A Combined Interval and Floating-Point Reciprocal Unit

Umut Küçükkabak and Ahmet Akkaş
Computer Engineering Department
Koç University
34450 Sarıyer, İstanbul, Turkey
umut@kucukkabak.com ahakkas@ku.edu.tr

***Abstract*— Interval arithmetic is one technique for accurate and reliable computing. Among interval arithmetic operations, division is the most time consuming operation. This paper presents the design and implementation of a combined interval and floating-point reciprocal unit. To compute the reciprocal of an operand, an initial approximation is computed first and then iterated twice by Newton-Raphson iteration. The combined interval and floating-point reciprocal unit computes the reciprocal of a double precision floating-point number in eleven clock cycles and the reciprocal of an interval in twenty-one clock cycles. The unit is implemented in VHDL and synthesized to estimate the area and the worst case delay. Simulation results showed that the least significant bit of the floating-point result cannot be guaranteed to be same for all cases compared to the result based on an infinite precision. For interval reciprocal, however, the true result is contained in the result interval.**

I. Introduction

Interval arithmetic provides an efficient method for monitoring errors in numerical computations and for solving problems that cannot be efficiently solved with floating-point arithmetic. To provide software support for fixed-precision interval arithmetic, interval arithmetic libraries have been developed. These libraries are INTLIB [1], PROFIL/BIAS [2], and InC++ [3]. Techniques for variable-precision arithmetic and interval arithmetic have also been combined in the extended scientific programming languages PASCAL-XSC [4], C-XSC [5], ACRITH-XSC [6], and VPI [7]. These languages are extensions to existing programming languages.

Compiler support for interval arithmetic has been developed. The GNU Fortran Compiler has been modified to provide support for single and double precision interval data types [8]. Although the interval-enhanced GNU Fortran compiler provides support for interval arithmetic, it suffers from function call overhead, because all interval operations are implemented by calls to runtime library routines. Sun Microsystems enhanced their Fortran 95 compiler to support intervals as intrinsic data types [9]. Sun's compiler has full support for extended interval data types. It also provides some compiler optimizations, which substantially improve the performance of interval code. The GNU gcc compiler has also been modified to inline interval operations [10]. To implement interval operations with compiler support, the compiler produces sequences of instructions that use existing floating point hardware. Even though inlining interval operations eliminates the function call overhead, the number of instructions to implement each interval operation is high.

The main disadvantage of software tools for interval arithmetic is their speed. Since the arithmetic operations are simulated in software, tremendous overhead occurs due to function calls, memory management, error checking, changing rounding modes, and exception handing.

To provide hardware support for interval arithmetic, a variable-precision, interval arithmetic coprocessor has been designed [11] in order to allow the programmer to specify the precision of the computation, determine the accuracy of the results, and recompute results with higher precision when necessary. The combined interval and floating-point multiplier [12], divider [13], adder/subtracter [14], and comparator/selector [15] have also been designed. These combined units support both floating-point and interval operations. In this paper, hardware support for interval and floating-point reciprocal operations is presented.

The remainder of this paper is organized as follows: Section II gives an overview to compute the reciprocal. Section III presents the design of our combined interval and floating-point reciprocal unit. Section IV gives area and delay estimates for the unit. Section V discusses the issues related to the presented unit and Section VI presents our conclusions.

II. An Overview to Compute Reciprocal

A method used to compute the reciprocal of an operand consists of two basic steps:

- Computation of an initial approximation,
- Two consecutive Newton-Raphson iterations.

A method proposed by Takagi [16] is used to obtain the initial approximation. The initial approximation is computed by multiplying a constant read through a table look-up with a modified input operand. The Newton-Raphson iterations construct the second step of the reciprocal computation. Each

iteration of Newton-Raphson yields a quadratic convergence to the correct result. In other words, error of the previous step squares itself in each Newton-Raphson iteration, which in turn results in doubled accuracy as compared to the previous approximation [17]. This ensures that, if more than thirteen correct bits are acquired in the initial approximation, it is guaranteed to obtain the correct reciprocal result for double precision floating-point number after two consecutive Newton-Raphson iterations ($14\ bits \rightarrow 28\ bits \rightarrow 56\ bits$).

