High Speed Associative Accumulation of Floating-point Numbers and Floating-point Intervals*

Ulrich Kulisch and Gerd Bohlender
Institut für Angewandte und Numerische Mathematik,
Karlsruher Institut für Technologie
D-76128 Karlsruhe, Germany
Ulrich.Kulisch@kit.edu,Gerd.Bohlender@kit.edu

Abstract

Floating-point arithmetic is the tool that is most commonly used for scientific computation. It is well known that conventional floating-point addition is not associative, i.e., for floating-point numbers a, b, c in general $a \boxplus (b \boxplus c) \neq (a \boxplus b) \boxplus c$. This, however, is a relict of old and poor technologies. We show in this paper, that n floating-point numbers $a_i, i = 1, 2, \ldots n$ can be added in a way that produces a result that is independent of the order in which the summands are added. This method does not round after each single addition. It accumulates the summands exactly into a modest fixed-point register on the arithmetic unit and rounds the sum to a floating-point number only once at the very end of the accumulation. By pipelining, the sum can be computed in the time the processor needs to read the summands, i.e., the sum is computed at extreme speed. The method presented here has the potential of replacing conventional methods for the implementation of floating-point addition and subtraction. It very naturally can be applied to the accumulation of n floating-point intervals. Here again the result is independent of the order in which the intervals are added.

1 Introduction

Conventionally, computing speed is measured in flops (floating-point operations per second). This assumes that the entire computation is reduced to the four elementary floating-point operations.

Beyond the four operations $+, -, \cdot$, and / several old mechanic calculators provided a number of compound operations such as *accumulate* or *multiply and accumulate*. This allowed a continuous accumulation of numbers and of products of numbers into different positions of a wide fixed-point register. This fixed-point

*Submitted: December 30, 2015; Revised: April 17, 2016; Accepted: April 21, 2016.

addition was the fastest way to use the computer. It was applied as often as possible. No intermediate results needed to be written 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 underflow 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 required, a rounding was performed only once at the very end of the accumulation. The method discussed here can be traced back to the old computer by G. W. Leibniz (1685). It shows an excellent feeling of the old mathematicians for efficiency in computing.

It is high time that such fast compound arithmetic operations are also offered on electronic computers. Old technologies (around 1970) did not allow this, and this old state of the art still determines the present computer architecture. The fact that neither the IEEE floating-point arithmetic standard 754 nor the standard IEEE 1788 for interval arithmetic provides and requires the accumulation and the dot product of two floating-point vectors as elementary arithmetic operations displays this tragic situation. Modern technology is very powerful. A modest fixed-point register for the accumulation of floating-point numbers and of simple products of floating-point numbers on the arithmetic unit can be provided very easily. It would be rewarded by extreme speed and increased accuracy.

Hardware implementations of the exact dot product (EDP) at Karlsruhe in 1993 [2] and at Berkeley in 2013 [3] show that the EDP can be computed in about 1/5th to 1/6th of the time needed for computing a possibly wrong dot product in conventional floating-point arithmetic. Influenced by the need for fast and exact computation of the dot product, modern processors by Intel and IBM provide register memory of 16 K bits on the arithmetic unit [8]. Hence, they differ from the traditional von Neumann architecture. This difference may be small but it is essential for high speed computation of accumulations and of the EDP. Access to register memory is much faster than access to main memory. About 1 K bits suffice for a real dot product, 2 K bits for an interval dot product, and 4 K bits for a complex interval dot product. One half of these bits suffice for high speed accumulation of floating-point numbers, of floating-point intervals, and of complex floating-point intervals, respectively. So the 16 K bits still give room for a number of interrupts.

Using a software routine for a correctly rounded dot product as alternative for a hardware implemented EDP leads to a comparatively slow process. A correctly rounded dot product is built upon a computation of the dot product in conventional floating-point arithmetic. This is already 5 to 6 times slower than an EDP. High accuracy is obtained by clever and sophisticated mathematical considerations which all together make it slower than the EDP by more than one magnitude. High speed and accuracy, however, are essential for acceptance and success of interval arithmetic.

