ARITHMETIC FOR VECTOR PROCESSORS

R. Kirchner and U. Kulisch

Fachbereich Informatik, Universität Kaiserlautern
Fakultät für Mathematik, Universität Karlsruhe
West Germany

Abstract: In electronic computers the elementary arithmetic operations are these days generally approximated by floating-point operations of highest accuracy. Vector processors and parallel computers often provide additional operations like "multiply and add", "accumulate" or "multiply and accumulate". Also these operations shall always deliver the correct answer whatever the data are. The user should not be oblighed to execute an error analysis for operations predefined by the manufacturer.

In the first part of this paper we discuss circuits which allow a fast and correct computation of sums and scalar products making use of a matrix shaped arrangement of adders and pipeline technology. In the second part a variant is discussed which permits a drastic reduction in the number of adders required. The methods discussed in this paper can also be used to build a fast arithmetic unit for micro computers in VLSI-technology.

1. Introduction

Modern computers of highest performance, the so-called vectorprocessors or supercomputers, are gaining considerably in importance in research and development. They serve for simulation of processes which cannot be measured at all or only with great effort, for solving large engineering design problems or for evaluation of large sets of measured data and for many other applications. It is commonly assumed that these computers open a new dimension for scientific computation. In sharp contrast to this is the fact that the arithmetic implemented on supercomputers differs only marginally from that of their slower predecessors, although results are much more sensitive to rounding errors, numerical instabilities, etc. due to the huge number of operations executed.

Research in numerical mathematics has shown that, with a more comprehensive and optimal vector arithmetic, reliable results can be more easily obtained when dealing with extensive and huge problems. Computers with this kind of arithmec have proved the significance of this development in many successful applications.

Until now, it has been assumed that an optimal vector arithmetic could not be implemented on supercomputers. The users, therefore, had to choose between either lengthy computation times and accurate results on general purpose computers or comparatively short computation times and possibly wrong results obtained on supercomputrs.

It was assumed, in particular, that correct computation of continued sums and scalar products, which are necessary for vector arithmetic, could not be implemented on supercomputers with pipeline processing. Well known circuits, which solve this problem, require several machine cycles for carrying out a single addition whereas a computer of highest performance with traditional arithmetic carries out one addition in each cycle[1]. This paper describes various circuits for the optimal computation of sums and scalar products at the speed of supercomputers. There is, in principle, no longer any reason to continue to accept inaccurate sums or scalar products by not using optimal vector arithmetic on vectorprocessors and supercomputers. The additional costs compared with the cost of the complete system are justified in any case. It takes the burden of an error analysis from the user.

The first electronic computers were developed in the middle of this century. Before then, highly sophisticated electromechanical computing devices were used. Several very interesting techniques provided the four basic operations of addition, subtraction, multiplication, and division. Many of these calculators were able to perform an additional operation which could be called "accumulating addition/subtraction" or continued summation. The machine was equipped with an input register of about 10 to 13 digits. Compared to that, the result register was much longer and had perhaps 30 digits. It was situated on a sled which could be shifted back and forth relatively to the input register. This allowed an accumulation of a large number of summands into different positions of the result register. There was no rounding executed after each addition. As long as no overflow occurred, this accumulating addition was error free. Addition was associative, the result being independent of the order in which the summands were added.

This accumulating addition without intermediate roundings was never implemented on electronic com-

[1] By a cycle time or a machine cycle we understand the time which the system needs to deliver a summand or a product, in case of a scalar product computation, to the addition pipeline.

CH2419-0/87/0000/0256$01.00 © 1987 IEEE

puters. Only recently, several /370 compatible systems have appeared which simulate this process on general purpose machines by accumulating into an area in main memory, which is kept in the cache memory for enhanced performance. [5], [6]. This allows the elimination of a large number of roundings and contributes essentially to the stability of the computational process. This paper desribes circuits for an implementation of the accumulating addition on very fast computers making use of pipelining and other techniques.

The first electronic computers executed their calculations in fixed-point arithmetic. Fixed-point addition and subtraction is error free. Even very long sums can be accumulated with only one final rounding in fixed-point arithmetic, if a carry counter is provided which gathers all intermediate positive or negative overflows or carries. At the very end of the summation a normalization and rounding is executed. Thus accumulation of fixed point numbers is associative again. The result is correct to one unit in the last figure and it is independent of the order in which the summands are added. Fixed-point arithmetic, however, imposed a scaling requirement. Problems needed to be preprocessed by the user so that they could be accommodated by the fixed-point number representation.

With the increasing speed of computers, problems that could be solved became larger and larger. The necessary pre-processing soon became an enormous burden.

The introduction of floating-point representation in computation largely eliminated this burden. A scaling factor is appended to each number in floating-point representation. The arithmetic itself takes care of the scaling. Multiplication and division require an addition, respectively subtraction, of the exponents which may result in a large change in the value of the exponent. But multiplication and division are relatively stable operations in floating-point arithmetic. Addition and subtraction, in contrast, are troublesome in floating-point.

As an example let us consider the two floating-point vectors

$$x = \begin{bmatrix} 10^{20} \\ 1223 \\ 10^{24} \\ 10^{18} \\ 3 \\ -10^{21} \end{bmatrix} , \qquad y = \begin{bmatrix} 10^{30} \\ 2 \\ -10^{26} \\ 10^{22} \\ 2111 \\ 10^{19} \end{bmatrix}$$

A computation of the inner or scalar product of these two vectors gives

$x.y = 10^{50} + 2,446 - 10^{50} + 10^{40} + 6,333 - 10^{40} = 8,779$

Most digital computers will return zero as the answer although the exponents of the data vary only within 5 % or less of the exponent range of large systems. This error occurs because the floating-point arithmetic in these computers is unable to cope with the large digit range required for this calculation.

Floating-point representation and arithmetic in computers was introduced in the middle of this centry. Computers then were relatively slow, being able to execute only about 100 floating-point operations in a second. The fastest computers today are able to execute billions of floating-point operations in a second. This is a gigantic gain in speed by a factor of 10^7 over the electronic computers of the early fifties. Of course, the problems that can be dealt with, have become larger and larger. The question is whether floating-point representation and arithmetic which already fails in simple calculations, as illustrated above, are still adequate to be used in computers of such gigantic speed for huge problems.

We think that the set of floating-point operations should be extended by a fifth operation, the "accumulating addition/subtraction" without intermediate rounding, an operation which was already available on many electromechanical calculators. It is the purpose of this paper to show that this additional operation can be executed with extreme speed. We realize this operation by adding the floating-point summands into a fixed-point number over the full floating-point range. Thus "accumulating addition/subtraction" is error free. Even very long chains of additions/subtractions can be executed with only a single rounding at the very end of the summation. Such "Accumulating addition/subtraction" is associative. The result is independent of the order in which the summands are added.

With the fifth operation "accumulating addition/-subtraction", we combine the advantages of fixed-point arithmetic - error free addition and subtraction even for very long sums - with the advantages of floating-point arithmetic - no scaling requirements.

2. The State of the Art

A normalized floating-point number z (in sign-magnitude representation) is a real number of the form

$$z = * \, m \cdot b^e \, .$$

Here $* \in \{+,-\}$ denotes the sign (sign(z)), m the mantissa (mant(z)), b the base of the number system and e the exponent (exp(z)). b is an integer number with b > 1. The exponent is an integer and lies between two integers $e1 \leq e2$. In general, $e1 < 0$ and $e2 > 0$. m is the mantissa. It is of the form

$$m = \sum_{i=1}^{l} z[i] \cdot b^{-i} \, .$$

Here, the z[i] denote the digits of the mantissa; $z[i] \in \{0,1,\ldots,b-1\}$ for all i = 1(1)n and $z[1] \neq 0$. l is the length of the mantissa. It denotes the number of mantissa digits carried along. The set of normalized floating-point numbers does not contain the number 0. In order to obtain a unique definition of 0 one can additionally define: sign(0) = +, mant(0) = .000 ... 0 (l zeros after the point) and exp(0) = e1. This kind of floating-point system depends on four constants b,l,e1 and e2. We denote it with S = S(b,l,e1,e2).

Let

$$u = (u_i) = \begin{bmatrix} u_1 \\ u_2 \\ . \\ . \\ u_n \end{bmatrix} \qquad v = (v_i) = \begin{bmatrix} v_1 \\ v_2 \\ . \\ . \\ v_n \end{bmatrix}$$

be two vectors, the components of which are normalized floating-point numbers, i.e. $u_i, v_i \in S$ forall $i = i(1)n$. The theory of computer arithmetic[1], [2], [3] demands that scalar products of two floating-point vectors u and v be computed with maximum accuracy by the computer for each relevant, finite n and different roundings. By doing so, millions of roundings can be eliminated in complicated calculations. This contributes essentially to the stability of the computational process and enlarges the reliability and accuracy of computed results. Furthermore, defect correction then becomes an effective mathematical instrument.

