Hardware Support for Interval Arithmetic

Reinhard Kirchner, Universität Kaiserslautern,
D-67653 Kaiserslautern
Ulrich W. Kulisch, Universität Karlsruhe, D-76128 Karlsruhe

Summary: A hardware unit for interval arithmetic (including division by an interval that contains zero) is described in this paper. After a brief introduction an instruction set for interval arithmetic is defined which is attractive from the mathematical point of view. These instructions consist of the basic arithmetic operations and comparisons for intervals including the relevant lattice operations. To enable high speed, the case selections for interval multiplication (9 cases) and interval division (14 cases) are done in hardware. The lower bound of the result is computed with rounding downwards and the upper bound with rounding upwards by parallel units simultaneously. The rounding mode must be an integral part of the arithmetic operation. Also the basic comparisons for intervals together with the corresponding lattice operations and the result selection in more complicated cases of multiplication and division are done in hardware. There they are executed by parallel units simultaneously. The circuits described in this paper show that with modest additional hardware costs interval arithmetic can be made almost as fast as simple floating-point arithmetic.

1 Introduction

Interval mathematics has been developed to a high standard over the last few decades. It provides methods which deliver results with guarantees. However, the arithmetic on existing processors makes these methods slow. This paper deals with the question of how interval arithmetic can effectively be provided on computers. This is an essential prerequisite for the superior and fascinating properties of interval mathematics to be more widely used in the scientific computing community. With more suitable processors, rigorous methods based on interval arithmetic could be comparable in speed to todays approximate methods. Interval arithmetic is a natural extension to floating-point arithmetic, not a replacement for it. With increasing speed of computers (gigaflops, teraflops, pentaflops) interval arithmetic becomes a principal and necessary tool for controlling the precision of a computation as well as the accuracy of the computed result.

In conventional numerical analysis Newton's method is the key algorithm for nonlinear problems. The method converges quadratically to the solution if the initial value of the iteration is already close enough to it. However, it may fail in finite as well as in infinite precision arithmetic even in the case of only a single solution in a given interval. In contrast to this the interval version of Newton's method is globally convergent. It never fails, not even in rounded arithmetic. Newton's method reaches its ultimate elegance and power in the Extended Interval Newton Method. It yields to enclosures of all single zeros in a given domain. It is quadratically convergent. The key operation to achieve these fascinating properties is division by an interval which contains zero. It separates different zeros from each other. A method which provides for computation of all zeros of a system of equations in a given domain is very frequently applied. This justifies taking division by an interval which contains zero into the basic set of interval operations, and implementing it by the hardware of the computer.

To handle critical situations it is generally agreed that interval arithmetic must be supplemented by some measure to extend the precision within a computation. An easy way to achieve this is an accurate *multiply and accumulate* instruction or, what is equivalent to it, an accurate scalar product. With it quadruple or multiple precision arithmetic can easily be provided. If the *multiply and accumulate* instruction[1] is implemented in hardware it is very fast. The data can be stored and moved as double precision floating-point numbers. Generally speaking, interval arithmetic brings guarantees and mathmematics into computation, while the accurate *multiply and accumulate* instruction brings higher (dynamic) precision and accuracy. Hardware support for the accurate *multiply and accumulate* instruction is discussed in the first chapter of the book [9].

The present paper is written for the hardware engineer who is supposed to implement the circuitries on a future processor. Our main intention, therefore, is to keep the paper simple, short, and selfcontained. The basic formulas for interval operations and comparisons are already carefully derived in the two books [5] and [6] of the second author. Division by an interval which contains zero is implemented using formulas which are derived in the paper [7].

For the near future the authors do not expect that a completely new instruction set architecture will grow within the present market of desktop and server computers. Thus in this paper hardware support for interval arithmetic is developed in a way which allows an easy incorporation into existing processors whithout any need to change the operating system.

[1] Here *multiply and accumulate* means continued accumulation of products, in contrast to 'multiply and add'.

During the last decade a fair amount of research has been done on various aspects of hardware support for interval arithmetic. A number of relevant papers is listed under References. The present paper gives a complete solution to the problem.

2 An Instruction Set for Interval Arithmetic

From the mathematical point of view the following instructions for interval operations and comparisons are desirable. In the following $A = [a_1, a_2]$ and $B = [b_1, b_2]$ denote interval operands, and $C = [c_1, c_2]$ denotes the result of an interval operation. For the operations $\circ \in \{+, -, \cdot, /\}$ with rounding downwards or upwards the symbols $\triangledown, \triangle, \circ \in \{+, -, \cdot, /\}$, are used respectively.