Iterative refinement or defect correction methods are basic ingredients for success of numerical and of interval analysis. It is the EDP which makes these techniques fast and successful. For details, see Chapter 9 in [19]. A hardware

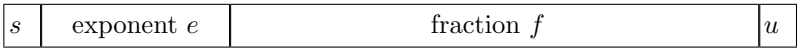

Figure 1: The floating-point number format.

supported EDP brings speed and accuracy to floating-point interval arithmetic. With it interval arithmetic becomes a fast and exception-free computing tool! High speed floating-point interval arithmetic carries the potential of replacing floating-point arithmetic in many applications.

The method for high speed accumulation of floating-point numbers and floating-point intervals presented here is a specialization of techniques for fast and exact computation of dot products of two floating-point vectors[1] – used in the XSC-languages since 1980 [9, 10, 11, 23, 24] – to the accumulation of simple floating-point numbers. This paper shows how simple and fast a set of floating-point numbers and floating-point intervals can be accumulated exactly. Of course, exact accumulation of floating-point numbers can be obtained as a particular case of an EDP. However, a special routine for it makes it even simpler and faster. It may even replace conventional implementations of the elementary floating-point operations addition and subtraction.

We now develop circuitry for fast and exact accumulation of floating-point numbers. We do this for a data format which is close to double precision. Having in mind that the target of our study is floating-point interval arithmetic, we avoid the huge exponent ranges of the IEEE 754 arithmetic standard. We choose a format that is more appropriate for floating-point interval arithmetic.

2 Basic Assumptions

Motivated by the book *The End of Error* by John Gustafson [4], which extends arithmetic for closed real intervals to just connected sets[2] of real numbers, we develop the method for a particular data format. The study, however, is not restricted to this format; the basic ideas can be applied to other formats as well. We assume that the floating-point number is represented by a 64 bit word as shown in Figure 1. Here one bit is used for the sign s, 9 bits are used for the exponent, 53 bits for the fraction and one bit for the ubit u. As usual the leading bit of the fraction of a normalized binary floating-point number is not stored, so the fraction actually consists of 54 bits. For the exponent, $e1$ subnormal numbers with a denormalized mantissa are permitted. The ubit u is used to indicate whether the number is exact ($u = 0$) or inexact ($u = 1$). The ubit is very useful for advanced interval arithmetic. If it is 0, the corresponding bracket is closed, and it is open if the ubit is 1.

If $\mathbb{F} = \mathbb{F}(b, l, e1, e2)$ denotes a floating-point system with base b, l digits in the mantissa, least exponent $e1$, and greatest exponent $e2$, we have $b = 2$,

[1] For details see Chapter 1 in [17], or Chapter 8 in [19].

[2] These can be closed, open, half-open, bounded or unbounded.

$l = 54$, $e1 = -255$, and $e2 = 256$. For numerical examples, we occasionally use base $b = 10$. We are now going to compute the sum

$$s := \sum_{\nu=1}^{n} a_\nu = a_1 + a_2 + ... + a_n, \quad a_i \in F(2, l, e1, e2),\ i = 1, 2, \ldots n. \qquad (1)$$

All floating-point numbers of this system can be taken into a fixed-point register of length $L = e2 + l + |e1| = 256 + 54 + 255 = 565$ bits without loss of information, see Figure 2. The size of this register is independent of the number of summands that are to be added: If one of the summands has an exponent 0,

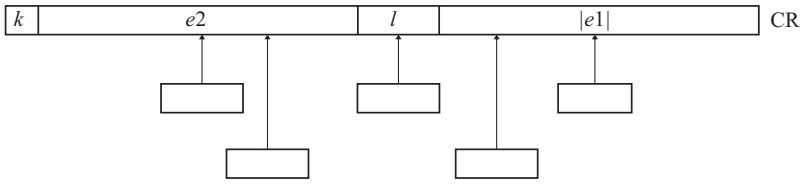

Figure 2: Complete register with long shift for exact accumulation of floating-point numbers.

its mantissa can be taken into a register of length l. If another summand has exponent 1, it can be treated as having an 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 $e2$ and $|e1|$. Hence, any summand can be treated as having an exponent 0 and be taken into a fixed-point register of length $e2+l+|e1|$ without loss of information. The contents of this register can be zero, positive or negative. This information is held in the status register. We stress the fact that subnormals (gradual underflow) are included in the representation. No particular restrictions on the digits of the mantissa are made in case of an exponent $e1$.

