC-languages (eXtended Scientific Computation) that have been developed at the author's institute [Kla91,Kla92,Kla93,Kla93a,Kul87,Kul87a,IAM90,IBM90]. We mention a few of these constructs and demonstrate their usefulness. Central to this is the idea to allow variables of the size of the LA to be defined in a user's program. For this purpose a new data type called *dotprecision* is introduced. A variable of the type *dotprecision* is a fixed-point variable with $L = k + 2e2 + 2l + 2|e1|$ digits of base b. See Fig. 1. As has been shown earlier, every finite sum of floating-point products $\sum_{i=1}^{n} a_i \times b_i$ can be represented as a variable of type *dotprecision*. Moreover, every such sum can be computed in a local store of length L on the SPU without loss of information. See Fig. 1. Along with the type *dotprecision* the following collection of constructs serve as primitives for developing expressions in a program which easily can be evaluated with the SPU instruction set:

dotprecision	new data type
:=	assignment from dotprecision
	to dotprecision or
	to real with rounding to nearest or
	to interval with roundings downwards and upwards
	depending on the type on the left hand side of the
	:= operator.

For variables of type dotprecision so called *dotprecision expressions* are permitted which are defined as sums of *simple expressions*. A *simple expression* is either a signed or unsigned constant or a variable of type real or a single product of two such objects or another dotprecision variable. All operations (multiplications and accumulations) are to be executed to full accuracy.

For instance, let x be a variable of type dotprecision and y and z variables of type real. Then in the assignment

```
x := x + y * z
```

the double length product of y and z is added to the variable x of type dotprecision and its new value is assigned to x.

The scalar product of two vectors $a = (a_i)$ and $b = (b_i)$ is now easily implemented with a variable x of type dotprecision as follows:

```
x := 0;
for i := 1 to n do x := x + a[i] * b[i];
y := x;
```

The last statement $y := x$ rounds the value of the variable x of type dotprecision into the variable y of type real by applying the standard rounding of the computer. y then has the value of the scalar product $a \boxdot b$ which is within a single rounding error of the exact scalar product $a \cdot b$.

For example, the method of defect correction or iterative refinement requires highly accurate computation of expressions of the form

$$\mathrm{a} \cdot \mathrm{b} - \mathrm{c} \cdot \mathrm{d}$$

with vectors a, b, c, and d. Employing a variable x of type dotprecision, this expression can now be programmed as follows:

```
x := 0;
for i := 1 to n do x := x + a[i] * b[i];
for i := 1 to n do x := x − c[i] * d[i];
y := x;
```

The result, involving $2n$ multiplications and $2n-1$ additions, is produced with but a single rounding operation.

In the last two examples y could have been defined to be of type interval. Then the last statement $y := x$ would produce an interval with a lower bound which is obtained by rounding the dotprecision value of x downwards and an upper bound by rounding it upwards. Thus, the bounds of y will be either the same or two adjacent floating-point numbers.

In the XSC-languages the functionality of the dotprecision type and expression is available also for complex data as well as for interval and complex interval data.

5 Scalar Product Units for Top-Performance Computers

By definition a top-performance computer is capable to read two data x and y to perform a product $x \times y$ into the arithmetic logical unit and/or the SPU simultaneously in one portion. Supercomputers are typical representatives of this kind of computers. Usually the floating-point word consists of 64 bits and the data bus is 128 or even more bits wide. However, also digital signal processors with a word size of 32 bits can belong into this class if two 32 bit words are read into the ALU and/or SPU in one portion. For these kind of computers both solutions A and B which have been sketched in sections 2.1 and 2.2 make sense and will be discussed. The higher the speed of the system the more hardware has to be invested. The most involved and expensive solution seems to be best suited to reveal the basic ideas. So we begin with solution A using a long adder for the data format double precision.

5.1 Long Adder for 64 Bit Data Word (Vector Processor) (Solution A)

In [Kir87] the basic ideas have been developed for a general data format. However, to be very specific we discuss here a circuit for the data format double precision of the IEEE-arithmetic standard 754. There the word size consists of 64 bits. The mantissa has 53 bits and the exponent 11 bits. The exponent covers a range from -1022 to $+1023$. The LA has 4288 bits. We assume again that the scalar product computation can be subdivided into a number of independent steps like

a) read a_i and b_i
b) compute the product $a_i \times b_i$
c) add the product to the LA.

Now by assumption the SPU can read the two factors a_i and b_i simultaneously in one portion. We call the time that is needed for this a *cycle*. Then, in a balanced design, steps b) and c) should both be performed in about the same time. Using well known fast multiplication techniques like Booth-Recoding and Wallace-tree this certainly is possible for step b). Here, the two 53 bit mantissas are multiplied. The product has 106 bits. The main difficulty seems to appear in step c). There, we have to add a summand of 106 bits to the LA in every *cycle*.

With solution A the addition is performed by a long adder and a long shift, both of $L = 4288$ bits. An adder and a shift of this size are necessarily slow, certainly too slow to process one summand of 106 bits in a single *cycle*. Therefore, measures have to be taken which speed up the addition as well as the shift. As a first step we subdivide the long adder into shorter pieces. Without loss of generality we assume that the pieces consist of 64 bits. [5] A 64 bit adder certainly is faster than a 4288 bit adder. Now each one of the 64 bit adders may produce a carry. We write these carries into carry registers between two adjacent adders. See Fig. 10.

If a single addition has to be performed these carries still have to be propagated. In a scalar product computation, however, this is not necessary. We assume that a large number of summands has to be added. We simply add the carry with the next summand to the next more significant adder. Only at the very end of the accumulation, when no more summands are coming, carries may have to be eliminated. However, every summand is relatively short. It

[5] other pieces are possible, see [Kir87,Kir88].

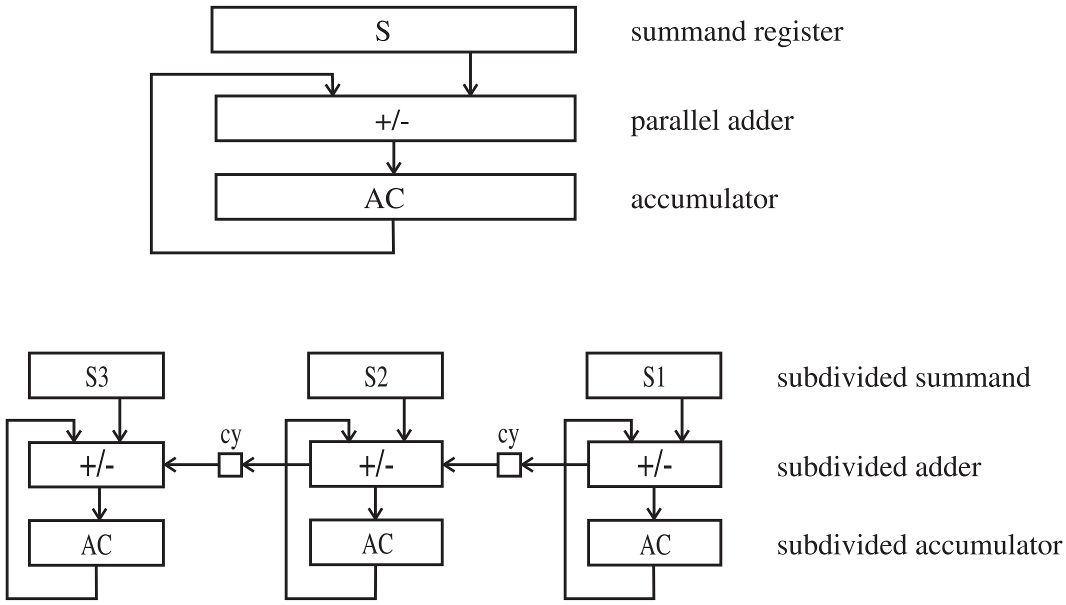


Fig. 10. Parallel and subdivided parallel adder

consists of 106 bits only. So during the addition of a summand, carries are only produced in a small part of the 4288 bit adder. The carry elimination, on the other hand, takes place during each step of the addition wherever a carry is left. So in an average case there will only be very few carries left at the end of the accumulation and a few additional *cycles* will suffice to absorb the remaining carries. Thus, subdividing the adder enables it to keep up in speed with steps a) and b) and to read and process a summand in each *cycle*.

The long shift of the 106 bit summand is slow also. It is speeded up by a matrix shaped arrangement of the adders. Only a few, let us assume here four of the partial adders are placed in a row. We begin with the four least significant adders. The four next more significant adders are placed directly beneath of them and so on. The most significant adders are forming the last row. The rows are connected as shown in Fig. 11.

In our example, where we have 67 adders of 64 bits, 17 rows suffice to arrange the entire summing matrix. Now the long shift is performed as follows: The summand of 106 bits carries an exponent. In a speed shifter of 106 to 256 bits it is shifted into a position where its most significant digit is placed directly above of the position in the long adder which carries the same exponent identification. The remaining digits of the summand are placed immediately to its right. Now the summing matrix reads this summand into the S-registers (summand register) of every row. The addition is executed in that row where the exponent identification coincides with the one of the summand.

It may happen that the most significant digit of the summand has to be shifted pretty far to the right. Then the remaining digits would hang over at the right

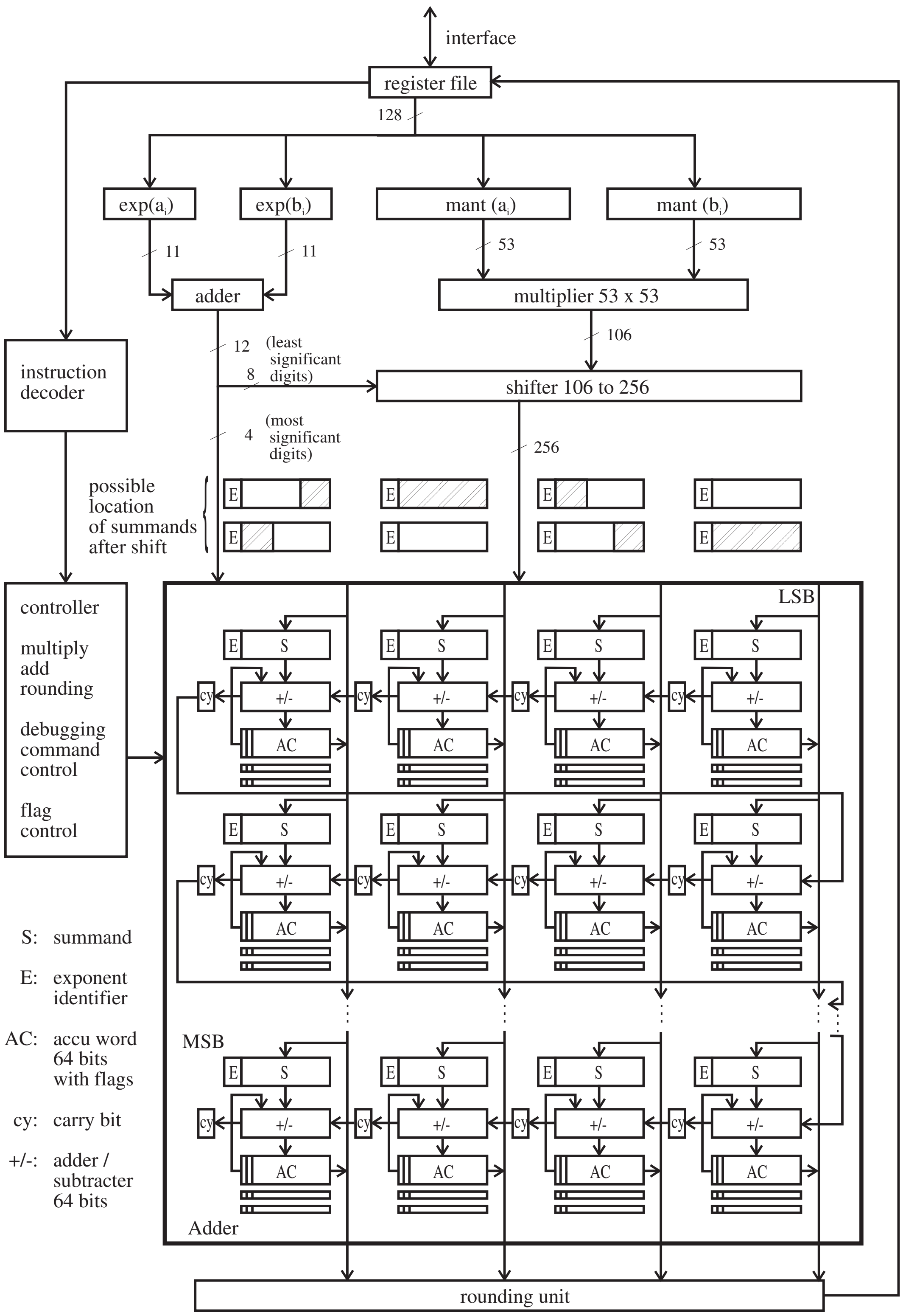


Fig. 11. Block diagram of a SPU with long adder for a 64 bit data word and 128 bit data supply

end of the shifter. These digits then are reinserted at the left end of the shifter by a ring shift. If now the more significant part of the summand is added in row r, its less significant part will be added in row $r-1$.

By this matrix shaped arrangement of the adders, the unit can perform both a shift and an addition in a single *cycle*. The long shift is reduced to a short shift of 106 to 256 bits which is fast. The remaining shift happens automatically by the row selection for the addition in the summing matrix.

