Hardware Design and Arithmetic Algorithms for a Variable-Precision, Interval Arithmetic Coprocessor

Michael J. Schulte and Earl E. Swartzlander, Jr.
Department of Electrical and Computer Engineering
University of Texas at Austin
Austin, TX 78712, USA

Abstract

This paper presents the hardware design and arithmetic algorithms for a coprocessor that performs variable-precision, interval arithmetic. The coprocessor gives the programmer the ability to specify the precision of the computation, determine the accuracy of the result, and recompute inaccurate results with higher precision. Direct hardware support and efficient algorithms for variable-precision, interval arithmetic greatly improve the speed, accuracy, and reliability of numerical computations. Performance estimates indicate that the coprocessor is 200 to 1,000 times faster than a software package for variable-precision, interval arithmetic. The coprocessor can be implemented on a single chip with a cycle time that is comparable to IEEE double-precision floating point coprocessors.

1: Introduction

As noted in [1], the computational speeds of the fastest computers have increased by a factor of roughly 100 during each of the last three decades, and an increase by a factor of 1,000 is expected during the present decade. This tremendous increase in computing power gives researchers the ability to solve previously intractable problems. The large number of arithmetic operations, however, makes it important to monitor and control errors in numerical computations.

As the number of arithmetic operations increases, the probability of inaccurate results due to roundoff error and catastrophic cancellation also increases. This calls for an increase in the precision of modern computers. Unfortunately, however, most modern computers only provide hardware support for floating point numbers with 64 bits or less. For example, computers which conform to the IEEE-754 double-precision floating point standard [2] use a 64-bit floating point format with an 11-bit exponent and 53-bit significand. As a result, today's numerically intensive applications may produce results which are completely inaccurate. On most computer systems, however, there is no efficient method to determine the accuracy of the result or increase the precision of the computation.

To overcome the numerical deficiencies of existing computer systems, several extended scientific programming languages that support variable-precision, interval arithmetic have been developed. These include ACRITH-XSC [3], PASCAL-XSC [4], C-XSC [5], and VPI [6]. Variable-precision arithmetic gives the programmer the ability to specify the precision (i.e., the number of bits or words in the significand) of the computation, based on the problem to be solved and the required accuracy of the results. Interval arithmetic [7] produces two values for each result, such that the true result is guaranteed to be between the two values. The distance between the two values indicates the accuracy of the result. The combination of variable-precision arithmetic and interval arithmetic gives the programmer the ability to set the precision of the computation, determine the accuracy of the result, and recompute inaccurate results using higher precision.

The main disadvantage of the extended scientific programming languages presented in [3-6] is their speed. Since these languages are designed for machines which support the IEEE 754 standard, all variable-precision, interval arithmetic operations are simulated in software. This adds tremendous overhead due to function calls, memory management, error and range checking, changing rounding modes, and exception handling. As reported in [8], changing the rounding mode on IEEE processors can take as long as executing six floating point additions, due to an inefficient user interface. The interval arithmetic routines discussed in [9] are approximately 40 times slower than their single-precision floating point equivalents. and routines which support variable-precision interval arithmetic (up to 56 decimal digits) are more than 1,200 times slower.

1063-6889/95 $4.00 © 1995 IEEE

To overcome the speed limitations of existing software techniques, direct hardware support is required. Early hardware designs for accurate arithmetic focused on supporting exact dot products [10], in which all arithmetic operations are mathematically exact and only a single rounding is performed at the very end. A survey of a hardware designs for computing exact dot product is presented in [11]. To produce accurate results when multiple computations are performed, exact dot product computations should be combined with variable-precision arithmetic [12]. Processors have also been designed which support variable-precision arithmetic, including CADAC [13], DRAFT [14], and CASCADE [15]. Although these processors improve the speed of variable-precision computations, they do not provide special instructions for interval arithmetic. CASCADE and CADAC use non-binary number representations, which complicates the hardware design and decreases compatibility with existing processors. DRAFT and CASCADE perform variable-precision integer arithmetic and do not have hardware support for floating point operations.

This paper presents a coprocessor which performs variable-precision, interval arithmetic. Because the arithmetic operations are implemented in hardware, this design offers a substantial speedup over existing software methods for controlling numerical error. It also uses non-redundant binary arithmetic, which simplifies the coprocessor design and enhances compatibility with other processors. An overview of interval arithmetic is given in Section 2. In Section 3, the number representation and hardware design of the coprocessor are presented. Section 4 describes hardware algorithms for the arithmetic operations, elementary function generation, dot product computation, and interval operations. Area, delay and performance estimates for the coprocessor are given in Section 5, followed by conclusions in Section 6.