If the register shown in Figure 2 is built as an accumulator with an adder, all summands can be added without loss of information. To accommodate possible overflows, it is convenient to provide a few, say k, more digits of base b on the left. These are used to count the number of overflows that occur during the accumulation. With such an accumulator, every sum (1) can be added without loss of information. As many as b^k overflows may occur and be accommodated without loss of information. In the worst case, presuming every sum causes an overflow, we can accommodate sums with $n \leq b^k$ summands. Since every sum of floating-point numbers can be exactly accumulated in this register, we call it a *Complete Register*, CR for short.

Assuming here that k is sufficiently large, $k = 50$ for instance, we get a register size of $k + e2 + l + |e1| = k + 565$ bits. This is a little more than 9 words

of 64 bits. The summands are shifted to the proper position and added. The final sum has to be in the single exponent range $e1 \leq e \leq e2$, otherwise it is not representable as a floating-point number, and the problem has to be scaled. Once more, we stress the fact that the size of this register only depends on the floating-point format. It is indeed independent of the number n of summands that are to be accumulated.

The size of the complete register would grow with the exponent range of the data format in use. If this range should be extremely large, as for instance in the case of an extended precision floating-point format, only an inner part of the register would be supported by hardware. The outer parts which are used very rarely could be simulated in software. The long data format of the IBM System/370 architecture covered a range of about 10^{-75} to 10^{75}, which is very modest. This architecture dominated the market for more than 25 years, and most problems could conveniently be solved with machines of this architecture within this range of numbers.

3 Fast Carry Resolution

The addition of a floating-point number into the complete register CR seems to be slow. It appears to require a long shift, and it may produce a carry propagation over several hundred bits. We now discuss a very fast solution for both problems for the data format outlined in Fig. 1. As soon as the principles are clear, the technique can easily be applied to other data formats as well. The mantissa here consists of $l = 54$ bits. We assume additionally that the complete register CR is subdivided into words of 64 bits. The mantissa of the summand touches at most two consecutive 64-bit words of the complete register, which are determined by the exponent of the summand. A shifter then aligns the 54 bit summand into the correct position for the subsequent addition into the two consecutive words of the CR. This addition may produce a carry (or a borrow in case of subtraction). The carry is absorbed by the next more significant 64-bit word of the complete register in which not all digits are 1 (or 0 for subtraction), as shown in Figure 3(a). For fast identification of this word, two information bits or flags are appended to each complete register word, as shown in Figure 3(b). One of these bits, the *all bits 1* flag, is set to 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 to 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 the summand into two consecutive words of the CR, a search is started for the next more significant word 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 bit 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 CR word contents when fetching a word from the CR, Figure 3(b). Borrows are handled in an analogous way.

