Reliable Computing 1 (3) (1995), pp. 325–342

# A software interface and hardware design for variable-precision interval arithmetic

Michael J. Schulte and Earl E. Swartzlander, Jr.

This paper presents a software interface and hardware design for variable-precision, interval arithmetic. The software interface gives the programmer the ability to specify the precision of the computation and determine the accuracy of the result Special instructions for vector and matrix operations are also provided. The hardware design directly supports variable-precision, interval arithmetic. This greatly improves the accuracy of the computation and is much faster than existing software methods for controlling numerical error. Hardware algorithms are presented for the basic arithmetic operations, exact dot products, and elementary functions. Area and delay estimates indicate that the processor can be implemented on a single chip with a cycle time that is comparable to existing IEEE double-precision floating point processors.

# Программный интерфейс и конструкция аппаратуры для интервальной арифметики переменной разрядности

М. Й. Шульте, Е. Е. Шварцландер, мл.

Описываются программный интерфейс и конструкция аппаратуры для интервальной арифметики переменной разрядности. Программный интерфейс дает программисту возможность управлять разрядностью вычислений, определяя точность результата. Также предусмотрены специальные инструкции для векторных и матричных операций. Конструкция аппаратуры напрямую поддерживает интервальную арифметику переменной разрядности, что значительно повышает точность вычислений и обеспечивает выигрыш в скорости в сравнении с существующими программными методами управления величиной численных погрешностей. Представлены аппаратно реализованные алгоритмы для основных арифметических операций, точных скалярных произведений и элементарных функций. Оценки времени вычислений и требуемой площади кристалла показывают, что соответствующий процессор может быть реализован на одном кристалле с рабочей частотой, сравнимой с существующими процессорами плавающей точки двойной точности стандарта IEEE.

## 1. Introduction

Advances in VLSI technology and computer architecture have lead to increasingly faster digital computers. During each of the last three decades, the computational speeds of the fastest computers increased by a factor of roughly 100 [1]. This increase in computing power has lead to the development of computer systems which perform billions of arithmetic operations per second and has given researchers the ability to solve previously intractable problems. The large number of arithmetic operations, however, has made it extremely important to keep track of and control errors in numerical computations.

Although the number of arithmetic operations performed by computers has increased by several orders of magnitude over the past few decades, the arithmetic precision of computers has remained relatively unchanged. For example, in 1967 the IBM 360/91 [2] had a 64-bit floating point format with a 7-bit exponent and a 57-bit significand (mantissa). In comparison, computers which conform to the IEEE–754 double-precision floating point standard [18] use a 64-bit floating point format with an 11-bit exponent and a 53-bit significand. 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. As a result, today's numerically intensive applications may produce results which are completely inaccurate.

As an example of the inaccuracies that can occur due to roundoff error and catastrophic cancellation, consider taking the dot product of the following two vectors:

$$A = [-10^{18}, 2246, 10^{27}, 10^{25}, 22, 10^{5}]$$
$$B = [10^{38}, 33, 10^{29}, -10^{22}, 1044, 10^{42}].$$

Although the correct value of the dot product is 97,086, the result computed using IEEE double-precision arithmetic is zero. On most computer systems, however, there is no method for determining whether or not the final result is correct.

To overcome the numerical limitations of existing computer systems, several scientific programming languages including PASCAL–XSC [11], ACRITH–XSC [29], C–XSC [19], and VPI [8] have been developed. These scientific programming languages are extensions to existing languages which allow the user to define abstract data types, overload functions, and create dynamic arrays. Special instructions are defined for vector and matrix operations, which are essential for scientific computations. Furthermore, these languages provide software support for variable-precision, interval arithmetic [23] in order to increase the reliability of the computation.

The main disadvantage of the extended scientific programming languages is their speed. The languages are designed for machines which support the IEEE 754 standard. As a result, all variable-precision, interval arithmetic operations must be simulated in software. This adds tremendous overhead due to function calls, memory management, error and range checking, and exception handling. The interval arithmetic routines discussed in [25] are approximately 40 times slower than their single-precision floating point equivalents. Routines which supported variable-precision, interval arithmetic (up to 56 decimal digits) are more than 1,200 times slower than the corresponding single-precision routines.

To overcome the speed limitations of existing scientific programming languages, direct hardware support is required. Previous designs, such as [6] and [7] improve the speed of variable-precision computations, but do not provide direct hardware support for interval arithmetic. This paper presents a software interface and hardware design which support variable-precision, interval arithmetic. Because the arithmetic operations are implemented in hardware, this system offers substantial speedups over existing software programs which are designed for general purpose computers. Section 2 presents an overview of the software interface to the processor. In Section 3, a hardware implementation of the processor is given. Section 4 discusses the algorithms used to perform the basic arithmetic operations, dot product accumulation, and elementary function evaluation. Area and delay estimates for the processor are given in Section 5, followed by conclusions in Section 6.

# 2. Software interface

This section and the following section describe the software interface and hardware design of the variable-precision, interval arithmetic processor. The main design goal is to obtain a balance between the functionality and complexity of the hardware and software designs. Systems which only provide software support for variable-precision, interval arithmetic are typically too slow, while pure hardware solutions are either too complex or not flexible enough for general purpose use. The design presented here uses a mix of hardware and software to provide an efficient, flexible, and numerically reliable system.

