Reliable Computing 2 (1) (1996), pp. 47–62

Variable-precision, interval arithmetic coprocessors

MICHAEL J. SCHULTE and EARL E. SWARTZLANDER, JR.

This paper presents hardware designs, arithmetic algorithms, and numerical applications for variable-precision, interval arithmetic coprocessors. These coprocessors give the programmer the ability to set the initial precision of the computation, determine the accuracy of the results, and recompute inaccurate results with higher precision. Variable-precision, interval arithmetic algorithms are used to reduce the execution times of numerical applications. Three hardware designs with data paths of 16, 32, and 64 bits are examined. These designs are compared based on their estimated chip area, cycle time, and execution times for various numerical applications. Each coprocessor can be implemented on a single chip with a cycle time that is comparable to IEEE double-precision floating point coprocessors. For certain numerical applications, the coprocessors are two to four orders of magnitude faster than a conventional software package for variable-precision, interval arithmetic.

Интервальные арифметические сопроцессоры переменной разрядности

М. Й. Шульте, Е. Е. Шварцландер, мл.

Представлены конструкция аппаратуры, используемые арифметические алгоритмы и приложения к решению численных задач для интервальных арифметических сопроцессоров переменной разрядности. Эти сопроцессоры позволяют программисту устанавливать начальную разрядность вычислений, определять точность результатов и заново вычислять неточные результаты с большей разрядностью. Для уменьшения времени выполнения в численных приложениях используются интервально-арифметические алгоритмы переменной разрядности. Рассмотрены три аппаратные схемы с шиной данных шириной 16, 32 и 64 бита. Эти схемы сравниваются по требуемой площади кристалла, продолжительности рабочего цикла и быстродействию в различных численных приложениях. Каждый из этих сопроцессоров может быть реализован на одном кристалле с рабочей частотой, сравнимой с сопроцессорами плавающей точки двойной точности стандарта IEEE. В некоторых численных приложениях наши сопроцессоры на два-четыре порядка быстрее, чем распространенные программные пакеты, реализующие интервальную арифметику переменной разрядности.

1. Introduction

Roundoff error and catastrophic cancelation in scientific computations can lead to results that are completely inaccurate [5, 26]. On most computer systems, however, there is no efficient method to increase the precision of the computation or determine the accuracy of results. Consequently, programmers are often forced to spend extra time developing and testing applications to ensure that they produce accurate and reliable results.

To improve the accuracy and reliability of numerical computations, several software tools have been developed. Software packages, such as [4, 6, 35], support variable-precision arithmetic, which gives the programmer the ability to specify the precision of the computation based on the problem to be solved and the desired accuracy of the results. Interval arithmetic libraries [16, 20] provide software support for fixed-precision interval arithmetic [1, 25]. Scientific programming languages [32] provide special instructions and data types for interval arithmetic and exact dot products [22]. Recently, techniques for variable-precision arithmetic and interval arithmetic have been combined in the extended scientific programming languages PASCAL–XSC [17], C–XSC [18], ACRITH–XSC [33], and VPI [12]. These languages provide data types and special instructions for variable-precision numbers, intervals, complex numbers, vectors, and matrices. When traditional numerical tools are inadequate, computer algebra systems provide symbolic or exact solutions [9, 34].

The main disadvantage of software tools for accurate and reliable arithmetic is their speed. Since the arithmetic operations are simulated in software, tremendous overhead occurs due to function calls, memory management, error and range checking, expression manipulation, changing rounding modes, and exception handling. The interval arithmetic routines discussed in [27] are approximately 40 times slower than their single-precision floating point equivalents. Routines that support variable-precision, interval arithmetic (up to 56 decimal digits) are more than 1,200 times slower than the corresponding single-precision routines. Arithmetic operations in the arbitrary precision library discussed in [31] are 50 to 100 times slower than equivalent floating point operations in hardware, even when no additional precision is required. Certain application programs that use computer algebra systems are approximately 3,000 times slower than equivalent numerical programs written in C [7].

To overcome the speed limitation of existing software tools, direct hardware support is required. To improve the accuracy and performance of vector and matrix operations, processors that support exact dot products have been designed [2, 13, 19]. To facilitate the use of interval arithmetic, these processors provide the four rounding modes specified by the IEEE 754 floating point standard [15]. Other processors, including CADAC [11], DRAFT [10], and Cascade [8], have been designed to provide hardware support for variable-precision arithmetic. Although these processors improve the speed of variable-precision computations, they do not provide special instructions for interval arithmetic or vector and matrix operations.

This paper presents hardware designs, arithmetic algorithms, and numerical applications for variable-precision, interval arithmetic coprocessors (VPIACs). Section 2 gives an overview of the number representation and hardware design of the coprocessors. Section 3 presents the algorithms used to perform variable-precision, interval arithmetic. In Section 4, area and delay estimates are given for VPIACs with data paths of 16, 32, and 64 bits. Cycle counts for arithmetic and interval operations are reported in Section 5, along with execution times for dot product computation, polynomial evaluation, and interval Newton methods. Conclusions are given in Section 6. A software interface to the VPIACs is presented in [29]. This paper is an extension of the research presented in [30].