This requires, for example, the execution of the following formulas by the computer:

$$u \odot v = \bigcirc \left(\sum_{i=1}^{n} u_i * v_i \right)$$

$$u \boxdot v = \square \left(\sum_{i=1}^{n} u_i * v_i \right)$$

$$u \overset{\cdot}{\bigtriangledown} v = \bigtriangledown \left(\sum_{i=1}^{n} u_i * v_i \right) \qquad \text{(I)}$$

$$u \overset{\cdot}{\bigtriangleup} v = \bigtriangleup \left(\sum_{i=1}^{n} u_i * v_i \right)$$

The multiplication- and addition-signs on the right side denote the correct multiplication and addition for real numbers. ○, □, ▽, △ are rounding symbols. ○ denotes a rounding to the nearest floating-point number, □ denotes the rounding towards zero, ▽ denotes the monotone downwardly directed rounding and △ denotes the monotone upwardly directed rounding.

For an execution of formula (I) first the products $u_i * v_i$ have to be correctly calculated by the computer. This leads to a mantissa of 2l digits and an exponent which lies in the range of $2e1-1 \leq e \leq 2e2$. So the computation of scalar products is reduced to the evaluation of sums of the following form:

$$\diamond \left(\sum_{i=1}^{n} w_i \right), \quad n \in \mathbb{N} \qquad \text{(II)}$$

Here the w_i are floating-point numbers of double length $w_i \in S(b,2l,2e1-1,2e2)$, for all $i = 1(1)n$. ◇ denotes a general rounding symbol, $\diamond \in \{\bigcirc, \square, \bigtriangledown, \bigtriangleup\}$. Measures have to be taken first to generate and represent the summands w_i correctly in the computer. In case of scalar products this can be done by [illegible] and [illegible] circuits.

For traditional general purpose computers there are several ways to correctly compute (I) and (II) mentioned in the literature. It is the intention of this paper to describe circuits for high speed computation of (I) and (II) on vector computers by means of pipeline techniques. These circuits have to accept and process one summand from (I) resp. (II) during each machine cycle. To assist in the understanding of the following material, we first refer to one of the possibilities mentioned in [4]:

We consider a register of $L = k + 2e2 + 2l + 2|e1|$ digits of base b, which should be placed in the arithmetic unit (Figure 1).

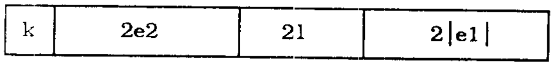


Figure 1

We divide this register into segments of length l (Fig. 2):

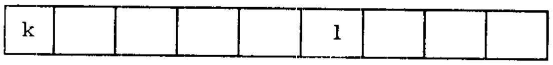


Figure 2

The summands in (I) and (II) are of length 2l. They fit therefore, digitwise into a subrange of length 3l of this storage. This part of the register, which is determined by the exponent of the summand, is selected and loaded into an accumulator of length 3l. The summand is loaded into a shiftregister of the same length, being correctly positioned according to the exponent, and then added into the accumulator (Figure 3).

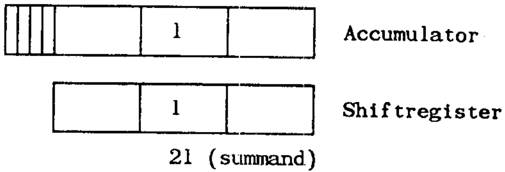


Figure 3

The addition may produce a carry. In order to catch this carry, a few more digits than the three words of length l can be read from the long register into the accumulator, which is extended to the left accordingly. If not all of these digits are b-1, the carry is caught by these additional digits. Since it is possible that all these additional digits are b-1, a loop has to be provided which then adds the carry to the following digits of the long register. This loop may possibly have to be activated several times.

The addition of the summands of (I) resp. (II) into the long register, Fig. 1 resp. Fig. 2, may still produce a carry on the very far left of the register. In order to catch such carries the long register is extended on the left by a few more (k) digits of base b (Fig. 1). Then, any sum (I) or (II) of n summands can be added without loss of information into the long register of length L. b^k carries may occur and can be processed without loss of information.

Here we conclude our description of one possibility to solve the problems (I) and (II). See [4].

What we just described belongs to the state of the art.

3. Fast Computation of Sums and Scalar Products

The method described above is not suited for the computation of (I) resp. (II) on vector processors or supercomputers. The process of reading, shifting, carry handling, possibly by a loop, and writing back is certainly too slow to be executed in one cycle time of only a few nsecs of these computers. A solution of the problem by a very long adder is also very costly and probably too slow.

We therefore discuss here a variant of the possibilities mentioned above which makes processing of a summand of (I) resp. (II) possible within a very short cycle time. In comparison to general purpose computers, vector processors and supercomputers achieve their high speed of computation by means of pipeline technology whereby during each machine cycle a result is obtained. If scalar products and sums are to be computed with high speed on vector processors or supercomputers, one has to develop circuits which accept and process one summand (resp. a product) per machine cycle. This is only possible if the addition is done by means of pipeline technology. This paper describes various circuits which allow this.
At first the most important issues and ideas of the circuitry are presented in the text referring to Figures 4 to 15. These Figures contain some more details which are not essential for a first understanding of the principles. These details are presented later in chapter 4 "Additional Remarks concerning the Figures".
The circuit described below consists of a shifter which is followed by a pipelined adder called summing matrix (Figure 4). The shifting device may be realized by standard technology and belongs to the state of the art.
The adder consists of registers of a total length of $S \geq L$. Here L denotes the length of the long register as outlined above[2] (Figure 1). The register length S is divided into r identical parts which are arranged as rows one below the other (Figure 5). r denotes the number of rows. All rows are of the same length. Each of these rows is divided into $c \geq 1$ independent adders A (see Figure 6). Thus the whole summing device consists of $r \cdot c$ independent adders. Each of these adders A has a width of a digits. Between two of these independent adders, carry handling must be possible. Also between the last adder of a row and the first one of the next row a carry handling must be possible. The complete summing device which we call the summing matrix SM, has a width of $S = a \cdot c \cdot r$ digits of base b. c denotes the number of columns of the summing matrix. It must be $S \geq L = k + 2e2 + 21 + 2 |e1|$ (Figures 5, 6).
The summing matrix contains $c \cdot r$ independent adders A. Each of these adders must be able to add a digits of base b in parallel within one machine cycle, and to register a carry which possibly may occur. Since each row of the summing matrix consists of c i[illegible] adders, h:= [illegible] digits can be added i[illegible] of the summ[illegible] matrix. Each of the r rows of the summing matrix must be at least as lo[illegible] mantissa l[illegible] of the summands which [illegible] be added. Each [illegible] it of the summing matrix is characterized by a certain exponent corresponding to the digit [illegible] tion. The upper right [illegible] the summing matrix carries the least significant digit, the low[illegible] part of the summing [illegible] carries the r[illegible] significant digit of the f[illegible] summing device ([illegible] 5, Figure 6).
Each summand [illegible] each product of [illegible] resp. (II) must now be [illegible] into the summing matrix at the proper positi[illegible] according to its e[illegible] t. The row selection is [illegible] d by the more [illegible] icant bits of the expon[illegible] div h)[3] and [illegible] selection of the columns [illegible] mined by the l[illegible] significant bits of the [illegible] t (exp mod h)[4] [illegible] s complies roughly with [illegible] lection of the [illegible] ing position in two steps [illegible] process descr[illegible] in Fig. 3.

The incoming s[illegible] ls resp. produc[illegible] re now first shifted in t[illegible] ifting unit ([illegible] el shifter, cross bar sw[illegible] into the corre[illegible] position according to th[illegible] onents. The s[illegible] is executed as a ringshif[illegible] s means that [illegible] part of the summand which [illegible] over the rig[illegible] end is reinserted at the [illegible] end of the shi[illegible] gister (Figure 6 upper pa[illegible] summands 2 and 3, Figure 8). The summand is di[illegible] ibuted onto the [illegible] independent parts of wid[illegible] f the shiftregi[illegible] r. Each part receives an e[illegible] identification according to a specific digi[illegible] it, e.g. the le[illegible] significant one (Figures 5 [illegible] and 10). The in[illegible] ual adders A also carry an [illegible] ent identifica[illegible]. The shifted and expanded s[illegible] nd now drops int[illegible] e top row of the summing m[illegible] and thereafter [illegible] ceeds row by row through t[illegible] umming matrix, [illegible] ng ahead one row in each ma[illegible] e cycle. The ad[illegible] ion is executed as soon a[illegible] exponent iden[illegible] ication of a transfer regis[illegible] in the summing m[illegible] rix coincides with the expon[illegible] identification p[illegible] of the summand.
A summand, whi[illegible] arrives at the su[illegible] ng unit, can remain connec[illegible] after shifting [illegible] the correct position withi[illegible] he shifting unit. In this case, the addition is executed in only [illegible] e row of the summing matrix. The shift procedu[illegible] however, can also cause an overhanging at the right end of the row. The overh[illegible] ng part then is reinserted by a ringshift at [illegible] left end of the shifting unit (see Figures 6 and 8). In this case, the addition of both parts [illegible] the summand is then executed in neighbouring r[illegible] of the summing matrix. If the most significant part of the summand, which was situated at the right end of the shifter, is added in row y then the addition of the least significant part, which was situated at the left end of the shifter, is added in row y - 1. This means the next less significant row (see Figure 9).
It is, however, not at all necessary that each

[2] or a part of it. A reduction of the length S is discussed below.