Every summand carries an exponent which in our example consists of 12 bits. The lower part of the exponent, i. e. the 8 less significant digits determine the shift width and with it the selection of the columns in the summing matrix. The row selection is obtained by the 4 more significant bits of the exponent. This complies roughly with the selection of the adding position in two steps by the process described in the context of Fig. 2. The shift width and the row selection for the addition of a product $a_i \times b_i$ to the LA are known as soon as the exponent of the product has been computed. Since the exponents of a_i and b_i consist of 11 bits only, the result of their addition is available very quickly. So while the multiplication of the mantissas is still being executed the shifter can already be switched and the addresses of the LA words for the accumulation of the product $a_i \times b_i$ can be selected.

The 106 bit summand touches at most three consecutive words of the LA. The addition of the summand is executed by these three partial adders. Each one of these adders can produce a carry. The carry of the left of these partial adders can with high probability be absorbed, if the addition always is executed over a full row of four adders. The fourth adder then can be chosen to be the next more significant one. This can reduce the number of carries that have to be resolved during future steps of the accumulation and in particular at the end.

In each step of the accumulation an addition only has to be activated in the selected row of adders and in those adders where a non zero carry is waiting to be absorbed. This adder selection can reduce the power consumption for the accumulation step essentially.

The carry resolution method that has been discussed so far is most natural. It is simple and does not require particular hardware support. If long scalar products are being computed it works very well. Only at the end of the accumulation, if no more summands are coming, a few additional *cycles* may be required to absorb the remaining carries. Then a rounding can be executed. This number of additional *cycles* for the carry resolution at the end of the accumulation, however, although it is small in general, depends on the data and is unpredictable. In case of short scalar products the time needed for these additional *cycles* may be unproportionally high and indeed exceed the addition time.

With the fast carry resolution mechanism that has been discussed in section 2.3 these difficulties can be overcome. At the cost of some additional hardware all carries can be absorbed immediately at each step of the accumulation. The method is displayed in Fig. 11 also. Two flag registers for the *all bits 0* and the *all bits 1* flags are shown at the left end of each partial accumulator word in the figure. The addition of the 106 bit products is executed by three consecutive partial adders. Each one of these adders can produce a carry. The carries between two of these adjacent adders can be avoided, if all partial adders are built as *Carry Select Adders*. This increases the hardware costs only moderately. The carry registers between two adjacent adders then are no longer necessary. The flags indicate which one of the more significant LA words will absorb the left most carry. During an addition of a product only these 4 LA words are changed and only these 4 adders need to be activated. The addresses of these 4 words are available as soon as the exponent of the summand $a_i \times b_i$ has been computed. During the addition step now simultaneously with the addition of the product the carry word can be incremented (decremented). If the addition produces a carry the incremented word will be written back into the local accumulator. If the addition does not produce a carry, the local accumulator word remains unchanged. Since we have assumed that all partial adders are built as *Carry Select Adders* this final carry resolution scheme requires no additional hardware. Simultaneously with the incrementation/decrementation of the carry word a second set of flags is set up for the case that a carry is generated. If the latter is the case the second set of flags is copied into the former word.

The accumulators that belong to partial adders in Fig. 11 are denoted by AC. Beneath of them a small memory is indicated in the figure. It can be used to save the LA contents very quickly in case that a program with higher priority interrupts the computation of a scalar product and requires the unit for itself.

In Fig. 11 the register for the summands carry an exponent identification denoted by E. It is very useful for the final rounding. The usefulness of the flags for the final rounding has already been discussed. They also serve for a fast clearance of the accumulator.

The SPU which has been discussed in this section seems to be costly. However, it consists of a large number of identical parts and it is very regular. This allows a very compact design. Furthermore the entire unit is simple. No particular exception handling techniques are to be dealt with by the hardware. Vector computers are the most expensive systems. A compact and simple solution, though expensive, is justified for these systems.

In this section as well as in section 5.4 we consider a computer which uses a 32 bit floating-point word and which is capable to read two such words into the ALU and/or SPU simultaneously in one portion. Digital Signal Processors may be representatives of this kind of computers. Real time computing requires very high computing speed and high accuracy in the result. As in the last section we call the time that is needed to read the two 32 bit floating-point words a *cycle*.

We first develop circuitry which realizes Solution A using a long adder and a long shift. To be very specific we assume that the data are given as single precision floating-point numbers conforming to the IEEE-arithmetic standards 754. There the mantissa consists of 24 bits and the exponent has 8 bits. The exponent covers a range from -126 to $+127$ (in binary). As discussed in Remark 3a) of section 2.2, 640 bits are a reasonable choice for the LA. It can be represented by 10 words of 64 bits.

Again the scalar product is computed by a number of independent steps like

a) read a_i and b_i,
b) compute the product $a_i \times b_i$,
c) add the product to the LA.

Each one of the mantissas of a_i and b_i has 24 bits. Their product has 48 bits. It can very fast be computed by a 24×24 bit multiplier using standard techniques like Booth-Recoding and Wallace tree. The addition of the two 8 bit exponents of a_i and b_i delivers the exponent of the product consisting of 9 bits.

The LA consists of 10 words of 64 bits. The 48 bit mantissas of the product touches at most two of these words. The addition of the product is executed by the corresponding two consecutive partial adders. Each one of these two adders can produce a carry. The carry between the two adjacent adders can immediately be absorbed if all partial adders are built as *Carry Select Adders* again. The carry of the more significant of the two adders will be absorbed by one of the more significant 64 bit words of the LA. The flag mechanism (see section 2.3) indicates which one of the LA words will absorb a possible carry. So during an addition of a summand the contents of at most 3 LA words are changed and only these three partial adders need to be activated. The addresses of these words are available as soon as the exponent of the summand $a_i \times b_i$ has been computed. During the addition step simultaneously with the addition of the product the carry word can be incremented (decremented). If the addition produces a carry the incremented word will be written back into the local accumulator. If the addition does not produce a carry, the local accumulator word remains unchanged. Since all partial adders are built as *Carry Select Adders* no additional hardware is needed for the carry resolution. Simultaneously with the incrementation/decrementation of the carry word a second set of flags is set up for the case that a carry is generated. If the latter is the case the second set of flags is copied into the former flag word.

Details of the circuitry just discussed are summarized in Fig. 12. The figure is organized in great similarity to Fig. 11 of the previous section. In order to avoid the long shift, the long adder is designed as summing matrix consisting of 2 adders of 64 bits in each row. For simplicity in the figure only 3 rows (of the 5 needed to represent the full LA) are displayed.

In a speed shifter of 48 to 128 bits the 48 bit product is shifted into a position where its most significant digit is placed directly above of the position in the long adder which carries the same exponent identification. The remaining digits of the summand are placed immediately to its right. If they hang over at the right end of the shifter, they are reinserted at the left end by a ring shift. Above of the summing matrix in Fig. 12 two possible positions of summands after the shift are indicated.

The summing matrix now reads the summand into its S-registers. The addition is executed by those adders where the exponent identification coincides with the one of the summand. The exponent of the summand consists of 9 bits. The lower part, i. e. the 7 less significant digits determine the shift width. The selection of the two adders which perform the addition is obtained by the more significant bits of the exponent.

In Fig. 12 again some memory is indicated for each part of the LA. It can be used to save the LA contents very quickly in case a program with higher priority interrupts the computation of a scalar product and requires the unit for itself.

In comparison with Fig. 11, Fig. 12 shows an additional 32 bit data path directly from the input register file to the speed shifter. This data path is supposed to allow a very fast execution of the operation *multiply and add fused*, $rnd(a \times b + c)$, which is provided by some conventional floating-point processors. While the product $a \times b$ is computed by the multiplier, the summand c can already be added to the LA.

The SPU which has been discussed in this section seems to be costly. While a single floating-point addition conveniently can be done with one 64 bit adder, here 640 full adders have been used in carry select adder mode. However, the advantages of this design are tremendous. While a conventional floating-point addition can produce a completely wrong result with only two or three additions, the new unit never delivers a wrong answer, even if millions of floating-point numbers or single products of such numbers are added. An error

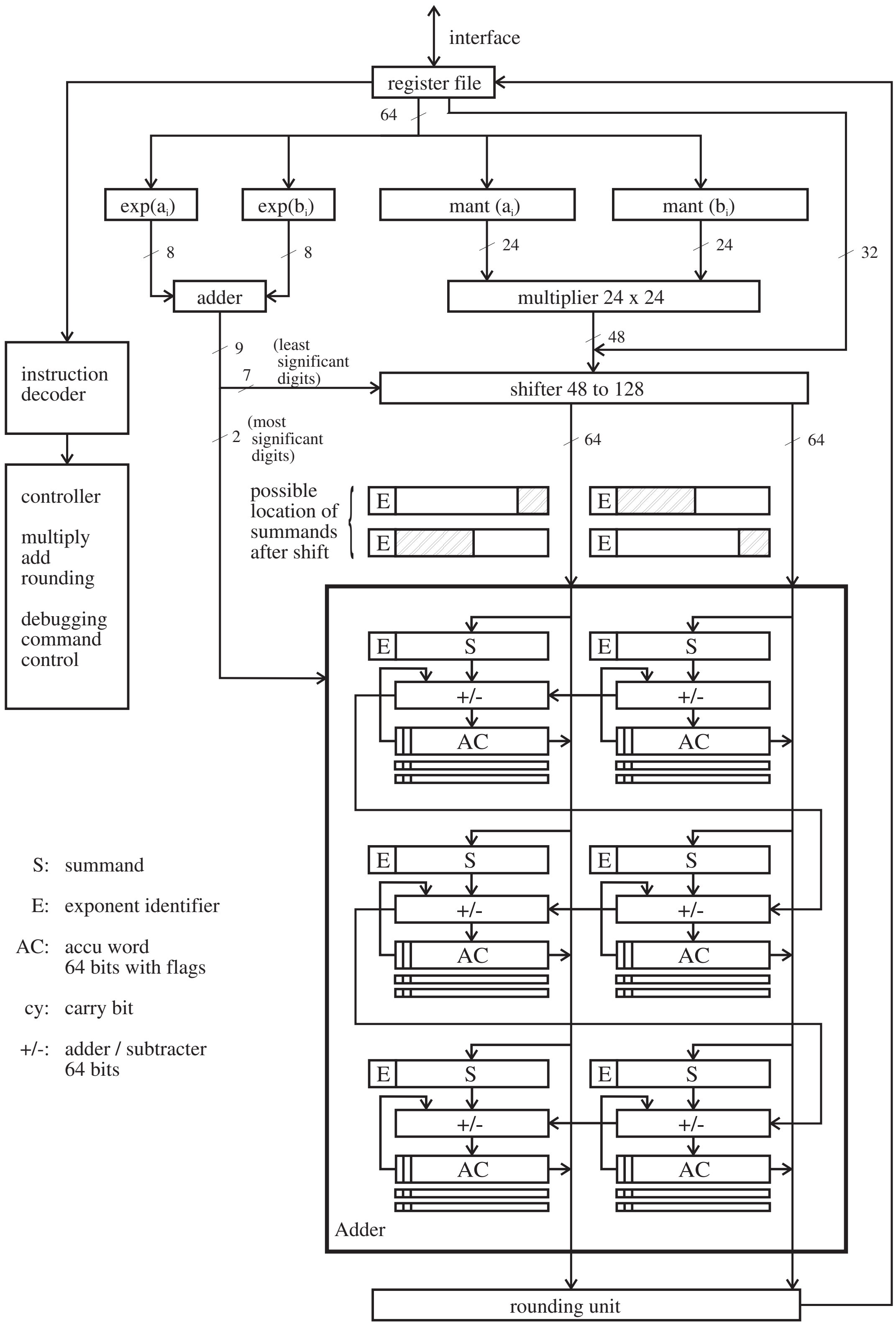


Fig. 12. Block diagram of a SPU with long adder for a 32 bit data word and 64 bit data supply

analysis is never necessary for these operations. The unit consists of a large number of identical parts and it is very regular. This allows a very compact design. No particular hardware has to be spent to deal with rare exceptions. Although the increase in adder equipment by a factor of 10, compared with a conventional floating-point adder, seems to be high, the number of full adders spent for the circuitry is not extraordinary. We stress the fact that for a Wallace tree in case of a standard 53×53 bit multiplier about the same amount of full adders is used. For fast conventional computers this is the state of the art multiplication since many years and nobody complains about high costs.

5.3 Short Adder with Local Memory on the Arithmetic Unit for 64 Bit Data Word (Vector Processor) (Solution B, preliminary)

In the circuits discussed in sections 5.1 and 5.2 adder equipment was provided for the full width of the LA. The long adder was subdivided into partial adders of 64 bits. In section 5.1 67 and in section 5.2 10 such units were used. During an addition of a summand, however, in section 5.1 only 4 and in section 5.2 only 3 of these units are activated. This raises the question whether adder equipment is really needed for the full width of the LA and whether the accumulation can be done with only 4 respectively 3 adders in accordance with Solution B of section 2.2. There the LA is kept as local memory on the arithmetic unit.

In this section we develop such a solution for the data format double precision. A principle solution using a short adder and local memory on the arithmetic unit was already discussed in section 3.2. There the data a_i and b_i to perform a product $a_i \times b_i$ are read into the SPU successively in two portions of 64 bits. This leaves 4 machine *cycles* to perform the accumulation in the pipeline.

Now we assume that the two data a_i and b_i for a product $a_i \times b_i$ are read into the SPU simultaneously in one portion of 128 bits. Again we call the time that is needed for this a *cycle*. In accordance with the solution shown in Fig. 11 and section 5.1 we assume again that the multiplication and the shift also can be done in one such read *cycle*. In a balanced pipeline, then, the circuit for the accumulation must be able to read and process one summand in each (read) *cycle* also. The circuit in Fig. 13 displays a solution. In great similarity to the summing matrix in Fig. 11 we assume there that the local memory LA is organized in 17 rows of four 64 bit words.