2. Hardware design

This section gives an overview of the number representation and hardware design for the VPIACs. The hardware is designed to handle the common case quickly, while still providing correct results and acceptable performance when extremely high precision is required. Each VPIAC functions as a tightly-coupled coprocessor that receives input data and instructions from the main processor. Standard floating point arithmetic is performed by the main processor, and the VPIAC handles all variable-precision, interval computations. The hardware supports the

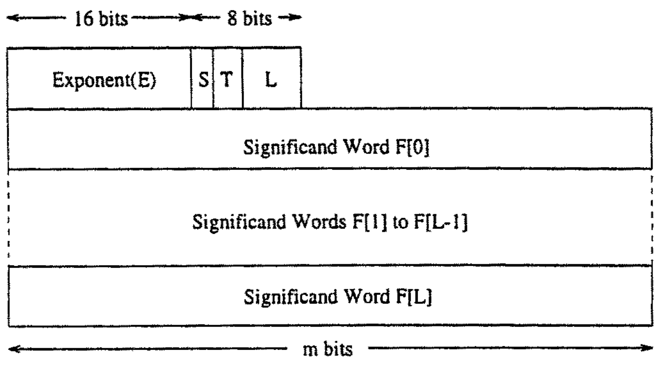

Figure 1. Variable-precision floating point format

four rounding modes specified in the IEEE 754 floating point standard [15]. In the following discussion, the data path size (i.e., the number of bits per word in the significand (mantissa) of the variable-precision number) is denoted by m, and a VPIAC with a data path of m bits is referred to as a m-bit VPIAC.

The format for variable-precision numbers is shown in Figure 1. Intervals are represented by two variable-precision numbers, which correspond to the interval endpoints. 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), and a significand (F) that 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 m-bit words in the significand. The words of the significand are stored from most significant $F[0]$ to least significant $F[L]$. The significand is normalized between 1 and 2. The value of a variable-precision floating point number VP is

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

Variable-precision numbers have a maximum precision of $32m$ bits and their range is approximately

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

In comparison, IEEE double-precision floating point numbers have a maximum precision of 53 bits and their range is approximately

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

A block diagram of the hardware unit that performs variable-precision, interval arithmetic is shown in Figure 2. 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 the register file, a m-bit by m-bit multiplier, a $2m$-bit adder, a 4 word by $2m$-bit selector, a long accumulator consisting of 64 $2m$-bit segments, a $2m$-bit shifter, and a 16-bit exponent adder and data path control unit.

The register file consists of two memory units: a 64-word by 32-bit header memory, and a 256-word by m-bit significand memory. Each header word contains the exponent, sign, type, and length of the variable-precision number, along with an index that points to the most significant word of the corresponding significand. When operations are performed on variable-precision numbers, the header words are first read. In the following cycles, the significand

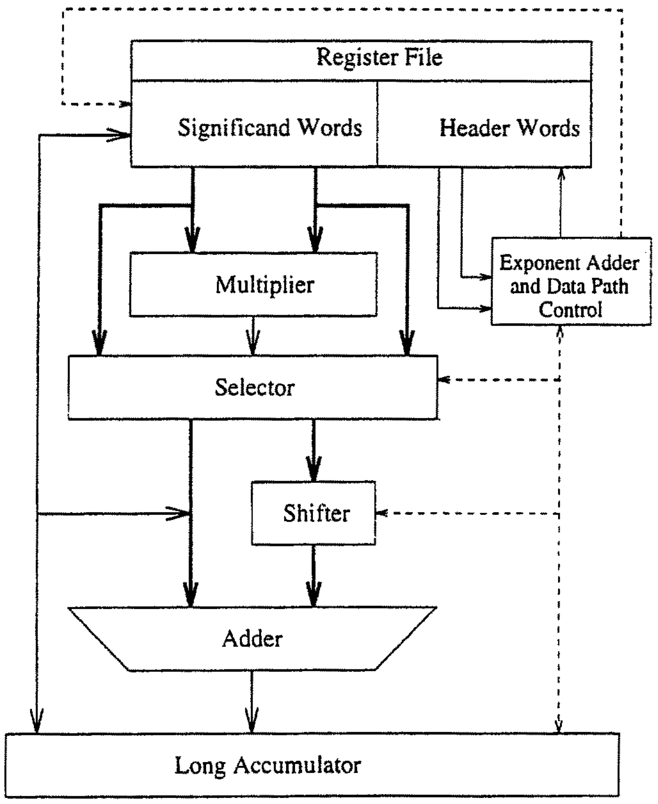

Figure 2. Arithmetic coprocessor hardware design

words are accessed based on the operation and the value of the index fields. The header and significand memories have two read ports and one write port. This allows two operand words to be read and one operand word to be written each cycle.

Significand words that are read from the register file go into the multiplier, the selector, or the long accumulator. The selector performs comparison operations and determines which values go into the adder and the shifter. The multiplier takes two m-bit significand words as inputs and computes the $2m$-bit product. The adder takes two $2m$-bit numbers as inputs and produces a $2m$-bit sum and a carry-out bit. The shifter takes a $2m$-bit number and shifts it by up to $2m$ bits.