2.1 Algebraic Operations

Addition $C := [a_1 \triangledown b_1, a_2 \triangle b_2]$

Subtraction $C := [a_1 \triangledown b_2, a_2 \triangle b_1]$

Multiplication

	$b_1 \geq 0$	$b_1 < 0 \leq b_2$	$b_2 < 0$
$a_1 \geq 0$	$[a_1 \triangledown b_1, a_2 \triangle b_2]$	$[a_2 \triangledown b_1, a_2 \triangle b_2]$	$[a_2 \triangledown b_1, a_1 \triangle b_2]$
$a_1 < 0 \leq a_2$	$[a_1 \triangledown b_2, a_2 \triangle b_2]$	$[\min(a_1 \triangledown b_2, a_2 \triangledown b_1), \max(a_1 \triangle b_1, a_2 \triangle b_2)]$	$[a_2 \triangledown b_1, a_1 \triangle b_1]$
$a_2 < 0$	$[a_1 \triangledown b_2, a_2 \triangle b_1]$	$[a_1 \triangledown b_2, a_1 \triangle b_1]$	$[a_2 \triangledown b_2, a_1 \triangle b_1]$

Division
$0 \notin B$

	$b_1 > 0$	$b_2 < 0$
$a_1 \geq 0$	$[a_1 \triangledown b_2, a_2 \triangle b_1]$	$[a_2 \triangledown b_2, a_1 \triangle b_1]$
$a_1 < 0 \leq a_2$	$[a_1 \triangledown b_1, a_2 \triangle b_1]$	$[a_2 \triangledown b_2, a_1 \triangle b_2]$
$a_2 < 0$	$[a_1 \triangledown b_1, a_2 \triangle b_2]$	$[a_2 \triangledown b_1, a_1 \triangle b_2]$

0 ∈ B

	$b_1 = b_2 = 0$	$b_1 < b_2 = 0$	$b_1 < 0 < b_2$	$0 = b_1 < b_2$
$a_2 < 0$	$[+\mathtt{NaN}, -\mathtt{NaN}]^1$	$[a_2 \triangledown b_1, +\infty]$	$[a_2 \triangledown b_1, a_2 \triangle b_2]^2$	$[-\infty, a_2 \triangle b_2]$
$a_1 \leq 0 \leq a_2$	$[-\infty, +\infty]$	$[-\infty, +\infty]$	$[-\infty, +\infty]$	$[-\infty, +\infty]$
$a_1 > 0$	$[+\mathtt{NaN}, -\mathtt{NaN}]^1$	$[-\infty, a_1 \triangle b_1]$	$[a_1 \triangledown b_2, a_1 \triangle b_1]^2$	$[a_1 \triangledown b_2, +\infty]$

In the table for division by an interval which contains zero the notation $[+\mathtt{NaN}, -\mathtt{NaN}]$ is used to represent the empty interval. Since the result of an interval operation is supposed to always be a single interval again, the results which consist of the union of two intervals are delivered and represented as improper intervals $[a_2 \triangledown b_1, a_2 \triangle b_2]$ and $[a_1 \triangledown b_2, a_1 \triangle b_1]$. In these particular cases the left hand bound is higher than the right hand bound.

2.2 Comments on the Algebraic Operations

Except for a few cases in the table for division by an interval which contains zero, the lower bound of the result is always obtained with an operation rounded downwards and the upper bound with an operation rounded upwards. Multiplication and division need all combinations of lower and upper bounds of input intervals depending on the signs of the bounds. Thus an operand selection has to be performed before the operation can be executed. However, in all cases of computing the bounds of the result interval the left hand operand is a bound of the interval $A = [a_1, a_2]$ and the right hand operand is always a bound of the operand $B = [b_1, b_2]$. Thus the operand selection can be executed for the bounds of A and B separately.

In one special case ($0 \in A$ and $0 \in B$) multiplication requires the computation of two result pairs. Then the interval hull of these is taken.

In the case that $0 \in B$ division may produce various special results like $+\infty$, $-\infty$ or the empty interval.