2: Interval arithmetic

Interval arithmetic was originally proposed as a tool for bounding roundoff errors in numerical computations [7]. It is also used to determine the effects of approximation errors and errors that occur due to non-exact inputs. Interval arithmetic produces two values for each result. The two values correspond to the endpoints of an interval, such that the true result is guaranteed to lie on this interval. The width of the interval (i.e., the distance between the two endpoints) indicates the accuracy of the result. Because of its usefulness in monitoring numerical error, interval arithmetic has been applied to several numerical problems including global optimization, function evaluation, differential equations, finding roots of polynomials, and solving systems of equations [16].

As defined in [7], a closed interval $X = [a, b]$ consists of the set of real numbers between and including the two endpoints a and b (i.e., $X = \{x: a \leq x \leq b\}$). A real number c is equivalent to the degenerate interval $[c, c]$. Interval arithmetic specifies how to add, subtract, multiply, and divide intervals. When performing interval arithmetic on computers, the interval endpoints may not be representable. In this case, the lower endpoint is rounded towards negative infinity and the upper endpoint is rounded towards positive infinity.

3: Hardware design

This section gives an overview of the hardware design for the variable-precision, interval arithmetic coprocessor. The hardware is designed to handle the common case quickly, while still providing correct results and acceptable performance when extremely high precision is required. The hardware unit functions as a tightly-coupled coprocessor, which receives input data and instructions from the main processor. Standard floating point arithmetic is performed by the main processor, and the coprocessor handles all variable-precision, interval arithmetic. The hardware supports the four rounding modes specified in the IEEE 754 floating point standard: round-to-nearest-even, round toward positive infinity, round toward negative infinity, and round toward zero. The software interface to the coprocessor is described in [17].

The format for variable-precision numbers is shown in Figure 1. Each variable-precision number consists of a 16-bit exponent field (E), a sign bit (S), a 2-bit type field (T), a 5-bit significand length field (L), an index field (I), and a significand (F) which consists of $L+1$ significand words ($F[0]$ to $F[L]$). The exponent is represented with a bias of 32,768. The sign bit is zero if the number is positive and one if it is negative. The type field indicates if a number is infinite, zero, or not-a-number. The length field specifies the number of 32-bit words in the variable-precision number, and the index field points to the most significant word of the significand. The significand words are stored from most significant $F[0]$ to least significant $F[L]$. The significand is normalized between 1 and 2, so that its most significant bit is always one. The value of a variable-precision floating point number VP is

$$VP = (-1)^S \times F \times 2^{E-32,768}$$

For variable-precision numbers, the maximum significand length is 32 32-bit words, or 1,024 bits. This gives a maximum precision of around 313 decimal digits and a range of

$$[2^{-32,768}, 2^{32,769}] \approx [10^{-9,864}, 10^{9,864}]$$

For comparison, the format for IEEE double-precision numbers is shown in Figure 2. These numbers consist of a sign bit (*S*), an 11-bit exponent (*E*), and a 52-bit significand (*F*). The exponent is represented using excess 1,023. The significand is normalized between 1 and 2 and uses a ***hidden one***. The value of an IEEE double-precision number *DP* is

$$DP = (-1)^S \times 1.F \times 2^{E-1,023}$$

IEEE double-precision numbers have a maximum precision of around 16 decimal digits and a range of

$$[2^{-1,022}, 2^{1,024}] \approx [10^{-307}, 10^{308}]$$

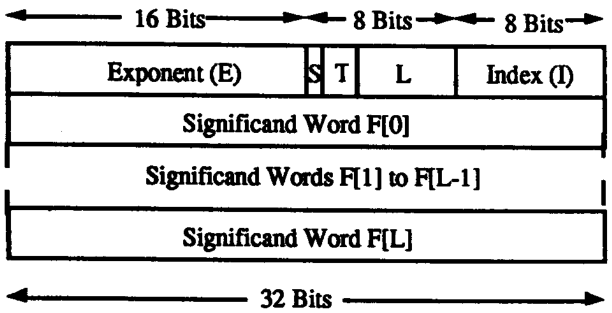

Figure 1: Variable-precision number format.

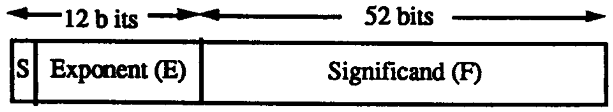

Figure 2: IEEE double-precision format.

A block diagram of the hardware unit which performs variable-precision, interval arithmetic is shown in Figure 3. Control signals are shown as dashed lines. The significand and exponent data paths are depicted as bold and plain lines, respectively. The main components of the hardware unit are a register file, a 32-bit by 32-bit multiplier, a 64-bit adder, a long accumulator consisting of 64 64-bit segments, and a 64-bit shifter. The hardware unit also has a 16-bit exponent adder and data path control unit which determines the exponent of the result and controls the register file, long accumulator, selector, and shifter.