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 operations without roundoff error or overflow. The implementation of the long accumulator is similar to the one presented in [2, 19]. The long accumulator consists of a 64 word by $2m$-bit dual-port-RAM, carry resolution logic, and rounding and normalization control. Temporary variable-precision values are stored in the dual-port-RAM, that contains one write port and one read/write port. Values are written to the RAM from either the adder or the register file. Values read from the RAM either go directly into the register file, or are fed back into the adder.

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

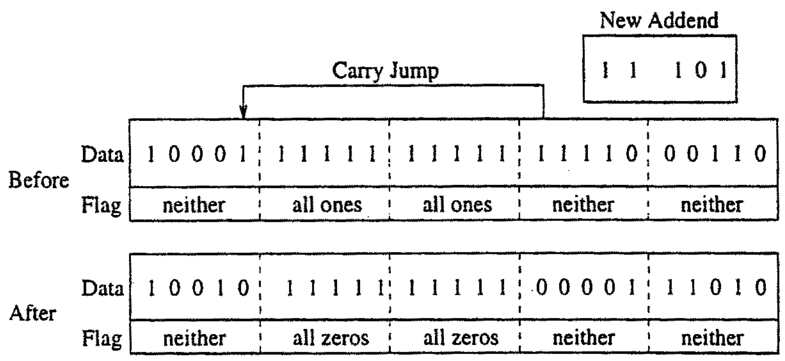

Figure 3. Adding a number to the long accumulator

of the long accumulator has a 2-bit flag associated with it that tells if the bits in the segment contain *all ones*, *all zeros*, or *neither* [2, 19]. A carry propagating into a segment that contains all ones will cause the flag to signal all zeros. Similarly, a borrow into a segment that contains all zeros will cause the flag to signal all ones. If a carry or a borrow comes into a segment that is neither all ones nor all zeros, the carry or borrow will not be propagated beyond that segment. The 2-bit flags and carry resolution logic determine the segment to which the carry is added. When a value is read from a segment of the long accumulator, the corresponding 2-bit flag is checked. If the flag indicates *all ones* or *all zero*, a constant is read back. Otherwise the value is read from the accumulator RAM.

Figure 3 demonstrates the accumulation process using five bit segments, when the addend is also five bits. The addend is added to two of the segments in the long accumulator. The exponent of the addend determines the segments accessed in the long accumulator and the amount that the addend is shifted. If a carry occurs after the second addition, it is added to the first segment that does not contain all ones. The flags for segments between the carry generation and carry resolution that indicated *all ones* are toggled to indicate *all zeros*, however the bits in these segments remain unchanged. A similar situation occurs for subtraction and borrow propagation.

Once the final result is computed, it is normalized and rounded to a variable-precision floating point number. The *all zeros* and *all ones* flags simplify normalizing and rounding the result, since they indicate the first non-zero segment of the long accumulator and help to determine the sticky bit [19].

3. Variable-precision, interval arithmetic algorithms

This section describes hardware algorithms for variable-precision, interval arithmetic. All intervals are stored in the register file using consecutive register words, with the lower endpoint stored first. For the variable-precision arithmetic operations, the two operands are denoted by 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 $\bigtriangledown$ and $\triangle$ denote round-toward-minus-infinity and round-toward-positive-infinity, respectively. For simplicity, all operands are assumed to have an n word (nm bit) significand (i.e., $L = n - 1$).

To perform variable-precision, floating point addition E_A and E_B are compared to determine the greater exponent. The operand with the larger exponent has its significand words written into the long accumulator. If the two operands have the same exponent, F_B is written into the long accumulator. If we assume that $E_B \geq E_A$, F_B is written into the long accumulator. In the subsequent cycles, F_A is added to the long accumulator, using a series of $2m$-bit additions, in which the carry-out of the i-th addition is the carry-in of the $(i+1)$-th addition. The difference between E_A and E_B is used to select the appropriate words from the long accumulator and determine the number of bits that each word of F_A is shifted.

If addition is performed on operands with different signs, or subtraction is performed on operands with the same sign, the number with the smaller magnitude is subtracted from the number with the larger magnitude and the sign of the result is set to the sign of the number with the larger magnitude. After the final result is computed the long accumulator is normalized and rounded to a specified precision. The final result is then stored back into the register file.

Interval addition and subtraction are defined as [25]

$$\begin{aligned} X + Y &= [\bigtriangledown(a+c), \triangle(b+d)], \\ X - Y &= [\bigtriangledown(a-d), \triangle(b-c)]. \end{aligned}$$

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

For floating point multiplication, the significands of the two operands are multiplied and the exponents are added. The sign of the result is zero if the signs of the multiplier and the multiplicand are the same, and one if they are different. Since the significand of the product is between 1 and 4, it may be necessary to shift the significand right one position and increment the exponent.

Variable-precision multiplication is performed by using the multiplier, adder, and long accumulator to generate and accumulate $2m$-bit partial products. In the first cycle, the exponents are read from the header memory and added to compute the exponent of the product. Each subsequent cycle, m-bits of the multiplier are multiplied by m-bits of the multiplicand to produce a $2m$-bit partial product that is added to the previously accumulated partial products. The sum of the partial products is stored in the long accumulator. To avoid excessive carry propagation, the less significant partial products are generated first, as shown in Figure 4. After the product is computed, it is rounded and stored back into the register file. To multiply two n word variable precision numbers n^2 partial products are generated and accumulated. A similar algorithm is used for computing the square of a number. However, due to the symmetry in the partial products of the square only $(n^2+n)/2$ partial products are generated and accumulated.