2.3 Comparisons and Lattice Operations

c is a value of type boolean.

equality $c := (a_1 = b_1 \wedge a_2 = b_2)$

[1] special encoding of empty interval

[2] special encoding of result $[-\infty, c_2] \cup [c_1, +\infty]$

less than or equal $c := (a_1 \leq b_1 \wedge a_2 \leq b_2)$

greatest lower bound $C := [\min(a_1, b_1), \min(a_2, b_2)]$

least upper bound $C := [\max(a_1, b_1), \max(a_2, b_2)]$

inclusion $c := (b_1 \leq a_1 \wedge a_2 \leq b_2)$

element of $c := (b_1 \leq a \wedge a \leq b_2)$, special case of inclusion

interval hull $C := [\min(a_1, b_1), \max(a_2, b_2)]$

intersection $C := [\max(a_1, b_1), \min(a_2, b_2)]$ or empty interval (encoded $[+\texttt{NaN}, -\texttt{NaN}]$)

check interval branch on $a_1 > a_2$ (checking for proper interval)

2.4 Comments on Comparisons and Lattice Operations

For comparisons and lattice operations no shuffling of bounds is needed. All combinations of minimum and maximum of lower and upper bounds do occur.

In IEEE arithmetic the bit string of nonnegative floating-point numbers can be interpreted as an integer for comparison purposes. Computation of minimum or maximum is comparison and selection.

Intersection and checking for an improper interval needs a comparison of the lower and the upper bound of the result interval. In all other cases the lower and the upper bound of the result interval can be computed independently.

3 General Circuitry for Interval Operations and Comparisons

3.1 Algebraic Operations

We assume that the data are read from and written into a register file or a memory-access-unit. Each memory cell consists of 128 bits for pairs of double precision floating-point numbers holding the lower and upper bound of an interval. Intervals are moved within the unit via busses of 128 bits.

There are three such bus systems in the interval arithmetic unit, one for the operand $A = [a_1, a_2]$, a second one for the operand $B = [b_1, b_2]$, and a third one for the result $C = [c_1, c_2]$. The interval arithmetic unit consists of two

major parts, one to perform the operand selection and the operations, the other to perform comparisons and result selection. See Figure 1.

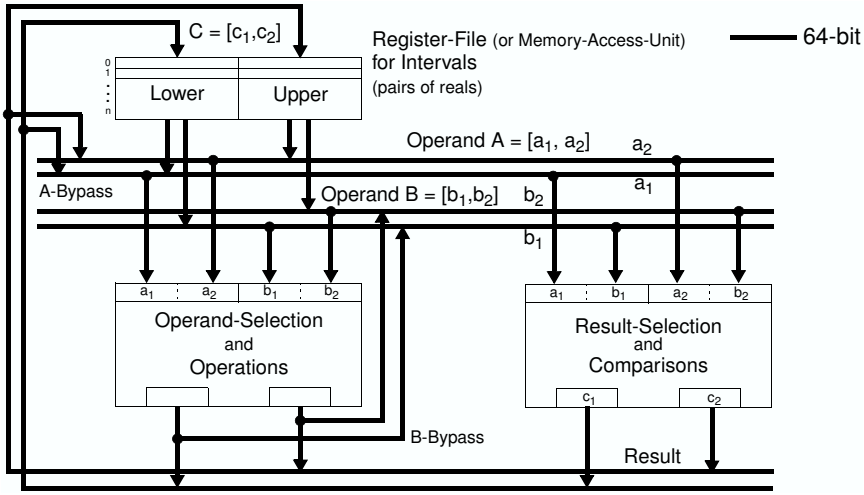

Figure 1: General circuitry for interval operations and comparisions.

An operation is performed as follows: The operands $A = [a_1, a_2]$ and $B = [b_1, b_2]$ are read from the memory onto the A-bus and B-bus and forwarded to the Operand-Selection and Operations Unit. Here from the lower and upper bounds of the operands multiplexers select various combinations of values depending on the operation, and on signs and nulls of all four values. Finally a pair of operations $\triangledown$ and $\triangle$, $\circ \in \{+, -, \cdot, /\}$, is performed on the selected values, one with rounding downwards and one with rounding upwards. See Figure 2. In many cases the computed values are already the desired result $C = [c_1, c_2]$. In these cases the result is forwarded to the memory via the C-bus. But there are exceptions, as for instance in the case of multiplication where both operands contain zero, or in the case of division by an interval that contains zero. See the tables in Section 1. In these cases the computed values are forwarded to the Comparison and Result-Selection Unit for further processing.

The selector signals o_{a1}, o_{a2}, o_{b1}, and o_{b2} control the multiplexers. The Operand-Selection and Operations Unit performs all arithmetic operations.

In the case of **addition** just the lower bounds of A and B are added with rounding downwards and the upper bounds are added with rounding upwards $[a_1 \triangledown b_1, a_2 \triangle b_2]$. The selector signals are set to $o_{a1} = 0$, $o_{a2} = 1$, $o_{b1} = 0$, and $o_{b2} = 1$.