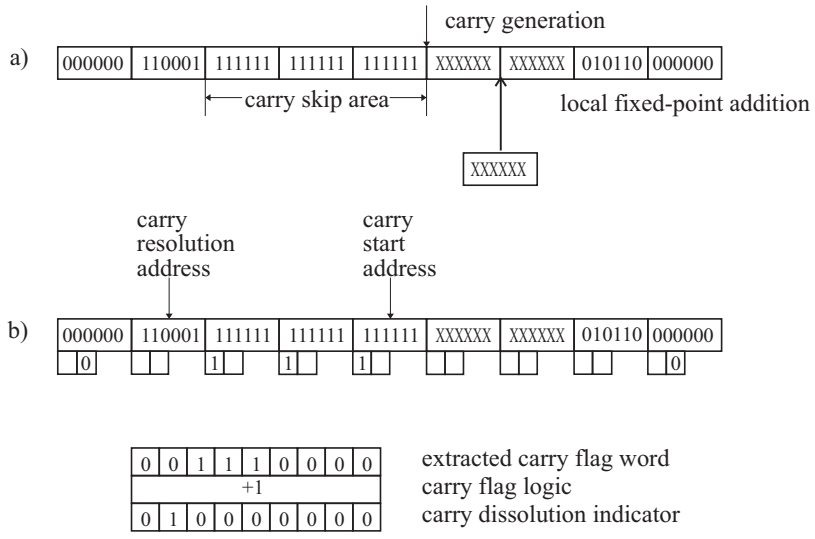

Figure 3: Fast carry resolution.

This carry handling scheme allows a very fast carry resolution. The exponent of the summand delivers the address for its addition. By using the flags, the carry word can be incremented/decremented simultaneously with the addition of the summand. If the addition of the summand produces a carry, the incremented/decremented carry word is written into the CR. Otherwise, nothing is changed.

Other, perhaps simpler methods for fast carry resolution are possible. We sketch here a second solution where adder equipment is provided for the entire width of the complete register CR. For more details in case of the dot product computation, see Chapter 1 in [17] or Chapter 8 in [19].

An adder for the entire width of the CR certainly is too slow to perform an addition in a single cycle. Hence, we subdivide the CR into shorter pieces (of 16, 32, 64 bits). The width of these segments is chosen such that a parallel addition can be performed in a single cycle. Now each one of these adder segments can produce a carry. These carries are written into carry registers between adjacent adders, as shown in Figure 4. If a single addition has to be performed, these carries have to be propagated straight away. If more than one summand has to be added, the carries are added to the next more significant adder in the next addition cycle together with the next summand. Only at the very end of the accumulation, when no more summands are coming, carries may have to be eliminated. However, the summands consist of 54 bits only. So during an

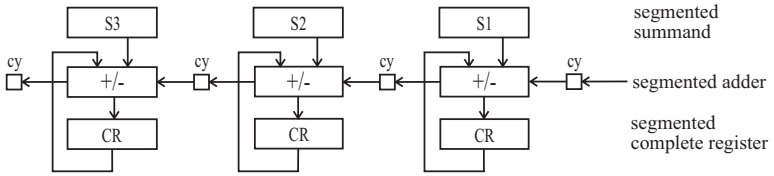

Figure 4: Segmented addition.

addition of a summand carries are only produced in a small part of the complete register CR. The carry elimination, on the other hand, takes place during each step of the accumulation whenever a carry is left. Hence 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 segmenting the adder enables it to read and process a summand in each cycle.

In each step of the accumulation, an addition only has to be activated for a small number (two) of the adder segments and for those adders where a non zero carry is waiting to be absorbed. This adder selection can reduce the power consumption for the accumulation step significantly.

4 Accumulation of Floating-point Numbers

We assume in this section that the data are stored in the double precision 64-bit format as outlined in Section 2. There the mantissa has 54 bits.

A central building block is the complete register CR. It is a fixed-point register wherein any sum of floating-point numbers can be represented without error. It allows accumulation of any finite number of floating-point numbers exactly or with a single rounding at the very end of the accumulation. As shown in Section 2, it consists of about 9 words of 64 bits.

The 54-bit summand is added to two consecutive words of the CR. These are determined by the exponent of the summand. After the addition, these two words are written back into the same two CR words that the portion has been read from.