In each *cycle* the multiplier supplies a product (summand) to be added in the accumulation unit. Every such summand carries an exponent which in our example consists of 12 bits. The 8 lower (less significant) bits of the exponent

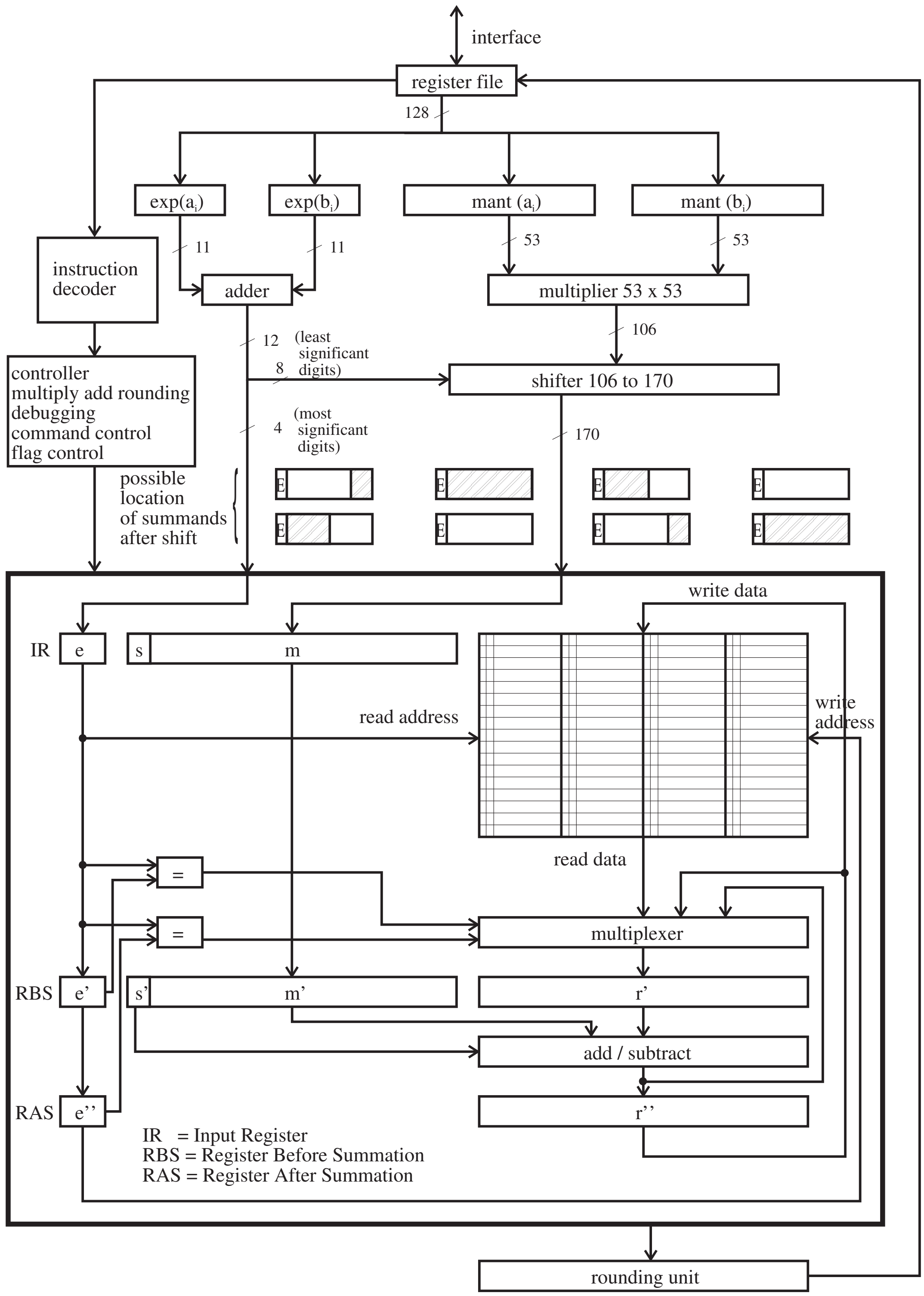


Fig. 13. Block diagram of a SPU with short adder and local store for a 64 bit data word and 128 bit data supply

determine the shift width. The row selection of the LA is obtained by the 4 most significant bits of the exponent. This complies roughly with the selection of the adding position in two steps by the process described in the context of Fig. 2. The shift width and the row selection for the addition of the product to the LA are known as soon as the exponent of the product has been computed. Since the exponents of a_i and b_i consist of 11 bits only, the result of their addition is available very quickly. So while the multiplication of the mantissa is still being executed the shifter can already be switched and the addresses for the LA words for the accumulation of the product $a_i \times b_i$ can be selected.

After being shifted the summand reaches the accumulation unit. It is read into the input register IR of this unit. The shifted summand now consists of an exponent e, a sign s and a mantissa m. The mantissa touches three consecutive words of the LA, while the exponent is reduced to the four most significant bits of the original exponent of the product.

Now the addition of the summand is performed in the accumulation unit by the following three steps:

(i) The local memory is addressed by the exponent e. The contents of the addressed part of the LA including the word which resolves the carry are transferred to the register before summation RBS. This transfer moves four words of 64 bits. The summand is also transferred from IR to the corresponding section of RBS. In Fig. 13 this part of the RBS is denoted by e', s' and m' respectively.

(ii) In the next *cycle* the addition or subtraction is executed in the add/subtract unit according to the sign. The result is transferred to the register after summation RAS. The adder/subtracter consists of 4 parallel adders of 64 bits which are working in carry select mode. The summand touches three of these adders. Each one of these three adders can produce a carry. The carries between two of these adjacent adders are absorbed by the carry select addition. The fourth word is the carry word. It is selected by the flag mechanism. During the addition step a 1 is added to or subtracted from this word in carry select mode. If the addition produces a carry the incremented/ decremented word will be selected. If the addition does not produce a carry this word remains unchanged. Simultaneously with the incrementation/decrementation of the carry word a second set of flags is set up which is copied into the flag word in the case that a carry is generated. In Fig. 13 two possible locations of the summand after the shift are indicated. The carry word always is the most significant word. An incrementation/decrementation of this word never produces a carry. Thus the adder/subtracter in Fig. 13 simply can be built as a parallel carry select adder.

(iii) In the next *cycle* the computed sum is written back into the same four memory cells of the LA to which the addition has been executed. Thus only one address decoding is necessary for the read and write step. A different bus called *write data* in Fig. 13 is used for this purpose.

In summary the addition consists of the typical three steps: 1. read the summand, 2. perform the addition and 3. write the sum back into the (local) memory. Since a summand is delivered from the multiplier in each *cycle*, all three phases must be active simultaneously, i. e. the addition itself must be performed in a pipeline. This means that it must be possible to read from the memory and to write into the memory in each *cycle* simultaneously. So two different data paths have to be provided. This, however, is usual for register memory.

The pipeline for the addition consists of three steps. Pipeline conflicts are very well possible. A pipeline conflict occurs if an incoming summand needs to be added to a partner from the LA which is still being computed and not yet available in the local memory. These situations can be detected by comparing the exponents e, e' and e'' of three successively incoming summands. In principle all pipeline conflicts can be solved by the hardware. Here we discuss the solution of two pipeline conflicts which with high probability are the most frequent occurrences.

One conflict situation occurs if two consecutive products carry the same exponent e. In this case the two summands touch the same three words of the LA. Then the second summand is unable to read its partner for the addition from the local memory because it is not yet available. This situation is checked by the hardware where the exponents e and e' of two consecutive summands are compared. If they are identical, the multiplexer blocks off the process of reading from the local memory. Instead the sum which is just being computed is directly written back into the register before summation RBS via the multiplexer so that the second summand can immediately be added without memory involvement.

Another possibility of a pipeline conflict occurs if from three successively incoming summands the first one and the third one carry the same exponent. Since the pipeline consists of three steps, the partner for the addition of the third one then is not yet in the local memory but still in the register after summation RAS. This situation is checked by the hardware also, see Fig. 13. There the two exponents e and e'' of the two summands are compared. In case of coincidence the multiplier again suppresses the reading from the local memory. Instead now, the sum of the former addition, the result of which is still in RAS, is directly written back into the register RBS before summation via the multiplexer. So also this pipeline conflict can be solved by the hardware without memory involvement.

The case $e = e' = e''$ is also possible. It would cause a reading conflict in the

	5			4	3	2	1	

Fig. 14. Carry propagation in case of a pipeline conflict

multiplexer. The situation can be avoided by writing a dummy exponent into e'' or by reading from the add/subtract unit with higher priority.

The product that arrives at the accumulation unit touches three consecutive words of the LA. A more significant fourth word absorbs the possible carry. The solution for the two pipeline conflicts just described work well, if this fourth word is the next more significant word. A carry is not absorbed by the fourth word if all its bits are one resp. zero. The probability that this is the case is $1 : 2^{64} < 10^{-18}$. In the vast majority of instances this will be the case.

If it is not the case the word which absorbs the carry is selected by the flag mechanism and read into the most significant word of the RBS. The addition step then again works well including the carry resolution. But difficulties occur in both cases of a pipeline conflict. Fig. 14 displays a certain part of the LA. The three words to which the addition is executed are denoted by 1, 2 and 3. The next more significant word is denoted by 4 and the word which absorbs the carry by 5.

In case of a pipeline conflict with $e = e'$ or $e = e''$ the following addition again touches the words 1, 2 and 3. Now the carry is absorbed either by word 4 or by word 5. Word 4 absorbs the carry if an addition is followed by an addition or a subtraction followed by a subtraction. Word 5 absorbs the carry if an addition is followed by a subtraction or vice versa. So the hardware has to take care that either word 4 or 5 is read into the most significant word of RBS in dependence of the operation which follows. The case that word 5 is the carry word again needs no particular care. Word 5 is already in the most significant position of the RBS. It is simply treated the same way as the words 1, 2 and 3. In the other case word 4 has to be read from the LA into RBS, simultaneously with the words 1, 2 and 3 from the add/subtract unit or from RAS into RBS. In this case word 5 is written into the local memory via the normal write path.

So far certain solutions for the possible pipeline conflicts $e = e'$ and $e = e''$ have been discussed. This are the most frequent but not the only conflicts that may occur. Similar difficulties appear if two or three successively incoming summands overlap only partially. In this case the exponents e and e' and/or e'' differ by 1 or 2 so that also these situations can be detected by comparison of the exponents. Another pipeline conflict appears if one of the two following summands overlaps with a carry word. In these cases summands have to be built up in parts from the adder/subtracter or RAS and the LA. Thus hardware solutions for these situations are more complicated and costly. We leave a detailed study of these situations to the reader/designer and offer the following alternative: The accumulation pipeline consists of three steps only. Instead of investing a lot of hardware logic for rare situations of a pipeline conflict it may be simpler and less expensive to stall the pipeline and delay the accumulation by one or two *cycles* whatever applies. It should be mentioned that other details as for instance the width of the adder that is used also can heavily change the design aspects. A 128 instead of a 64 bit adder width which was assumed here could simplify several details.

It was already mentioned that the probability for the carry to run further than the fourth word is less than 10^{-18}. A particular situation where this happens occurs if the sum changes its sign from a positive to a negative value or vice versa. This can happen frequently. To avoid a complicated carry handling procedure in this case a small carry counter of perhaps three bits could be appended to each 64 bit word of the LA. If these counters are not zero at the end of the accumulation their contents has to be added to the LA. For further details see [Kul86], [Kir88].

As was pointed out in connection with the unit discussed in section 3.2, the addition of the summand actually can be carried out over 170 bits only. Thus the shifter that is shown in Fig. 13 can be reduced to a 106 to 170 bits shifter and the data path from the shifter to the input register IR as well as the one to RBS also need to be 170 bits wide only. If this possible hardware reduction is applied, the summand has to be expanded to the full 256 bits when it is transferred to the adder/subtracter.

5.4 Short Adder with Local Memory on the Arithmetic Unit for 32 Bit Data Word (DSP) (Solution B, preliminary)

Now we consider again a 32 bit data word. We assume that two of these are read simultaneously into the SPU in one read *cycle*. The LA is kept as local memory on the SPU. We assume that the addition of a summand, which now consists of a 48 bit product, can be done by three adders of 64 bits including the carry resolution. Multiplication of the mantissas and addition of the exponents are done in full accordance to the upper part of the circuits shown in Fig. 12. The shift is executed similar to the one in Fig. 12. We shall comment on it later. The appropriately shifted product then reaches the accumulation unit. A block diagram of this unit is shown in Fig. 15.

We assume that the multiplication and the shift can be performed in one read *cycle*. Then, a shifted product reaches the input register IR of the accumulation unit in each (read) *cycle*. The accumulation unit must add and process one summand in each such *cycle*. The addition itself is performed by the following three steps, see Fig. 15.