In the case of **subtraction** the bounds of B are exchanged by the operand selection. Then the subtraction is performed, the lower bound is computed with rounding downwards and the upper bound with rounding upwards

$[a_1 \triangledown b_2, a_2 \triangle b_1]$. The selector signals are set to $o_{a1} = 0$, $o_{a2} = 1$, $o_{b1} = 1$, and $o_{b2} = 0$.

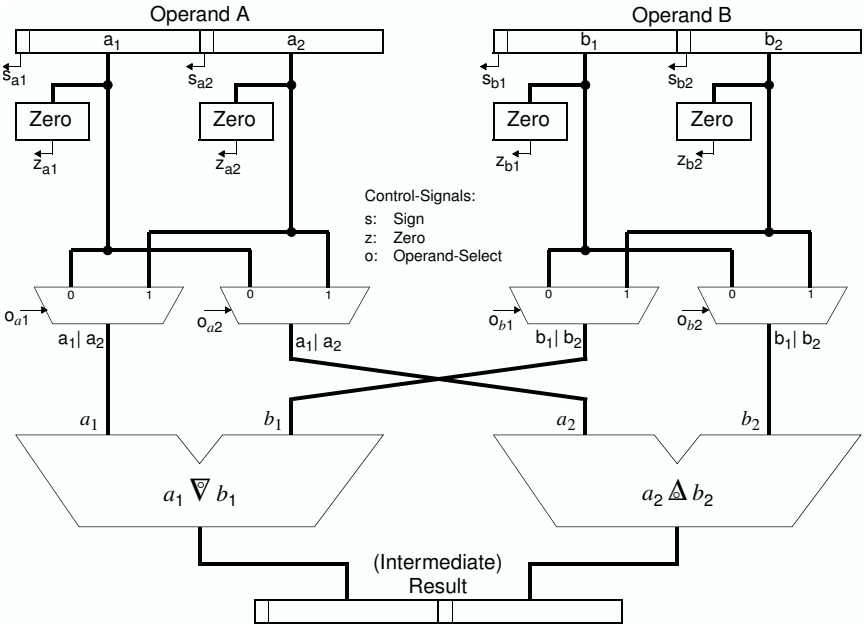

Figure 2: Operand Selection and Operations Unit.

Multiplication is a little more complicated. The various multiplications are performed in the following way:

If both operands A and B do not contain zero ($s_{a1} \cdot \overline{s_{a2}} \cdot s_{b1} \cdot \overline{s_{b2}} = 0$)

then the bounds are shuffled, the result $[a_1 \triangledown b_1, a_2 \triangle b_2]$ is computed with the selected bounds, and delivered to the target via the C-bus.

else

1. The bounds are shuffled for the first multiplication by operand selection, the first partial result $[a_1 \triangledown b_2, a_1 \triangle b_1]$ is computed, and it is forwarded to the Comparison and Result-Selection Unit via the A-bus.
2. The bounds are shuffled for the second variation by operand selection, the second partial result $[a_2 \triangledown b_1, a_2 \triangle b_2]$ is computed, and it is forwarded to the Comparison and Result-Selection Unit via the B-bus.
3. In the Comparison and Result-Selection Unit the hull of the two multiplications is selected and as final result delivered to the target via the C-bus.

In the case of multiplication the multiplexers are controlled by the following selector signals:[2]

$$o_{a1} = s_{b2} + \overline{s_{a1}} \cdot s_{b1} + ms \qquad o_{a2} = \overline{s_{b1}} + \overline{s_{a1}} \cdot \overline{s_{b2}} + ms$$
$$o_{b1} = \overline{ms}(s_{a2} + s_{a1} \cdot \overline{s_{b2}}) \qquad o_{b2} = \overline{s_{a1}} + \overline{s_{a2}} \cdot \overline{s_{b1}} + ms$$

These signals are computed by the signs of the bounds of the interval operands $A = [a_1, a_2]$ and $B = [b_1, b_2]$. In the expressions the signal ms is zero in the case 1. and it is one in the case 2. of the conditional statement above.

In the case of multiplication every operand selector signal can be realized by two or three gates!

Division is a little simpler than multiplication. It is organized following a similar pattern :

If B does not contain zero ($\overline{s_{b1}} \cdot \overline{z_{b1}} + s_{b2} = 1$)

then the bounds are shuffled, the result $[a_1 \triangledown b_1, a_2 \triangle b_2]$ is computed with the selected bounds, and delivered to the target via the C-bus.

else
1. The bounds are shuffled, the arithmetic operation $[a_1 \triangledown b_1]$ or $[a_2 \triangle b_2]$ is computed with the selected bounds, and it is forwarded to the Comparison and Result-Selection Unit together with the selection code for special values.
2. In the Comparison and Result-Selection Unit the result is generated from arithmetic values and/or special values ($-\infty|+\texttt{NaN}|+\infty|-\texttt{NaN}$), and it is forwarded to the target via the C-bus.