Interval multiplication is defined as [25]

$$X \times Y = \left[\bigtriangledown\left(\min(ac, ad, bc, bd)\right), \triangle\left(\max(ac, ad, bc, bd)\right)\right].$$

Rather than computing all four products and then comparing the results, the endpoints to be multiplied to form the upper and lower endpoints of the product are determined by examining the sign bits of a, b, c, and d [14]. With this technique, only two variable-precision multiplications are required to perform interval multiplication, unless

$$a < 0 < b \quad \text{AND} \quad c < 0 < d.$$

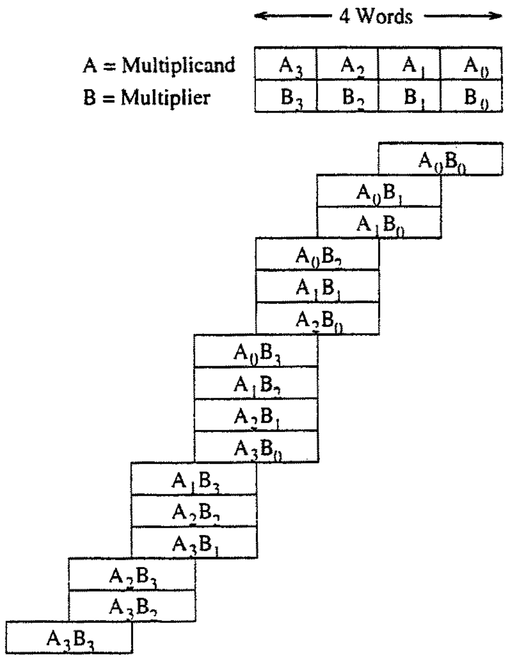

Figure 4. Variable-precision multiplication (4 words by 4 words)

Interval squaring is defined as

$$\begin{aligned} X^2 &= [a^2, b^2] && (a \geq 0), \\ X^2 &= [b^2, a^2] && (b \leq 0), \\ X^2 &= [0, \max(a, b)^2] && (a < 0 < b). \end{aligned}$$

For variable-precision multiplication and squaring, a method similar to the one proposed in [21] is used to guarantee correct rounding and reduce the number of partial products that are required.

For floating point division the significands of the two operands are divided and the exponents are subtracted. The sign of the result is zero if the signs of X and Y are the same, and one if they are different. Since the significand of the result is between $1/2$ and 2, it may be necessary to shift the quotient left one position and decrement the exponent.

The algorithm used to perform division is a variation of the short reciprocal divide algorithm [24], which has been modified for variable-precision, interval arithmetic. This algorithm uses an approximation to the reciprocal of the divisor to generate and accumulate successive quotient digits. The divide algorithm requires $n^2 + n$ single precision multiplications and $2(n^2 + n)$ single precision additions to divide two n word numbers and produce a correctly rounded n word quotient.

Interval division is defined as [25]

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

unless $0 \in Y$. In this case, the quotient interval is infinite and extended interval arithmetic is used [14]. Similar to interval multiplication, 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.

A similar algorithm is used to compute square roots. It 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 to n words of precision. Since the square root is monotonically increasing, an interval square root is defined as

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

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

Accurate dot products are essential for scientific applications. The dot product of two vectors $X = [x_1, x_2, \ldots x_k]$ and $Y = [y_1, y_2, \ldots, y_k]^T$ is defined as

$$X \cdot Y = \sum_{i=1}^{k} x_i \cdot y_i.$$

For each x_i, y_i pair in the dot product a variation of the multiplication algorithm is used to compute a new product and add it to the long accumulator. The segments chosen from the long accumulator and the amount that the new product is shifted is determined by the exponent of the new product. The *all ones* and *all zeros* flags help reduce carry propagation over long distances. After the entire dot product is computed, it is normalized and rounded to the specified precision.

To compute the lower endpoint of an interval dot product, the lower endpoint of each interval multiplication is computed and added to the long accumulator. Once the lower endpoints of all k products have been accumulated, the value in the long accumulator is normalized, rounded toward negative infinity, and stored back to the register file. After resetting the long accumulator to zero, the upper endpoint of the dot product is computed by accumulating the upper endpoint of each interval multiplication. After the upper endpoint of the dot product is computed, it is normalized, rounded toward positive infinity, and stored back to the register file.