In general, programs consist of sections of code which require variable-precision, interval arithmetic and sections of code for which standard floating-point arithmetic is sufficient. To maximize performance, the programmer is able to specify the precision of the computation and whether or not interval arithmetic is to be used. The programmer can also test the accuracy of the computation and perform the computation again with higher precision if the accuracy requirements are not met.

The software interface presented here is an extension of the C++ programming language. Program 1 illustrates the use of variable precision, interval arithmetic. In this example, the precision of the computation is initialized to four 64-bit words (256 bits). The variables $a$, $b$, and $c$ are IEEE double-precision numbers, and the variables $X$, $Y$, and $Z$ are variable-precision intervals. The values of $a$, $b$, and $c$ are first read in from standard input. $X$ is initialized to $[\min(a,b), \max(a,b)]$, and $Y$ is set to $[c,c]$. Since $X$ and $Y$ have more precision than $a$, $b$, and $c$, their trailing bits are set to zero. Alternatively, if an interval assignment is made which results in a loss of precision, then the lower endpoint is automatically rounded downward (RD) and the upper endpoint is rounded upward (RU). After this, $X$ and $Y$ are added together to produce $Z = [\mathrm{RD}(a+c), \mathrm{RU}(b+c)]$.

```
main(){
  double a, b, c;              /* double-precision values */
  precision(4);                /* set precision to 4 words */
  cin >> a >>b >> c;           /* read in values */
  vp_interval X(a,b),Y(c),Z; /* intervals X, Y and Z */
  Z = X + Y;                   /* Z = [RD(a + c),RU(b + c)] */
}
```

Program 1. Variable-precision, interval arithmetic code

To efficiently support numerical computations, additional data types are defined. These include combinations of vectors, matrices, intervals, complex numbers, and variable-precision numbers. The numerical data types are shown in Program 2. Data types with a vp prefix contain variable-precision entries, while the other data types contain IEEE double-precision entries. Operations on both types of numbers are supported directly by the hardware. The hardware also provides support for exact dot products and complex arithmetic operations.

Program 3 shows code which uses some of the numerical data types. In this example, the matrices $A$, $B$, and $C$ contain IEEE double-precision, complex, interval entries. $X$ is a variable-precision, complex, interval vector. The matrices $A$ and $B$ have sizes $m$ by $k$ and $k$ by $n$, respectively. Initially, the entries for matrices $A$ and $B$ are read in, and each element in the vector $X$ is set to zero. Next $C$, is dynamically allocated as an $m$ by $n$ matrix and receives

the complex, interval product of $A$ and $B$. After this, a loop is entered which sums the rows of $C$, into the vector $X$.

```
real, vector, matrix;                    (real)
vp_real, vp_vector, vp_matrix;           (vp_real)
complex, cvector, cmatrix;               (complex)
vp_complex, vp_cvector, vp_cmatrix;      (vp_complex)
interval, ivector, imatrix;              (real interval)
vp_interval, vp_ivector, vp_imatrix;     (vp_real interval)
cinterval, civector, cimatrix;           (complex interval)
vp_cinterval, vp_civector, vp_cimatrix; (vp_complex interval)
```

Program 2. Numerical data types

```
main() {
  precision(8);                /* set precision to 8 words */
  int m, n, k;                 /* matrix, vector sizes */
  cin >> m >> n >> k;          /* read in matrix sizes */
  cimatrix A(m,k), B(k,n), C; /* complex interval matrices */
  vp_civector X(n) = 0;        /* complex interval vector */
  cin >> A >> B;               /* read in vectors A and B */
  C = A*B;                     /* matrix multiplication */
  for (i = 0; i < m; i++)      /* loop to sum rows of C */
    X = X + C[i];              /* add row i of C to X */
  cout << X;                   /* output X */
}
```

Program 3. Code for numerical data types

Accurate function evaluation is also provided for each of the numerical data types. For ease of programming the standard functions are overloaded so that the same function name can be used for different types. In the example shown in Program 4, the sine of $x$ is first computed using IEEE double-precision. Then, the sine of each element in the interval vector $X$ is computed, and each interval result is stored to 4 words (256 bits) of precision. After this, the sine of each element in matrix $A$ is computed.

```
main( ) {
  precision(4);          /* precision is 4 words */
  int n;                 /* size of matrix and vector */
  double x, y;           /* IEEE double-precision */
  cin >> x >> n;         /* read in values */
  vp_ivector X(n), Y;    /* vp interval vectors X and Y*/
  matrix A(n,n), B;      /* real matrices A and B */
  cin >> X >> A;         /* read in vector X and matrix A */
  y = Sin(x);            /* sine of IEEE double */
  Y = Sin(X);            /* sine of each vector element */
  B = Sin(A);            /* sine of each matrix element */
}
```