[3] div denotes i[illegible] ger division,
i.e. 24 di[illegible]) = 2.

[4] mod denotes t[illegible] remainder of inte[illegible] r division,
i.e. 24 mo[illegible] 1) = 4.

transfer unit carries a complete exponent identification. It is sufficient to identify the row by the exponent part exp $\underline{d}$ [illegible] h of the summands in the shifter and to use it for selection of row y. The distinction whether the addition has to be executed in row y or in row y − 1 is made by a bit connected with each transfer register or by a suitable column signal which distinguishes the transfer registers of a row. (The principle is illustrated by the diagrams shown in Figures 11 and 12).

The addition may cause [illegible]ries between the independent adders A. Carry [illegible]isters between the independent adders absorb [illegible] carries. In the next machine cycle these carries are added into the next more significant adder A, possibly together with another summand. In this way, during each machine cycle one summand [illegible] be fed into the summing matrix, although [illegible] carry handling of one summand may take several machine cycles. The method displayed in the [illegible]res shows one of diverse possibilities to handle [illegible] carries. There may be carry presencing or lo[illegible]d or other techniques applied to speed up the carry processing within one row. In any way, the [illegible]ming matrix allows the carry processing to be [illegible]uted independently of the summations and in p[illegible]el with the processing that has to be done a[illegible], e.g. adding further summands or reading out [illegible] result.

In principle, the summi[illegible]trix can only process positive summands. Neg[illegible] summands or positive subtrahends are therefo[illegible]rked and at the proper place not added but su[illegible]d. Here negative carries instead of positi[illegible]ies may occur. Similar to positive carries th[illegible] to be processed possibly over several mac[illegible]cles. In other words: The independent adders [illegible] be able to carry out additions as well as s[illegible]tions and to process positive and negative [illegible]s in both cases (Figure 6, 12).

The design of the comp[illegible]mming device containing the summing matrix described herewith can depend on the technolo[illegible]ed. We have mentioned already that the width [illegible] the individual adders A has to be chosen in [illegible] way that an addition over the complete width [illegible] be executed within one machine cycle. Each ro[illegible]e summing matrix must be at least as wide [illegible] individual summands. The shorter the rows [illegible]e [illegible]ster the summands can be shi[illegible]d into [illegible]ght position. On the other hand, shortening [illegible] width of the rows of the summing matrix i[illegible]s the number of rows and with it, the numb[illegible]ipeline steps for the complete su[illegible]ation pr[illegible]

After inpu[illegible]f the l[illegible]mma[illegible] the rows can be read [illegible]tarti[illegible] with the [illegible] significant row, provided the [illegible]w in que[illegible] does not require any carry ha[illegible]g. In th[illegible]se the carries first have [illegible] be [illegible]moved. [illegible]adout process can use the [illegible]me [illegible] path [illegible]ch the summands pass throu[illegible] matrix. [illegible]e result rows follow the [illegible]st [illegible]and on [illegible] t[illegible]ugh the transfer regi[illegible]rs. [illegible]ring th[illegible] [illegible]ut process additions and carry ha[illegible]dling i[illegible] more significant rows may still [illegible] executed simultaneously with the readout [illegible]ess the [illegible] to the required floating-[illegible]t format [illegible] executed. The result can [illegible] stored a[illegible] intermediate long variable [illegible]r [illegible]her proc[illegible]. Several rounding possibil[illegible] can be c[illegible]out simultaneously as mentioned in [4]. During the readout process the computation of a new scalar product resp. a new sum can be started.

The width a of the independent adders A depends on the technology used and on the cycle time of the system. The width should be as large as possible. But on the other hand, it must permit the addition over the a digits in one machine cycle. (In the case of a scalar product, a machine cycle is the time in which the system delivers a product).

Depending on the technology there are several possibilities of transportation of the summands to one of the r rows of the summing matrix SM.

The method described above is based on the idea that each of the independent adders A is supplemented by a transfer register of the same width (plus tag-register for exponent identification and +/- control). During each machine cycle, each transfer register can pass on its contents to the transfer register in the corresponding position in the next row and receive a digit sequence from the transfer register in the corresponding position in the previous row. Attached to the transfer registers is the tag-register for exponent identification (Figure 5 and Figure 6). The contents of this register are always compared with the exponent identification of the corresponding adder. In case of coincidence, the addition resp. subtraction is activated (Figures 5, 6 and 12).

Alternatives to this procedure are also possible.

1. One of these alternatives could be to transfer the summand in one machine cycle directly into the appropriate row of transfer registers of the summing matrix as determined by the exponent. During the following machine cycle, the addition is executed. Simultaneously, a new summand can be transferred to the same, or another row, so that an addition in each machine cycle is carried out.
2. The procedure is similar to 1. The intermediate storage of the summands in transfer registers, however, is not necessary if it is possible to execute the transfer- and addition-process in one machine cycle. In this case, no transfer registers are necessary. The output of the result then also takes place directly.
3. The transfer of the summands to the target row can be carried out not only sequentially and directly but also with several intermediate steps, for example, by binary selection.

Each one of these alternatives also allows a direct and therefore faster readout of the result without dropping step by step through the transfer registers.

To each independent adder A of length a belongs a transfer register TR which is basically of the same length. The number of adders A resp. transfer registers TR in a row is chosen in such a way that the mantissa length $\bar{m}$ of the summands plus the length of the transfer registers t (=a) becomes less or equal to the length of the row ($\bar{m} + a \leq h = c \cdot a$). In this way, an overlapping of the less significant part of the mantissa with its most significant part in one transfer register is avoided. For typical floating-point formats this condition may result in long rows of the summing matrix or in short widths a of the adders resp. transfer registers. The former case causes lengthy shifts while the latter case causes more carries (Figure 6 upper part and Figure 8).

This disadvantage can be avoided by providing several (≥ 2) partial transfer registers for each adder of length a. Each partial transfer register TR of length $t \leq a$ carries its own exponent identification. Finally, the length t of the transfer registers can be chosen independently of the length a of the adders A. Both only need to be integer divisors of the row length of the summing matrix $h = a \cdot c = t \cdot n$ (see Figures 13, 14 and 15).

Figures 6 and 13 show, in particular, that the summing matrix has a very systematic structure and that it can be realized by a few, very simple building blocks. It is suitable, therefore, for realization in various technologies.

Based on the same principle also, summands which consist of products of three and more factors can be added correctly.

If the summing matrix is to be realized in VLSI-technology it may happen that the complet summing matrix does not fit on a single chip. One should then try to develop components for the columns of the summing matrix since the number of connections (pins) between adjacent columns is much smaller than between neighbouring rows.

The following remarks and Figures 4 to 15 provide a more detailed description of the structure of the summing matrix and its functioning.

4. Additional Remarks concerning the Figures

The following abbreviations are used in the Figures:

A	Adder
AC	Accumulator Register
CY	Carry
E	Tag-Register for Exponent Identification
LSB	Least Significant Bit
MSB	Most Significant Bit
SM	Summing Matrix
SR	Shifter
TR	Transfer Register

Figure 4 shows a structure diagram of the complete summing circuitry and illustrates the interaction of different parts of the whole circuitry, such as: separation of the summands into sign, exponent and mantissa, shifting unit, summing matrix, controller and rounding unit.

Figure 5: As mentioned in the text, we assume that $S \geq L$. Figure 5 shows the case $S > L$. There, for both the first and last rows part of the row is covered by transfer registers only. For the whole summing matrix this means that transfer registers exist for S digits but adders for L digits only. L is chosen such that it is a multiple of a.

The dotted lines through the independent adders A indicate that the transfer wires bypass the adders. Above the transfer registers, the tag-register for exponent identification is indicated by a box. This register is part of the transfer register.

Figure 6 sh[illegible] block diagram [illegible] e summing matrix. It is [illegible] on a s[illegible] format which uses 4 bit[illegible] ribe one di[illegible] se b.