The register file consists of two memory units: a 64-word by 32-bit header memory, and a 256-word by 32-bit significand memory, as shown in Figure 4. Each header words contains the exponent, sign, type, length, and index of the number. When operations are performed on variable-precision numbers, the header word is first read from memory. In the following cycles, the significand is accessed based on the value of the index field. This two-level design allows a maximum of 64 variable-precision numbers to be stored on the coprocessor chip. Other variable-precision numbers are stored in main memory. The header and significand memories each have two read ports and one write port.

Significand words are read from the register file and go into the multiplier, selector, or long accumulator. The selector performs comparison operations and determines which values go into the adder and the shifter. The shifter shifts a value by up to 64 bits, before it enters the adder.

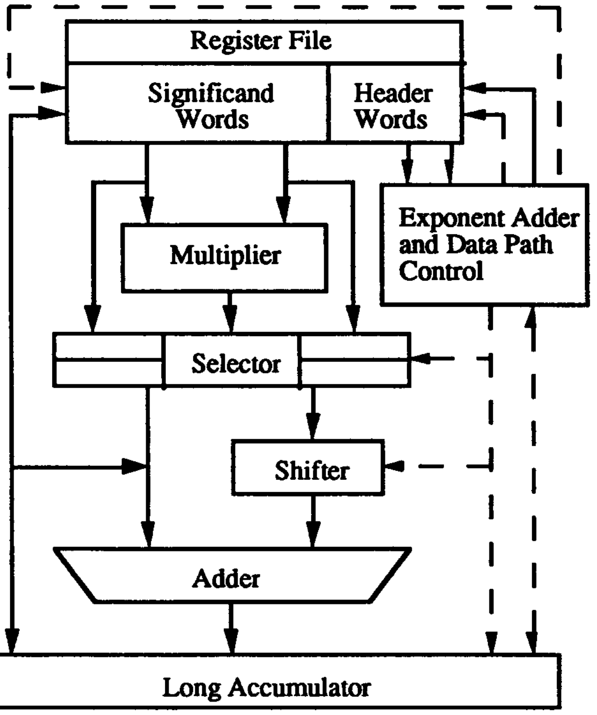

Figure 3: Hardware unit.

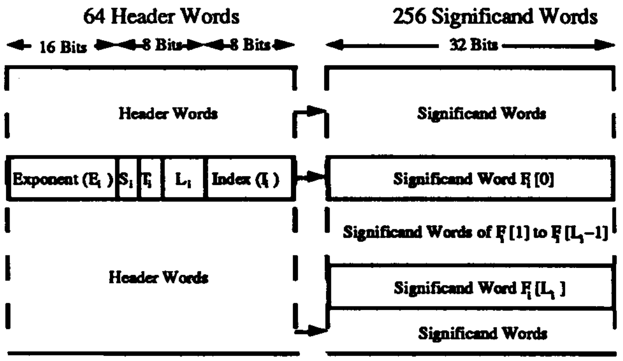

Figure 4: Register file.

The long accumulator stores intermediate variable-precision results. It functions as an extremely long fixed point register and is useful for performing variable-precision arithmetic algorithms without roundoff error or overflow. The implementation of the long accumulator is similar to the one presented in [28]. The long accumulator consists of a 64-word by 64-bit dual-port RAM, 64 2-bit flags, carry resolution and flag generation logic, and rounding and normalization control. Variable-precision values are stored in the dual-port-RAM, which contains one write port and one read/write port.

When adding a number to the long accumulator, it is possible for the carry to propagate over several segments, resulting in a large number of additions. To prevent this, each segment of the long accumulator has a 2-bit flag associated with it that tells if the bits in the segment are all ones, all zeros, or neither [28]. A carry propagating into a segment that contains all ones will cause the flag to signal all zeros. Similarly, a borrow from a segment which has all zeros causes the flag signal all ones. If a carry or a borrow come into a segment which has neither all ones nor all zeros, the carry is not propagated beyond that segment.

Figure 5 demonstrates the accumulation process using five-bit segments, when the new addend is also five bits. The addend is added to two of the segments in the long accumulator. If a carry occurs after the second addition, it is added to the first word that does not contain all ones. The 2-bit flags, and flag generation and carry resolution logic determine the segment to which the carry is added. Once the final result is computed, it is normalized, rounded to a specified precision, and stored back in the register file. The *all ones* and *all zeros* flags simplify normalizing and rounding the result, since they indicate the first non-zero word of the result and help to determine the sticky bit, which is needed to implement correct rounding.