A. Initial Approximation

The method proposed by Takagi [16] is used to obtain an initial approximation for the reciprocal operation. With this method, an input operand, $X = [1.x_1, x_2, x_3, ..., x_n]$, is divided into two parts as shown in Equations (1) and (2)

$$X_1 = [1.x_1 x_2 x_3 ... x_m] \quad (1)$$

$$X_2 = [0.x_{m+1} x_{m+2} ... x_n] \times 2^{-m} \quad (2)$$

where, $X = X_1 + X_2$.

The initial approximation is computed as

$$X^{-1} \simeq C' \cdot X' \quad (3)$$

where, C' is the refined constant as given in Equation (4) and X' is the modified operand as given in Equation (5). The mathematical computation which clarifies the refinement process of a constant and the modification of an input operand can be found in [16] and [18].

$$C' = (X_1)^{-2} - (X_1)^{-3} \cdot 2^{-m} + (X_1)^{-4} \cdot 7 \cdot 2^{-2m-3} \quad (4)$$

$$X' \simeq [1.x_1 x_2 x_3 ... x_m \tilde{x}_{m+1} \tilde{x}_{m+2} ... \tilde{x}_n] \quad (5)$$

where; complement of x_i is denoted by $\tilde{x}_i$.

In our implementation, the reciprocal of a normalized double precision IEEE floating-point number is computed [19]. Therefore, n is 52. A trade-off appears while selecting the number m, as it is determinant on both the ROM size ($2^m \times 2m$ bits) and the accuracy of the initial approximation ($2m \pm 3$ bits). Utmost care is taken to keep the ROM size minimum in order to prevent the cost of hardware. On the other hand, the accuracy obtained by initial approximation has to be sufficient to yield a correct result after two Newton-Raphson iterations. Among many experiments performed on the implemented unit, the most effective m, which yields correct results at all trials, was shown to be 12 so that the employed ROM should have the size of:

$$2^{12} \times 24\ bits = 2^{10} \times 96\ bits = 96\ Kbits = 12\ KB \quad (6)$$

B. Newton-Raphson Iteration

After initial approximation is computed, the well known Newton-Raphson method is applied twice to increase the accuracy of the initial approximation. The Newton-Raphson formula derived from Taylor series expansion for reciprocal uses the iterative equation given in Equation (7):

$$x_{i+1} = x_i(2 - X x_i) \quad (7)$$

Using the form in Equation (7), two multiplications and one subtraction operation (which may be handled by bitwise inversion) are required for computation of x_{i+1}. Due to the nature of the method itself, it is possible to employ the method to increase the accuracy of an initially obtained result. Newton-Raphson method is applied twice in an iterative manner in this work, in order to increase the accuracy of the initial approximation of reciprocal. In each Newton-Raphson iteration, the number of true bits in the mantissa approximately doubles itself.

III. Combined Interval and Floating-Point Reciprocal Unit

A block diagram of the combined interval and floating-point reciprocal unit is shown in Figure 1. The design consists of four pipeline stages. The first pipeline stage constructs the table look-up and operand modification phase. The second pipeline stage performs the multiplication. The third pipeline stage adds the multiplier outputs in carry-save format and manipulates the results if shift or bitwise inversion is required. The last pipeline stage is used to round the result for only interval operation.

A. Reciprocal of a Floating-Point Number

The combined reciprocal unit accepts as input a floating-point or an interval operand. When it is used for floating-point operation, only the REG1 register is used to hold the input. The reciprocal of a floating-point number takes total of eleven clock cycles. The first three clock cycles yields the initial reciprocal approximation. Each Newton-Raphson iteration is completed in four clock cycles. Since Newton-Raphson iteration is applied twice, a total of eight clock cycles is dedicated for two Newton-Raphson iterations.

The first pipeline stage, consisting of ROM and the operand modifier (OpMod), is only used in the first clock cycle of the reciprocal operation to obtain the refined constant, C' (Constant), and the modified operand, X' (Modified_Op). Therefore, pipeline registers are not used between the first and the second pipeline stages.