Program 4. Code for standard functions

```
main( ) {
  vp_real tolerance = 1e-6;    /* specify error tolerance */
  int prec, n;                 /* precision and sizes */
  int att=0, max_att=10;       /* number of attempts */
  cin >> prec >> n;            /* read in values */
  precision(prec);             /* set precision */
  vp_interval X;               /* X is prec words */
  vp_vector A(n),B(2*n);       /* vp vectors A and B*/
  cin >> A >> B;               /* get inputs for A and B */
restart:                       /* beginning of loop */
  for (i = 0; i < n; i++) {
    X = X + (A[i]*B[2*i]);     /* accumulate in  X */
  }
  if (X.width() > tolerance){ /* if error is too large */
    precision(prec++);         /* increment precision */
    goto restart;              /* redo computation */
    if (att++ > max_att)       /* if too many attempts */
      exit(1);                 /* then exit */
  }
}
```

Program 5. Recomputing with higher-precision

Special instructions allow the precision of the computation to be set and the accuracy of the result to be determine. In Program 5, the *precision* function is used to set the precision of the computation. The $X.width()$ function indicates the width of interval $X$. If the width is too large, the interval is recomputed using higher precision. In this example, the maximum error tolerance is $10^{-6}$. If the width of interval $X$ is greater than this value, the precision is increased by one word and another attempt is made to compute the result. This process continues until the desired accuracy is achieved or too many unsuccessful attempts at computing the result have occurred.

# 3. Processor hardware design

This section gives an overview of the hardware design for a processor which supports variable-precision, interval arithmetic. The hardware is designed to handle the common case quickly, while still providing correct results and acceptable performance for less frequent situations in which the data is especially ill-conditioned. The design incorporates the extra hardware required for variable-precision interval arithmetic directly into the design of the floating point processor. Alternatively, a tightly-coupled variable-precision co-processor could be designed which works in conjunction with the floating point processor

The processor described here supports both normalized IEEE double-precision and variable-precision floating point numbers. Denormalized numbers are handled in software. The format for IEEE double-precision numbers is shown in Figure 1. IEEE double-precision numbers consist of a sign-bit (S), an 11-bit exponent (E), and a 52-bit significand (F). The exponent is represented with a bias of 1023. A normalized IEEE double-precision number DP has a significand between

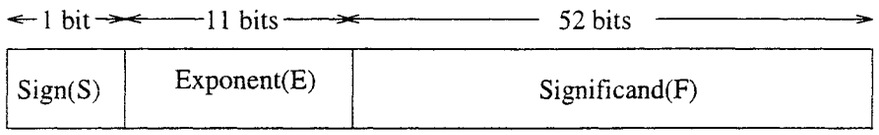


Figure 1. IEEE double-precision floating point format

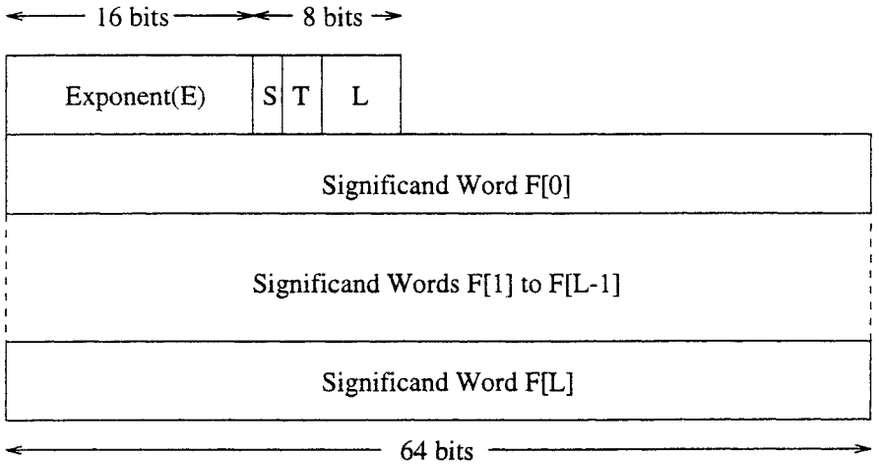


Figure 2. Variable-precision floating point format

1 and 2 and uses a hidden one. Its value is

$$\mathrm{DP} = (-1)^S \times 1.F \times 2^{E-1023}.$$

The format for variable-precision numbers is shown in Figure 2. 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 ($F$) 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 64-bit words in the significand. The words of the significand are stored from least significant $F[0]$ to most significant $F[L]$. The significand is normalized between 1 and 2, but does not use a hidden one. The value of a variable-precision floating point number VP is

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

For variable-precision numbers, the maximum significand length is 32 64-bit words, or 2048 bits. This gives a maximum precision of approximately 616 decimal digits. The range of positive, variable-precision numbers is approximately