1. The product which is already in IR touches two successive 64 bit words of the LA. These words are addressed by the exponent e of the product. The contents of these two words of the LA and the word which absorbs the carry are transferred from the LA to the register part r' of RBS. This transfer moves three 64 bit words. The summand in IR is also transferred to the corresponding section of RBS. This part is denoted by e', s' and m' in Fig. 15.
2. In the next step the addition or subtraction is executed in the add/subtract unit according to the sign. The result is transferred to the register RAS. The adder/subtracter consists of three 64 bit adders which are working in carry select mode. So the carries between the lower two of these adders are absorbed by the carry select addition. The carry word is the most significant one. An incrementation/decrementation of this word never produces a carry. Thus the adder/subtracter in Fig. 15 simply can be built as a parallel adder.
3. In the next *cycle* the computed sum is written back into the same three memory cells of the LA to which the addition has been executed. The *write* bus is used for this purpose. Fig. 15. Thus only one address decoding is necessary for the read and write step.

Since a summand is delivered from the multiplier in each *cycle*, all three of these phases must be active simultaneously, i. e. the addition must be performed in a pipeline. This means, in particular, that it must be possible to read from the LA and to write into the LA simultaneously in each *cycle*. Therefore, two different data paths have to be provided, Fig. 15.

The pipeline for the addition consists of three steps. Pipeline conflicts again are very well possible. A pipeline conflict occurs if an incoming summand needs to be added to a partner from the LA which is still being computed and not yet available in the local memory. These situations can be detected by comparing the exponents e, e' and e'' of three successively incoming summands. In principle all pipeline conflicts can be solved by the hardware. We discuss here the solution of two pipeline conflicts which with high probability are the most frequent occurrences.

One conflict situation occurs if two consecutive products carry the same exponent e. In this case the two summands touch the same two words of the LA. Then the second summand is unable to read its partner for the addition from the LA because it is not yet available. This situation is checked by the hardware where the exponent e and e' of two consecutive summands are compared. In case of coincidence the process of reading from the LA is blocked off. Instead the sum which is just being computed is directly written back into the register RBS so that the second summand can immediately be added without memory involvement.

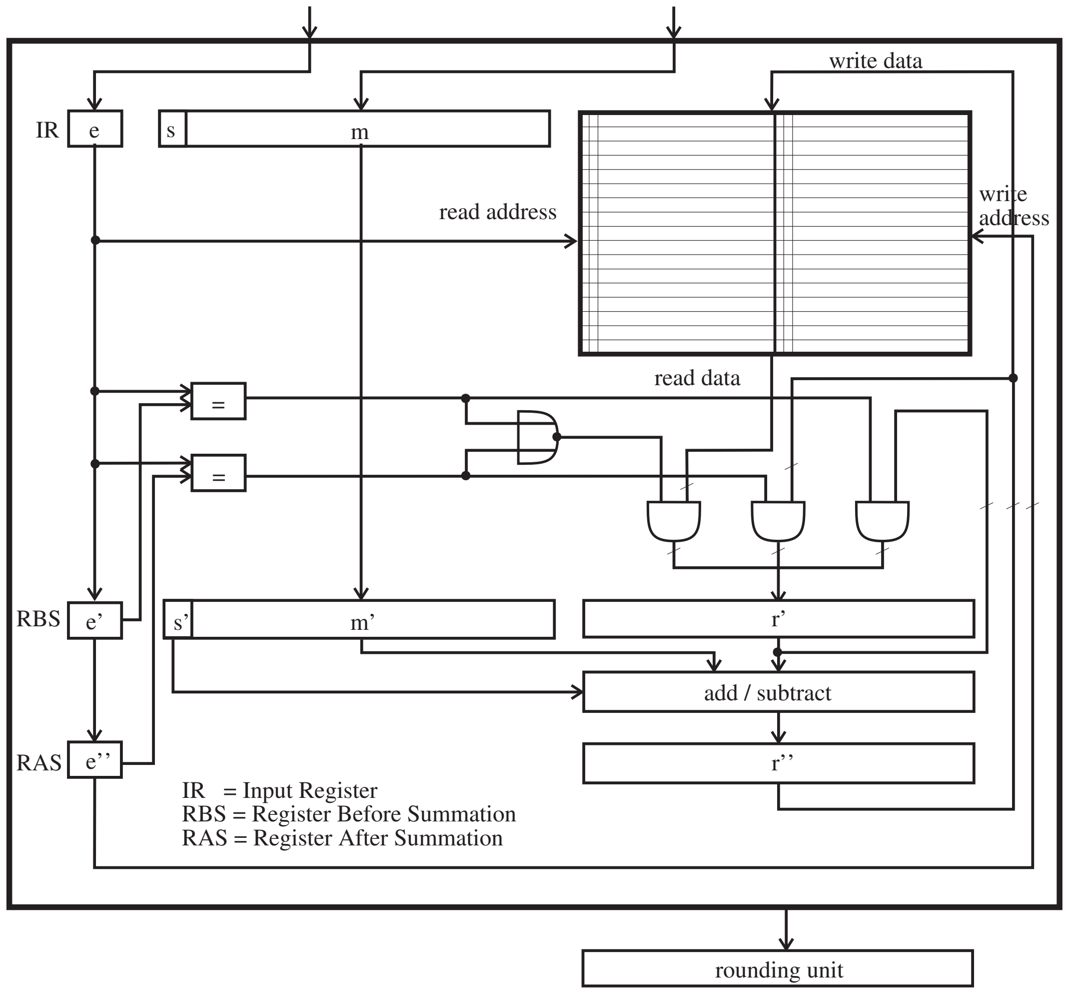


Fig. 15. Block diagram for a SPU with short adder and local store for a 32 bit data word and 64 bit data supply

Another possibility of a pipeline conflict occurs if from three successively incoming summands the first one and the third one carry the same exponent. Since the pipeline consists of three phases the partner for the addition of the third one then is not yet in the LA but still in the register RAS. This situation is checked by the hardware as well, see Fig. 15. There the two exponents e and e'' are compared. In case of coincidence again the process of reading from the LA is blocked off. Instead now, the result of the former addition, which is still in RAS, is directly written back into RBS. Then the addition can be executed without LA involvement.

The case $e = e' = e''$ is also possible. It would cause a conflict in the selection unit which in Fig. 15 is shown directly beneath of the LA. The situation can be avoided by writing a dummy exponent into e'' or by reading from the add/subtract unit with higher priority. This solution is not shown in Fig. 15.

The product that arrives at the accumulation unit touches two consecutive

		4			3	2	1	

Fig. 16. Carry propagation in case of a pipeline conflict

words of the LA. A more significant third word absorbs the possible carry. The solution for the two pipeline conflicts work well, if this third word is the next more significant word of the LA. The probability that this is not the case is less than 10^{-18}. In the vast majority of instances this will be the case.

If it is not the case the word which absorbs the carry is selected by the flag mechanism and read into the most significant word of the RBS. The addition step then works well again including the carry resolution. But difficulties can occur in both cases of a pipeline conflict. Fig. 16 shows a certain part of the LA. The two words to which the addition is executed are denoted by 1 and 2. The next more significant word is denoted by 3 and the word which absorbs the carry by 4.

In case of a pipeline conflict with $e = e'$ or $e = e''$ the following addition again touches the words 1 and 2. Now the carry is absorbed either by the word 3 or by the word 4. Word 3 absorbs the carry if an addition is followed by an addition or a subtraction is followed by a subtraction. Word 4 absorbs the carry if an addition is followed by a subtraction or vice versa. So the hardware has to take care that either word 3 or word 4 is read into the most significant word of RBS in dependence of the operation which follows. The case that the word 4 is the carry word again needs no particular care. Word 4 is already in the most significant position of the RBS. It is simply treated the same way as the words 1 and 2. In the other case word 3 has to be read from the LA into RBS simultaneously with the words 1 and 2 from the add/subtract unit or from RAS into RBS. In this case word 4 is written into the local memory via the normal write path.

So far solutions for the two pipeline conflicts $e = e'$ and $e = e''$ have been discussed. This are not the only conflicts that may occur. Similar difficulties appear if two or three successively incoming summands overlap only partially. In this case the exponents e and e' and/or e'' differ by 1 so that these situations can be detected by comparison of the exponents also. Another pipeline conflict appears if one of the following two summands overlaps with a carry word. In these cases summands have to be built up in parts from the adder/subtracter or RAS and the LA. Thus hardware solutions for these situations are more complicated and costly. We leave a detailed study of these situations to the reader/designer and offer the following alternative. The accumulation pipeline consists of three steps only. Instead of investing a lot of hardware logic for very rare situations of a pipeline conflict it may be simpler and less expensive to stall the pipeline and delay the accumulation by one or two *cycles* whatever applies.

The product consists of 48 bits. So the summand never touches the 16 least significant bits of word 1. The most significant third 64 bit word of the adder is supposed to absorb the carry. It can be built as an incrementer/decrementer by halfadders. Thus, in comparison with Fig. 12, the shifter can be reduced to a 48 to 112 bit shifter and the data path from the shifter to the input register IR as well as the one to RBS also needs to be 112 bits wide only. If this possibility is chosen, the summand has to be expanded to the full 192 bits when it is read into the adder/subtracter.

The circuits that have been discussed so far are based on the assumption that the LA is organized in words of 64 bits and that the width of the partial adder that is used also consists of 64 bits. It should be mentioned that these assumptions, although realistic, are nevertheless somehow arbitrary and that other choices are very well possible and may lead to simpler or better solutions. The LA could as well be organized in words of 128 or only 32 bits. The width of the partial adder could also be 128 or 32 bits. All these possibilities allow interesting solutions for the different cases that have been discussed in this paper. We leave it to the reader to play with these combinations and select the one which fits best to a given hardware environment. With increasing word size the probability for a pipeline conflict decreases.

6 Hardware Accumulation Window

So far it was assumed in this paper that the SPU is newly designed and incorporated as an integral part into the arithmetic unit of the processor. Now we discuss the question what can be done if not enough register space for the LA is available on the processor.

The final result of a scalar product computation is assumed to be a floating-point number with an exponent in the range $e1 \leq e \leq e2$. If this is not the case, the problem has to be scaled. During the computation of the scalar product, however, summands with an exponent outside of this range may occur. The remaining computation then has to cancel all the digits outside of the range $e1 \leq e \leq e2$. So in a normal scalar product computation, the register space outside this range will be used less frequently. It was already mentioned earlier in this paper that the conclusion should not be drawn from this consideration that the register size can be restricted to the single exponent range in order to save some silicon area. This would require the instalment of complicated exception handling routines in software or in hardware. The latter may finally require as much silicon. A software solution certainly is much slower. The hardware requirement for the LA in case of standard arithmetics are modest and the necessary register space really should be invested.

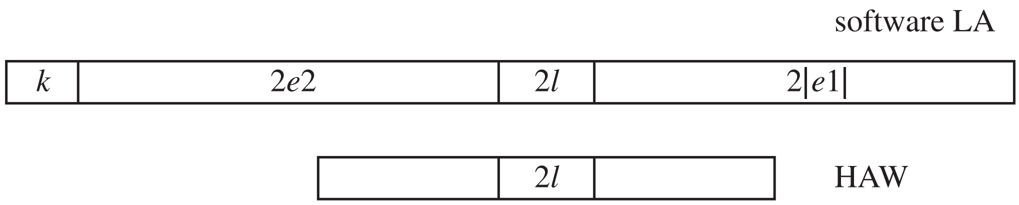


Fig. 17. Hardware Accumulation Window (HAW)

However, the memory space for the LA on the arithmetic unit grows with the exponent range of the data format. If this range is extremely large, as for instance in case of an extended precision floating-point format, then only an inner part of the LA can be supported by hardware. We call this part of the LA a Hardware Accumulation Window (HAW). See Fig. 17. The outer parts of this window then must be handled in software. They probably appear less often.

There are still other reasons that recommend the development of techniques for the computation of the accurate scalar product using a HAW. Many conventional computers on the market do not provide enough register space to represent the full LA on the CPU. Then a HAW is one choice which allows a fast and correct computation of the scalar product in many cases.

Another possibility is to place the LA in the user memory, i. e. in the data cache. In this case only the start address of the LA and the flag bits are put into (fixed) registers of the general purpose register set of the computer. This solution has the advantage that only a few registers are needed and that a longer accumulator window or even the full LA can be provided. This reduces the necessity to handle exceptions. The disadvantage of this solution is that for each accumulation step, four memory words must be read and written in addition to the two operand loads. So the scalar product computation speed is limited by the data cache to processor transfer bandwidth and speed. If the full long accumulator is provided this is a very natural solution. It has been realized on several IBM, SIEMENS and HITACHI computers of the /370 architecture in the 1980ies [IBM84,IBM86,IBM90,SIE86].

A faster solution certainly is obtained for many applications with a HAW in the general purpose register set of the processor. Here only a part of the LA is present in hardware. Overflows and underflows of this window have to be handled by software. A full LA for the data format double precision of the IEEE-arithmetic standard 754 requires 4288 bits or 67 words of 64 bits. We assume here that only 10 of these words are located in the general purpose register set.

Such a window covers the full LA that is needed for a scalar product computation in case of the data format single precision of the IEEE-arithmetic standard 754. It also allows a correct computation of scalar products in case of the data format long of the /370 architecture as long as no under- or overflows occur. In this case $64 + 28 + 63 = 155$ hexadecimal digits or 620 bits are required. With a HAW of 640 bits all scalar products that do not cause an under- or overflow could have been correctly computed on these machines. This architecture was successfully used and even somehow dominated the market for more than 20 years. This example shows that even if only a HAW of 640 bits is available, the vast majority of scalar products will execute on fast hardware.

Of course, also in case that only a HAW is available, all scalar products should be computed correctly. Any operation that over- or underflows the HAW must be completed in software. This requires a complete software implementation of the LA, i. e. a variable of type dotprecision. All additions that do not fit into the HAW must be executed in software into this dotprecision variable.