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

$$\begin{aligned} \text{hull}(X,Y) &= [\min(a,c), \max(b,d)], \\ \text{intersection}(X,Y) &= [\max(a,c), \min(b,d)], \\ \text{midpoint}(X) &= (a+b)/2, \\ \text{width}(X) &= b-a. \end{aligned}$$

The intersection and hull operations take two variable-precision intervals and return a variable-precision interval. The width and midpoint operations, on the other hand, take one variable-precision interval and return a variable-precision floating point number. To determine the minimum and maximum values, the exponent adder and the operand selector are used. If the two numbers being compared have the same sign and exponent, then the selector compares their significand words from most significant to least significant to determine which number is greater. Thus, it takes at most $2n$ significand comparisons to determine the upper and lower endpoints for interval hull. Interval intersection requires at most $3n$ significand comparisons, since after the lower and upper endpoints are determined, a test is made to ensure that the upper endpoint is greater than or equal to the lower endpoint. If it is not, a warning is signaled. The interval midpoint and width operations are implemented using the variable precision addition and subtraction algorithms, respectively. If the result is not representable, it is rounded to the nearest variable-precision floating number. The division by two in the midpoint operation is implemented by decrementing the exponent of $a+b$ by one. To compare intervals, relational operators such as equal to, subset, superset, is-contained-in, and disjointness are also provided as defined in [17].

4. Area and delay estimates

Table 1 gives the hardware requirements for three VPIACs with data paths of 64, 32, and 16 bits. The number of words is denoted by w and the number of bits is denoted by b. For example, the long accumulator for the 64-bit VPIAC consists of 64 words, each of which is 128 bits long.

Area estimates are given in Table 2, based on data from a 1.0 micron CMOS standard cell library [23]. The estimates for the multiplier assume that multiplication is implemented using a Reduced Area Multiplier [3], followed by a carry look-ahead adder. The area of each

Component	64-bit VPIAC	32-bit VPIAC	16-bit VPIAC
Multiplier	64b by 64b	32b by 32b	16b by 16b
Adder	128b	64b	32b
Significand Memory	256w by 64b	256w by 32b	256w by 16b
Header Memory	64w by 32b	64w by 32b	64w by 32b
Long Accumulator	64w by 128b	64w by 64b	64w by 32b
Shifter	128b	64b	32b
Operand Selector	4w by 128b	4w by 64b	4w by 32 b
Exponent Add/Sub	16b	16b	16b
Latches	128b and 64b	64b and 32b	32b and 16b

Table 1. Hardware requirements

Component	64-bit VPIAC	32-bit VPIAC	16-bit VPIAC
Multiplier	49.4	15.2	4.9
Adder	4.2	2.1	1.0
Significand Memory	35.5	17.8	8.8
Header Memory	4.4	4.4	4.4
Long Accumulator	26.3	13.0	6.5
Shifter	8.2	3.9	1.9
Operand Selector	7.8	4.1	2.0
Exponent Add/Sub	0.6	0.6	0.6
Latches	4.8	2.6	1.4
Pads, Space etc.	84.7	38.2	18.9
Total	225.9	101.9	50.4

Table 2. Area estimates (mm^2)

Component	64-bit VPIAC	32-bit VPIAC	16-bit VPIAC
Multiplier (Reduction)	20.0	14.0	10.5
Multiplier (Final Add)	18.4	13.8	9.3
Adder	18.4	13.8	9.3
Significand Memory	8.2	7.4	7.0
Header Memory	6.8	6.8	6.8
Long Accumulator	7.8	7.0	6.8
Shifter	8.9	8.2	7.8
Operand Selector	4.2	3.5	3.2
Exponent Add/Sub	4.4	4.4	4.4
Latches	2.0	2.0	2.0
Cycle Time	22.0	16.0	12.5

Table 3. Delay and cycle time estimates (ns)

component is estimated by calculating the total size of the macrocells (e.g., AND gates, full adders, half adders, etc.) that make up the component and then adding an additional 50 percent for internal wiring. 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 estimated chip areas for the 64-bit, 32-bit, and 16-bit VPIACs are 225.9 mm^2, 101.9 mm^2, and 50.4 mm^2, respectively. In comparison, an IEEE double-precision coprocessor in the same technology has a total area of 100.8 mm^2 [30].

Delay and cycle time estimates are given in Table 3. The delay of each component is computed by taking the worst case delay of the critical path and adding 25 percent for unexpected delays and clock skew. The multipliers use two cycles. In the first cycle the partial products are generated and reduced to two numbers. In the second cycle these two numbers are added together to produce the product. The cycle time for each design is the sum of the multiplier reduction delay and the latch delay. The cycle times for the 64-bit, 32-bit, and 16-bit VPIACs are 22.0 ns, 16.0 ns, and 12.5 ns, respectively. An IEEE double-precision coprocessor in the same technology has a cycle time of 20.0 ns [30].

5. Cycle counts and execution times

Table 4 shows the number of cycles required for operations on both point and interval operands. The number of m-bit words in each operand is denoted by n. Short multiplication refers to the product of a one word multiplier and an n word multiplicand. For the dot product operation, k represents the number of elements in the two vectors whose dot product is computed. The cycle counts reported include the cycles needed for instruction fetch, instruction decode, reading the operands from the register file, performing the operation, rounding the result, and storing the rounded result back into the register file. The cycle counts given assume that the operands are already in the register file.