The operand selection is controlled by the following selector signals:

$$o_{a1} = s_{b2} + s_{a1} \cdot s_{b1} \qquad o_{a2} = \overline{s_{b1}} + s_{a2} \cdot \overline{s_{b2}}$$
$$o_{b1} = \overline{s_{a1}} + \overline{s_{a2}} \cdot s_{b1} \qquad o_{b2} = s_{a2} + s_{a1} \cdot s_{b1}$$

In the case of division every operand selector signal can be realized by two gates!

In the cases of division by an interval which contains zero the result is an improper interval where one or both bounds is not a real number or where

[2]Note: A negative sign is a 1. A bar upon a logical value means inversion. In the expressions a dot stands for a logical and, and a plus for a logical or.

the left hand bound is higher than the right hand bound. In Newton's method, for instance, the following operation then is an intersection with a regular interval. Treatment of this and similar cases is left to software. The instruction *check interval* supplies a test for improper intervals.[3]

3.2 Comparisons and Result-Selection

In this unit comparisons and minima and maxima are to be computed. We mention again that in IEEE arithmetic the bit string of nonnegative floating-point numbers can be interpreted as an integer for comparison purposes. Computation of a minimum or maximum consists of a compare plus selection. Thus the arithmetic that has to be done in this unit is relatively simple. Again the computation of the lower bound and the upper bound of the result is done in parallel and simultaneously. No bound shuffling is necessary in this unit. See Figure 3.

The comparisons for **equality**, **less than or equal** and **set inclusion** are done by comparing the bounds, combining the results and setting a flag in the state register. For the operation **element of** an interval $[a, a]$ is built in software. Then a test for inclusion is applied.

The minimum and maximum computations for the operations **greatest lower bound**, **least upper bound**, **interval hull**, and the result selection in the case of multiplication where $0 \in A$ and $0 \in B$ are executed by comparing the bounds and selecting the higher and the lower, respectively. Then the result is delivered to the target.

The computation of the **intersection** is a little more complicated. First the bounds of the operands are compared and the higher lower bound and the lower upper bound are selected. If $\max(a_1, b_1) \leq \min(a_2, b_2)$, the intersection, which is the interval $[\max(a_1, b_1), \min(a_2, b_2)]$ is delivered to target. Else the empty interval is delivered. It is represented by $[+\texttt{NaN}, -\texttt{NaN}]$.

In the case of division by an interval which contains zero, an interval with bounds like $+\texttt{NaN}$, $-\infty$, $-\texttt{NaN}$, $+\infty$ has to be delivered. These alternatives are selected in the Comparison and Result-Selection Unit.

Now we give the various selector signals that appear in Figure 3.

[3] Within the given framework of existing processors only one interval can be delivered as result of an operation. Other solutions which use special registers or flags or new exceptions would require an adaption of the operating system.

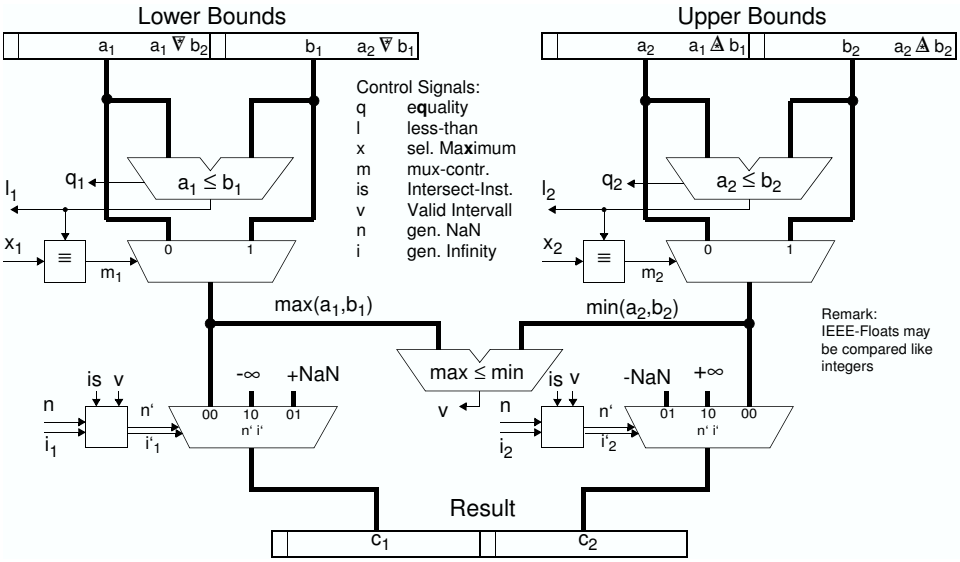

Figure 3: Comparisons and Result Selection Unit.