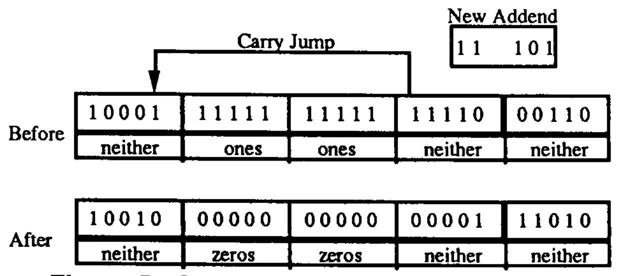

Figure 5. Accumulation of a new addend.

4: Arithmetic algorithms

This section describes hardware algorithms for the arithmetic operations, elementary function generation, accurate dot products, and interval operations. All intervals are stored in the register file using consecutive registers words, with the lower endpoint stored first. For the variable-precision arithmetic operations, the two operands are denoted as A and B with significands F_A and F_B and exponents E_A and E_B, respectively. For variable-precision, interval arithmetic operations, the intervals are $X = [a, b]$ and $Y = [c, d]$. The symbols ∇ and Δ denote rounding toward negative and positive infinity, respectively.

For variable-precision addition and subtraction, E_A and E_B are compared to determine the greater exponent. The operand with the greater exponent has its significand written into the long accumulator. Assuming that $E_B \geq E_A$ and addition is being performed, F_B is written into the long accumulator. In the following cycles, F_A is added to the long accumulator using a series of 64-bit additions, in which the carry-out of the i^{th} addition is used as carry-in for the $(i+1)^{th}$ addition. After the final result is computed, the long accumulator is normalized and rounded to a specified precision. The final result is then stored in the register file.

Interval addition and subtraction are defined [7] as:

$$X + Y = [\nabla(a + c), \Delta(b + d)]$$
$$X - Y = [\nabla(a - d), \Delta(b - c)]$$

Thus, interval addition (subtraction) requires two variable-precision additions (subtractions). The lower endpoint is computed first and rounded toward negative infinity. The upper endpoint is then computed and rounded towards positive infinity.

Variable-precision multiplication is performed by using the multiplier, adder, and long accumulator to generate and accumulate 64-bit partial products. During the first cycle, the exponents of the two operands are added to compute the exponent of the product. Each subsequent cycle, 32 bits of the multiplier are multiplied by 32 bits of the multiplicand to produce a new 64-bit partial product which is added to the sum of the previously accumulated partial products. The sum of the partial products is stored in the long accumulator. Figure 6 shows the multiplication process for a 4-word by 4-word (128-bit by 128-bit) multiply. To reduce carry propagation, the less significant partial products are generated first.

If the multiplicand and multiplier contain m and n words $(m \geq n)$, then $m \cdot n$ single-precision multiplications and additions are required. If the final result is rounded to m words, the n least significant words of the product do not need to be computed. In this case, a method proposed in [19] is used to reduce the number of single-precision multiplications and additions to

$$m \cdot n - (n^2 - 3n) / 2 - 1$$

This is possible, because only the $m + 1$ most significant columns of partial products are likely to contribute to the rounded product. For the 4 word by 4 word multiplication shown in Figure 6, the partial products $A_0 \cdot B_0$, $A_0 \cdot B_1$, and $A_1 \cdot B_0$ are not likely to effect the product when it is rounded to 4 words. A quick test determines if the omitted partial products can change the value of the rounded product. If they can, the product is computed to full precision and then rounded.

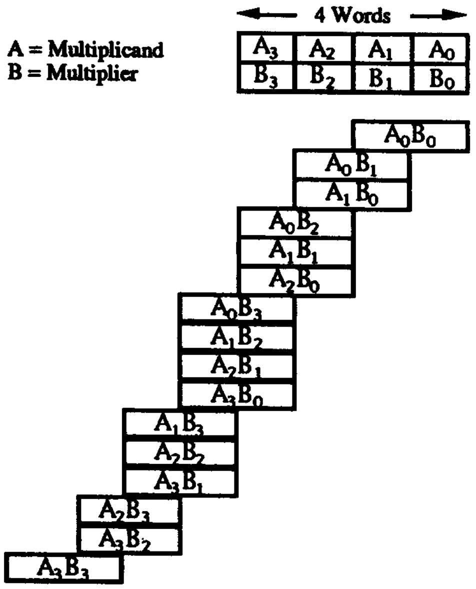

Figure 6: Variable-precision multiplication.

Interval multiplication is defined [7] as:

$X \times Y = [\nabla min(ac,ad,bc,bd), \Delta max(ac,ad,bc,bd)]$

Rather than computing all four products and then comparing the results, the endpoints to be multiplied together to form the upper and lower endpoints of the result are determined by examining the sign bits *a*, *b*, *c*, and *d* [18]. With this technique, only two variable-precision multiplications are required to perform interval multiplication, unless

$a < 0 < b \quad AND \quad c < 0 < d$

The method proposed in [19] guarantees correct directional rounding.