The algorithms used for addition, subtraction, short multiplication, hull, intersection, midpoint, and width are $O(n)$, and the algorithms for square, multiplication, division, and square root are $O(n^2)$. The algorithm for dot product computation is $O(k \cdot n^2)$. Although algorithms with better asymptotic complexities exist, they require more control logic and are slower for

Operation	point operands	interval operands
Addition/Subtraction	$2n+8$	$4n+12$
Short Multiplication	$2n+12$	$4n+20$
Square	$n^2/2+2n+12$	$n^2+4n+22$
Multiplication	n^2+n+12	$2n^2+2n+22$
Division	$3n^2+4n+20$	$6n^2+8n+38$
Square Root	$3n^2+6n+26$	$6n^2+12n+48$
Dot Product	$k(2n^2+12)+2n+20$	$k(4n^2+26)+4n+36$
Hull	not applicable	$2n+8$
Intersection	not applicable	$3n+12$
Midpoint	not applicable	$2n+8$
Width	not applicable	$2n+8$

Table 4. Cycle counts

Bits	64-bit VPIAC	32-bit VPIAC	16-bit VPIAC	VPI-SP
64	5.41 (410)	5.50 (403)	9.15 (243)	2220
128	7.57 (659)	11.7 (426)	28.5 (175)	4990
256	16.1 (950)	36.4 (421)	105 (146)	15300
512	50.1 (1010)	135 (407)	413 (133)	54900
1,024	186 (1130)	529 (397)	1640 (128)	210000

Table 5. Execution times for point dot product (μs)

Bits	64-bit VPIAC	32-bit VPIAC	16-bit VPIAC	VPI-SP
64	11.4 (409)	11.5 (405)	18.7 (249)	4660
128	15.7 (629)	23.9 (413)	57.3 (172)	9880
256	32.8 (896)	73.3 (401)	211 (139)	29400
512	101 (1010)	270 (378)	826 (123)	102000
1,024	372 (1050)	1060 (368)	3290 (119)	390000

Table 6. Execution times for interval dot product (μs)

small and moderate precisions. Hardware support and efficient implementation of the interval operations allow them to be executed in approximately twice as many cycles as the equivalent operations on point operands.

Tables 5 and 6 show execution times for point and interval dot products, with $k = 16$. The number of bits $(n \cdot m)$ is varied from 64 to 1,024 bits. For comparison, the execution times of the VPI software package (VPI-SP) are also given [12]. The ratio of the VPI-SP's execution time to the corresponding processor's execution time is given in parenthesis. The cycle counts for point and interval dot products are

$$\begin{aligned} \text{CyclesDotPoint} &= 32n^2+2n+212, \\ \text{CyclesDotInterval} &= 64n^2+4n+452. \end{aligned}$$

For the coprocessors, the execution time is computed as the product of the number of cycles

Bits	64-bit VPIAC	32-bit VPIAC	16-bit VPIAC	VPI-SP
64	10.6 (262)	9.60 (307)	12.0 (232)	2780
128	13.2 (478)	15.4 (410)	27.0 (234)	6310
256	21.1 (910)	34.5 (557)	81.0 (237)	19200
512	47.5 (1450)	104 (662)	285 (241)	68800
1,024	143 (1860)	365 (726)	1080 (245)	265000

Table 7. Execution times for point polynomial evaluation (μs)

Bits	64-bit VPIAC	32-bit VPIAC	16-bit VPIAC	VPI-SP
64	18.5 (303)	17.3 (324)	22.5 (249)	5600
128	23.8 (508)	28.8 (420)	52.5 (230)	12100
256	39.6 (927)	67.2 (546)	161 (228)	36700
512	92.4 (1440)	205 (649)	569 (234)	133000
1,024	282 (1770)	728 (685)	2150 (232)	499000

Table 8. Execution times for interval polynomial evaluation (μs)

and the cycle time. For example, for the 32-bit VPIAC, if the precision of the computation is 512 bits, then $n = 16$, CyclesDotPoint = 8,436, and CyclesDotInterval = 16,900. Since the cycle time for the 32-bit VPIAC is 16 ns, the execution times for point and interval dot products are

$$\begin{array}{lllll} \text{ExecDotPoint} & = 16 \times 8.436 & = 134{,}976 \text{ ns} & \approx & 135\mu s, \\ \text{ExecDotInterval} & = 16 \times 16{,}900 & = 270{,}400 \text{ ns} & \approx & 270\mu s. \end{array}$$

The execution times of the VPI software package are determined by running one thousand iterations of the operation on a 40 MHz Sparc IPX processor and taking the average execution time.

The 64-bit VPIAC has the shortest execution times, but the largest area requirements. The 16-bit VPIAC has the smallest area requirements, but the longest execution times. The 32-bit VPIAC offers a good compromise between the two designs, with fairly low area and good execution times. When the precision is relatively low (i.e., 128 bits or less), the execution times of the 32-bit VPIAC and the 64-bit VPIAC are quite close. For example, for point dot product computations with 64 bits of precision, the 32-bit VPIAC has an execution time that is only 1.7 percent greater than the execution time of the 64-bit VPIAC. This occurs because although the 64-bit VPIAC requires fewer cycles to compute the dot product, it has a longer cycle time. For 16 element dot product computations, the VPIACs are 119 to 1130 times faster than the VPI-SP.