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

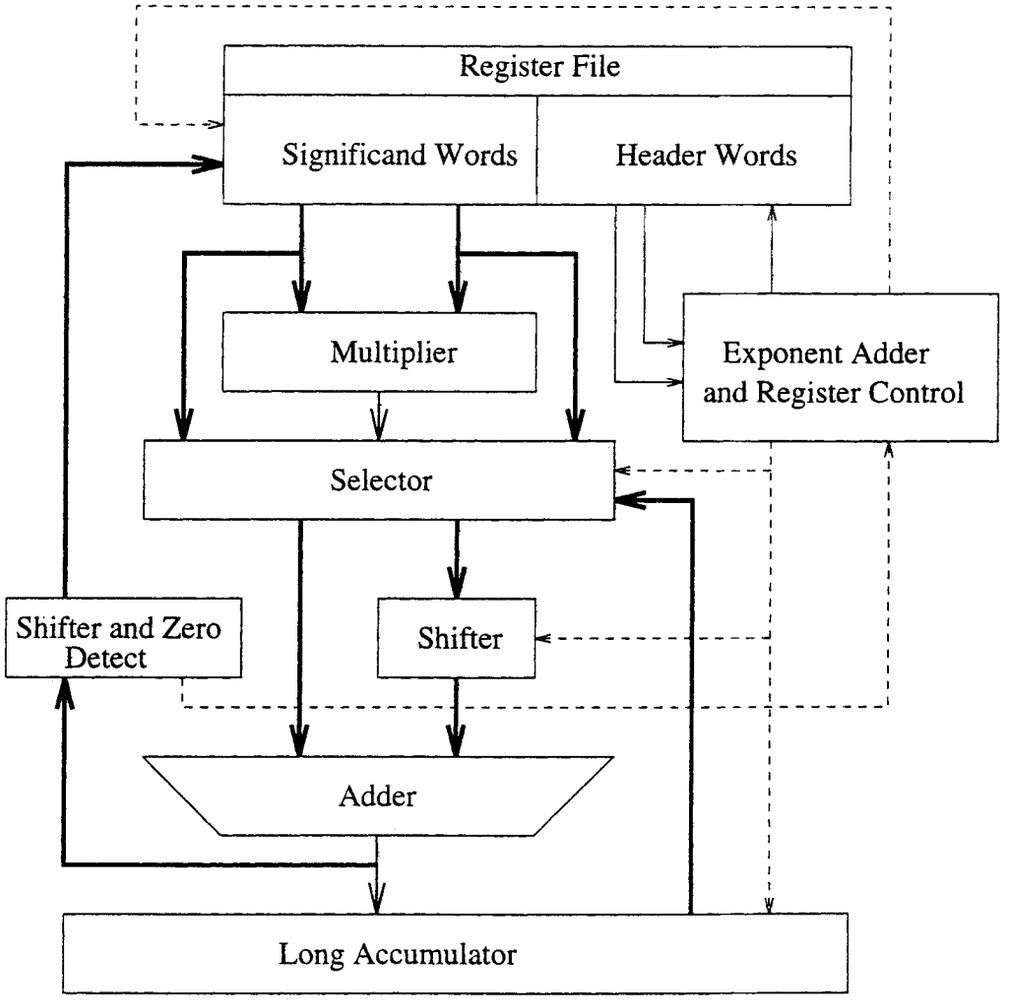


Figure 3. Arithmetic processor hardware design

In comparison, IEEE double-precision floating point numbers have a maximum precision of 53 bits or approximately 16 decimal digits. The range for positive, IEEE double-precision numbers is approximately

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

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 the register file, a 64-bit by 64-bit multiplier, a 133-bit adder, a long accumulator consisting of 64 128-bit segments, and two 128-bit shifters. The hardware unit also includes an exponent adder which determines the exponent of the result and helps control the register file, long accumulator, selector, and shifters. The selector stores temporary values and determines which values go into the adder.