$$
\begin{aligned}
n &= n_1 = n_2 = (\overline{z_{a1}} \cdot \overline{s_{a1}} \cdot \overline{s_{a2}} + s_{a2}) \cdot z_{b2} \cdot s_{b1} && \text{3 Gates} \\
i_1 &= z_{a1} \cdot z_{b1} + z_{a1} \cdot s_{b1} \cdot \overline{s_{b2}} + s_{a1} \cdot \overline{s_{a2}} \cdot s_{b1} \cdot \overline{s_{b2}} && \text{7 Gates} \\
&\quad + \overline{s_{a2}} \cdot \overline{z_{b1}} \cdot z_{b2} + s_{a1} \cdot z_{b1} \cdot \overline{z_{b2}} + s_{a1} \cdot \overline{s_{a2}} \cdot z_{b1} \\
i_2 &= z_{a1} \cdot z_{b1} + z_{a1} \cdot s_{b1} \cdot \overline{s_{b2}} + s_{a1} \cdot \overline{s_{a2}} \cdot s_{b1} \cdot \overline{s_{b2}} && \text{4 Gates} \\
&\quad + s_{a1} \cdot \overline{s_{a2}} \cdot z_{b1} + \overline{s_{a2}} \cdot z_{b1} \cdot \overline{z_{b2}} + s_{a2} \cdot \overline{z_{b1}} \cdot z_{a2} && \text{(common subexpressions)} \\
i'_1 &= \overline{is} \cdot i_1 \\
i'_2 &= \overline{is} \cdot i_2 \\
n' &= n_1 = n_2 = \overline{is} \cdot n + n \cdot \overline{v}
\end{aligned}
$$

The logical expressions given in this section were developed from function tables of up to several hundred entries. Then minimization was performed which leads to the equations given. We do not replicate the tables and the minimization here because all this is a standard procedure in circuit design. The function tables contain a high amount of don't care entries which allows realization with a very small number of gates. (Since the dont' care entries may be used during minimization in different ways various designs may end up with different equations of equal complexity)

4 Alternative Circuitry for Interval Operations and Comparisons

In Figure 2 the operand selection and the execution of the interval operations together form the Operand Selection and Operations Unit. This is justified since the time needed for the operand selection is negligible in comparison to the time needed to perform the arithmetic operations.

It may, however, be useful to separate the operand selection from the operations. Separation into two units would make it easier to use these for other purposes. Examples are ordinary arithmetic or shuffling of parts of the operands. Many processors are being built which provide already several independent arithmetic units (super scalar processors). These then easily may be used as part of the interval operations unit. Figure 4 shows a corresponding circuit for interval operations and comparisons.

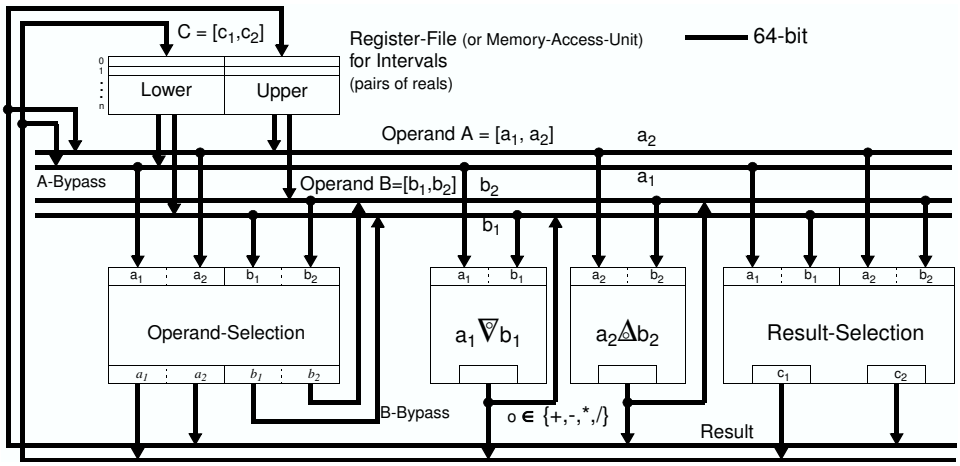

Figure 4: General circuitry for interval operations and comparisons.