Tables 7 and 8 show the execution times for point and interval polynomial evaluation, for a 20 term polynomial. Horner's rule is used, so that each term in the polynomial requires 1 addition/subtraction and one multiplication. The cycle counts for evaluating a 20 term polynomial are

$$\begin{array}{lll} \text{CyclesPolyPoint} & = & 20n^2 + 60n + 400, \\ \text{CyclesPolyInterval} & = & 40n^2 + 120n + 680. \end{array}$$

When the precision is 64 bits, the 32-bit VPIAC actually has a shorter execution time than the 64-bit VPIAC, due to its shorter cycle time. As the precision increases, the n^2 term dominates

Bits	64-bit VPIAC	32-bit VPIAC	16-bit VPIAC	VPI-SP
64	10.1 (6310)	10.8 (5900)	14.5 (4390)	63700
128	13.4 (11100)	18.5 (8050)	34.1 (4370)	149000
256	23.2 (20100)	43.7 (10700)	103 (4530)	467000
512	54.6 (32100)	132 (13300)	362 (4830)	1750000
1,024	166 (39600)	463 (14200)	1360 (4830)	6570000

Table 9. Execution times for interval newton method (μs)

and the 64-bit VPIAC has the shortest execution times. For 20 term polynomial evaluations, the VPIACs are 228 to 1860 times faster than the VPI-SP.

Table 9 shows the execution times for one iteration of an interval Newton method [25]. The interval Newton method takes an interval X_i which includes a zero of the function $f(x)$, and computes a tighter interval X_{i+1} which includes the same zero. It employs the following iterative equation:

$$X_{i+1} = \left(\text{midpoint}(X_i) - \frac{f(\text{midpoint}(X_i))}{f'(X_i)}\right) \cap X_i.$$

For the execution times shown in Table 9, $f(x)$ and its derivative $f'(x)$ are chosen as

$$\begin{aligned} f(x) &= 10x^2 - 5x + 3\sqrt{x} - 17, \\ f'(x) &= 20x + \tfrac{3}{2\sqrt{x}} - 5. \end{aligned}$$

Each iteration of the algorithm requires six additions/subtractions, 5 short multiplications, 1 square, 2 divisions, 2 square roots, 1 midpoint operation, and one intersection. Each of these operations is performed on interval operands. The number of cycles per iteration is

$$\text{CyclesNewtonInterval} = 25n^2 + 93n + 386.$$

For all reported interval Newton methods, the 16-bit VPIAC has the longest execution times. The 64-bit VPIAC has the shortest execution times, except when the precision is 64 bits. For interval newton method, the VPIACs are 4370 to 39600 times faster than the VPI-SP. The long execution times for the VPI-SP is primarily due to its slow computation of the interval square root.

6. Conclusions

This paper examined hardware designs for VPIACs with data path widths of 64-bits, 32-bits, and 16-bits. The 16-bit VPIAC has the shortest cycle time and uses the least amount of area, but has longest execution times for the applications examined. The 64-bit VPIAC has the shortest execution times for most applications, but uses the largest amount of area and has the longest cycle time. The 32-bit VPIAC offers a good compromise between the two designs. It uses less than half the area of the 64-bit VPIAC, and has comparable execution times for low to moderate precisions. The design of the 32-bit VPIAC is described in more detail in [28].

The VPIACs give the programmer the ability to set the initial precision of the computation, determine the accuracy of the results, and recompute inaccurate results with higher precision.

They can also be used to evaluate the accuracy of programs before running them on a general purpose processor, or to select between various programs based on their accuracy for given inputs. Direct hardware support for variable-precision, interval arithmetic greatly improves the accuracy and reliability of the computation, and is much faster than existing software techniques for controlling numerical error. The coprocessors are two to four orders of magnitude faster than the VPI software package for variable-precision point and interval applications.

Acknowledgments

Special thanks are extended to Professor Jeff Ely at Lewis and Clark University who provided the variable-precision, interval arithmetic package that was modified to simulate the hardware design presented in this paper. The authors are also grateful to Professor Ulrich Kulisch, Professor George Corliss, Professor R. Baker Kearfott, and Christoph Baumhof who supplied useful background information. Comments and suggestions provided by the anonymous referees were helpful in revising the paper.

References

[1] Alefeld, G. and Herzberger, J. *Introduction to interval computations.* Academic Press, New York, 1983.

[2] Baumhof, C. *A new VLSI vector arithmetic coprocessor for the PC.* In: "Proceedings of the 12th Symposium on Computer Arithmetic", 1995, pp. 210–215.

[3] Bickerstaff, K. C., Schulte, M. J., and Swartzlander, Jr., E. E. *Parallel reduced area multipliers.* Journal of VLSI Signal Processing **9** (1995), pp. 181–192.

[4] Blum, B. I. *An extended arithmetic package.* Communications of the ACM **8** (1965), pp. 318–320.

[5] Bohlender, G. *What do we need beyond IEEE arithmetic?* In: Ullrich, Ch. (ed.) "Computer Arithmetic and Self-Validating Numerical Methods", Academic Press, 1990, pp. 1–32.

[6] Brent, R. P. *A FORTRAN multiprecision arithmetic package.* ACM Transactions on Mathematical Software **4** (1978), pp. 57–70.

[7] Buchberger, B. *Groebner bases in Mathematica: enthusiasm and frustration.* In: "Programming Environments for High-level Scientific Problem Solving", 1991, pp. 80–91.