Width of A [illegible] bytes = [illegible]
Number of a[illegible] in one row c [illegible]
Number of r[illegible] SM $r = 8$
$k = 20$ carry [illegible], $l = 14$ dig[illegible] the mantissa
$e1 = -64$ and [illegible] = 64.
Users of /3[illegible] tible [illegible] recognize this data [illegible] their [illegible] sion format.
$L = 20 + 2 \cdot$ [illegible] $2 \cdot 14 +$ [illegible] = [illegible]04 digits of 4 bits = 1[illegible].
Width of the [illegible] mplete summing matrix
$S = a \cdot c \cdot$ [illegible] $4 \cdot 5 \cdot 8$ byte[illegible] 0 bytes $\geq L =$ 152 bytes.
In this ex[illegible] the width t of [illegible] ansfer registers equal [illegible] width of the [illegible]: $t = a = 4$ bytes.
The upper [illegible] of the Figure shows several positions of [illegible] ands.

Figure 7 def[illegible] the exponent [illegible] ates x and y of the dig[illegible] e summing ma[illegible] horizontal, y vertical [illegible] coordinate[illegible] obtained according to [illegible] ollowing formu[illegible]

e_o denotes [illegible] e reference po[illegible] e digit with the le[illegible] exponent in t[illegible] rix (at the upper [illegible] t end).

e_1 denote[illegible] least significa[illegible] digit of the adder.

e_m denotes [illegible] most signifi[illegible] nt digit of theadde[illegible]
If the [illegible] and the last row [illegible] the complete matrix [illegible] ins adders ov[illegible] e full width then e_1 [illegible] $_o$ and $e_m = e_o +$ [illegible]

e denotes [illegible] e exponent of a digit to be added.

$e-e_o$ denotes [illegible] e distance to the [illegible] st significant end of [illegible] matrix.

$y = (e-e_o)$ [illegible] h is the row [illegible] nate in which the di[illegible] with the exponent [illegible] s added.

$x = (e-e_o)$ [illegible] h indicates the distance to the least s[illegible] ficant end of row y.

Figures 8 a[illegible] describe the task of the shift unit and its [illegible] lation to the generation of the exponent ide[illegible] ication which will be transferred into the summ[illegible] g matrix with the mantissa.

The task of t[illegible] shift unit is:

1. adjust [illegible] ntissa to the co[illegible] position for its add[illegible] on, if necessary b[illegible] ring shift.
2. fill th[illegible] emaining positions of the transfer registe[illegible] resp. the row with zeros.

Figure 8 sho[illegible] the shifted mantissa in both possible cases.

Figure 9 desc[illegible] bes the shift process. Two cases are to be dis[illegible] nguished:

1. $x = (e-e$ [illegible] mod $h \geq \bar{m}$: no overhanging,
   t[illegible] whole mantissa is added in one r[illegible].
2. $x < \bar{m}$: [illegible] overhanging,
   t[illegible] mantissa is [illegible] dded in two

successive rows.
Part m_1 remains within the width of the row. The overhanging part m_2 is reinserted at the left of the row. Both parts are furnished with a corresponding exponent identification. Part m_2 will be selected for addition in row y-1 whereas part m_1 will be added in row y.

The shifted and expanded mantissa row drops row by row through the matrix as a transfer row. Before that, each transfer section is characterized by its exponent which carries the information where the addition has to be executed.

Figure 10 shows the exponent identification of the sections of the transfer rows. Each row of the transfer matrix consists of n transfer sections of length t. Figure 10 denotes the exponent identification t_e (transfer exponent) of these transfer sections of the matrix. [illegible] e_t denotes the exponent of the e.g. least significant digit of a transfer section then this transfer section can be characterized by the exponent identification t_e with $t_e = (e_t - e_o)$ div t.
Before a summand enters the matrix, each transfer section of the summand receives an exponent identification. During [illegible] through the matrix, this exponent identification is then compared with t_e. Equality triggers addition. The lower part of Figure 10 shows how transfer sections of the summand get their exponent identification.
A mantissa with the exponent e (= exponent of its most significant digit) receives the exponent identification $(e - e_o)$ div $t = e_m$ in the most significant transfer section, and exponent identification $e_m - 1$, $e_m - 2$, etc. in the less significant transfer sections.
Figure 10 shows in [illegible] part the two typical cases. (Addition of complete summand in one row resp. in two consecutive rows).

Figure 11 explains the modified adder selection by row identification. This row identification is transfered through the matrix with the transfer row. The adders are triggered off as soon as the row identification and the row index coincide. The row selection switch RS generates two selection signals which activate the adders of the row in question (see Figure 12, too). An activating signal [illegible] via the wire "z-selection" if the row identification equals the row index. An activating [illegible] is sent via the wire "z-1-selection" if [illegible] is the row index.
Then the transfer sections only carry the information (z-1, z-summation) [illegible]
Since the transfer [illegible] only contain positive values the information [illegible] addition or subtraction is addition [illegible] transfer [illegible]
Thus the controlled [illegible] transfer registers with specific information [illegible] each row which leads about to the structure [illegible] in Figure 11.

Figure 12 shows a block diagram for an adder cell. For simplicity the case t = a is selected. The cell contains centrally an "adder/subtractor" and a "partial accumulator section". The right upper corner shows the corresponding transfer register with wires from the next less significant row and to the next more significant row.
Additionally, the transfer register contains a tag register for "z/z-1" identification which indentifies through which selection wire the cell can be activated. The "adder/subtractor" receives the operands from the "partial accumulator section" and in case of selection from the transfer register. Zero is added if no selection takes place. In addition, the carry (positive or negative) arriving from the right is processed during each addition/subtraction and, if necessary, a carry is passed on to the next adder cell on the left. This carry is temporarily stored in an auxiliary register. Figure 15 further shows a control wire which selects the operation (addition/subtraction) as well as a control wire for the read out process (at the bottom of the figure). All control wires traverse the whole row.

Figure 13 is very similar to Figure 6. It shows one row of the summing matrix, but with $t < a$. The Figure is based on the same data format as Figure 6, i.e.: one digit of basis b is described by 4 bits, k = 20 carry digits, l = 14 digits in the mantissa, e1 = -64 and e2 = 64. Furthermore:
Width of AC: a = 4 bytes = 32 bits.
Number of adders in one row c = 4.
Number of rows in SM r = 10.
$L = 20 + 2 \cdot 64 + 2 \cdot 14 + 2 \cdot 64 = 304$ digits per 4 bits = 152 bytes.
Width of the complete summing matrix
$S = a \cdot c \cdot r = 4 \cdot 4 \cdot 10$ bytes = 160 bytes $\geq L$ = 152 bytes.
In this example the width of the transfer registers is smaller than the width a of the adders: $t = \frac{a}{2} = 2$ bytes.
This permits a smaller row width of only c = 4 adders.
The upper part of the Figure shows the position of a summand of $\bar{m} = 2 \cdot l = 14$ bytes at a critical position.

Figure 14 shows another case where the width of the adders differs from that of the transfer registers ($t \neq a$). In the Figure the transfer registers are shown without exponent identification. Dotted lines again indicate transfer wires which bypass the adder in question.

Figure 15 shows a section of a row of the summing matrix with $t \neq a$. Here the case $3t = 2a$ has been selected. It shows how digits of the same transfer register are distributed and added into neighbouring adders.

5. Summation with only one Row of Adders

We now discuss a further variant of the above circuitry for which adders exist only for one row of the summing matrix. The complete structure of this variant is similar to the one before (Figure 16). I.e. the complete circuitry consists of an input adjusting unit, the summing unit with the actual accumulator and a device for carry handling, result row filtering and rounding.

The complete fixed-point word, over which summation takes place, is divided into rows and columns, as before. The transfer width and the adder width, however, must now be identical. The width can be chosen according to the criteria as outlined above. The columns of the matrix shaped summing unit are now completely disconnected, i.e. no transmission of carries takes place between the individual columns of the matrix during the process of summation. The carries occurring during the summation are collected in carry counters and processed at the end of the summation process.

Figure 17 shows the circuit of a "column" of the matrix shaped summing unit. The full "long accumulator" is spread over the various columns of the summing unit. The part allotted to one column is called "accu-memory", see (1) in Figure 17.[5]

To each cell of the accu-memory belongs a carry counter. The collection of carry counters of a column is called "carry-memory", see (2) in Figure 17. In these cells of the carry-memory all carries emerging from the adder/subtractor are collected and incorporated in the result at the very end of the summing process. The individual cells of the carry-memory must be so wide that they can take a carry (positive or negative) from each summand. For a vector length of 128 one needs, for example, 7 bits plus a sign bit resp. an 8 bit number in twos'-complement.

In Figure 17, for example, the column width is 32 bits and the width of the individual carry-memory cells is 16 bits. This allows a correct computation of sums with less than or equal to 32 K summands. The exponent identification (in Figure 17) has a width of e bit; consequently the column has 2^e cells resp. the memory matrix 2^e rows.