A 54- out of 118-bit shifter is used to align the summand onto the relevant word boundaries. Figure 5 shows a block diagram for the accumulation. The addition can be executed by a 54-bit adder. It may cause a carry. The carry is absorbed by incrementing (or decrementing in the case of a borrow) a more significant word of the CR as determined by the carry handling scheme. The entire process can be pipelined. The pipeline is sketched in Figure 6. The exponent of the summand consists of 9 bits. The six low order (less significant) bits are used to perform the shift. The 3 more significant bits deliver the CR address to which the summand has to be added, so the originally very long shift

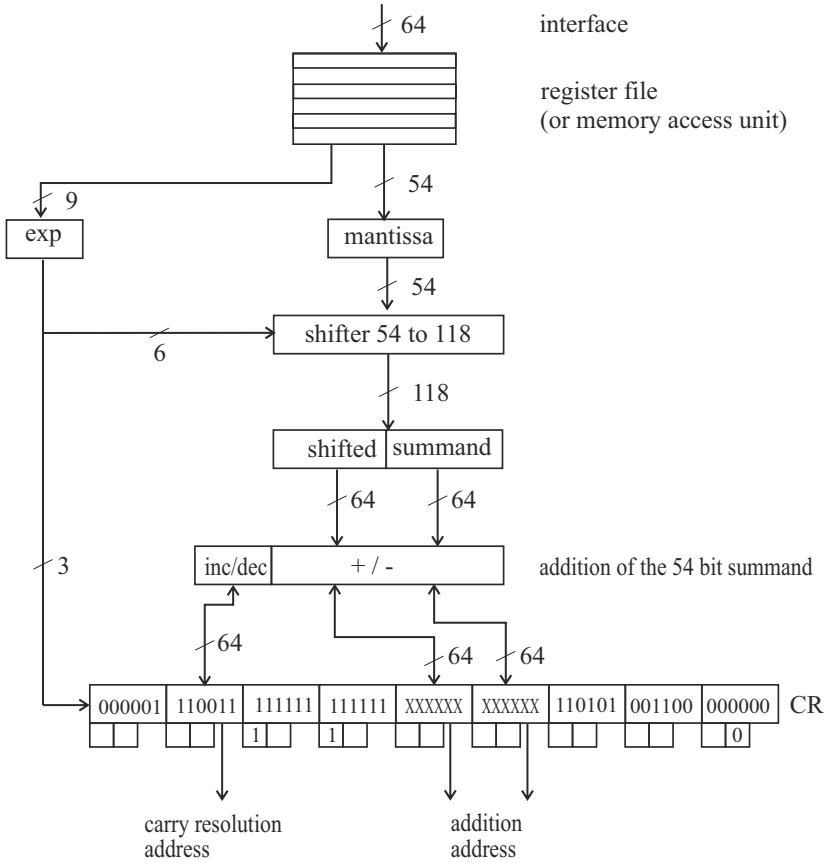

Figure 5: Block diagram for fast and exact accumulation of floating-point numbers.

is split into a short shift and an addressing operation. The shifter performs a relatively short shift operation. The addressing selects the two words of the CR for the addition.

The carry logic determines the word that absorbs the carry. All these address decodings can be hard wired. The result of each addition is written back into the same CR words to which the addition has been executed. The two carry flags appended to each accumulator word are indicated in Figure 5. In practice the flags are kept in separate registers.

We stress that in the circuit just discussed, virtually no unoverlapped computing time is needed for the arithmetic. In the pipeline, the arithmetic is performed in the time that is needed to read the data into the accumulation unit.

We consider a computer which is able to read data into the arithmetic logical

unit in portions of 64 bits. If the 54-bit summand spans two consecutive 64-bit words of the complete register, a closer look shows that the 10 least significant bits of those two words are never changed by addition of the summand. Thus the adder needs to be 54 bits wide only. Figure 5 shows a sketch for the accumulation of a summand.

In the circuit, a 54- to 118-bit shifter is used. Sophisticated logic is used for the generation of the carry resolution address, since this address must be generated very quickly. Only one address decoder is needed to find the starting address for an addition. The more significant part of the summand is added to the contents of the CR word with the next address. A tree structured carry logic now determines the CR word which absorbs the carry.