There are three situations where the HAW can not correctly accumulate the product:

- the exponent of the product is so large that the product does not (completely) fit into the HAW. Then the product is added in software to the dotprecision variable.
- the exponent of the product is so small that the product does not (completely) fit into the HAW. Then the product is added in software to the dotprecision variable.
- the product fits into the HAW. But its accumulation causes a carry to be propagated outside the range of the HAW. In this case the product is added into the HAW. The carry must be added in software to the dotprecision variable.

If at the end of the accumulation the contents of the software accumulator are non zero, the contents of the HAW must be added to the software accumulator to obtain the correct value of the scalar product. Then a rounding can be performed if required. If at the end of the accumulation the contents of the software accumulator are zero, the HAW contains the correct value of the scalar product and a rounded value can be obtained from it.

Thus, in general, a software controlled full LA supplements a HAW. The software routines must be able to perform the following functions:

- clear the software LA. This routine must be called during the initialization of the HAW. Ideally, this routine only sets a flag. The actual clearing is only done if the software LA is needed.
- add or subtract a product to/from the software LA.
- add or subtract a carry or borrow to/from the software LA at the appropriate digit position.
- add the HAW to the software LA. This is required to produce the final result

when both the HAW and the software LA were used. Then a rounding can be performed.

– round the software LA to a floating-point number.

With this software support scalar products can be computed correctly using a HAW at the cost of a remarkable software overhead and a considerable time penalty for products that fall outside the range of the HAW.

An alternative to the HAW-software environment just described is to discard the products that underflow the HAW. A counter variable is used to count the number of discarded products. If a number of products were discarded, the last bits of the HAW must be considered invalid. A valid rounded result can be generated by hardware if these bits are not needed. If this procedure fails to produce a useful answer the whole accumulation is repeated in software using a full LA.

A 640 bit HAW seems to be the shortest reasonable hardware window. If this much register space is not available, a software implementation probably is the best solution.

If a shorter HAW must be implemented, then it should be a movable window. This can be represented by an exponent register associated to the hardware window. At the beginning of an accumulation, the exponent register is set so that the window covers the least significant portion of the LA. Whenever a product would cause the window to overflow, its exponent tag is adjusted, i. e. the window moves to the left, so that the product fits into the window. Products that would cause an underflow are counted and otherwise ignored. The rounding instruction checks whether enough significant digits are left to produce a correctly rounded result or whether too much cancellation did occur. In the latter case it is up to the user to accept the inexact result or to repeat the whole accumulation in software using a full LA.

Using this technique a HAW as short as 256 bits could be used to perform rounded scalar product computation and QUAD arithmetic. However, it would not be possible to perform many other nice and useful applications of the optimal scalar product with this type of scalar product hardware as for instance a *long real arithmetic*.

The software overhead caused by the reduction of the full width of the LA to a HAW represents a trade off between hardware expenditure and runtime.

With the accurate scalar product operators for multiple precision arithmetic including multiple precision interval arithmetic can easily be provided. This enables the user to use higher precision operations in numerically critical parts of a computation. Experience shows that if one runs out of precision in a certain problem class one often runs out of *double* or *extended* precision very soon as well. It is preferable and simpler, therefore, to provide the principles for enlarging the precision than any fixed higher precision. To allow fast execution of a number of multiple precision arithmetics the HAW should not be too small.

7 Theoretical Foundation of Advanced Computer Arithmetic and Shortcomings of Existing Processors and Standards

Arithmetic is the basis of mathematics. Advanced computer arithmetic expands the arithmetic and mathematical capability of the digital computer in the most natural way. Instead of reducing all calculations to the four elementary operations for floating-point numbers, advanced computer arithmetic provides twelve fundamental data types or mathematical spaces with operations of highest accuracy in a computing environment.

Besides of the real numbers, the complex numbers form the basis of analysis. For computations with guarantees one needs the intervals over the real and complex numbers as well. The intervals bring the continuum on the computer. An interval between two floating-point bounds represents the continuous set of real numbers between these two floating-point bounds.

The twelve fundamental data types or mathematical spaces consist of the four basic data types real, complex, interval and complex interval as well as the vectors and matrices over these types. Arithmetic operations in the computer representable subsets of these spaces are defined by a general mapping principle which is called a semimorphism. These arithmetic operations are distinctly different from the customary ones in these spaces which are based on elementary floating-point arithmetic.

If M is any one of these twelve data types (or mathematical spaces) and N is its computer representable subset, then for every arithmetic operation $\circ$ in M, a corresponding computer operation $\boxdot$ in N is defined by

$$\text{(RG)} \quad a \boxdot b := \square(a \circ b) \qquad \text{for all } a, b \in N \text{ and all operations } \circ \text{ in } M,$$

where $\square : M \to N$ is a mapping from M onto N which is called a rounding if it has the following properties:

$$\text{(R1)} \quad \square a = a \qquad \text{for all } a \in N \qquad \text{(projection)}$$

$$\text{(R2)} \quad a \leq b \Rightarrow \square a \leq \square b \qquad \text{for } a, b \in M \qquad \text{(monotonicity)}$$

The concept of semimorphism requires additionally that the rounding is antisymmetric, i. e. that it has the property

$$\text{(R3)} \quad \square(-a) = -\square(a) \qquad \text{for all } a \in M \qquad \text{(antisymmetry)}$$

For the interval spaces among the twelve basic data types — the intervals over the real and complex numbers as well as the intervals over the real and complex vectors and matrices — the order relation in (R2) is the subset relation $\subseteq$. A rounding from any interval set M onto its computer representable subset N is defined by properties (R1), (R2) (with $\leq$ replaced by $\subseteq$), plus the additional property

$$\text{(R4)} \quad a \subseteq \square a \qquad \text{for all } a \in M \qquad \text{(inclusion)}$$

These interval roundings are also antisymmetric, that is, they satisfy property (R3). [Kul76,Kul81].

Additional important roundings from the real numbers onto the floating-point numbers are the montone downwardly and upwardly directed roundings with the property

$$\text{(R4)} \quad \triangledown a \leq a \text{ resp. } a \leq \triangle a \qquad \text{for all } a \in M \qquad \text{(directed)}$$

These directed roundings are uniquely defined by (R1), (R2) and (R4), see [Kul76,Kul81]. Arithmetic operations are also defined by (RG) with the roundings $\triangledown$ and $\triangle$.

With the five rules (RG) and (R1, 2, 3, 4), a large number of arithmetic operations is defined in the computer representable subsets of the twelve fundamental data types or mathematical spaces. (RG) means that every computer operation should be performed in such a way that it produces the same result as if the mathematically correct operation were first performed in the basic space M and the exact result then rounded into the computer representable subset N. In contrast to the traditional approximation of the arithmetic operations in the product spaces by floating-point arithmetic, all operations with the properties (RG), (R1) and (R2) are optimal in the sense that there is no better computer representable approximation to the true result (with respect to the prescribed rounding). I. e. between the correct and the computed result of an operation there is no other element of the corresponding computer representable subset. This can easily be seen: Let $a, b \in N$, and $\alpha \in N$ the greatest lower and $\beta \in N$ the least upper bound of the correct result $a \circ b$ in M, i. e.

$$\alpha \leq a \circ b \leq \beta,$$

then

$$\Box\alpha \underset{(R1)}{=} \alpha \underset{(R2)}{\leq} \Box(a \circ b) \underset{(RG)}{=} a \boxdot b \underset{(R2)}{\leq} \Box\beta \underset{(R1)}{=} \beta \qquad (\#)$$

Thus, all semimorphic computer operations are of 1 ulp (unit in the last place) accuracy. 1/2 ulp accuracy is achieved in the case of rounding to nearest. In the product spaces the order relation is defined componentwise. So in the product spaces property (#) holds for every component.

Fig. 18 shows a table of the twelve basic arithmetic data types and corresponding operators as they are provided by the programming language PASCAL-XSC [Kla91,Kla93]. All data types and operators are predefined available in the language. The operations can be called by the operator symbols shown in the table. An arithmetic operator followed by a less or greater symbol denotes an operation with rounding downwards or upwards, respectively. The operator $+*$ takes the interval hull of two elements, $**$ means intersection. Also all outer operations that occur in Fig. 18 (scalar times vector, matrix times vector, etc.) are defined by the five properties (RG), (R1, 2, 3, 4), whatever applies. A count of all inner and outer predefined operations in the figure leads to a number of about 600 arithmetic operations.

Fig. 19 lists the same data types in their usual mathematical notation. There $\mathbb{R}$ denotes the real and $\mathbb{C}$ the complex numbers. A heading letter V, M and I denotes vectors, matrices and intervals, respectively. R stands for the set of floating-point numbers and D for any set of higher precision floating-point numbers. If M is any set, $\mathbb{P}M$ denotes the power set, which is the set of all subsets of M. For any operation $\circ$ in M a corresponding operation $\circ$ in $\mathbb{P}M$ is defined by $A \circ B := \{a \circ b \mid a \in A \wedge b \in B\}$ for all $A, B \in \mathbb{P}M$.

For each set-subset pair in Fig. 19, arithmetic in the subset is defined by semimorphism. These operations are different in general from those which are performed in the product spaces if only elementary floating-point arithmetic is furnished on the computer. Semimorphism defines operations in a subset N of a set M directly by making use of the operations in M. It makes a direct link between an operation in M and its approximation in the subset N. For instance, the operations in MCR (see Fig. 19) are directly defined by the operations in $M\mathbb{C}$, and not in a round about way via $\mathbb{C}$, $\mathbb{R}$, R, CR, and MCR as it would have to be done by using elementary floating-point arithmetic only.

The properties of a semimorphism can be derived as necessary conditions for an homorphism between ordered algebraic structures [Kul76,Kul81]. It is easy to see that repetition of semimorphism is again a semimorphism. A careful analysis of the requirements of semimorphism is given in [Kul76,Kul81]. The resulting algebraic and order structure are studied there under the mapping pro-

right op. / left operand	integer real complex	interval cinterval	rvector cvector	ivector civector	rmatrix cmatrix	imatrix cimatrix
monadic	+, −	+, −	+, −	+, −	+, −	+, −
integer real complex	+, +<, +> −, −<, −> *, *<, *> /, /<, /> +*	+, −, *, / +*	*, *<, *>	*	*, *<, *>	*
interval cinterval	+, −, *, / +*	+, −, *, / +*, **	*	*	*	*
rvector cvector	*, *<, *> /, /<, />	*, /	+, +<, +> −, −<, −> *, *<, *> +*	+, −, * +*		
ivector civector	*, /	*, /	+, −, * +*	+, −, * +*, **		
rmatrix cmatrix	*, *<, *> /, /<, />	*, /	*, *<, *>	*	+, +<, +> −, −<, −> *, *<, *> +*	+, −, * +*
imatrix cimatrix	*, /	*, /	*	*	+, −, * +*	+, −, * +*, **

Fig. 18. Predefined arithmetic data types and operators of PASCAL-XSC.

perties (RG) and (R1, 2, 3, 4). Many properties of both the order structure and the algebraic structure are invariant under a semimorphism. Because of (R2) with respect to $\leq$ or $\subseteq$ the order structure is not changed if we move from a set into a subset in any row of Fig. 19, while the algebraic structure is considerably weakened. The concept of semimorphism and its explicit five rules (RG), (R1, 2, 3, 4) are used as an axiomatic definition of computer arithmetic in the XSC-languages [Kla91,Kla92,Kla93,Kla93a,Kul87,Kul87a,IAM90,IBM90].

In the theory of computer arithmetic it is ultimately shown, that all arithmetic operations of the twelve fundamental numerical data types of Fig. 18 or spaces of Fig. 19 can be provided in a higher programming language by a modular technique, if on a low level, preferably in hardware, 15 fundamental operations are available: the five operations $+, -, \times, /, \cdot$, each one with the three roundings $\square, \bigtriangledown, \bigtriangleup$. Here $\cdot$ means the scalar product of two vectors, $\square$ is a monotone, antisymmetric rounding, e. g. rounding to nearest, and $\bigtriangledown$ and

$$
\begin{array}{rrrr}
 & \mathbb{R} \supset & D \supset & R \\
 & V\mathbb{R} \supset & VD \supset & VR \\
 & M\mathbb{R} \supset & MD \supset & MR \\
 & & & \\
\mathbb{PR} \supset & I\mathbb{R} \supset & ID \supset & IR \\
\mathbb{P}V\mathbb{R} \supset & IV\mathbb{R} \supset & IVD \supset & IVR \\
\mathbb{P}M\mathbb{R} \supset & IM\mathbb{R} \supset & IMD \supset & IMR \\
 & & & \\
 & \mathbb{C} \supset & CD \supset & CR \\
 & V\mathbb{C} \supset & VCD \supset & VCR \\
 & M\mathbb{C} \supset & MCD \supset & MCR \\
 & & & \\
\mathbb{PC} \supset & I\mathbb{C} \supset & ICD \supset & ICR \\
\mathbb{P}V\mathbb{C} \supset & IV\mathbb{C} \supset & IVCD \supset & IVCR \\
\mathbb{P}M\mathbb{C} \supset & IM\mathbb{C} \supset & IMCD \supset & IMCR
\end{array}
$$

Fig. 19. Table of the spaces occurring in numerical computations.

$$
\begin{array}{cccccccl}
\boxplus & \boxminus & \boxtimes & \boxslash & \square & & a \boxdot b & = \square \sum_{i=1}^{n} a_i \times b_i \\
\bigtriangledown\!\!\!\!\!\!{\scriptstyle +} & \bigtriangledown\!\!\!\!\!\!{\scriptstyle -} & \bigtriangledown\!\!\!\!\!\!{\scriptstyle \times} & \bigtriangledown\!\!\!\!\!\!{\scriptstyle /} & \bigtriangledown & & a \bigtriangledown\!\!\!\!\!\!{\scriptstyle \cdot}\; b & = \bigtriangledown \sum_{i=1}^{n} a_i \times b_i \\
\bigtriangleup\!\!\!\!\!\!{\scriptstyle +} & \bigtriangleup\!\!\!\!\!\!{\scriptstyle -} & \bigtriangleup\!\!\!\!\!\!{\scriptstyle \times} & \bigtriangleup\!\!\!\!\!\!{\scriptstyle /} & \bigtriangleup & & a \bigtriangleup\!\!\!\!\!\!{\scriptstyle \cdot}\; b & = \bigtriangleup \sum_{i=1}^{n} a_i \times b_i
\end{array}
$$

Fig. 20. The fifteen fundamental operations for advanced computer arithmetic.