The Operand Selection Unit and the Operations Unit then would look as shown in Figure 5 and Figure 6. We will not discuss these circuits in further detail. Their functionality should be clear from what has been said already.

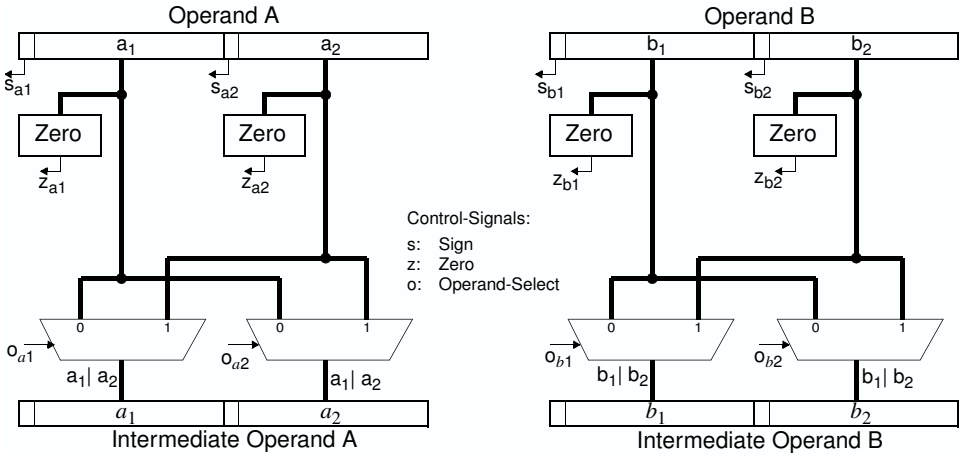

Figure 5: Operand Selection Unit.

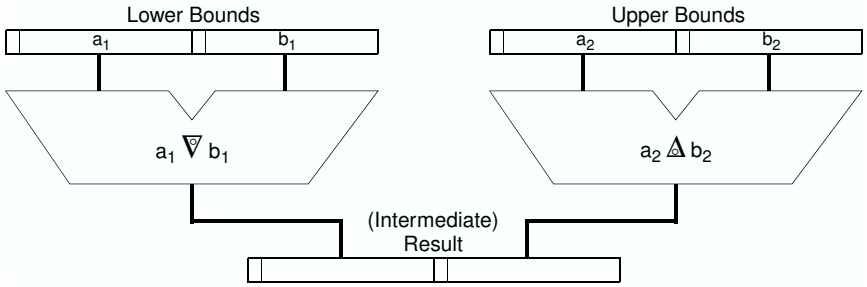

Figure 6: Arithmetic Operations Unit.

5 Closing Remarks

In summary it can be said that a hardware unit for fast interval arithmetic has a very regular structure. Interval arithmetic is just regular arithmetic for pairs of reals with particular roundings, plus operand selection, plus clever control.

It is interesting to note that most of what is needed is already available on current x86-processors. Figure 7 shows figures from various publications by Intel.

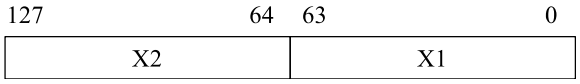

Figure 6. Packed double precision floating-point data type

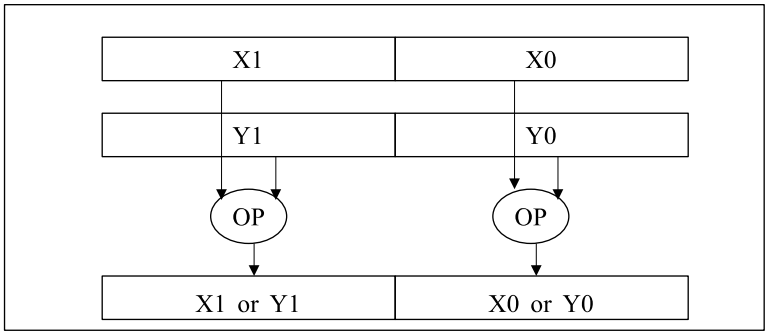

Figure 11-3. Packed Double-Precision Floating-Point Operation

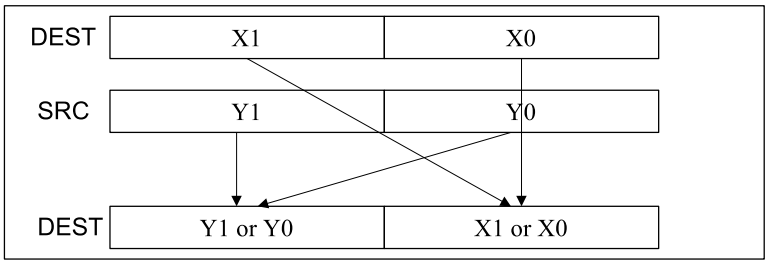

Figure 11-5. SHUFFD Instruction Packed Shuffle Operation

Figure 7: Figures from various Intel publications.