Figure 6 sketches a pipeline for this kind of addition. In the figure we assume that two machine cycles are needed to decode and read one 64-bit word into the accumulation unit. In the block diagram for the accumulation in Figure 5, data busses of 64 bits are frequently used. In a pipeline, the arithmetic is performed

cycle	**read**	**shift**	**add/subt**
	address decoding a_i		
	read a_i		
	address decoding a_{i+1}	shift a_i	
	read a_{i+1}	address decoding CR	
	address decoding a_{i+2}	shift a_{i+1}	add/subt a_i
	read a_{i+2}	address decoding CR	store result and flags
	address decoding a_{i+3}	shift a_{i+2}	add/subt a_{i+1}
	read a_{i+3}	address decoding CR	store result and flags
	address decoding a_{i+4}	shift a_{i+3}	add/subt a_{i+2}
	read a_{i+4}	address decoding CR	store result and flags

Figure 6: Pipeline for the accumulation of floating-point numbers.

in the time which is needed to read the data into the accumulation unit. Here, we assume that with the necessary address decoding, this requires two cycles for the 54-bit summand.

An adder width of a power of 2 may simplify shifting as well as address decoding. The lower bits of the exponent of the summand control the shift operation while the higher bits are used directly as the starting address for the accumulation of the summand into the CR.

The two flag registers appended to each complete register word are indicated in Figure 5. In practice, the flags are kept in separate registers.

If not processed any further, the exact result of the accumulation usually has to be rounded into a floating-point number. The flag bits that are used for the fast carry resolution can also be used for the rounding. By looking at the flag bits, the leading result word in the complete register can easily be identified. This and the next CR word are needed to compose the mantissa of the result. This 128 bit quantity must be shifted to form a normalized mantissa of a 64-bit number.

In floating-point interval arithmetic, two complete registers are used for the representation of the lower and the upper bound. For the correct rounding downwards or upwards, it is necessary to check whether any of the discarded bits is a one. This is done by testing the remaining bits of the complete register by looking at the *all bits 0* flags of the following words. If these are not all zero, the rounding upwards or downwards is executed by enlarging the $54th$ bit of the mantissa by 1 in magnitude.

To develop its full power, the method discussed here requires a small amount of register memory ($k + 565$ bits) on the arithmetic unit. Access to this memory is considerably faster than access to the main memory of the computer. Modern processors by leading manufacturers provide this much memory on the arithmetic unit. See [8], for instance.

Finally in the final sum, the ubit still has to be set. It is set to 0 only if the ubits of all summands have been 0. In all other cases, the ubit of the final sum is set to 1. A ubit 0 stands for a closed interval bracket, and a ubit 1 stands for an open interval bracket. We illustrate the influence of the ubit on interval addition by a simple example: $[3,3]+[1,2]+[4,5)+(-1,\infty) = [4,5]+[4,5)+(-1,\infty) = [8,10)+(-1,\infty) = (7,\infty)$.

The method for fast and exact accumulation of floating-point numbers discussed in this paper is the basic ingredient for high speed accumulation of n floating-point intervals $A_i = [a_{i1}, a_{i2}] \in \mathbb{IF}$, $i = 1, 2, \ldots n$. Again the result is obtained with extreme speed and increased accuracy. The following formula speaks for itself

$$\diamondsuit \sum_{\nu=1}^{n} A_\nu = \diamondsuit \left\langle \sum_{\nu=1}^{n} a_{\nu 1}, \sum_{\nu=1}^{n} a_{\nu 2} \right\rangle = \left\langle \bigtriangledown \sum_{\nu=1}^{n} a_{\nu 1}, \bigtriangleup \sum_{\nu=1}^{n} a_{\nu 2} \right\rangle, \qquad (2)$$