$\bigtriangleup$ are the monotone downwardly and upwardly directed roundings from the real numbers into the floating-point numbers. All 15 operations $\boxdot, \bigtriangledown\!\!\!\!\!\!{\scriptstyle \circ}\;, \bigtriangleup\!\!\!\!\!\!{\scriptstyle \circ}\;$, with $\circ \in \{+, -, \times, /, \cdot\}$, Fig. 20, are defined by (RG). In case of the scalar product, a and b are vectors $a = (a_i)$, $b = (b_i)$ with any finite number of components.

The IEEE arithmetic standards 754 and 854 offer 12 of these operations: $\boxdot, \bigtriangledown\!\!\!\!\!\!{\scriptstyle \circ}\;, \bigtriangleup\!\!\!\!\!\!{\scriptstyle \circ}\;$, with $\circ \in \{+, -, \times, /\}$. These standards also prescribe specific data formats. A general theory of computer arithmetic is not bound to these data formats. By adding just three more operations, the optimal scalar products $\boxdot, \bigtriangledown\!\!\!\!\!\!{\scriptstyle \cdot}\;, \bigtriangleup\!\!\!\!\!\!{\scriptstyle \cdot}\;$, all operations in the usual product spaces of numerical mathematics can be performed with 1 or 1/2 ulp accuracy in each component.

Remark 1: With this information it seems to be relatively easy to provide advanced computer arithmetic on processors which offer the IEEE arithmetic standard 754. The standard seems to be a step in the right direction. All that is additionally needed are the three optimal scalar products $\boxdot$, $\triangledown$ and $\vartriangle$. If they are not supported by the computer hardware they could be simulated. One possibility to simulate these operations certainly would be to place the LA into the user memory, i. e. in the data cache. This possibility was already mentioned in section 6.

However, a closer look into the subject reveals severe difficulties and disadvantages which result in unnecessary performance penalties. So that at a place where an increase in speed is to be expected, a severe loss of speed results into the bargain.

A first severe drawback comes from the fact that IEEE arithmetic separates the rounding from the operation. First the rounding mode has to be set. Then an arithmetic operation can be performed. In a conventional floating-point computation this does not cause any difficulties. The rounding mode is set only once. Then a large number of arithmetic operations is performed with this rounding mode. However, when interval arithmetic is performed, the rounding mode has to be switched very frequently. In the computer the lower bound of the result of every interval operation has to be rounded downwards and the upper bound rounded upwards. Thus the rounding mode has to be set for every arithmetic operation. If setting the rounding mode and the arithmetic operation are equally fast, this slows down interval arithmetic unnecessarily by a factor of two in comparison to a conventional floating-point arithmetic. On the Pentium processor setting the rounding mode takes three cycles, the following operation only one!! Thus an interval operation is 8 times slower than the corresponding floating-point operation. The rounding should be part of the arithmetic operation as required by the postulate (RG) of the axiomatic definition of (advanced) computer arithmetic. Every one of the rounded operations $\boxdot, \triangledown\!\!\!\!\!\circ\,, \vartriangle\!\!\!\!\!\circ\,$, $\circ \in \{+, -, \times, /\}$ should be executed in a single cycle! The rounding must be an integral part of the operation.

A second severe drawback comes from the fact that all the commercial processors that perform IEEE-arithmetic in case of multiplication only deliver a rounded product to the outside world. Computation of an accurate scalar product requires products of the full double length. These products have to be simulated from outside on the processor. This slows down the multiplication by a factor of 10 in comparison to a rounded hardware multiplication. In a software simulation of the accurate scalar product the products of double length then have to be accumulated into the LA. This process is again slower by a factor of 5 in comparison to a (possibly wrong) hardware accumulation of products in floating-point arithmetic. Thus in summary a factor of at least 50 for the runtime is the trade-off for an accurate computation of the scalar product on existing processors. This is too much to be easily accepted by the user. Again at a place where an increase in speed by a factor of at least two is to be expected if the scalar product is supported by hardware, a severe loss of speed is obtained by processors which have not been designed for the purpose they are needed for.

A third severe drawback is the fact that no reasonable interface to the programming languages is required by existing computer arithmetic standards. The majority of operations shown in Fig. 18 can be provided in a programming language which allows operator overloading. Operator overloading, however, is not enough to call the twelve operations $\boxdot, \bigtriangledown\!\!\!\!\!\!\circ\,, \bigtriangleup\!\!\!\!\!\!\circ\,, \circ \in \{+, -, \times, /\}$ which are provided by all IEEE-arithmetic processors in a higher programming language. A general operator concept is necessary for ease of programming (three real operations for $+, -, \times, /$). This solution has been chosen in PASCAL-XSC. In C-XSC which has been developed as a C++ class library, the 8 operators $\bigtriangledown\!\!\!\!\!\!\circ$ and $\bigtriangleup\!\!\!\!\!\!\circ\,, \circ \in \{+, -, \times, /\}$ are hidden in the interval operations and not openly available. This is necessary because C++ does not allow three different operators for addition, subtraction, multiplication and division for the data type real.

Computer arithmetic is an integral part of all programming languages. The quality of the arithmetic operations should be an integral part of the definition of all programming languages. This can easily be done. All operations that are shown in Fig. 18 can be defined by the five simple rules (RG) and (R1, 2, 3, 4). In particular the eight operations $\bigtriangledown\!\!\!\!\!\!\circ$ and $\bigtriangleup\!\!\!\!\!\!\circ\,$, with $\circ \in \{+, -, \times, /\}$ are defined by (RG), (R1), (R2) and (R4). All interval operations are defined by (RG), (R1), (R2), (R3) and (R4). All other operations that appear in Fig. 18 can be defined by (RG), (R1), (R2) and (R3) with the additional information whether rounding to nearest, towards infinity or towards zero is required. A precise definition of advanced computer arithmetic thus turns out to be short and simple.

IEEE-arithmetic has been developed as a standard for microprocessors in the early eighties at a time when the microprocessor was the 8086. Since that time the speed of microprocessors has been increased by several magnitudes. IEEE-arithmetic is now even provided and used by super computers, the speed of which again is faster by several magnitudes. All this is no longer in balance. With respect to arithmetic many manufacturers believe that realization of the IEEE-arithmetic standard is all that is necessary to do. In this way the existing standards prove to be a great hindrance to further progress. Advances in computer technology are now so profound that the arithmetic capability and repertoire of computers should be expanded to prepare the digital computer for the computations of the next century. The provision of Advanced Computer Arithmetic is the most natural way to do this.

Remark 2: A vector arithmetic coprocessor chip for the PC has been developed in a CMOS VLSI gate array technology at the authors institute in

1993/94. It is connected with the PC via the PCI-bus. The PCI- and EMC-interface are integrated on chip. At its time the chip computed the accurate scalar product between two and four times faster than the PC an approximation in floating-point arithmetic. With increasing clock rate of the PC the PCI-bus turned out to be a severe bottle neck. To keep up with the increased speed the SPU must be integrated into the arithmetic logical unit of the processor and interconnected by an internal bus system.

The chip, see Fig. 21, realizes the SPU that has been discussed in section 3.1, 207,000 transistors are needed. About 30 % of the transistors and the silicon area are used for the local memory and the flag registers with the carry resolution logic. The remaining about 70 % of the silicon area is needed for the PCI/EMC-interface and the chips own multiplier, shifter, adder and rounding unit. All these units would be superfluous if the SPU would be integrated into the arithmetic unit of the processor. A multiplier, shifter, adder and rounding unit are already there. Everything just needs to be arranged a little differently and adjusted to each other. Thus finally the SPU requires less transistors and silicon area than is needed for the exception handling of the IEEE-arithmetic standard. Logically the SPU is much more regular and simpler. With it a large number of exceptions that can occur in a conventional floating-point computation are avoided.

Modern computer technology allows to spend millions of transistors on a single chip. This allows to put solutions into the computer hardware which even an experienced computer user is totally unaware of. Due to the insufficient knowledge and familiarity with the technology, the design tools and implementation techniques, obvious and easy solutions are not required by mathematicians emphatically enough. The engineer on the other side, who is familiar with these techniques is not aware of the consequences for mathematics.

Remark 3: In addition to the numerical data types and operators displayed in Fig. 18 the XSC-languages provide an array type *staggered (staggered precision)* [Ste84,Ste89] for multiple precision data. A variable of type *staggered* consists of an array of variables of the type of its components. Components of the *staggered* type can be of type *real* or of type *interval*. The value of a variable of type *staggered* is the sum of its components. Addition and subtraction of such multiple precision data can easily be performed in the LA. Multiplication of two variables of this type can be computed easily and fast by the accurate scalar product. Division is performed iteratively. The multiple precision data type *staggered* is controlled by a global variable called *stagprec*. If *stagprec* is 1, the *staggered* type is identical to its component type. If, for instance, *stagprec* is 4 each variable of this type consists of an array of four variables of its component type. Again its value is the sum of its components. The global variable *stagprec* can be increased or decreased at any place in a program. This enables the user to use higher precision data and operations in numerically critical parts of his computation. It helps to increase software reliability. The elementary functions for the type *staggered* are also available in the XSC-languages for the component types *real* and *interval* [Bra87,Kra87]. In the case that *stagprec* is 2, a data type is encountered which occasionally is denoted as double-double or quadruple precision.

Acknowledgements

This text summarizes both an extensive research activity that went on during the past twenty years and the experience gained through various implementations of the entire arithmetic package on diverse processors. The text is also based on lectures held at the Universität Karlsruhe during the preceding 25 years. While the collection of research articles that contribute to this paper is not too large in number, I refrain from a detailed review of them and refer to the list of references. This text synthesizes and organizes diverse contributions into a coherent setting. In many cases more detailed information can be obtained from original doctoral theses.

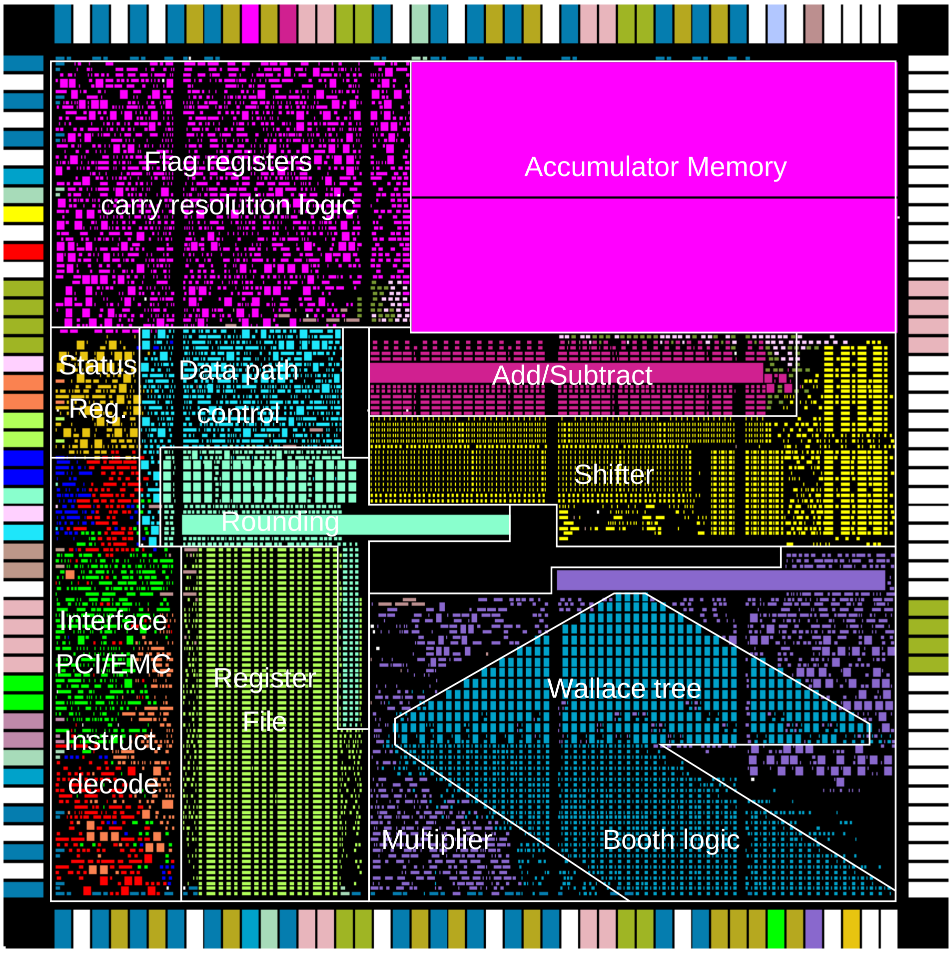
I am grateful to all those colleagues and co-workers who have contributed through their research to the development of advanced computer arithmetic as it is presented in this paper. In particular I would like to mention and thank Gerd Bohlender, Willard L. Miranker, Reinhard Kirchner, Siegfried M. Rump, Thomas Teufel, Harald Böhm, Andreas Knöfel and Christoph Baumhof.