In the 1^{st} clock cycle, the ROM takes as inputs the most significant 12-bit of the mantissa from REG1 register, in which the input operand is saved in 64-bit IEEE compliant format [19]. In this cycle, the CONTROL signal is set to zero. The value of CONTROL signal for different clock cycles is

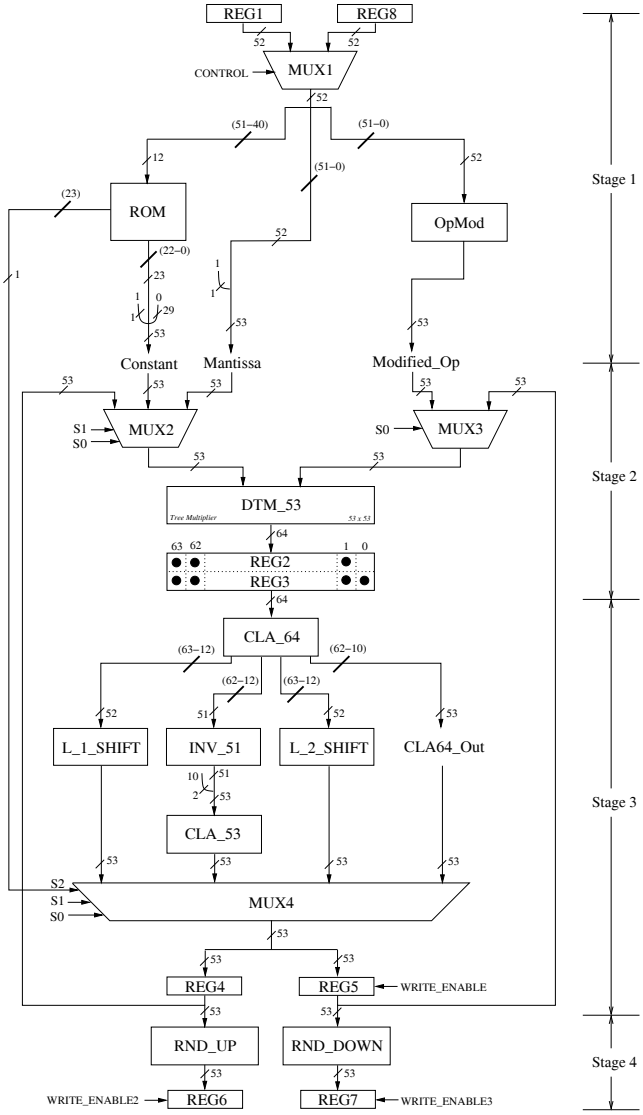


Fig. 1. Combined Interval and Floating-Point Reciprocal Unit

given in Table I. The ROM is constructed with the values that are computed using Equation (4).

On the other side, the mantissa of the operand is supplied to the OpMod unit. The OpMod unit is constructed with forty inverters, in which the most significant 12-bit of the mantissa stays the same, whereas the least significant 40-bit are bitwise inverted, as given in Equation (5). Within the OpMod unit, a logical one is appended as a most significant bit of the 52-bit word in order to form X' given in Equation (5).

In the 2^{nd} clock cycle, Constant, C', and Modified_Op, X', are selected by MUX2 and MUX3 multiplexors. These values are multiplied by the 53×53 Dadda Tree Multiplier, DTM_53, and the results in carry-save format are stored in the registers REG2 and REG3.

Effective Clock Cycle	CONTROL
1	0
10	1
other	X
Note: **X = Don't Care**	

TABLE I
CONTROL SIGNAL FOR MUX1

Cycle	Select Bits			Selected Input		
	S2	**S1**	**S0**	**MUX2**	**MUX3**	**MUX4**
2,3	0	0	0	Constant	Modified_Op	L_1_SHIFT
4,5,8,9	0	0	1	Mantissa	REG5	INV_51
7	0	1	0	X	X	L_2_SHIFT
6,10,11	0	1	1	REG4	REG5	CLA64_Out
2	1	0	0	Constant	Modified_Op	X
4,5,8,9	1	0	1	Mantissa	REG5	INV_51
3,7	1	1	0	X	X	L_2_SHIFT
6,10,11	1	1	1	REG4	REG5	CLA64_Out
Note: S2 = First Bit of ROM, X = Don't Care						

TABLE II
SELECTION BITS FOR MUX1, MUX2 AND MUX3

In the 3^{rd} clock cycle, the contents of registers REG2 and REG3 are added by 64-bit Carry Lookahead Adder, CLA_64. Depending on the most significant bit of the ROM, either *shift left 1-bit* (L_1_SHIFT) or *shift left 2-bit* (L_2_SHIFT) output is selected by MUX4. Table II shows the selection bits for MUX2, MUX3 and MUX4 multiplexors based on the S2 bit. L_1_SHIFT unit takes as inputs the most significant 52-bit of the CLA_64 output and shifts it to the left 1-bit. Finally, an implicit one is appended as a leading bit of the word for the oncoming multiplication in the L_1_SHIFT unit. L_2_SHIFT is conducted in similar manner to L_1_SHIFT except that it shifts the CLA_64 output to the left 2-bit. The implicit one is as well appended as the most significant bit of the word.