The algorithm used to perform division A/B is a variation of the short reciprocal divide algorithm [20], which is modified for variable-precision, interval arithmetic. This algorithm initially calculates a reciprocal approximation $R \approx 1/B$ which is accurate to 34 bits. During the i^{th} iteration, a quotient digit q_i is added to the quotient approximation Q_i through the computations

$$q_i = R \cdot P'_i$$
$$P_{i+1} = P_i - q_i \cdot B$$
$$Q_{i+1} = Q_i + q_i$$

where P_i is the i^{th} partial remainder and P'_i is the i^{th} partial remainder truncated to 34 bits. The quotient digit q_i is computed by rounding $R \cdot P'_i$ away from zero to 32 bits. Initially, $P_0 = A$ and $Q_0 = 0$. After computing the quotient, a correction step produces a correctly rounded quotient and exact remainder for any of the IEEE rounding modes [21].

Details for the variable-precision short reciprocal divide algorithm and a similar algorithm for square root are presented in [22]. The divide algorithm requires $n^2 + n$ single precision multiplications and $2n^2 + 2n$ single precision additions to divide two n word numbers. The square root algorithm requires $(n^2 + 3n)/2$ single precision multiplications and $(3n^2 + 5n)/2$ single precision additions to compute the square root of an n word number. These algorithms require 3 to 5 times fewer arithmetic operations than well-known algorithms based on Newton-Raphson iteration [23].

Interval division is defined [7] as:

$X/Y = [\nabla min(a/c,a/d,b/c,b/d), \Delta max(a/c,a/d,b/c,b/d)]$

if Y does not contain zero (i.e., $c < 0$ OR $d > 0$). Otherwise, the quotient interval is infinite and *extended interval arithmetic* is used [18]. The sign bits are examined to determine which endpoints are divided to compute the endpoints of the quotient, and only two variable-precision divisions are required. An interval square root is defined as:

$$\sqrt{X} = [\nabla(\sqrt{a}), \Delta(\sqrt{b})]$$

provided that $a \geq 0$. Otherwise, one or both endpoints of the result are not-a-number.

Elementary function evaluation is performed by polynomial approximations, using variations of the algorithms given in [24], [25]. These approximations have the form

$$f(x) \approx p_n(x) = a_0 + a_1 \cdot x + \ldots + a_n \cdot x^n = \sum_{i=0}^{n} a_i \cdot x^i$$

where $f(x)$ is the function to be approximated, $p_n(x)$ is a polynomial of degree n, and a_i is the coefficient of the i^{th} term. The function is approximated on a specified input interval and argument reduction is employed for values outside this interval, as specified in [24]. To reduce the number of multiplications, Horner's rule is applied, which has the form

$$p_n(x) = a_0 + x \cdot (a_1 + \ldots + x \cdot (a_{n-1} + x \cdot a_n) \ldots)$$

Thus, each term in the polynomial requires one addition and one multiplication.

Interval arithmetic is defined for the elementary functions as follows: If an elementary function $f(x)$ is monotonically increasing on $X = [a, b]$, the resulting interval is

$$f(X) = [\nabla f(a), \Delta f(b)]$$

For monotonically decreasing functions the resulting interval is

$$f(X) = [\nabla f(b), \Delta f(a)]$$

For functions which are neither monotonically increasing nor decreasing on $[a, b]$, the function is evaluated at its local minimum and maximum and at the interval endpoints to determine the resulting interval. For example, if $X = [-1, 2]$, then $\sin(X) = [\nabla\sin(-1), 1]$, since $sin(X)$ has a local maximum of 1 at $\pi / 2$.

Correct rounding of the elementary functions is prohibitively expensive, since there is no known analytical method to determine in advance the number of guard digits required to produce a correctly rounded result [26]. Instead, faithful rounding [27] is used to guarantee that the maximum error is no greater than two units in the last place and that all intervals endpoints are rounded in the proper direction.

Accurate dot products are essential for scientific applications. The dot product of two k-element vectors

$$V = [v_1, v_2, \ldots, v_k] \qquad W = [w_1, w_2, \ldots, w_k]^T$$

is defined as:

$$V \cdot W = \sum_{i=1}^{k} v_i \cdot w_i$$

For each v_i, w_i pair, their product is computed and added to the long accumulator. The segments chosen from the long accumulator and the amount that the product is shifted is determined by the exponent of the new product. The *all ones* and *all zeros* flags prevent carries from propagating over several segments. After the entire dot product is computed, it is normalized, rounded, and stored back in the register file.