During the normal summation process the following happens:

1. The mantissa section MANT, sign sg, and exponent identification EPI reach the input register RI, (3).
2. In the next cycle
   - the memory is addressed through EPI and the accu-part as well as the carry part are transferred to the corresponding section of the register before the sum-mation RBS, (4);
   - the mantissa section, sg, and EPI are also, transferred to the corresponding section of RBS, (5).
3. In the next cycle
   - addition resp. subtraction according to sg is executed in the adder/subtracter (6). The result is transferred to the corresponding section of the register after the summation RAS, (7). According to the carry, the carry-part is adjusted in (8) by +1, -1 or not at a[illegible] [illegible] transferred to RAS, (9);
   - EPI is [illegible] [illegible] to RAS, (10).
4. In the [illegible]
   - [illegible] addresses the memory, and [illegible] together with the carry-[illegible] back into the memory.

Since in e[illegible] [illegible] mantissa section is supplied, [illegible] be pipelined. This means, [illegible] phases need to be active [illegible] be possible therefore, to r[illegible] [illegible] and to write i[illegible] the same [illegible] memory during each machine cycle [illegible] is usual for register memories.

If in [illegible] the same accu- and carry-memory [illegible] addressed, the previously described [illegible] lead to a wrong result, since in the second cycle the result of the just started summing [illegible] should be read, which does not [illegible]. We [illegible] a typical pipeline conflict. [illegible] can be overcome by duplicating [illegible] memory several times which, [illegible].

Therefore, [illegible] an [illegible]ier alternative. We suppose that [illegible] consecutive cycles mantissa sections [illegible] exponent identification arrive. [illegible] following two cases:

a) direct[illegible] [illegible]her;
b) with [illegible] exponent identification in between [illegible] arbitrarily often and mixed.

We first deal with case a).

a) The register EPI [illegible] and EPI of RBS contain [illegible] exponent identification. The two are compared in [illegible] and in case of coincidence the read process from the memory to RBS is locked off in part (13) of the selection unit (12). Instead, the result of the addition of the first of the two consecutive summands is directly transferred to RBS via (14) so that the second summand can immediately be added.

   Furthermore, (15) causes a dummy exponent to be read into EPI of RAS. So, if in the same cycle a further third value with the same exponent identification is transferred to RI the case EPI/RI = EPI/RBS = EPI/RAS is avoided. This case would cause a conflict in the selection unit (12).

   Thus, consecutive summands with the same exponent identification can be added without memory involvement. The intermediate values may be written into the memory or discarded (storage blockade on). Only the last value must be written into the memory via RAS.

We now deal with case b).

b) Three values EPI_1, EPI_2, EPI_3 with $EPI_1 = EPI_3 \neq EPI_2$. In this case EPI/RI and EPI/RAS contain the same exponent identification.The two registers are compared in (16). In the following cycle the contents of RAS is directly transferred to RBS through part (17) of the selection unit (12). The read process from the memory is again suppressed in (13). The intermediate value may be written into the memory. It can also be suppressed.

In this way, any consecutive mantissa sections can

[5] The numbers enclosed in round parentheses in the text indicate in the corresponding Figure that part of the circuitry which is marked with the same number.

be added and the carries colle in the carry
counters.
We now consider the process of r g the result.
The central read control produ ontinuous ad-
dresses so that the accu-memory ead from the
least significant to the mos ificant row.
This sequence is a must he necessary
carry handling. The add es the memory
through the multiplexer (
Wires (19), (20) for transfer of carries lead
from column to column. The carry- s of a column
are fed to the next more signific column. There
they are taken into the mantiss ction of RBS.
To get there the multiplexer is switched
over. The carry, which is re he twos'-com-
plement for convenience, first to be changed
into sign-magnitude-represent d, if neces-
sary, expanded in length 2 next cycle,
the carry is added and a possible
1bit-carry (positive or ansferred to
the unit for preparing t er temporary
storage in RAS. The ab men d carry can
there be stored either in par he RAS-carry
register or in a 2bit auxiliary ter (23).

During the process of re g advisable to
delete the particular st mmediately by
a circuitry part which is This can, for
example, be done by writi to it. If va-
rious scalar products res o be accumu-
lated, the process of rea started until
the computation of the f inished. The
summands are continuousl ed into the
accu-carry-memory.
From the most significant carry part of
the memory is transferre xiliary carry
register, (24) in Figure e, this car-
ry is transferred with a ne cycle via
wire (20) to the least s lumn to have
it available for the rea s of the more
significant row.
The final carry treatment ns a single-
resp. multi-stage pipelin still remain-
ing carries are included . At the end
of this part of the cir ady rows of
the result appear, the ficant ones
first.
In another part of the 6), which is
shown in Figure 18, the the signifi-
cant digits must be fo significant
digit of the more signi er (28) con-
tains the result sign; s (preferably
zero) means positive, la ual 1, deci-
mal 9, hexadecimal F) me . t is advis-
able to initialize both ith zero. The
circuitry for filtering h significant
information now checks presented to
the circuit whether th t one digit
not equal to the sign d ored in the
higher significant regis this is the
case or if there is no s dig g. 1..8 in a
decimal system) at posit 28 the transfer
is enabled for the actu t clock cycle
to fill both registers consecutive
rows. If, however, the already en-
abled in the previous cy st be reen-
abled for one cycle only. circuit (29)
may therefore be descri llowing state
table with entries er enable".

State	output of sign 0	check 1
1	1/0	2/1
2	1/1	3/1
3	1/0	3/1

The transfer into the registers ends if only rows with sign digits follow. Finally, in both registers those rows appear, which contain the mantissa of the floating-point result. One obtains the exponent from the position as well as from the initial address resp. from the number of cycles necessary for reading. Furthermore, the information required for the rounding is easily obtained during output. It serves for a possible adaptation of the result.
The circuitry shown in Figure 17 may be varied to reduce the number of input/output lines, e.g. by transferring the carry count (19) through the MANT inputs. The Figure is intended just to show principles, and not tricky details.

6. Systems with large Exponent Range and further Remarks

Many computers have a very modest exponent range. This is for instance the case for the system /370 architecture. If in the decimal system, for instance, l = 17, e1 = -75 and e2 = 75 the full length $L = k + 2e2 + 2l + 2\ |e1|$ of the registers (see Figure 1 and Figure 2) can more or less easily be provided. Then sums and scalar products of the form (I) and (II) can be correctly computed for all possible combinations of the data by the technique discussed in this paper without ever getting an overflow or an interrupt.
However, there are also computers on the market with a very large exponent range of several hundred or thousand. In such a case it may be costly to provide the full register lengths of $L = k + 2e2 + 2l + 2\ |e1|$ for the techniques discussed in this paper. It is most useful then to reduce the register lengths to the single exponent range and instead of L to choose $L^* = k + e2 + 2l + |e1|$ or even a smaller range $e' \leqq e \leqq e''$ with $e1 < e'$ and $e'' < e2$ and correspondingly $L' = k + e'' + 2l + |e'|$.
Traditionally, sums and scalar products are computed in the single exponent range $e1 \leqq e \leqq e2$. If $|e1|$ and e2 are relatively large most scalar products will be correctly computable within this range or even in $e' \leqq e \leqq e''$. Whenever, in this case, the exponent of a summand in a sum or scalar product computation exceeds this range $e' \leqq e \leqq e''$ an overflow has to be signalled which may cause an interrupt.
In such a case the exponent range could be extended to a larger size on the negative or the positive side or even on both sides. We may very well assume that the necessity for such an extension of the exponent range occurs rather rarely. The supplementary register extensions, which are necessary for the techniques discussed in this paper, could then, for instance, be arranged in the main memory of the system and the summation within the

extended register part may then be executed in software. Such procedure would slow down the computation of scalar products in rather rare cases. But it still always will deliver the correct answer.

We further discuss a few slightly different methods how to execute accumulating addition/subtraction and the scalar product summation on processors with large exponent range.

On a more sophisticated processor the exponent range covered by the summing matrix could even be made adjustable to gain most out of this special hardware. This could be done by an automatic process of three stages:

1. A special vector instruction analyzes the two vectors and computes the exponent range that covers most of the summands or products of the vector components. This step may be discarded if the best range is already known.
2. The summing matrix gets properly adjusted to the range found in 1. and in a vector instruction the fitting part of the summand or products is accumulated into the summing matrix. If a summand or product does not fit into it it can be dealt by one of the two alternatives:
   a) Interrupt the accumulation and add that summand or product by software to the not covered extended parts of the accumulator which resides in main memory.
   b) Do not interrupt the accumulation, but discard this summand or product and mark this element in a vector flag register. Later the marked elements are added by software to the extended parts of the accumulator. This second way avoids interrupting and restarting the pipeline and will thus lead to higher performance than a).
3. In a final step the content of the summing matrix part of the accumulator is properly inserted between the extended parts to get the complete result in form of a correspondingly long variable in main memory.