Finally I wish to thank the Electrotechnical Laboratory, Agency of Industrial Science and Technology at Tsukuba, Japan for providing me the opportunity to write this article in a pleasant scientific environment without constantly being interrupted by the usual University business. I especially owe thanks to Satoshi Sekiguchi for being a wonderful host personally and scientifically. I am looking forward and eagerly await advanced arithmetic on commercial computers.

References

[Ada93] Adams, E.; Kulisch, U.(eds.): **Scientific Computing with Automatic Result Verification.** I. Language and Programming Support for Verified Scientific Computation, II. Enclosure Methods and Algorithms with Automatic Result Verification, III. Applications in the Engineering Sciences. Academic Press, San Diego, 1993 (ISBN 0-12-044210-8).

[Alb77] Albrecht, R.; Kulisch, U. (Eds.): **Grundlagen der Computerarithmetik.** Computing Supplementum



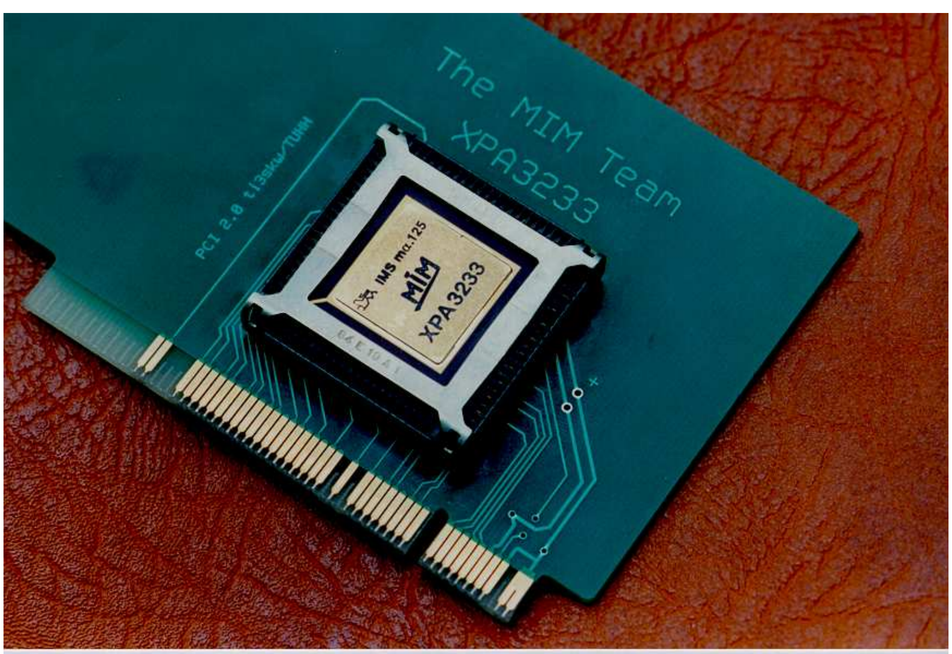


Fig. 21. Functional units, chip and board of the vector arithmetic coprocessor XPA 3233

1. Springer-Verlag, Wien / New York, 1977.

[Alb93] Albrecht, R.; Alefeld, G.; Stetter, H.J. (Eds.): **Validation Numerics – Theory and Applications.** Computing Supplementum **9**, Springer-Verlag, Wien / New York, 1993.

[Ale74] Alefeld, G.; Herzberger, J.: **Einführung in die Intervallrechnung.** Bibliographisches Institut (Reihe Informatik, Nr. 12), Mannheim / Wien / Zürich, 1974 (ISBN 3-411-01466-0).

[Ale83] Alefeld, G.; Herzberger, J.: **An Introduction to Interval Computations.** Academic Press, New York, 1983 (ISBN 0-12-049820-0).

[Apo68] Apostolatos, N.; Kulisch, U.; Krawczyk, R.; Lortz, B.; Nickel, K.; Wippermann, H.-W.: *The Algorithmic Language Triplex-ALGOL 60.* Numerische Mathematik **11**, pp. 175-180, 1968.

[Bau92] Baumhof, Ch.: *Behavioural Description of A Scalar Product Unit.* Universität Karlsruhe, ESPRIT Project OMI/HORN, Deliverable Report D1.2/2, Dez. 1992.

[Bau95] Baumhof, Ch.: *A New VLSI Vector Arithmetic Coprocessor for the PC.* In [ARITH, Vol. 12, pp. 210-215], 1995.

[Bau96] Baumhof, Ch.: *Ein Vektorarithmetik-Koprozessor in VLSI-Technik zur Unterstützung des Wissenschaftlichen Rechnens.* Dissertation, Universität Karlsruhe, 1996.

[Bau96a] Baumhof, Ch.; Bohlender, G.: *A VLSI Vector Arithmetic Coprocessor for the PC.* Proceedings of WAI'96 in Recife/Brasil, RITA (Revista de Informática Teórica e Aplicada), Extra Edition, October 1996.

[Ble87] Bleher, J. H.; Kulisch, U.; Metzger, M.; Rump, S. M.; Ullrich, Ch.; Walter, W.: *FORTRAN–SC: A Study of a FORTRAN Extension for Engineering/Scientific Computation with Access to ACRITH.* Computing **39**, pp. 93-110, Nov. 1987.

[Boh77] Bohlender, G.: *Floating-Point Computation of Functions with Maximum Accuracy.* IEEE Transactions on Computers, Vol. C-26, no. 7, July 1977.

[Boh78] Bohlender, G.: *Genaue Berechnung mehrfacher Summen, Produkte und Wurzeln von Gleitkommazahlen und allgemeine Arithmetik in höheren Programmiersprachen.* Dissertation, Universität Karlsruhe, 1978.

[Boh81] Bohlender, G.; Grüner, K.; Kaucher, E.; Klatte, R.; Krämer, W.; Kulisch, U.; Miranker, W. L.; Rump, S. M.; Ullrich, Ch.; Wolff v. Gudenberg, J.: *PASCAL–SC: A PASCAL for Contemporary Scientific Computation.* IBM Research Report RC 9009 (#39456) 8/25/81, 79 pages, 1981.

[Boh81a] Bohlender, G.; Kaucher, E.; Klatte, R.; Kulisch, U.; Miranker, W. L.; Ullrich, Ch.; Wolff v. Gudenberg, J.: *FORTRAN for Contemporary Numerical Computation.* IBM Research Report RC 8348. Computing **26**, pp. 277-314, 1981.

[Boh90] Bohlender, G.: *What Do We Need Beyond IEEE Arithmetic?* In [Ull90, pp. 1-32], 1990.

[Boh98] Bohlender, G.: *Literature List on Enclosure Methods and Related Topics* Institut für Angewandte Mathematik, Universität Karlsruhe, Report, 1998.

[Boe83] Böhm, H.: *Berechnung von Polynomnullstellen und Auswertung arithmetischer Ausdrücke mit garantierter maximaler Genauigkeit.* Dissertation, Universität Karlsruhe, 1983.

[Bra87] Braune, K: *Hochgenaue Standardfunktionen für reelle und komplexe Punkte und Intervalle in beliebigen Gleitpunktrastern.* Dissertation, Universität Karlsruhe, 1987.

[Cap88] Cappello, P. R.; Miranker, W. L.: *Systolic Super Summation.* IEEE Transactions on Computers **37** (6), pp. 657-677, June 1988.

[Cap88a] Cappello, P. R.; Miranker, W. L.: *Systolic Super Summation with Reduced Hardware.* IBM Research Report RC 14259 (#63831), IBM Research Division, Yorktown Heights, New York, Nov. 30, 1988.

[Erb92] Erb, H.: *Ein Gleitpunkt-Arithmetikprozessor mit mehrfacher Präzision zur verifizierten Lösung linearer Gleichungssysteme.* Dissertation, Fakultät für Informatik, Universität Karlsruhe, 1992.

[Ham87] Hamada, H.: *A New Real Number Representation and its Operation.* In [ARITH, Vol. 8, pp. 153-157], 1987.

[Ham92] Hammer, R.: *Maximal genaue Berechnung von Skalarproduktausdrücken und hochgenaue Auswertung von Programmteilen.* Dissertation, Universität Karlsruhe, 1992.

[Ham93] Hammer, R.; Hocks, M.; Kulisch, U.; Ratz, D.: **Numerical Toolbox for Verified Computing I: Basic Numerical Problems.** (Vol. II see [Kra98], version in C++ see [Ham95]) Springer–Verlag, Berlin / Heidelberg / New York, 1993.

[Ham95] Hammer, R.; Hocks, M.; Kulisch, U.; Ratz, D.: **C++ Toolbox for Verified Computing: Basic Numerical Problems.** Springer–Verlag, Berlin / Heidelberg / New York, 1995.

[Her94] Hergenhan, A.: *Spezifikation und Entwurf einer hochleistungsfähigen Gleitkomma–Architektur.* Diplomarbeit, Technische Universität Dresden, 1994.

[Hoe95] Hoefflinger, B.: *Next-Generation Floating-Point Arithmetic for Top-Performance PCs.* The 1995 Silicon Valley Personal Computer Design Conference and Exposition, Conference Proceedings, pp. 319-325, 1995.

[Hof93] Hoff, T.: *How Children Accumulate Numbers or Why We Need a Fifth Floating-Point Operation.* In: Jahrbuch Überblicke Mathematik, S. 219-222, Vieweg Verlag, 1993.

[Ker94] Kernhof, J.; Baumhof, Ch.; Höfflinger, B.; Kulisch, U.; Kwee, S.; Schramm, P.; Selzer, M.; Teufel, Th.: *A CMOS Floating-Point Processing Chip for Verified Exact Vector Arithmetic.* European Solid State Circuits Conference 94 ESSCIRC, Ulm, Sept. 1994.

[Kir87] Kirchner, R.; Kulisch, U.: *Arithmetic for Vector Processors.* In [ARITH, Vol. 8, pp. 256-269], 1987.

[Kir88] Kirchner, R.; Kulisch, U.: *Accurate Arithmetic for Vector Processing.* Journal of Parallel and Distributed Computing **5**, special issue on "High Speed Computer Arithmetic", pp. 250-270, 1988.

[Kla91] Klatte, R.; Kulisch, U.; Neaga, M.; Ratz, D.; Ullrich, Ch.: **PASCAL–XSC — Sprachbeschreibung mit Beispielen.** Springer-Verlag, Berlin/Heidelberg/New York, 1991 (ISBN 3-540-53714-7, 0-387-53714-7).

[Kla92] Klatte, R.; Kulisch, U.; Neaga, M.; Ratz, D.; Ullrich, Ch.: **PASCAL–XSC — Language Reference with Examples.** Springer-Verlag, Berlin/Heidelberg/New York, 1992.

[Kla93] Klatte, R.; Kulisch, U.; Lawo, C.; Rauch, M.; Wiethoff, A.: **C–XSC, A C++ Class Library for Extended Scientific Computing.** Springer-Verlag, Berlin/Heidelberg/New York, 1993.

[Kla93a] Klatte, R.; Kulisch, U.; Neaga, M.; Ratz, D.; Ullrich, Ch.: **PASCAL–XSC — Language Reference with Examples (In Russian).** Moscow, 1994.

[Kno91] Knöfel, A.: *Hardwareentwurf eines Rechenwerks für semimorphe Skalar- und Vektoroperationen unter Berücksichtigung der Anforderungen verifizierender Algorithmen.* Dissertation, Universität Karlsruhe, 1991.

[Kno91a] Knöfel, A.: *Fast Hardware Units for the Computation of Accurate Dot Products.* In [ARITH, Vol. 10, pp. 70-74], 1991.

[Kra87] Krämer, W.: *Inverse Standardfunktionen für reelle und komplexe Intervallargumente mit a priori Fehlerabschätzungen für beliebige Datenformate.* Dissertation, Universität Karlsruhe, 1987.

[Kra89] Krämer, W.; Walter, W.: *FORTRAN–SC: A FORTRAN Extension for Engineering/Scientific Computation with Access to ACRITH, General Information Notes and Sample Programs.* pp 1–51, IBM Deutschland GmbH, Stuttgart, 1989.

[Kra98] Krämer, W.; Kulisch, U.; Lohner, R.: **Numerical Toolbox for Verified Computing II: Theory, Algorithms and Pascal-XSC Programs.** (Vol. I see [Ham93,Ham95]) Springer–Verlag, Berlin / Heidelberg / New York, to appear 1998.

[Kul71] Kulisch, U.: *An axiomatic approach to rounded computations.* TS Report No. 1020, Mathematics Research Center, University of Wisconsin, Madison, Wisconsin, 1969, and Numerische Mathematik **19**, pp. 1-17, 1971.

[Kul75] Kulisch, U.: *Formalization and Implementation of Floating-Point Arithmetic.* Computing **14**, pp. 323-348, 1975.

[Kul76] Kulisch, U.: **Grundlagen des Numerischen Rechnens — Mathematische Begründung der Rechnerarithmetik.** Reihe Informatik, Band 19, Bibliographisches Institut, Mannheim/Wien/Zürich, 1976 (ISBN 3-411-01517-9).

[Kul81] Kulisch, U.; Miranker, W. L.: **Computer Arithmetic in Theory and Practice.** Academic Press, New York, 1981 (ISBN 0-12-428650-x).