To compute an interval dot product, the lower endpoint of the dot product is computed, followed by the upper endpoint. The lower endpoint of the dot product is computed by calculating the lower endpoint of each interval multiplication and adding it to the long accumulator. Once the lower endpoints of all k products are accumulated, their sum is normalized, rounded toward negative infinity, and stored back in the register file. After clearing the long accumulator, the upper endpoint of the dot product is computed by accumulating the upper endpoint of each interval multiplication. After, the upper endpoint is computed, it is normalized, rounded toward positive infinity, and stored back in the register file..

To efficiently support interval arithmetic, several interval operations are provided. These include interval intersection, hull, absolute value, width, and midpoint, which are defined as:

$$intersection(X, Y) = [max(a, c), min(b, d)]$$
$$hull(X, Y) = [min(a, c), max(b, d)]$$
$$abs(X) = max(|a|, |b|)$$
$$width(X) = b - a$$
$$midpoint(X) = (a + b)/2$$

The interval intersection and hull operations take two variable-precision, intervals and return a variable-precision interval. The midpoint, width, and absolute value operations take one variable-precision interval and return a variable-precision floating point number. Interval relational operators such as equal to, subset, superset, is-contained-in, and disjointness are also provided as defined in [4].

The operand selector is used to determine minimum and maximum values for interval intersection, hull, and absolute value. If the exponents and the signs of the two numbers being compared are the same, then the selector compares their significand words from most significant to least significant to determine which number is greater. The absolute value of a variable-precision number is computed by setting its sign bit to zero. The midpoint and width operations are implemented using the variable-precision addition and subtraction algorithms, respectively. The division by two in the midpoint operation is implemented by decrementing the exponent of $a + b$.

5: Area, delay and performance estimates

Table 1 gives area and delay estimates for the variable-precision, interval arithmetic coprocessor (VPIAC). These estimates are based on data from a 1.0 micron CMOS standard cell library [29]. The estimates for the multiplier assume that multiplication is implemented using a Reduced Area multiplier [30], followed by carry-lookahead addition. The area of each component is estimated by calculating the total size of the macrocells (e.g., AND gates, full adders, half adders, etc.) which make up the component, plus an additional 50 percent for internal wiring [29]. The total area is estimated as the sum of the component areas plus an additional 60 percent for control logic, global routing, unused space, and pad area. The total chip area is estimated as 101.9 mm^2. The delay for each component is computed by taking the worst case delay of the critical path and adding 25 percent for unexpected delays and clock skew [29]. The multiplier has the longest component delay which is 27.8 ns; 14.0 ns for partial product reduction and 13.8 ns for carry-lookahead addition. Assuming these two stages of the multiplication are pipelined and allowing an additional 2 ns for register latching, the coprocessor can have a cycle time of 16 ns (60 MHz).

Table 2 gives the area and delay estimates for an IEEE double-precision floating point coprocessor (IEEE DPC). It has a 53-bit by 53-bit multiplier, a 106-bit adder, a 32-word by 64-bit register file, a 106-bit normalizer, a 106-bit shifter, an 11-bit exponent add/subtract unit, and 53 and 106-bit latches. It requires a total area of 100.8 mm^2. The worst case delay comes from the multiplier, which has a component delay of 34.6 ns; 18.0 ns for partial product reduction and 16.6 ns for carry-lookahead addition. If these two stages of the multiplication are pipelined, the coprocessor can have a cycle time of 20 ns (50 MHz). Compared to the IEEE DPC, the VPIAC has almost the same area and a cycle time which is 20 percent shorter. The shorter cycle time of the variable-precision, interval arithmetic coprocessor is due to its narrower data path.

Table 1: Estimates for the VPIAC

Unit	*Area (mm^2)*	*Delay(ns)*
Multiplier	15.2	27.8
Carry-lookahead adder	2.1	13.8
Significand words	17.8	7.4
Header words	4.4	6.6
Long accumulator	13.0	7.0
Shifter	3.9	8.2
Operand selector	4.1	3.5
Exponent add/subtract	0.6	4.4
Latches	2.6	2.0
Global routing, pads, etc.	38.2	*
Total	101.9	*

Table 2: Estimates for an IEEE DPC.

Unit	*Area (mm^2)*	*Delay(ns)*
Multiplier	37.4	34.6
Carry-lookahead adder	4.3	16.6
Register file	4.4	6.2
Normalizer	7.2	9.0
Shifter	6.9	8.8
Exponent add/subtract	0.4	4.0
Latches	2.4	1.8
Global routing, pads, etc.	37.8	*
Total	100.8	*

Execution time estimates for the VPIAC are shown in Tables 3, 4, and 5 for interval addition, multiplication, and division. The precision of the computation is varied from 32 to 1,024 bits (i.e., from 1 to 32 words). For comparison purposes the execution times of the same operations using the VPI software package [6] are given, along with the speedup achieved by the variable-precision, interval arithmetic coprocessor.