Another cure of the overflow situation $e \notin [e', e'']$ may be the following: Summands with an exponent e, which is less than e', are not added, but gathered on a "negative heap". Similarily summands with an exponent, which is greater than e", are gathered on a "positive heap". The negative and the positive heap may consist of a bit string or a vector flag register where each summand or vector component is represented by a bit. This bit is set zero if the summand was already added. It is set 1 if the component belongs to the corresponding heap. After a first summation pass over all summands the computed sum is stored. Then the positive and/or negative heap is shifted into te middle of the exponent range $e' \leq e \leq e''$ by an exponent transformation and then added by the same procedure. After possibly several such steps the stored parts of the sum are put together and the final sum is computed. In many cases it will be possible to obtain the final result without summing up the negative heap.

Another possibility to obtain the correct result with a reduced register length L' = k + e' + 21 + e" is the following: The process of summation starts as usual. As soon as the exponent e of a summand exce [illegible] rar [illegible] ', e"] an exponent part is bui [illegible] ich [illegible] rets the digit sequence of L' [illegible] ry [illegible] ntissa of a normalized floatin [illegible] e normalization, in general, wi [illegible] a [illegible]. Then a "positive heap" is no [illegible] y. And in most cases it will be pos [illegible] n the correct rounded result witho [illegible] g [illegible] possibly still necessary "negat [illegible] od computes all accumulating s [illegible] cts correctly without consider [illegible] eg [illegible] aps as long as less than e" - e' [illegible] an [illegible] e negative heap can only influen [illegible] le [illegible] significant digits of L'.

The reductio [illegible] ful [illegible] mulator length L to a smaller si [illegible] ma [illegible] e exponent under- or overflows i [illegible] on processes. This always makes [illegible] nt [illegible] ing routine necessary. Whatever [illegible] procedure represents a trade off b [illegible] rd [illegible] expenditure and runtime.

A rather pri [illegible] er [illegible] dling would consist in a tradition [illegible] or [illegible] the positive and negative heap. I [illegible] is [illegible] sage should be delivered to the [illegible] t [illegible] result is probably not precise.

In the conte [illegible] g [illegible] languages the accumulator of l [illegible] = [illegible] " + 21 + e' represents a new data t [illegible] wh [illegible] le called precise. As long as no e [illegible] un [illegible] or overflow occurs $(e' \leq e \leq e'')$ ad [illegible] it on of [illegible] les of type real, of products of [illegible] ri [illegible] as well as of scalar products of [illegible] cto [illegible] to a variable of this type can pr [illegible] el [illegible] ated and it is error free. Accumu [illegible] f [illegible] ariables, products or scalar produ [illegible] a [illegible] ble of type precise is associative. [illegible] ndependent of the order in which [illegible] added.

Vectorproces [illegible] the fastest computers which are pr [illegible] av [illegible] al le. Their main field of application [illegible] sc nti [illegible] computation. It should be natural [illegible] v cto [illegible] essors compute vector operations c rr ctly. [illegible] vector operations consist basically of the [illegible] ponentwise addition and subtraction, the compo [illegible] wise multiplication and the scalar product. The [illegible] nplementation of highly accurate ve [illegible] ad [illegible] subtraction and componentwise mul [illegible] ic ti [illegible] longs to the state of the art. The [illegible] pu ti [illegible] f accurate scalar products has be [illegible] ea [illegible] w [illegible] n this paper.

Due to their [illegible] h [illegible] ee [illegible] computation, vectorprocessors must, [illegible] wever, also be able to support an automatic er [illegible] anays [illegible] sp. verification of the computed res [illegible] o [illegible] to achieve this it is necessary th [illegible] all op [illegible] ions, mentioned above, such as componentwise [illegible] ition/subtraction, componentwise multiplication and scalar products can optionally be called with several roundings, in particular with the monotone downwardly directed rounding, the monotone [illegible] wardly directed rounding and the roun [illegible] to the least including interval. We do not [illegible] uss the implementation of these roundings he [illegible] It belongs to the state of the art. For fur [illegible] information we refer to the literature.

Finally, we re [illegible] rk that the methods and procedures outlined in th [illegible] paper are also suitable to add up sums of produc [illegible] correctly which consist of more than two factors, for example

$$\sum_{i=1}^{n} a_i \cdot b_i * \quad .$$

7. Application to Mul[illegible]ple P[illegible]n Arithmetic

We show in this chapter that [illegible]ential parts of multiple precision arithm[illegible]n easily be executed with high speed if a f[illegible]calar product unit is available.

We consider

1. Double Precision Arithmeti[illegible]

1.1 Sum and Difference

It is clear that sums [illegible] two o[illegible]ble precision summands a + b or a + b [illegible] c .. [illegible]an be accumulated. The same holds f r sur[illegible]ctors or matrices.

1.2 Product

If a product a • b of tw doul[illegible]sion factors a and b has to be comp[illegible]ted, [illegible]actor can be represented as a sum of two si[illegible]recision numbers $a = a_1 + a_2$ and $b = b_1 + b$[illegible]ere a_1 and b_1 represent the first (hi[illegible]her si[illegible]ant) l digits and a_2 and b_2 repres[illegible] the l[illegible]ower significant) l digits of a [illegible]. The [illegible]lication then requires the execution [illegible] a sc[illegible]duct:

$$a \cdot b = (a_1 + a_2) \quad {}_1 + b$$

$$a_1b_1 + a_1b_2 + a_2b_1 \quad a_2b_2 \tag{1}$$

where each summand is [illegible] doul[illegible]cision. These can be added by the te[illegible] de[illegible]ped in this paper.

Similarly, products of [illegible]re t[illegible] factors can be computed. As in (1) products [illegible] double precision numbers are expressed b[illegible]calar product of single precision [illegible]ers. [illegible] right hand side of (1) each [illegible] is [illegible]le precision number which can [illegible]resse[illegible] sum of two single precision n[illegible] In t[illegible]f a product of four double preci[illegible] numb[illegible] leads to the following formulas, w[illegible] are [illegible]planatory.

$$a{\cdot}b{\cdot}c{\cdot}d = (a{\cdot}b) \; (\quad) =$$

$$\sum_{i=1}^{8} a^i \cdot \sum_{i=1}^{8} c^i = \quad \sum_{j} a^i \quad {}^j$$

$$\text{with } a \cdot b = \sum_{i=1}^{8} a^i \quad \cdot d = \sum^{8} \quad .$$

Thus a•b•c•d can b[illegible]d [illegible] of 64 products of two singl[illegible]r[illegible]ion [illegible]ach.

The case of product[illegible] o [illegible]uble precision matrices is a li[illegible] m[illegible]cult. But it can, in principle, b[illegible] reat[illegible]arily. If a product of two double [illegible]s has to be computed the two matric[illegible] ar[illegible] rst [illegible]presented as sums of two single precision matrices. Multiplication of these sums then leads to a sum of products of single precision matrices:

$$a \cdot b = (a_1 + a_2)\,(b_1 + b_2) =$$

$$a_1b_1 + a_1b_2 + a_2b_1 + a_2b_2 \tag{2}$$

Each component of the products on the right hand side of (2) is computed as a scalar product. Thus each component of the product matrix a • b consists of a sum of scalar products which itself is a scalar product.

In case of matrix products, which consist of more than two double precision matrix factors, one has to take into account that the components of (2) may already be pretty long. They may consist of 10 or 20 consecutive digit sequences of single precision lengths. These sums of single precision matrices then have to be multiplied with other such sums, which leads to a sum of matrix products. Each component of this sum can be computed as a scalar product of single precision numbers.

2.

Arithmetic of triple precision is a special case of quadruple precision arithmetic.

3. Quadruple Precision Arithmetic

3.1 Sum and Difference

Each summand of quadruple precision can be represented as a sum of two double precision summands. Thus sums of two or more quadruple precision summands can be added as expressed by the following formulas:

$$a + b = a_1 + a_2 + b_1 + b_2$$

$$a + b + c + \ldots + z =$$
$$a_1+a_2+b_1+b_2+c_1+c_2 + \ldots + z_1 + z_2 \; .$$

Sums of quadruple precision vectors or matrices can be treated correspondingly.

3.2 Products

Each quadruple precision number can be represented as a sum of four single precision numbers $a = a_1 + a_2 + a_3 + a_4$. Multiplication of such sums requires the execution of a scalar product:

$$a \cdot b = (a_1+a_2+a_3+a_4) \cdot$$

$$(b_1+b_2+b_3+b_4) = \sum_{i=1}^{4} \sum_{j=1}^{4} a_i \cdot b_j \tag{3}$$

Similarily, products of more than two quadruple precision factors can be computed. We indicate this process by the following formulas, which are self-explanatory.

$$a{\cdot}b{\cdot}c{\cdot}d =$$

$$(a{\cdot}b)\;(c{\cdot}d) = \Big(\sum_{i=1}^{4} \sum_{j=1}^{4} a_i\, b_j\Big)\;\Big(\sum_{i=1}^{4} \sum_{j=1}^{4} c_i d_j\Big) =$$

[6] High speed scientific [illegible]putat[illegible]on [illegible] usually done in the long data f[illegible] D[illegible]ecision here means the double [illegible] le[illegible] that format. If the usual long [illegible] is a[illegible]alled double precision our d[illegible] [illegible]esponds to quadruple or exten[illegible]

$$= \left(\sum_{i=1}^{32} a^i\right)\left(\sum_{j=1}^{32} c^j\right) = \sum_{i=1}^{32}\sum_{j=1}^{32} a^i c^j \; . \qquad (4)$$

There the 16 double precision summands $a_i b_j$ and $c_i d_j$ of the two factors of (4) are each represented as sums of two single precision-numbers. This leads to the product of the two sums over 32 single precision numbers a^i resp. c^j in the next line.