The MUX4 output is stored into both registers REG4 and REG5, at the end of the 3^{rd} clock cycle. Note that, REG5 is a register with WRITE_ENABLE signal, where in some cases, the register value will not be updated in order to keep the previous result for future operations. Table III summarizes the clock cycles in which REG5 is enabled. At the end of the 3^{rd} clock cycle, the initial reciprocal approximation, x_i, is obtained with an accuracy of $2m \pm 3$ bits and it is stored into registers REG4 and REG5. The computed initial approximation will be iterated twice by Newton-Raphson method in the next eight clock cycles. Note that, each Newton-Raphson iteration takes four clock cycles. The second and third pipeline stages are used twice in each Newton-Raphson iteration.

At the beginning of the 4^{th} clock cycle, 53-bit mantissa of the input operand, Mantissa, is selected by MUX2. MUX3 selects the input coming from register REG5, which contains

Effective Clock Cycle	WRITE_ENABLE
3, 7, 11	1
else	0

TABLE III
WRITE_ENABLE SIGNAL FOR REG5

$$\frac{1}{X} = \begin{cases} \oslash & : \text{if interval } X \text{ is empty} \\ [-\infty, +\infty] & : \text{if } x_l \leq 0 \leq x_u \\ [\ \bigtriangledown(\frac{1}{x_u}), \bigtriangleup(\frac{1}{x_l})] & : \text{elsewhere} \end{cases}$$

Fig. 2. Algorithm for Interval Reciprocal Operation

the initial reciprocal approximation. These two values are multiplied in the 4^{th} clock cycle. The multiplication result will yield Xx_i in Equation (7).

In the 5^{th} clock cycle, the multiplication results stored in REG2 and REG3 register are added by CLA_64. In order to obtain the mathematical expression, $2 - Xx_i$, the required operation is to take two's complement of Xx_i, provided by CLA_64. Excluding the implicit one, 51-bit of the multiplication result is bitwise inverted by INV_51. Later, an implicit one and a consequent zero are appended as the most significant 2-bit of the inverted word. The inverted result is incremented by *one*, using CLA_53. Finally, $2-Xx_i$ is obtained and saved in register REG4. However, in this clock cycle, WRITE_ENABLE signal is set to zero and the register REG5 is not updated. Hence, the register REG5 still preserves the initial approximation result, x_i.

During the 6^{th} clock cycle, MUX2 selects the value in REG4, which is $2 - Xx_i$, and MUX3 selects the value of REG5, in which the initial approximation, x_i, is saved. Their multiplication is performed in this clock cycle to yield $x_i(2-Xx_i)$.

In the 7^{th} clock cycle, products in carry-save format stored in registers REG2 and REG3 are added by CLA_64. Later, MUX4 selects the output of L_2_SHIFT unit and the result is saved in registers REG4 and REG5. At the end of the 7^{th} clock cycle, first iteration of the Newton-Raphson, x_{i+1}, is available in both registers REG4 and REG5. This concludes the first Newton-Raphson iteration and the number of accurate bits is doubled compared to the initial reciprocal approximation.

The 8^{th}, 9^{th}, 10^{th} and 11^{th} clock cycles are dedicated to conduct the second Newton-Raphson iteration. These are obviously the repetition of the 4^{th}, 5^{th}, 6^{th} and 7^{th} clock cycles, respectively. The exception here is that, in the last clock cycle, MUX4 selects the exact multiplication result, which is shown as CLA64_Out in Figure 1. The final reciprocal result is saved in both registers REG4 and REG5.