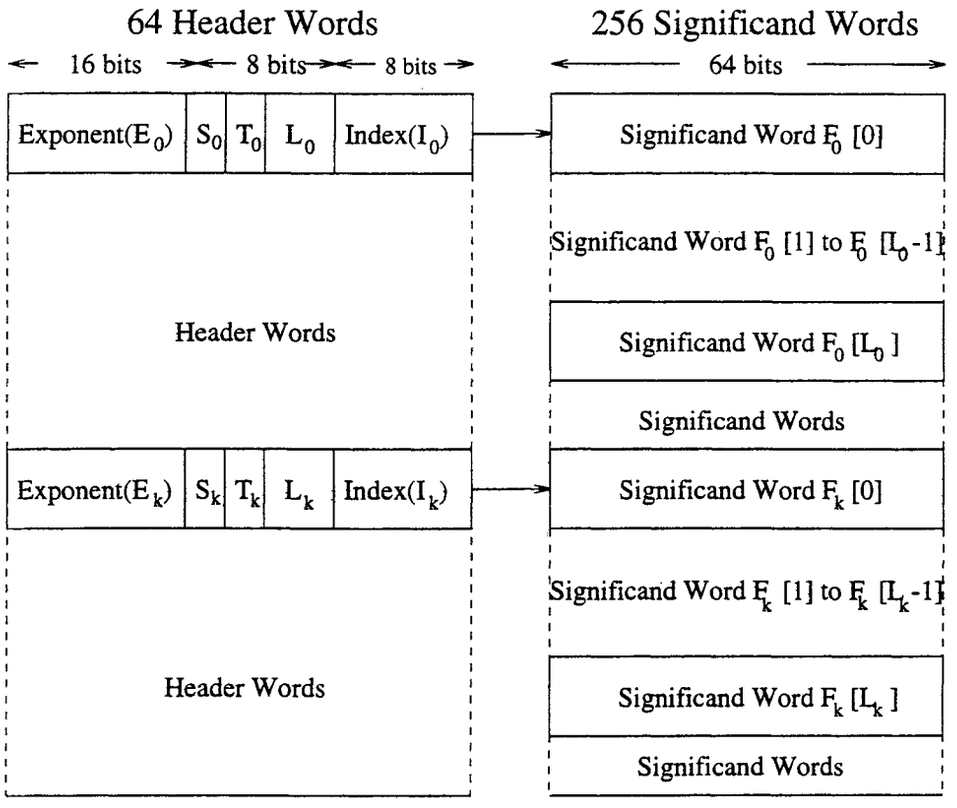


Figure 4. Header and significand memories

The register file consists of two memory units: a 64-word by 32-bit header memory, and a 256-word by 64-bit significand memory, as shown in Figure 4. Each header word contains the exponent, sign, type, and length of the variable-precision number, along with an index which points to the least 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 significands words are accessed based on the value of the index fields. For IEEE double-precision numbers, the significand is stored in the significand memory (with the leading 11 bits set to zero), and the sign bit and exponent are stored in the header memory. The value of the length field is zero, and the type field is used to indicate if the double-precision number is infinite, zero, or not-a-number. 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 in each cycle.

Operands which are read from the register file go into either the multiplier or the selector. The selector determines which values go into the adder and the shifter, based on the instruction being performed and the exponents of the operands. The long accumulator is used to store intermediate variable-precision results, as well as the partial sum of dot products.

# 4. Arithmetic algorithms

In this section hardware algorithms for variable-precision, interval arithmetic are described. All intervals are stored in the register file using consecutive memory words with the lower endpoint stored first. The hardware supports the four rounding modes specified in the IEEE 754 floating point standard: round-to-nearest-even (RN), round-toward-plus-infinity (RU), round-toward-minus-infinity (RD), and round-toward-zero (RZ).

## 4.1. Arithmetic operations

For floating point addition and subtraction, the exponents of the operands $A$ and $B$ must be equal before performing the operation. If the exponents of $A$ and $B$ are $E_a$ and $E_b$, with $E_b \geq E_a$, and the significands are $F_a$ and $F_b$, this is achieved by shifting $F_a$ to the right by $(E_b - E_a)$ bits and setting the exponent of the result to $E_b$. If addition is performed on operands with different signs, or subtraction is performed on operands with the same sign, the smaller number is subtracted from the larger number and the sign of the result is set to the sign of the larger number. After the subtraction, leading zeros may appear in the result, if the exponents of the two operands differ by less than two. These are removed by shifting the result to the left, and incrementing the exponent by an amount equal to the number of leading zeros.

Figure 5 shows variable-precision addition for C = A + B, where $N_a$, $N_b$, and $N_c$ denote the number of bits in the significands $F_a$, $F_b$, and $F_c$, respectively ($N_a$, $N_b$, and $N_c$ are multiples of 64). For variable-precision addition and subtraction, it may be necessary to shift one of the operands by several bits before adding them together. This is accomplished by accessing different words from the register file and using the shifter to align the bits in $F_a$ with the bits in $F_b$. The $\lceil (E_b - E_a + N_a - N_b)/64 \rceil$ least significant words of $F_c$ are equal to the $(E_b - E_a + N_a - N_b)$ least significant bits of $F_a$ (plus trailing zeros) and do not required any additions. The $\lceil (N_b + E_a - E_b)/64 \rceil$ words in $F_a$ and $F_b$ which overlap require full additions. These additions are performed as a series of 64-bit additions, in which the carry-out of the $i$-th addition is the carry-in of the $(i+1)$-th addition. The $\lfloor (E_b - E_a)/64 \rfloor$ most significant words of $F_c$ are equal to the corresponding words of $F_b$, with the possible addition of a carry. Unless there is a carry into the $\lfloor (E_b - E_a)/64 \rfloor$ most significant words of $F_b$, these words are copied directly into $F_c$, without any addition.

Addition and subtraction of the intervals $X = [a, b]$ and $Y = [c, d]$ are defined as [23]

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

Thus, interval addition (or subtraction) requires two variable-precision additions (or subtractions). The lower endpoint is computed first and rounded toward negative infinity. The upper endpoint is then computed and rounded towards positive infinity. Only three guard digits, including a sticky bit, are required to implement correct rounding [15].