[8] Carter, T. M. *Cascade: hardware for high/variable precision arithmetic.* In: "Proceedings of the 9th Symposium on Computer Arithmetic", 1989, pp. 184–191.

[9] Char, B. W. et al. *A tutorial introduction to Maple.* Journal of Symbolic Computation **2** (1986), pp. 179–200.

[10] Chiarulli, D. M., Rudd, W. G., and Buell, D. A. *DRAFT: a dynamically reconfigurable processor for integer arithmetic.* In: "Proceedings of the 7th Symposium on Computer Arithmetic", 1985, pp. 309–318.

[11] Cohen, M. S., Hull, T. E., and Hamarcher, V. C. *CADAC: a controlled-precision decimal arithmetic unit.* IEEE Transactions on Computers **C–32** (1983), pp. 370–377.

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

[13] Hafner, K. *Chips for high precision arithmetic.* In: Ullrich, Ch. (ed.) "Computer Arithmetic and Self-Validating Numerical Methods", Academic Press, 1990, pp. 33–54.

[14] Hansen, E. *Global optimization using interval analysis.* Marcel Dekker, 1992.

[15] *IEEE Standard 754 for binary floating point arithmetic.* IEEE, 1985.

[16] Kearfott, R. B., Dawande, M., Du, K., and Hu, C. *Algorithm 737: INTLIB: a portable FORTRAN 77 interval standard function library.* ACM Transactions on Mathematical Software **20** (1994), pp. 447–459.

[17] Klatte, R., Kulisch, U., Neaga, M., Ratz, D., and Ullrich, Ch. *PASCAL–XSC: language reference with examples.* Springer-Verlag, 1991.

[18] Klatte, R., Kulisch, U., Wiethoff, A., Lawo, C., and Rauch, M. *C–XSC: a C++ class library for extended scientific computing.* Springer-Verlag, 1993.

[19] Knöfel, A. *Fast hardware units for the computation of accurate dot products.* In: "Proceedings of the 10th Symposium on Computer Arithmetic", 1991, pp. 70–75.

[20] Knüppel, O. *PROFIL/BIAS—a fast interval library.* Computing **53** (1994), pp. 277–288.

[21] Krandick, W. and Johnson, J. R. *Efficient multiprecision floating point multiplication with optimal directional rounding.* In: "Eleventh Symposium on Computer Arithmetic", 1993, pp. 228–233.

[22] Kulisch, U. W. and Miranker, W. L. *Computer arithmetic in theory and in practice.* Academic Press, 1981.

[23] *LSI Logic 1.0 micron cell-based products databook.* LSI Logic Corporation, 1991.

[24] Matula, D. W. *A highly parallel arithmetic unit for floating point multiply, divide with remainder and square root with remainder.* In: "1989 IMACS/GAMM International Symposium on Scientific Computing, Computer Arithmetic, and Validated Numerics", 1989.

[25] Moore, R. E. *Interval analysis.* Prentice Hall, 1966.

[26] Ratz, D. *The effects of the arithmetic of vector computers on basic numerical methods.* In: Ullrich, Ch. (ed.) "Contributions to Computer Arithmetic and Self-Validating Numerical Methods", J.C. Baltzer, 1990, pp. 499–514.

[27] Reuter, E. K. et al. *Some experiments using interval arithmetic.* In: "Proceedings of the 4th Symposium on Computer Arithmetic", 1978, pp. 75–81.

[28] Schulte, M. J. and Swartzlander, Jr., E. E. *A hardware design and arithmetic algorithms for a variable-precision, interval arithmetic coprocessor.* In: "Proceedings of the 12th Symposium on Computer Arithmetic", 1995, pp. 222–229.

[29] Schulte, M. J. and Swartzlander, Jr., E. E. *A software interface and hardware design for variable-precision interval arithmetic.* Reliable Computing **1** (3) (1995), pp. 325–342.

[30] Schulte, M. J. and Swartzlander, Jr., E. E. *Designs and applications for variable-precision, interval arithmetic coprocessors.* Reliable Computing, to appear.

[31] Schwartz, J. *Implementing infinite precision arithmetic.* In: "Proceedings of the 9th Symposium on Computer Arithmetic", 1989, pp. 10–17.

[32] Ullrich, Ch. *Programming languages for enclosure methods.* In: Ullrich, Ch. (ed.) "Computer Arithmetic and Self-Validating Numerical Methods", Academic Press, 1990, pp. 115–136.

[33] Walter, W. V. *Acrith-XSC a Fortran-like language for verified scientific computing.* In: Adams, E. and Kulisch, U. (eds) "Scientific Computing with Automatic Result Verification", Academic Press, 1993, pp. 45–70.

[34] Wolfram, S. *Mathematica: a system for doing mathematics by computer.* Addison-Wesley, 1988.

[35] Wyatt, W. T., Lozier, D. W., and Orser, D. J. *A portable extended precision arithmetic package and library with FORTRAN precompiler.* ACM Transactions on Mathematical Software **2** (1976), pp. 209–231.

Received: June 27, 1995
Revised version: September 8, 1995

Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin
Texas 78712
USA