In this implementation, m is 12 and two Newton-Raphson iterations are applied to obtain more accurate result. Simulation results, among hundreds of normalized floating point numbers, showed that 52-bit accuracy including the implicit one is obtained. However, since an infinite precision to compute the final result is not available, the proposed design does not guarantee the correctness of the least significant bit of mantissa compared to IEEE compliant reciprocal operation. More clearly, in most of the cases investigated, the obtained reciprocal result is same as the reciprocal result provided by an IEEE compliant FPU with the default round to nearest even mode. There are also a few cases in which the computed reciprocal result is 1 ulp more than the results obtained by IEEE compliant FPU. A different initial approximation method and two Newton-Raphson iterations were used in the hardware implementation presented in [20] and it was reported that the similar accuracy for double precision floating-point number was obtained.

B. Reciprocal of an Interval

Reciprocal of an interval is straight forward and obeys interval division rules in which the dividend is unity. Reciprocal of an interval can be implemented with the algorithm given in Figure 2, where $X = [x_l, x_u]$ is input interval and its lower and upper endpoints are x_l and x_u, respectively. As seen from the algorithm, there are three cases. In the first case, the result interval is empty interval if the input interval is empty. The empty interval is denoted as $\oslash$ and corresponds to an interval that does not contain any real numbers. On IEEE compliant processors, empty intervals are represented using Not-a-Number (NaN) for each interval endpoint. In the second case, entire interval is returned as a result interval if input interval contains zero. The entire interval, denoted as $[-\infty, +\infty]$, corresponds to an interval that contains all real numbers. If input interval neither is an empty interval nor contains zero then the result interval is computed as it is shown in the last case of the algorithm, where $\bigtriangledown$ and $\bigtriangleup$ are used to indicate rounding directions towards negative infinity and positive infinity, respectively.

Reciprocation of an interval operand is performed as follows. It is assumed that x_l is saved in register REG1 and x_u is saved in register REG8. The CONTROL signal is set to zero in the 1^{st} clock cycle. As the multiplexor MUX1 selects the mantissa of the lower endpoint of the operand, the reciprocal operation follows the same procedure of the floating-point reciprocation. At the end of the 11^{th} clock cycle, the unrounded reciprocal of x_l is ready and saved in REG4.

In the 12^{th} clock cycle, the unrounded reciprocal of the lower endpoint of operand stored in register REG4 is rounded towards positive infinity by RND_UP unit. The upwards rounded result constitutes the upper endpoint of reciprocal interval and guarantees the containment. To round positive infinity, the RND_UP unit adds/subtracts *one* to/from the

least significant bit of the result saved in register REG4 if the lower endpoint of input interval is positive/negative. Rounded result, which forms the upper endpoint of the reciprocal interval result, is saved in register REG6, where WRITE_ENABLE2 signal is only set in the 12^{th} clock cycle.

Since the combined unit is pipelined, it is possible to initiate the computation for the reciprocal of upper endpoint in the 10^{th} clock cycle. In the same clock cycle, the second Newton-Raphson iteration for the reciprocal computation of the lower endpoint is performed using the second pipeline stage. Therefore, at the end of the 20^{th} clock cycle, the unrounded reciprocal of the upper endpoint is ready and the result is saved in register REG5.

In the 21^{st} clock cycle, available result in REG5 is read and RND_DOWN unit subtracts/adds *one* from/to the result in order to round the result towards negative infinity if the upper endpoint of input interval positive/negative. Within the same clock cycle, WRITE_ENABLE3 signal is set, and the result, which forms the lower endpoint of the reciprocal interval result, is saved in REG7. Note that, rounding units, RND_UP and RND_DOWN, does not increase the clock cycle time and performs their dedicated operations in one clock cycle.

As a conclusion, in twenty-one clock cycles, the reciprocal of an interval is computed and the results are available in the registers REG6 and REG7, which constitute the upper and lower endpoint of the reciprocal result, respectively. Thus, the latency of interval reciprocation is twenty-one clock cycles. Since initiation interval is the number of clock cycles that must elapse between issuing two operations of a given type, the initiation interval is eighteen clock cycles for interval reciprocal operation. When the combined unit is used to compute the reciprocal of a double precision floating-point number, the latency is eleven clock cycles and the initiation interval is nine clock cycles.