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

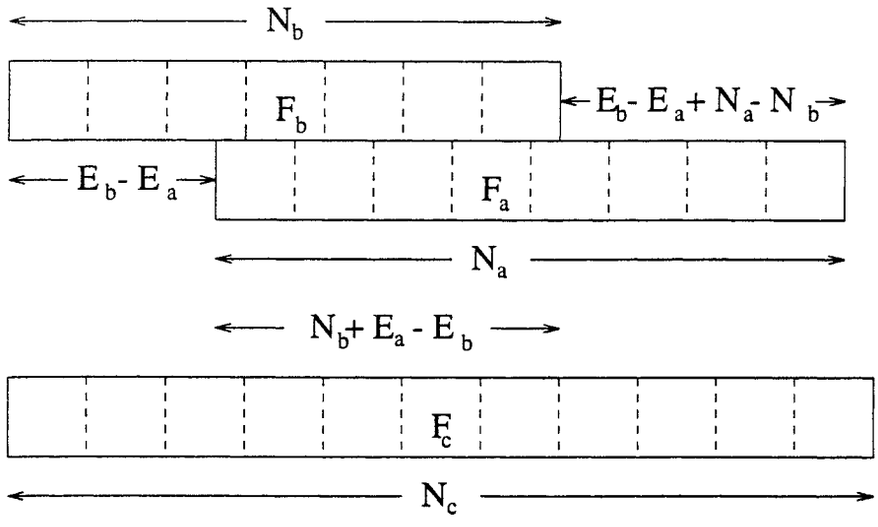


Figure 5. Variable-precision addition with exponent shift

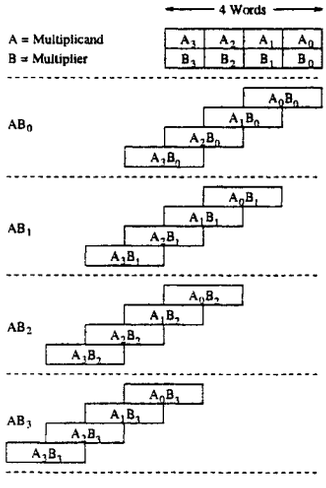


Figure 6. Variable-precision multiplication (4 word by 4 word)

Variable-precision multiplication is performed by using the multiplier and adder repetitively to generate and accumulate 128-bit partial products. During the first cycle, the exponents are read from the header memory and added to compute the exponent of the product. Each subsequent cycle, 64-bits of the multiplier are multiplied by 64-bits of the multiplicand to produce a new 128-bit partial product which is added to the previous result. 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 (256 bit by 256 bit) multiply. To avoid excessive carry propagation, the partial product generation is reordered so that the less significant partial products are generated first, as shown in Figure 7.

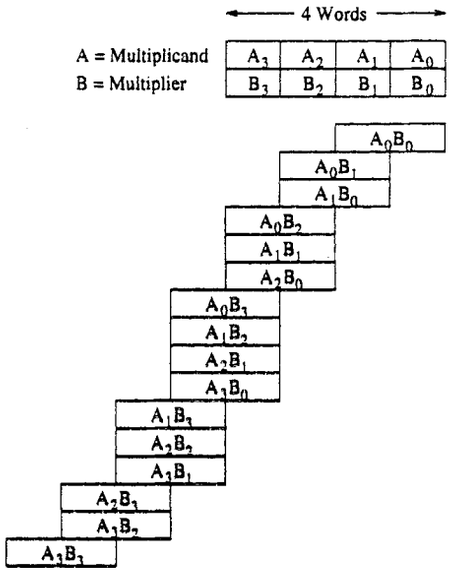


Figure 7. Reordered variable-precision multiplication (4 word by 4 word)

If the multiplicand and multiplier contain $m$ and $n$ 64-bit segments, respectively ($m \geq n$), then $m \cdot n$ single-precision multiplications and additions are required. If the final result is rounded to $m$ bits, then a method proposed in [16] is used to reduce the number of single-precision multiplications and additions to $m \cdot n - (n^2 - 3 \cdot n)/2 - 1$. This reduction in the number of operations is possible because only the $m + 1$ most significant columns of partial products are likely to contribute to the rounded product. For example, for the 4 word by 4 word multiplication shown in Figure 7, the partial products $A_0 \cdot B_0$, $A_0 \cdot B_1$, and $A_1 \cdot B_0$ will most likely have no effect on the product when it is rounded to 4 words. A quick test is used to determine 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.

Multiplication of the intervals $X = [a, b]$ and $Y = [c, d]$ is defined as [23]

$$X \times Y = \left[\mathrm{RD}\big(\min(ac, ad, bc, bd)\big), \mathrm{RU}\big(\max(ac, ad, bc, bd)\big)\right].$$

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 of the interval endpoints of $X$ and $Y$ [12]. With this technique, only two variable-precision multiplications are required, unless the condition

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

holds. The method proposed in [16] guarantees correct directional rounding.

For floating point division $X/Y$, the significands of the two operands are divided and the exponents are subtracted. The sign of the result is positive 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 final quotient left one position while decrementing the exponent.

The algorithm used for performing variable-precision division is based on Newton-Raphson iteration. As presented in [10], the Newton-Raphson division algorithm computes the reciprocal of the divisor $Y$ by the following iterative equation

$$b_{i+1} = b_i(2 - Yb_i)$$

where $b_i$ is the approximation to $1/Y$ after the $i$-th iteration. Each iteration approximately doubles the number of accurate bits in the result. Since the accuracy of the approximation increases with each iteration, lower precision computations are used for earlier iterations and higher precision computations are used for later iterations. After a suitable number of iterations, $b_i$ is multiplied by the dividend $X$ to obtain the quotient $Q$. A correction step is then employed to ensure that the quotient is correctly rounded [21].

Each iteration requires two multiplications and a subtraction. To start the algorithm, an initial approximation to $1/Y$ is required. This approximation is made by a table-lookup on the most significand bits of $Y$ [28]. If the initial approximation has $k$ bits of accuracy, then approximately $\lceil \log_2(p/k) \rceil$ iterations are required to compute a reciprocal which is accurate to $p$ bits. A similar algorithm which avoids division and is based on Newton-Raphson iteration is used to compute square roots [24].