If a product of two quadruple precision matrices is to be computed each factor is represented by a sum of four single precision floating-point matrices as in (3).

Multiplication of these sums leads to a sum of matrix products. Each component of these matrix products is computed as a scalar product. The sum of these scalar products is again a scalar product.

It was the intention of this section to demonstrate that with a fast accumulating addition/subtraction or scalar product unit a big step towards multiple precision arithmetic, even for product spaces, can be done.

8. Literature

[1] U. Kulisch: Grundlagen des Numerischen Rechnens - Mathematische Begründung der Rechnerarithmetik, Bibliographisches Institut, Mannheim 1976

[2] U. Kulisch and W.L. Miranker: Computer Arithmetic in Teory and Practice, Academic Press 1981

[3] U. Kulisch and W.L. Miranker: The Arithmetic of the Digital Computer: A New Approach, SIAM-Review, March 1986, pp. 1-40

[4] IBM System /370 RPQ, High Accuracy Arithmetic, Publication Number SA 22-7093-0

[5] High Accuracy Arithmetic, Subroutine Library, General Information Manual, IBM Program Number 5664-185

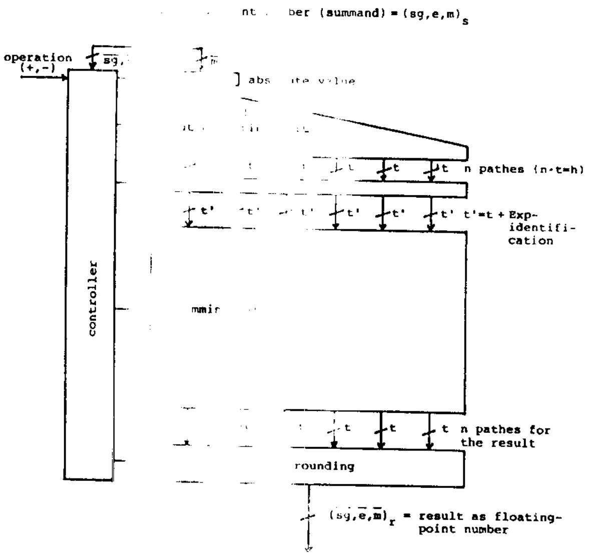
[6] High Accuracy Arithmetic, Subroutine Library, Program Description and User's Guide, IBM Program Number 5664-185, Publication Number GC 33-6163

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

[8] G. Bohlender and T. Teufel: BAP-SC: A Decimal Floating-Point Processor for Optimal Arithmetic, to appear in: Computer Arithmetic, Scientific Computing and Programming Languages (E. Kaucher, U. Kulisch, Ch. Ullrich, Eds), B.G. Teubner, 1987

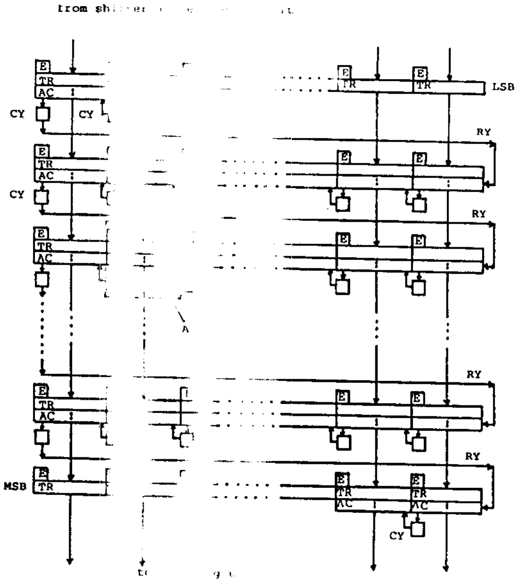
[9] Arithmos Benutzerhandbuch, SIEMENS AG., Bestell-Nr.: U 2900-J-Z 87-1

For a supplementary bibliography see the literature listed in [3].



ber (summand) = (sg,e,m)s
operation (+,-)
sg
abs value
n pathes (n·t=h)
t
t'
t'=t + Exp-identification
controller
n pathes for the result
rounding
(sg,e,m)r = result as floating-point number

Figur[illegible] ructu[illegible] [illegible]hole circuitry
lue d[illegible] [illegible]umber of figures of 'value'



E: tag-re[illegible]ste[illegible] expon[illegible] id[illegible]ification
CY: carry [illegible]is[illegible]
TR: transf[illegible] re[illegible]ter
A: adder
AC: accumul[illegible]ting r[illegible] [illegible]n-cluding arithme[illegible] [illegible]ircuitry

LSB: least significant bit
MSB: most significant bit
[illegible]: number of rows
RY: row carry

each row co[illegible]ains c a[illegible]ers of a digits and n transfer registers of t digits [illegible].e. h=[illegible]·a [illegible] n·t (here: n=c and t=a)

Figure 5: [illegible]cture o[illegible] th[illegible] s[illegible]ming matrix

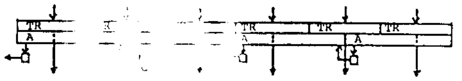


Figure [illegible]: Tr[illegible]nsf[illegible] reg[illegible]rs and adders of different wi[illegible]th [illegible] a, [illegible] instance: t=4, a=6). The transf[illegible] regi[illegible]ters are represented without tag-fi[illegible]ds [illegible]or exponent identification

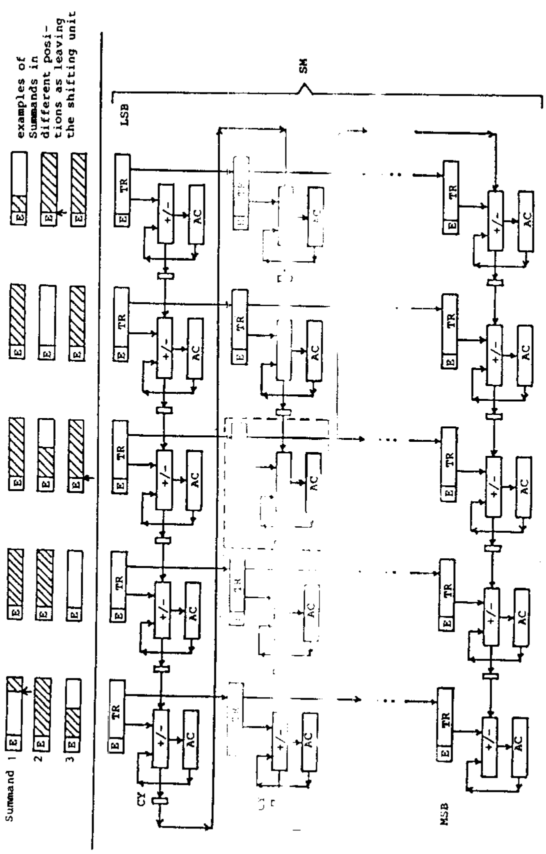

examples of Summands in different positions as leaving the shifting unit
Summand 1
2
3
E
SM
LSB
MSB
TR
+/-
AC
CY

Figure 6: Summing matrix SM consisting of h=c·r independent adders A
E: tag-register for exponent identification, TR: transfer register,
AC: accumulator register, CY: carry, ↑: most significant digit of summand

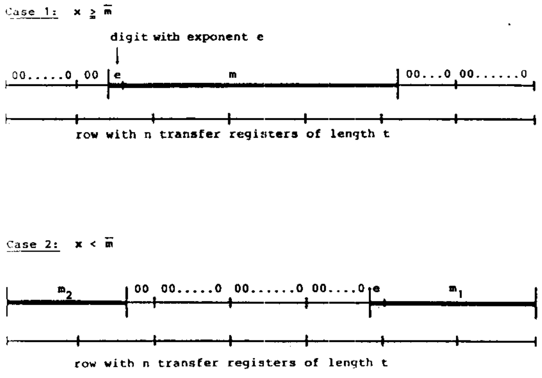
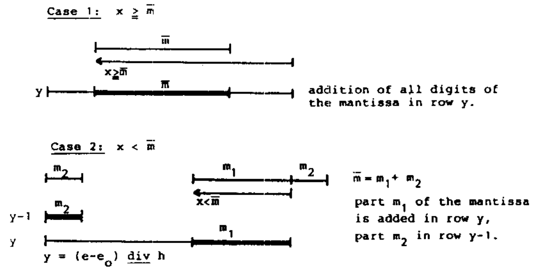