where $\diamondsuit$, $\bigtriangledown$, and $\bigtriangleup$ denote the roundings $\diamondsuit : \mathbb{IR} \to \mathbb{IF}$, $\bigtriangledown : \mathbb{R} \to \mathbb{F}$, and $\bigtriangleup : \mathbb{R} \to \mathbb{F}$. The latter two are the monotone downwardly resp. upwardly directed roundings. Here, of course, two complete registers are needed for the exact accumulation of the lower bounds $a_{\nu 1}$ and of the upper bounds $a_{\nu 2}, \nu = 1, 2, \ldots n$. Each one of the angle brackets shown here can be open or closed. This depends on the brackets of the intervals $A_i, i = 1, 2, \ldots n$. Again the bracket on a particular side of the interval can be closed only if the bracket of every summand is closed on that side of the interval. In other words: the ubit of the sum is obtained by the logical *or* of the ubits of all summands in combination with the rounding information.

The method discussed in this paper is not new. It has been used for exact evaluation of the dot product of two vectors the components of which are floating-point numbers in the XSC-languages (PASCAL-XSC, ACRITH-XSC, C-XSC) [9, 10, 11, 23, 24] since 1980. Hardware solutions are discussed at detail in the first chapter of the book [17] and in chapter eight of the book [19]. Its reduction to the accumulation of floating-point numbers discussed here even has the potential of replacing conventional methods for the implementation of floating-point addition and subtraction. Formula (2) for the accumulation of floating-point intervals shows that the computation of the lower and the upper

bound are independent of each other, so by doubling the hardware, both bounds can be computed simultaneously. This would reduce the speed for accumulating n floating-point intervals to the speed of accumulating n floating-point numbers.

Summary: The paper shows that fixed-point accumulation of floating-point numbers and floating-point intervals is superior to accumulation in floating-point arithmetic with respect to **speed and accuracy.** The summands are added into a fixed-point register on the arithmetic unit which covers the full floating-point range. No intermediate roundings or normalizations are performed. No intermediate results need to be stored and read in again for the next operation. The result is independent of the order in which the summands are added. By pipelining, the accumulation can be done in the time the processor needs to read the data, that is, no other method can be faster. For the given data, fixed-point accumulation of the floating-point summands is exact. If desired, only one rounding is performed at the very end of the accumulation., and it delivers a correctly rounded result.

The method described in this paper is not new at all. It can be traced back to the very early computer by G. W. Leibniz.

In case of an unstable (not so well conditioned) problem, it may happen that the best possible floating-point arithmetic does not compute a correct solution. In such a case, higher precision (multiple precision) arithmetic has to be used. 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 complete register as a long fixed-point variable. In Section 9 of [19], the operations $+, -, \cdot, /$, and the square root for multiple precision numbers and intervals are defined. On the basis of these operations, algorithms for the elementary functions also can be and are provided in the XSC-languages [9, 10, 11, 24]. The concept of function and operator overloading in these languages makes applications of multiple precision real and interval arithmetic very simple.

Acknowledgements

The authors owe thanks to John Gustafson and Goetz Alefeld for useful comments on the paper. They are also grateful to two anonymous referees. Their comments led to major improvements of the paper.

References

[1] E. Adams, U. Kulisch, (eds.), *Scientific Computing with Automatic Result Verification*, Academic Press, 1993.

[2] Ch. Baumhof, *A new VLSI vector arithmetic coprocessor for the PC*, in: Institute of Electrical and Electronics Engineers (IEEE), S. Knowles and W. H. McAllister (eds.), *Proceedings of 12th Symposium on Computer*

Arithmetic ARITH, Bath, England, July 19–21, 1995, pp. 210–215, IEEE Computer Society Press, Piscataway, NJ, 1995.

[3] D. Biancolin and J. Koenig, *Hardware Accelerator for Exact Dot Product*, ASPIRE Laboratory, University of California, Berkeley, 2015.

[4] J. L. Gustafson, *The End of Error*. CRC Press, A Chapman and Hall Book, 2015.

[5] R. Hammer, M. Hocks, U. Kulisch and D. Ratz, *Numerical Toolbox for Verified Computing I: Basic Numerical Problems*, Springer, 1993.