For interval division, the reciprocal of the lower and upper endpoints of the divisor are first computed. Interval multiplication is then used to produce the lower and upper endpoints of the quotient, as specified below

$$Q = X/Y = [a, b] \times [1/d, 1/c]$$
$$X/Y = \Big[\mathrm{RD}\big(\min(a/c, a/d, b/c, b/d)\big), \mathrm{RU}\big(\max(a/c, a/d, b/c, b/d)\big)\Big].$$

An adjustment step, similar to the one presented in [21], is used to ensure that both endpoints are correctly rounded. If the divisor interval contains zero, the resulting interval will contain positive or negative infinity. In this situation, extended interval arithmetic [12], which allows division by intervals containing zeros, is employed.

## 4.2. Dot product computation

Accurate dot products are essential for scientific applications. Given two vectors $X = [x_1, x_2, \ldots, x_n]$ and $Y = [y_1, y_2, \ldots, y_n]^T$, and a specified rounding mode $\theta$, the result of an exact dot product operation [17] is defined as

$$\theta(X \cdot Y) = \theta\left(\sum_{i=1}^{n} x_i \cdot y_i\right)$$

where all arithmetic operations are mathematically exact and only a single rounding is performed at the very end. To avoid overflow and rounding problems, exact dot products are only supported in hardware for IEEE double-precision numbers. The long accumulator is used to store the partial sum of the exact dot product. For each term in the dot product, the new product and the appropriate words from the long accumulator are added together. The words chosen from the long accumulator and the amount that these words are shifted is determined by the exponent of the new product.

When adding the new product to the partial dot product, it is possible for carries to propagate over long distances, resulting in a large number of additions. To prevent this, each

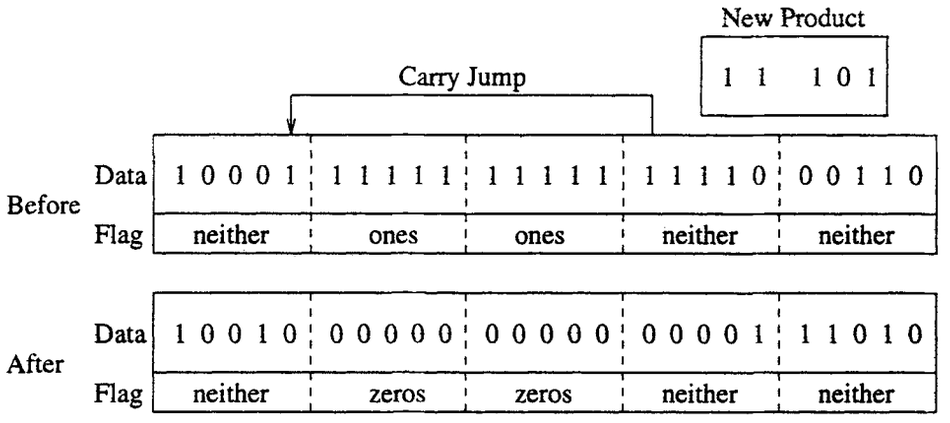


Figure 8. Accumulation of products for an accurate dot product

segment of the long accumulator has a flag associated with it which tells if the bits in the segment contain all ones, all zeros, or neither [13, 14]. A carry propagating into a segment which contains all ones will cause the flag to signal all zeros. Similarly, a borrow into a segment which contains all zeros will cause the flag to signal all ones. If a carry or a borrow comes into a segment which is neither all ones nor all zeros, the carry will not be propagated beyond that segment. Using this technique limits the number of additions that are performed for each element in the dot product to three; two to add the new product and one to resolve the carry. Figure 8 demonstrates the accumulation process using five bit segments. The new product 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. After the accumulation, the words which contained all ones now contain all zeros. Once the entire dot product is computed, it is normalized, rounded to a specified precision, and stored back in the register file. The *all ones* and *all zeros* flags also help simplify the normalization and rounding process, because they indicate the most significand word of the result and are used to determine the sticky bit.

As shown in [4], if a number representation uses at most $LMAX$ bits for the significand and the exponent ranges from $e_{\min}$ to $e_{\max}$, the dot product of two vectors each containing $2^N$ numbers can be computed exactly if the number of bits in the long accumulator is

$$\text{BITS\_LA} = N + 2 \times (\text{LMAX} + e_{\max} + |e_{\min}|).$$

Since BITS_LA = 8,192 and $2 \times (L + e_{\max} + |e_{\min}|) = 4{,}198$, $N = 8{,}192 - 4{,}198 = 3{,}994$. Thus, the exact dot product of two vectors containing $2^{3{,}994}$ IEEE double-precision numbers can be obtained.

To compute an exact interval dot product, it is necessary to determine the lower and upper endpoints of each multiplication. The lower endpoint is added to a portion of the long accumulator which contains the lower endpoint of the dot product. The upper endpoint is added a portion of the long accumulator which contains the upper endpoint of the dot product. After the upper and lower endpoints for the entire dot product have been computed, they are outward rounded to produce the final result.