IV. Area and Delay Estimates

In order to estimate the area and the worst-case delay, the combined interval and floating-point reciprocal unit is implemented and simulated in VHDL. The design was synthesized using Mentor Graphics' *LeonardoSpectrum* synthesis tool and TSMC 0.25 micron CMOS standard cell library. Table IV gives area estimates for the components of the whole unit, which were obtained using an operating voltage of 2.5 Volts and a temperature of 25 degrees centigrade.

Note that, the column *# of Gates* in Table IV refers to the number of equivalent gates for each type of components and the results are normalized such that an equivalent gate corresponds to the area of a single minimum-size inverter. The unit requires 50,762 gates and 12 KB ROM.

The total area given in Table IV is calculated by adding the referred *# of Gates* values, taking into account that the 53-bit register is utilized four times, the 64-bit register is utilized four times and 53-bit Carry Lookahead Adder is utilized three times in the combined reciprocal unit.

Component	# of Gates	Reference in the Unit
REG (64-bit)	521	REG1, REG2, REG3, REG8
REG (53-bit)	536	REG4, REG5, REG6, REG7
MUX (3-to-1, 53-bit)	518	MUX2
MUX (2-to-1, 53-bit)	445	MUX3
MUX (4-to-1, 53-bit)	479	MUX4
MUX (2-to-1, 52-bit)	210	MUX1
CLA (53-bit)	1011	CLA_53
CLA (64-bit)	1154	CLA_64
INVERTERS	91	INV_51, OpMod
Dadda Tree Multiplier	40811	DTM_53
Total Area	**50762**	

TABLE IV
AREA ESTIMATES FOR EACH COMPONENT

Pipeline Stage	Delay (ns)
1	2.62
2	5.62
3	4.34
4	2.23
Clock Cycle Time	**5.62**

TABLE V
DELAY ESTIMATES FOR EACH PIPELINE STAGE

To determine the clock cycle time of the combined reciprocal unit, the four pipeline stages of the unit are synthesized. As shown in Table V, the second pipeline stage has the worst case delay and it determines the clock cycle time.

V. Discussion

When the combined reciprocal unit is used to compute the reciprocal of a floating-point number, the least significant bit of the result is not guaranteed to be same compared to a result based on an infinite precision. This may not be a problem especially for digital signal processing applications. Actually, single precision version of the same unit could be sufficient enough for digital signal processing and multimedia applications.

Even though the containment of the true result for interval reciprocal operation is not proven mathematically, simulations of hundreds of interval reciprocal computation showed that the true result is definitely contained in the computed reciprocal interval. However, the disadvantage of the hardware support for interval reciprocal operation is that the possibility of an additional 1 ulp error in each endpoint computation. For some cases, whose frequency is extremely low, if both endpoints are computed with an additional 1 ulp error, then the range of the result interval would be 2 ulp wider compared to the interval result obtained using an infinite precision.

Advantages of the design are its speed and its ability to replace the division operation for interval and floating-point arithmetic. If the combined interval and floating-point division unit given in [13] is used, then the floating-point and interval division operation takes twenty-eight and fifty-seven clock cycles, respectively. However, it is possible to complete floating-point division in fourteen cycles and the interval division operation in twenty-five cycles, if the reciprocal result is used in the subsequent multiplication operation, where the floating-point and interval multiplications take three and four clock cycles, respectively.

There are two drawbacks of replacing division operation with a reciprocal and a multiplication operations. One of them is that the proposed combined reciprocal unit requires more hardware than the hardware required to implement the most commonly used Radix-4 SRT division algorithm. However, hardware requirement of the reciprocal unit can be reduced if the existing multiplier is shared. It is reported by Oberman that the effect of this structural hazard on excess CPI is between 0.02 and 0.04 if a multiplication based-division is implemented with a latency of approximately thirteen clock cycles [21]. Of course, it may not be feasible to share existing multiplier for applications in which the number of division and multiplication operations are high. In that case, the multiplier unit can be shared if a processor has more than one multiplier like in [22]. The second drawback is that the subsequent multiplication operation may introduce an additional rounding error.

VI. Conclusion

This paper introduces a hardware unit that supports both interval and floating-point reciprocal operations. When the combined reciprocal unit is used to compute the reciprocal of a floating-point number, it takes eleven clock cycles and the least significant bit of the result is not guaranteed to be same compared to a result based on an infinite precision. When the combined reciprocal unit is used to compute the reciprocal of an interval, it takes twenty-one clock cycles and the true result is contained in the result interval.