[6] R. Hammer, M. Hocks, U. Kulisch and D. Ratz, *C++ Toolbox for Verified Computing: Basic Numerical Problems*. Springer, 1995.

[7] R. Hammer, M. Hocks, U. Kulisch and D. Ratz, *Numerical Toolbox for Verified Computing I: Basic Numerical Problems*, MIR, Moskau, 2005 (in Russian).

[8] INTEL, Intel Architecture Instruction Set Extensions Programming Reference, 319433-017, December 2013, `https://software.intel.com/sites/default/files/managed/b4/3a/319433-024.pdf`.

[9] R. Klatte, U. Kulisch, M. Neaga, D. Ratz and Ch. Ullrich, *PASCAL-XSC – Sprachbeschreibung mit Beispielen*, Springer, 1991. See also `http://www2.math.uni-wuppertal.de/~xsc/` or `http://www.xsc.de/`.

[10] R. Klatte, U. Kulisch, M. Neaga, D. Ratz and Ch. Ullrich, *PASCAL-XSC – Language Reference with Examples*, Springer, 1992. See also `http://www2.math.uni-wuppertal.de/~xsc/` or `http://www.xsc.de/`.
Russian translation MIR, Moscow, 1995, third edition 2006. See also `http://www2.math.uni-wuppertal.de/~xsc/` or `http://www.xsc.de/`.

[11] R. Klatte, U. Kulisch, C. Lawo, M. Rauch and A. Wiethoff, *C-XSC – A C++ Class Library for Extended Scientific Computing*, Springer, 1993. See also `http://www2.math.uni-wuppertal.de/~xsc/` or `http://www.xsc.de/`.

[12] U. Kulisch, *Grundlagen des Numerischen Rechnens – Mathematische Begründung der Rechnerarithmetik*, Informatik 19, Bibliographisches Institut, Mannheim Wien Zürich, 1976.

[13] U. Kulisch and W. L. Miranker, *Computer Arithmetic in Theory and Practice*, Academic Press, 1981.

[14] U. Kulisch and W. L. Miranker (eds.), *A New Approach to Scientific Computation*, Academic Press, 1982.

[15] U. Kulisch, T. Teufel and B. Hoefflinger, Genauer und trotzdem schneller: Ein neuer Coprozessor für hochgenaue Matrix- und Vektoroperationen. Titelgeschichte, *Elektronik* **26** (1994), 52–56.

[16] U. Kulisch, R. Lohner, A. Facius, *Perspectives on Enclosure Methods*, Springer, 2001.

[17] U. Kulisch, *Advanced Arithmetic for the Digital Computer – Design of Arithmetic Units*, Springer, 2002.

[18] U. Kulisch, *An Axiomatic Approach to Computer Arithmetic with an Appendix on Interval Hardware*, LNCS, 7204, pp. 484 – 495, Springer, 2012.

[19] U. Kulisch, *Computer Arithmetic and Validity – Theory, Implementation, and Applications*, de Gruyter, Berlin, 2008, ISBN 978-3-11-020318-9, second edition 2013, ISBN 978-3-11-030173-1.

[20] U. Kulisch, *Mathematics and Speed for Interval Arithmetic – A Complement to IEEE P1788*. Prepared for and sent to IEEE P1788 in January 2014. To be published.

[21] U. Kulisch, *Up-to-date Interval Arithmetic – From closed intervals to connected sets of real numbers*, LNCS, Springer 2016.

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

[23] IBM, *IBM High-Accuracy Arithmetic Subroutine Library (ACRITH)*, IBM Deutschland GmbH (Department 3282, Schönaicher Strasse 220, D-71032 Böblingen), 1983, third 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.

[24] IBM, *ACRITH–XSC: IBM High Accuracy Arithmetic – Extended Scientific Computation. Version 1, Release 1*, IBM Deutschland GmbH (Department 3282, 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.

[25] J. D. Pryce (Ed.), *P1788, IEEE Standard for Interval Arithmetic*, `http://standards.ieee.org/findstds/standard/1788-2015.html`.