On an Intel Pentium 4, for instance, eight registers are available for words of 128 bits (xmm0, xmm1, ..., xmm7). The x86-64 processors even pro-

vide 16 such registers. These registers can hold pairs of double precision floating-point numbers. They can be viewed as bounds of intervals. Parallel operations like $+$, $-$, $\cdot$, $/$, min, max, and compare can be performed on these pairs of numbers. What is not available and would be needed is for one of the two operations to be rounded downwards and the other one rounded upwards. The last picture in Figure 7 shows that even shuffling of bounds is possible under certain conditions. This is half of operand selection needed for interval arithmetic. Also nearly all of the data paths are available on current x86-processors. Thus full hardware support of interval arithmetic would probably add less than 1%, more likely less than 0.1% to a current Intel or AMD x86 processor chip.

Full hardware support of fast interval arithmetic on RISC processors may cost a little more as these lack pairwise processing. But most of them have two arithmetic units and use them for super scalar processing. What has to be added is some sophisticated control.

There are still some interesting questions and work that should be done. Runtime statistics of interval instructions would be interesting, also the development of typical benchmark programs.

Acknowledgement: The authors gratefully acknowledge the help of Neville Holmes who went carefully through the manuscript, sending back corrections and suggestions that led to many improvements.

References

[1] Akkas, A.: *Instruction Set Enhancements for Reliable Computations,* Ph.D. Dissertation, Lehigh University, January 2002.

[2] Akkas, A.: *A Combined Interval and Floating-point Comparator/Selector,* IEEE 13th International Conference on Application-specific Systems, Architectures and Processors, pp. 208-217, San Jose, USA, July, 2002.

[3] Chiriaev, D. and Walster, G. W.: *Interval Arithmetic Specification,* available from Internet URL http://www.mscs.mu.edu/∼globsol/readings.html, 1998.

[4] Kolla, R.; Vodopivec, A.; Wolff v. Gudenberg, J.: *The IAX Architecture - Interval Arithmetic Extension,* Report No. 225, Institut fuer Informatik, Universitaet Wuerzburg, 1999.

[5] Kulisch, U.: *Grundlagen des numerischen Rechnens - Mathematische Begründung der Rechnerarithmetik,* Bibiographisches Institut, Mannheim, Wien, Zuerich, 1976.

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

[7] Kulisch, U.: *Interval Arithmetic Revisited.* This paper is published in the two books [9] and [8].

[8] Kulisch, U. W.; Lohner, R.; Facius, A. (eds.): *Perspectives on Enclosure Methods,* Springer-Verlag, Wien, New York, 2001.

[9] Kulisch, U.: *Advanced Arithmetic for the Digital Computer - Design of Arithmetic Units,* Springer-Verlag, Wien, New York, 2002.

[10] Stine, J. E.; Schulte, M. J.: *A Combined Interval and Floating Point Multiplier,* Proceedings of the 8th Great Lakes Symposium on VLSI, Lafayette, LA, pp. 208-213, February, 1998.

[11] Schulte, M. J. and Swartzlander, E. E. Jr.: *A Family of Variable-Precision, Interval Arithmetic Processor,* IEEE Transactions on Computers, No. 5, Vol. 49, pp. 387-398, May, 2000.

[12] Stine, J. E.: *Design Issues for Accurate and Reliable Arithmetic,* Ph.D. Dissertation, Lehigh University, January 2001.

[13] Stine, J. E.; Schulte, M. J.: *A Case for Interval Hardware on Superscalar Processors,* in Scientific Computing, Validated Numerics, and Interval Methods, pp. 53-68, Kluwer Academic Publishers, 2001.

[14] Wolff v. Gudenberg, J.: *Hardware Support for Interval Arithmetic,* in Scientific Computing and Validated Numerics, Proceedings of the International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics - SCAN '95, Kluwer Academic Publishers, 1996.

Contact: `Kirchner@informatik.uni-kl.de`
`Ulrich.Kulisch@math.uka.de`