The execution times for the VPIAC are determined by multiplying the number of cycles to perform the operation by the cycle time (16 ns). The number of cycles needed to perform interval addition, multiplication, and division with a precision of *n* words (i.e., *32n* bits) are

$$Cycles\_interval\_add = 4n + 12$$
$$Cycles\_interval\_mult = 2n^2 + 4n + 22$$
$$Cycles\_interval\_div = 6n^2 + 8n + 38$$

The execution times for the VPI software package were determined by running 1,000 iterations of the operation on a 40 MHz Sparc processor and taking the average execution time.

The speedup is computed as

$$Speedup = \frac{Execution\ time\ VPI}{Execution\ time\ VPIAC}$$

Based on the values given in Tables 3, 4, and 5, the VPIAC is around 200 to 1,000 times fasrer than the VPI software package, For the three arithmetic operations examined, interval division has the largest speedup and the interval addition has the smallest.

Table 3: Interval addition execution times (sec).

Precision (bits)	*VPIAC*	*VPI*	*Speedup*
32	$2.6{\cdot}10^{-7}$	$9.6{\cdot}10^{-5}$	369
64	$3.2{\cdot}10^{-7}$	$1.1{\cdot}10^{-4}$	343
128	$4.5{\cdot}10^{-7}$	$1.3{\cdot}10^{-4}$	289
256	$7.0{\cdot}10^{-7}$	$1.9{\cdot}10^{-4}$	271
512	$1.2{\cdot}10^{-6}$	$2.9{\cdot}10^{-4}$	242
1,024	$2.2{\cdot}10^{-6}$	$5.0{\cdot}10^{-4}$	227

Table 4: Interval multiplication execution times (sec).

Precision (bits)	*VPIAC*	*VPI*	*Speedup*
32	$4.2{\cdot}10^{-7}$	$1.1{\cdot}10^{-4}$	262
64	$5.4{\cdot}10^{-7}$	$1.9{\cdot}10^{-4}$	352
128	$9.9{\cdot}10^{-7}$	$5.1{\cdot}10^{-4}$	515
256	$2.7{\cdot}10^{-6}$	$1.7{\cdot}10^{-3}$	630
512	$9.1{\cdot}10^{-6}$	$6.6{\cdot}10^{-3}$	725
1,024	$3.3{\cdot}10^{-5}$	$2.6{\cdot}10^{-2}$	788

Table 5: Interval division execution times (sec).

Precision (bits)	*VPIAC*	*VPI*	*Speedup*
32	$8.3{\cdot}10^{-7}$	$5.6{\cdot}10^{-4}$	675
64	$1.2{\cdot}10^{-6}$	$1.1{\cdot}10^{-3}$	917
128	$2.7{\cdot}10^{-6}$	$2.8{\cdot}10^{-3}$	1,037
256	$7.8{\cdot}10^{-6}$	$8.2{\cdot}10^{-3}$	1,051
512	$2.7{\cdot}10^{-5}$	$2.8{\cdot}10^{-2}$	1,037
1,024	$1.0{\cdot}10^{-4}$	$1.0{\cdot}10^{-1}$	1,000

6: Conclusions

This paper presented a coprocessor which implements variable-precision, interval arithmetic. Efficient hardware algorithms for the arithmetic operations, elementary function generation, accurate dot products, and interval operations help to improve performance. The coprocessor gives the programmer the ability to determine the accuracy of the result and recompute inaccurate results using higher precision. It can also be used to evaluate the accuracy of programs before running them on a general purpose processor.

By providing hardware support for variable-precision, interval arithmetic, a substantial speedup over existing software methods is achieved. Performance estimates indicate that the coprocessor is around 200 to 1,000 times faster than the VPI software packages for interval addition, multiplication, and division. The variable-precision interval arithmetic algorithms presented in this paper have been simulated in C++. A behavioral level simulation of the coprocessor using VHDL is in progress.

References

[1] E. Adams and U. Kulisch, "Introduction," *Scientific Computing with Automatic Result Verification (E. Adams and U. Kulisch eds.)*, Academic Press, Inc., pp. 1-12, 1993.

[2] "IEEE Standard 754 for Binary Floating Point Arithmetic," *ANSI/IEEE Standard No. 754*, American National Standards Institute, Washington DC, 1985.

[3] W.V. Walter, "Acrith-XSC A Fortran-like Language for Verified Scientific Computing," *Scientific Computing with Automatic Result Verification (E. Adams and U. Kulisch eds.)*, Academic Press, Inc., pp. 45-70, 1993.