References

[1] R. B. Kearfott, M. Dawande, K. Du, and C. Hu. Algorithm 737: INTLIB: A portable FORTRAN 77 interval standard function library. ACM *Trans. Math. Software*, 20:447–459, 1994.

[2] O. Knuppel. PROFIL/BIAS - a fast interval library. *Computing*, 53:277–288, 1994.

[3] E. Hyvonen and S. De Pascale. InC++ library family for interval computations. *Supplement to the Internat, Workshop on Applications of Interval Computations, ed. V. Kreinovich, Internat. J. Reliable Comput.*, pages 85–90, 1995.

[4] R. Klatte, U. Kulisch, M. Neaga, D. Ratz, and Ch. Ullrich. PASCAL-XSC*: Language Reference with Examples*. Springer-Verlag, 1991.

[5] R. Klatte, U. Kulisch, A. Wiethoff, C. Lawo, and M. Rauch. C-XSC*: A C++ Class Library for Extended Scientific Computing*. Springer-Verlag, 1993.

[6] W. V. Valter. ACRITH-XSC: A fortran-like language for verifi ed scientifi c computing. In E. Adams and U. Kulisch, editors, *Scientific Computing with Automatic Result Verification*, pages 45–70. Academic Press, 1993.

[7] J. S. Ely. The VPI software package for variable precision interval arithmetic. *Interval Computations*, 2:135–153, 1993.

[8] M.J. Schulte, V. Zelov, A. Akkas, and J.C. Burley. The interval-enhanced gnu fortran compiler. *Reliable Comput.*, (53):311–322, 1999.

[9] Interval arithmetic in the forte^{tm} fortran 95 compiler. Available at http://developers.sun.com/prodtech/cc/compilers_index.html.

[10] A. Akkas, M.J. Schulte, and J.E. Stine. Intrinsic compiler support for interval arithmetic. *Numerical Algorithms - Special Issue: SCAN'2002 International Conference*, 37:13–20, 2004.

[11] M. J. Schulte and E. E. Swartzlander, Jr. A hardware design and arithmetic algorithms for a variable-precision, interval arithmetic coprocessor. In *Proceedings of the 12th Symposium on Computer Arithmetic*, pages 163–171, July 1995.

[12] J. E. Stine and M. J.Schulte. A combined interval and fbating-point multiplier. In *8th Great Lakes Symposium on VLSI*, pages 208–213, February 1998.

[13] J. E. Stine and M. J. Schulte. A combined interval and fbating-point divider. In *Thirty-Second Asilomar Conf. on Signals, Systems and Computers*, pages 218–222, November 1998.

[14] J. E. Stine. *Design Issues for Accurate and Reliable Arithmetic*. PhD thesis, Graduate Department of Computer Science, Lehigh University, 2001.

[15] A. Akkas. A combined interval and fbating-point comparator/selector. In *IEEE 13th Internat. Conf. on Application-specific Systems, Architectures, and Processors*, pages 208–217, 2002.

[16] N. Takagi. Generating a power of an operand by a table look-up and a multiplication. In *Proceedings of the 13th Symposium on Computer Arithmetic*, pages 126–131, July 1997.

[17] S. C. Chapra and R. P. Canale. *Numerical Methods for Engineers*, volume 3. McGraw-Hill, 1998.

[18] U. Kucukkabak. Design and implementation of reciprocal unit for interval and fbating-point arithmetic. Master's thesis, Graduate School of Engineering, Koc University, 2005.

[19] *ANSI/IEEE 754-1985 Standard for Binary Floating-Point Arithmetic*, 1985.

[20] D. L. Fowler and J. E. Smith. An accurate, high speed implementation of division by reciprocal approximation. In *Proceedings of 9th Symposium on Computer Arithmetic*, pages 60–67, Sept. 1989.

[21] Stuart F. Oberman and Michael J. Flynn. Design issues in division and other fbating-point operations. *IEEE Transactions on Computers*, 46(2):154–161, February 1997.

[22] A. Akkas and M.J. Schulte. Effi cient dual-mode fbating-point multiplier design and implementation. *submitted to Journal of Systems Architecture*, 2004.