[Kul81a] Kulisch, U.: *Schaltungsanordnung und Verfahren zur Bildung von Skalarprodukten und Summen von Gleitkommazahlen mit maximaler Genauigkeit.* Patentschrift DE 3144015 A1, 1981.

[Kul82] Kulisch, U.; Ullrich, Ch. (Eds.): **Wissenschaftliches Rechnen und Programmiersprachen.** Proceedings of Seminar held in Karlsruhe, April 2–3, 1982. Berichte des German Chapter of the ACM, Band 10, B. G. Teubner Verlag, Stuttgart, 1982 (ISBN 3-519-02429-2).

[Kul83] Kulisch, U.; Miranker, W. L. (Eds.): **A New Approach to Scientific Computation.** Proceedings of Symposium held at IBM Research Center, Yorktown Heights, N. Y., 1982. Academic Press, New York, 1983 (ISBN 0-12-428660-7).

[Kul84] Kulisch, U.; Miranker, W. L.: *The Arithmetic of the Digital Computer: A New Approach.* IBM Research Center RC 10580, pp. 1-62, 1984. SIAM Review, Vol. 28, No. 1, pp. 1-40, March 1986.

[Kul86] Kulisch, U.; Kirchner, R.: *Schaltungsanordnung zur Bildung von Produktsummen in Gleitkommadarstellung, insbes. von Skalarprodukten.* Patentschrift DE 3703440 C2, 1986.

[Kul87] Kulisch, U. (Ed.): **PASCAL–SC: A PASCAL extension for scientific computation**, Information Manual and Floppy Disks, Version IBM PC/AT; Operating System DOS. B. G. Teubner Verlag (Wiley-Teubner series in computer science), Stuttgart, 1987 (ISBN 3-519-02106-4 / 0-471-91514-9).

[Kul87a] Kulisch, U. (Ed.): **PASCAL–SC: A PASCAL extension for scientific computation**, Information Manual and Floppy Disks, Version ATARI ST. B. G. Teubner Verlag, Stuttgart, 1987 (ISBN 3-519-02108-0).

[Kul89] Kulisch, U. (Ed.): **Wissenschaftliches Rechnen mit Ergebnisverifikation — Eine Einführung.** Ausgearbeitet von S. Geörg, R. Hammer und D. Ratz. Vol. 58. Akademie Verlag, Berlin, und Vieweg Verlagsgesellschaft, Wiesbaden, 1989.

[Kul94] Kulisch, U.; Teufel, T.; Hoefflinger, B.: *Genauer und trotzdem schneller, Ein neuer Coprozessor für hochgenaue Matrix- und Vektoroperationen.* Titelgeschichte, Elektronik **26**, 1994.

[Lic88] Lichter, P.: *Realisierung eines VLSI-Chips für das Gleitkomma-Skalarprodukt der Kulisch-Arithmetik.* Diplomarbeit, Fachbereich 10, Angewandte Mathematik und Informatik, Universität des Saarlandes, 1988.

[Mei87] Meis, T.: *Brauchen wir eine Hochgenauigkeitsarithmetik?* Elektronische Rechenanlagen, Carl Hanser Verlag, pp. 19-23, 1987.

[Mue91] Müller, M.; Rüb, Ch.; Rülling, W.: *Exact Accumulation of Floating-Point Numbers.* In [ARITH, Vol. 10, pp. 64-69], 1991.

[Mue93] Müller, M.: *Entwicklung eines Chips für auslöschungsfreie Summation von Gleitkommazahlen.* Dissertation, Universität des Saarlandes, Saarbrücken, 1993.

[Pic72] Pichat, M.: *Correction d'une somme en arithmétique à virgule flottante.* Numerische Mathematik **19**, pp. 400-406, 1972.

[Pri91] Priest, D. M.: *Algorithms for Arbitrary Precision Floating Point Arithmetic.* In [ARITH, Vol. 10, pp. 132-143], 1991.

[Rum80] Rump, S. M.: *Kleine Fehlerschranken bei Matrixproblemen.* Dissertation, Universität Karlsruhe, 1980.

[Rum83] Rump, S. M.: *How Reliable are Results of Computers? / Wie zuverlässig sind die Ergebnisse unserer Rechenanlagen?* In: *Jahrbuch Überblicke Mathematik*, Bibliographisches Institut, Mannheim, 1983.

[Rum83a] Rump, S. M.; Böhm, H.: *Least Significant Bit Evaluation of Arithmetic Expressions in Single-Precision.* Computing **30**, pp. 189-199, 1983.

[Sch92] Schmidt, L.: *Semimorphe Arithmetik zur automatischen Ergebnisverifikation auf Vektorrechnern.* Dissertation, Universität Karlsruhe, 1992.

[Ste84] Stetter, H. J.: *Sequential Defect Correction for High-Accuracy Floating-Point Algorithms.* Lecture Notes in Mathematics, Vol. 1006, pp. 186-202, Springer-Verlag, 1984.

[Ste89] Stetter, H. J.: *Staggered Correction Representation, a Feasible Approach to Dynamic Precision.* In: *Proceedings of the Symposium on Scientific Software*, edited by Cai, Fosdick, Huang, China University of Science and Technology Press, Beijing, China, 1989.

[Suz96] Suzuki, H.; Morinaka, H.; Makino, H.; Nakase, Y.; Mashiko, K.; Sumi, T,: *Leading-Zero Anticipatory Logic for High-Speed Floating-Point Addition.* IEEE Journal of Solid-State Circuits, Vol. 31, No. 8, August 1996.

[Tan92] Tangelder, R.J.W.T: *The Design of Chip Architectures for Accurate Inner Product Computation.* Dissertation, Technical University Eindhoven, 1992. ISBN 90-9005204-6.

[Teu84] Teufel, T.: *Ein optimaler Gleitkommaprozessor.* Dissertation, Universität Karlsruhe, 1984.

[Ull90] Ullrich, Ch. (Ed.): **Computer Arithmetic and Self-Validating Numerical Methods.** (Proceedings of SCAN 89, held in Basel, Oct. 2-6, 1989, invited papers). Academic Press, San Diego, 1990.

[Wal90] Wallis, P. J. L. (Ed.): **Improving Floating-Point Programming.** J. Wiley, Chichester, 1990 (ISBN 0 471 92437 7).

[Wal89] Walter, W.: *FORTRAN–SC: A FORTRAN Extension for Engineering / Scientific Computation with Access to ACRITH, Language Reference and User's Guide.* 2nd ed., pp. 1-396, IBM Deutschland GmbH, Stuttgart, Jan. 1989.

[Wil63] Wilkinson, J.: **Rounding Errors in Algebraic Processes.** Prentice-Hall, Englewood Cliffs, New Jersey, 1963.

[Win85] Winter, Th.: *Ein VLSI-Chip für Gleitkomma-Skalarprodukt mit maximaler Genauigkeit.* Diplomarbeit, Fachbereich 10, Angewandte Mathematik und Informatik, Universität des Saarlandes, 1985.

[Win90] Winter, D. T.: *Automatic Identification of Scalar Products.* In [Wal90], 1990.

[Yil89] Yilmaz, T.; Theeuwen, J.F.M.; Tangelder, R.J.W.T.; Jess, J.A.G.: *The Design of a Chip for Scientific Computation.* Eindhoven University of Technology, 1989 and pp. 335-346 of Proceedings of the Euro-Asic Symposium, Grenoble, Jan.25-27, 1989.

[Yoh73] Yohe, J.M.: *Roundings in Floating-Point Arithmetic.* IEEE Trans. on Computers, Vol. C-22, No. 6, June 1973, pp. 577-586.

[ARITH] Institute of Electrical and Electronics Engineers: **Proceedings of x-th Symposium on Computer Arithmetic ARITH.** IEEE Computer Society Press. IEEE Service Center, 445 Hoes Lane, P.O.Box 1331, Piscataway, NJ 08855-1331, USA.

Editors of proceedings; place of conference; date of conference.

(i) Shively, R.R.; Minneapolis; June 16, 1969.
(ii) Garner, H.L.; Atkins, D.E.; Univ Maryland, College Park; May 15 – 16, 1972.
(iii) Rao, T.R.N.; Matula, D.W.; SMU, Dallas; Nov. 19 – 20, 1975.
(iv) Avizienis, A.; Ercegovac, M.D.; UCLA, Los Angeles; Oct. 25 – 27, 1978.
(v) Trivedi, K.S.; Atkins, D.E.; Univ Michigan, Ann Arbor; May 18 – 19, 1981.
(vi) Rao, T.R.N.; Kornerup, P.; Univ Aarhus, Denmark; June 20 – 22, 1983.

(vii) Hwang, K.; Univ Illinois, Urbana; June 4 – 6, 1985.
(viii) Irwin, M.J.; Stefanelli, R.; Como, Italy; May 19 – 21, 1987.
(ix) Ercegovac, M.; Swartzlander, E.; Santa Monica; Sept. 6 – 8, 1989.
(x) Kornerup, P.; Matula, D.; Grenoble, France; June 26 – 28, 1991.
(xi) Swartzlander Jr., E.; Irwin, M. J.; Jullien, G.; Windsor, Ontario; June 29 – July 2, 1993.
(xii) Knowles, S.; Mc Allister, W. H.; Bath, England; July 19 – 21, 1995;
(xiii) Lang, Th.; Muller, J.-M.; Takagi, N.; Asilomar, California; July 6 – 9, 1997;

[IAM80] IAM: *PASCAL-XR: PASCAL for eXtended Real arithmetic.* Joint research project with Nixdorf Computer AG. Institute of Applied Mathematics, University of Karlsruhe, Postfach 6980, D-76128 Karlsruhe, Germany, 1980.

[IAM88] IAM: *FORTRAN–SC: A FORTRAN Extension for Engineering / Scientific Computation with Access to ACRITH.* Institute of Applied Mathematics, University of Karlsruhe, Postfach 6980, D-76128 Karlsruhe, Germany, Jan. 1989.
1. Language Reference and User's Guide, 2nd edition.
2. General Information Notes and Sample Programs.

[IAM90] IAM: *ACRITH–XSC, A Programming Language for Scientific Computation.* Syntax Diagrams. Institute of Applied Mathematics, University of Karlsruhe, Postfach 6980, D-76128 Karlsruhe, Germany, 1990.

[IBM84] IBM: *IBM System/370 RPQ. High Accuracy Arithmetic.* SA 22-7093-0, IBM Deutschland GmbH (Department 3282, Schönaicher Strasse 220, D-71032 Böblingen), 1984.

[IBM86] IBM: **IBM High-Accuracy Arithmetic Subroutine Library (ACRITH).** IBM Deutschland GmbH (Department 3282, Schönaicher Strasse 220, D-71032 Böblingen), 3rd edition, 1986.
1. General Information Manual. GC 33-6163-02.
2. Program Description and User's Guide. SC 33-6164-02.
3. Reference Summary. GX 33-9009-02.

[IBM86a] IBM *Verfahren und Schaltungsanordnung zur Addition von Gleitkommazahlen.* Europäische Patentanmeldung, EP 0 265 555 A1, 1986.

[IBM90] IBM: **ACRITH–XSC: IBM High Accuracy Arithmetic — Extended Scientific Computation. Version 1, Release 1.** IBM Deutschland GmbH (Schönaicher Strasse 220, D-71032 Böblingen), 1990.
1. General Information, GC33-6461-01.
2. Reference, SC33-6462-00.
3. Sample Programs, SC33-6463-00.
4. How To Use, SC33-6464-00.
5. Syntax Diagrams, SC33-6466-00.

[IEEE81] IEEE: *A Proposed Standard for Binary Floating-Point Arithmetic.* IEEE Computer, March 1981.

[IEEE85] American National Standards Institute / Institute of Electrical and Electronics Engineers: *A Standard for Binary Floating-Point Arithmetic.* ANSI/IEEE Std. 754-1985, New York, 1985 (reprinted in SIGPLAN **22**, 2, pp. 9-25, 1987). Also taken over as IEC Standard 559:1989.

[IEEE87] American National Standards Institute / Institute of Electrical and Electronics Engineers: *A Standard for Radix-Independent Floating-Point Arithmetic.* ANSI/IEEE Std. 854-1987, New York, 1987.

[IMACS89] IMACS; GAMM: *IMACS-GAMM Resolution on Computer Arithmetic.* In Mathematics and Computers in Simulation **31**, pp. 297-298, 1989. In Zeitschrift für Angewandte Mathematik und Mechanik **70**, no. 4, p. T5, 1990.

[IMACS93] IMACS; GAMM: *GAMM-IMACS Proposal for Accurate Floating-Point Vector Arithmetic.* GAMM, Rundbrief 2, pp. 9-16, 1993. Mathematics and Computers in Simulation, Vol. **35**, IMACS, North Holland, 1993. News of IMACS, Vol. 35, No. 4, pp. 375-382, Oct. 1993.

[NUM91] Numerik Software GmbH: **PASCAL–XSC: A PASCAL Extension for Scientific Computation. User's Guide.** Numerik Software GmbH, Haid-und-Neu-Straße 7, D-76131 Karlsruhe, Germany / Postfach 2232, D-76492 Baden-Baden, Germany, 1991.

[SIE86] SIEMENS: **ARITHMOS (BS 2000) Unterprogrammbibliothek für Hochpräzisionsarithmetik. Kurzbeschreibung, Tabellenheft, Benutzerhandbuch.** SIEMENS AG, Bereich Datentechnik, Postfach 83 09 51, D-8000 München 83. Bestellnummer U2900-J-Z87-1, Sept. 1986.