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

[6] J.E. Ely, "The VPI Software Package for Variable-Precision Interval Arithmetic," *Interval Computations*, vol. 2, pp. 135-153, 1993.

[7] R.E. Moore, *Interval Analysis*, Prentice-Hall, Englewood Cliffs, N.J., 1966.

[8] A. Knofel, "A Hardware Kernel for Scientific/Engineering Computations," *Scientific Computing with Automatic Result Verification (E. Adams and U. Kulisch eds.)*, Academic Press, Inc., pp. 549-570, 1993.

[9] E.K. Reuter *et al.*, "Some Experiments Using Interval Arithmetic," *Proceedings of the 4th Symposium on Computer Arithmetic*, pp. 75-81, 1978.

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

[11] G. Bohlender, "What Do We Need Beyond IEEE Arithmetic?," *Computer Arithmetic and Self-Validating Numerical Methods (C. Ullrich ed.)*, Academic Press, Inc., pp. 1-32, 1990.

[12] W.M. Kahan and E. LeBlanc, "Anomalies in the IBM ACRITH Package," *Proceedings of the 7th Symposium on Computer Arithmetic*, pp. 322-331, 1985.

[13] M.S. Cohen, T.E. Hull, and V.C. Hamarcher, "CADAC: A Controlled-Precision Decimal Arithmetic Unit," *IEEE Transactions on Computers*, Vol. C-32, pp. 370-37 7, 1983.

[14] D.M. Chiarulli, W.G. Rudd, and D.A. Buell, "DRAFT: A Dynamically Reconfigurable Processor for Integer Arithmetic," *Proceedings of the 7th Symposium on Computer Arithmetic*, pp. 309-318, 1985.

[15] T.M. Carter, "Cascade: Hardware for High/Variable Precision Arithmetic," *Proceedings of the 9th Symposium on Computer Arithmetic*, pp. 184-191, 1989.

[16] R.E. Moore (Ed.), *Reliability in Computing: The Role of Interval Methods in Scientific Computations*, Academic Press, 1988.

[17] M.J. Schulte and E.E. Swartzlander, Jr., "A Software Interface and Hardware Design for Variable-Precision Interval Arithmetic," *Interval Computations*, 1995 (in press).

[18] E. Hansen, *Global Optimization Using Interval Analysis*, Marcel Dekker, New York, 1992.

[19] W. Krandick and J.R. Johnson, "Efficient Multiprecision Floating Point Multiplication with Optimal Directional Rounding," *Proceedings of the 11th Symposium on Computer Arithmetic*, pp. 228-233, 1993.

[20] D.W. Matula, "A Highly Parallel Arithmetic Unit for Floating Point Multiply, Divide with Remainder and Square Root with Remainder," *SCAN-89*, Basel, October 1989.

[21] G. Bohlender, W. Walter, P. Kornerup, and D.W. Matula, "Semantics for Exact Floating Point Operations," *Proceedings of the 10th Symposium on Computer Arithmetic*, pp. 22-27, 1991.

[22] M.J. Schulte and E.E. Swartzlander, Jr. , "Algorithms for Exact Variable-Precision Divide and Square Root", in preparation.

[23] R.P. Brent, "The Complexity of Multiple-Precision Arithmetic," *The Complexity of Computational Problem Solving (R.S. Andressen and R.P. Brent eds.)* University of Queensland Press, pp. 126-165, 1976.

[24] K. Braune, "Standard Functions for Real and Complex Point and Interval Arguments with Dynamic Accuracy," *Computing with Automatic Result Verification (H.J. Stetter and U. Kulisch eds.)*, Springer-Verlag, Wien, pp. 159-184, 1988.

[25] D.M. Smith, "Efficient Multiprecision Evaluation of Functions," *Mathematics of Computation*, Vol. 52, pp. 131-134, 1989.

[26] M.J. Schulte and E.E. Swartzlander, Jr., "Hardware Designs for Exactly Rounded Elementary Functions," *IEEE Transactions on Computer*, Vol. 43, pp. 964-973, August, 1994.

[27] M. Daumas and D.W. Matula, "Rounding of Floating Point Intervals," *Research Report 93-06*, Laboratoire de l'Informatique du Paralle'lisme, Ecole Normale Supe'rieure de Lyon, France, 1993.

[28] A. Knofel, "Fast Hardware Units for the Computation of Accurate Dot Products," *Proceedings of the 10th Symposium on Computer Arithmetic*, pp. 70-75, 1991.

[29] *LSI Logic 1.0 Micron Cell-Based Products Databook*, LSI Logic Corporation, Milpitas, California, 1991.

[30] K.C. Bickerstaff, M.J. Schulte, and E.E. Swartzlander, Jr., "Reduced Area Multipliers," *Proceedings 1993 Application Specific Array Processors*, pp. 478-489, 1993.