Figure 8: Task of the shift unit

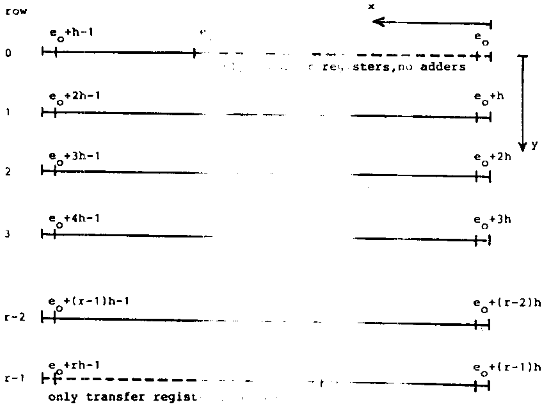

Case 1: x ≥ m̄
m̄
x≥m̄
y
m̄
addition of all digits of the mantissa in row y.
Case 2: x < m̄
m2
m1
m2
m̄ = m1 + m2
x<m̄
y-1
m2
part m1 of the mantissa is added in row y, part m2 in row y-1.
y
m1
y = (e-e0) div h

e: exponent of the most significant digit of the mantissa

$\bar{m}$: length of mantissa, number of digits of the mantissa

h: number of digits of a row of the summing matrix

Figure 9: Description of the shift process



e_o: reference poi[illegible] [illegible] exponent in the matrix

e_1: exponent of t[illegible] [illegible] digit of the adders

e_m: exponent of t[illegible] [illegible] digit of the adders

h: length of row [illegible] b), h = c · a
$L = e_m - e_1 + 1$, S[illegible]

e: exponent of a [illegible]

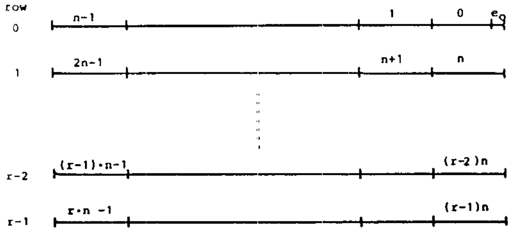
$e - e_o$: distance to th[illegible] end of the matrix

$y = (e - e_o)$ div h: row [illegible] [illegible] exponent e

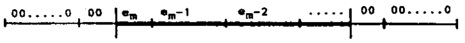
$x = (e - e_o)$ mod h: dis[illegible] [illegible]nificant end of row y

Figure 7: Exponent c[illegible] [illegible]its in the summing matrix

Exponent identification t_e of the transfer sections of the matrix:



A mantissa transfer row with mantissa exponent e gets the following exponent identifications in the corresponding transfer sections: $e_m, e_{m-1}, \ldots$.



with $e_m = (e - e_o)$ div t

This is independent of the location in the transfer row. For instance:

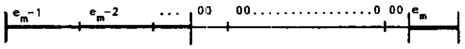
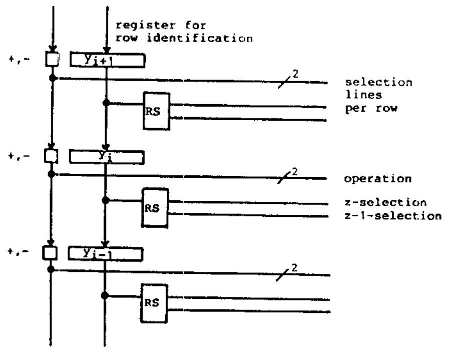
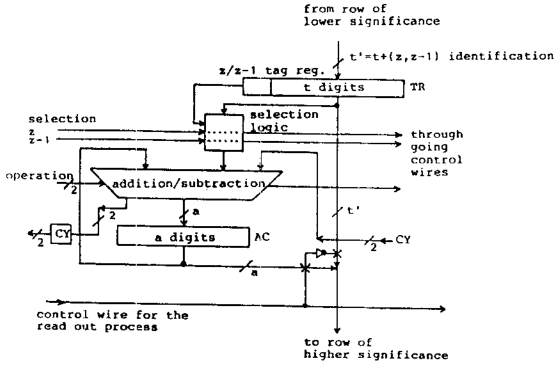


Figure 10: Exponent identification of the sections of the transfer rows



i: index of the summands (i=1,...,m)
RS: row selection

Figure 11: Simplified adder selection by row identification y_i



CY: carry with sign
selection logic: add/subtr, if (selection-z) and (tag-z) or (selection-(z-1) and (tag-(z-1) add/subtr. zero else

Figure 12: Structure of a section of the matrix row y, for a = t

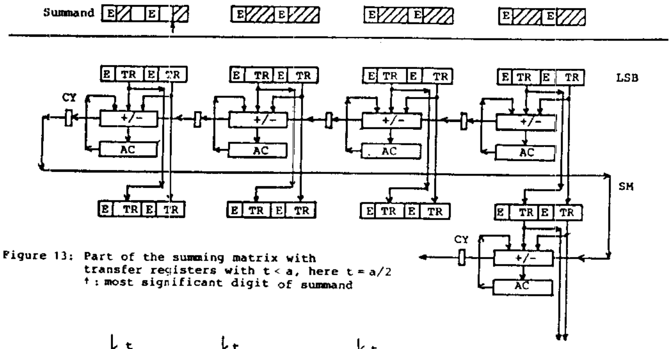


Figure 13: Part of the summing matrix with transfer registers with t < a, here t = a/2
†: most significant digit of summand

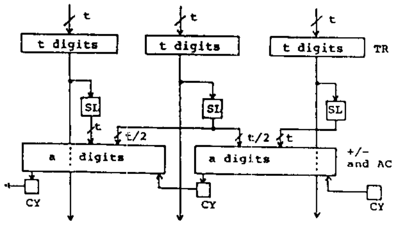
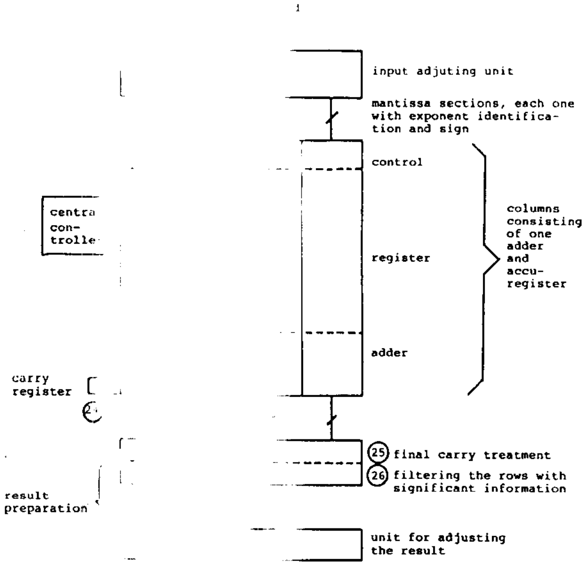


Figure 15: Structure of a section with several adder / subtracters and transfer registers (for example 2a = 3t) in a simplified representation without tag-fields for exponent identification and control lines.
SL: selection



Figur· ·mming unit ·f adders

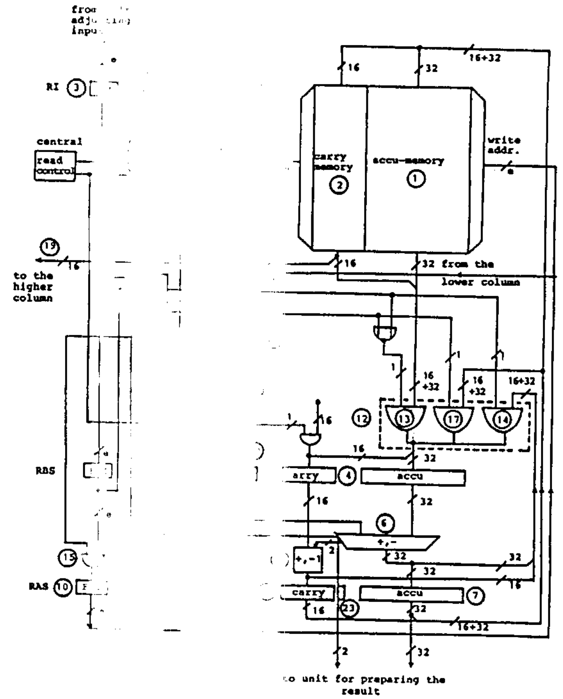
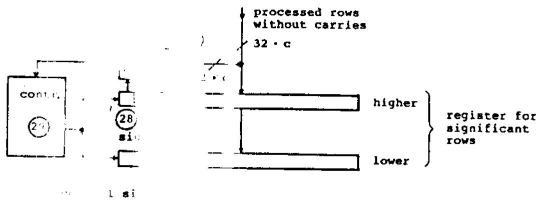


Figure 17:



Fi jur· rows with significant
of columns