### 4.3. Elementary function evaluation

Elementary function evaluation is performed by polynomial approximations. These approximations have the form

$$f(x) \approx p_{n-1}(x) = a_0 + a_1 \cdot x + a_2 \cdot x^2 + \cdots + a_{n-1} \cdot x^{n-1} = \sum_{i=0}^{n-1} a_i \cdot x^i$$

where $f(x)$ is the function to be approximated, $p_{n-1}(x)$ is a polynomial of degree $n-1$, and $a_i$ is the coefficient of the $i$-th term. With this method, the elementary functions are approximated using variable-precision addition and multiplication. The functions are approximated on a specified input interval and argument reduction is employed for values outside this interval [9]. To reduce the required number of multiplications, Horner's rule [22] is applied, so that the approximation takes the form

$$p_{n-1}(x) = a_0 + x \cdot \Big(a_1 + x \cdot \Big(a_2 + x \cdot \Big(a_3 + \cdots + x \cdot (a_{n-2} + x \cdot a_{n-1}) \ldots\Big)\Big)\Big).$$

With Horner's rule, a polynomial approximation of degree $n-1$ requires $n-1$ multiplications and $n-1$ additions.

If a Chebyshev series approximation of degree $n-1$ is used for the approximation, the maximum approximation error on an interval $[a, b]$ is

$$E_n(x) \leq \left(\frac{b-a}{4}\right)^n \cdot \frac{2f^n(\xi)}{n!} \quad (a \leq \xi \leq b)$$

where $\xi$ is the point on $[a, b]$ where the $n$-th derivative of $f(x)$ has its maximum value [22]. To reduce the number of terms in the approximation, the interval on which the function is computed is divided into subintervals and separate coefficients are used for each subinterval. The coefficients for each subinterval are determined by a table-lookup on the most significant bits of $x$. A more detailed description of this process is given in [26, 27].

The evaluation of an elementary function $f(x)$ on an interval $X = [a, b]$ is performed as follows: If $f(x)$ is monotonically increasing on $[a, b]$, the resulting interval is

$$f(X) = \Big[\mathrm{RD}\big(f(a)\big), \mathrm{RU}\big(f(b)\big)\Big].$$

If $f(x)$ is monotically decreasing on $[a, b]$ the resulting interval is

$$f(X) = \Big[\mathrm{RD}\big(f(b)\big), \mathrm{RU}\big(f(a)\big)\Big].$$

If $f(x)$ is are neither monotonically increasing nor decreasing on $[a, b]$, $f(x)$ is evaluated at its local minimum and maximum, and at the interval endpoints to determine the resulting interval. For example, if $\sin(x)$ is evaluated on the interval $[-1, 2]$, then the resulting interval is $\Big[\mathrm{RD}\big(\sin(-1)\big), 1.0\Big]$, since $\sin(X)$ has a local maximum of 1.0 at $\pi/2$.

## 5. Area and delay estimates

Table 1 gives area and delay estimates for the variable-precision, interval arithmetic processor. These estimates are based on data from a 1.0 micron CMOS standard cell library [20]. The

estimates for the multiplier assume that multiplication is implemented using a Reduced Area Multiplier [3], followed by a carry look-ahead adder [5]. 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 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 chip area is estimated to be 227.5 $mm^2$. The component delays are computed by taking the worst case delay of the critical path and adding 25 percent for unexpected delays and clock skew. The worst case component delay comes from the multiplier which has a delay of 37.6 ns; 19.2 ns for partial product reduction and 18.4 ns for carry look-ahead addition. Assuming these two stages of the multiplication are separately pipelined and allowing an additional 2 ns for control and register latching, the processor could have a cycle time of less than 22 ns (45 MHz).

| Unit | Area ($mm^2$) | Delay (ns) |
|---|---|---|
| Multiplier (64 bits by 64 bits) | 49.4 | 37.6 |
| Carry look-ahead adder (133 bits) | 5.2 | 18.4 |
| Significand memory (256 words by 64 bits) | 35.5 | 8.2 |
| Header memory (64 words by 32 bits) | 4.4 | 7.2 |
| Long accumulator (64 words by 128 bits) | 17.8 | 7.8 |
| Shifter + zero detect (128 bits) | 8.5 | 9.2 |
| Shifter (128 bits) | 8.2 | 8.9 |
| Operand selector (4 words by 128 bits) | 7.8 | 4.2 |
| Exponent add/subtract (16 bits) | 0.6 | 4.4 |
| Latches | 4.8 | 2.0 |
| Control logic, global routin, pads, etc | 82.7 | * |
| Total | 227.5 | * |

Table 1. Estimates for the variable-precision, interval arithmetic processor

Table 2 gives the area and delay estimates for a IEEE double-precision floating point processor, using the same assumptions as above. The worst case component delay comes from the multiplier which has a delay of 34.6 ns; 18.0 ns for partial product reduction and 16.6 ns for carry look-ahead addition. If these two stages of the multiplication are separately pipelined, the processor can have a cycle time of less than 20 ns (50 MHz). Compared to the variable-precision, interval arithmetic processor, the IEEE double-precision processor uses approximately 56 percent less area and has a cycle time which is approximately 9 percent shorter. The main increase in area comes from the increased size of the arithmetic units, the larger register file, and the memory needed for the long accumulator.

# 6. Conclusions

This paper presented the software interface and hardware design for a computer system which supports variable-precision, interval arithmetic. The software interface allows the programmer

| Unit | Area ($mm^2$) | Delay (ns) |
|---|---|---|
| Multiplier (53 bits by 53 bits) | 37.4 | 34.6 |
| Carry look-ahead adder (106 bits) | 4.3 | 16.6 |
| Register file (16 words by 64 bits) | 4.4 | 6.2 |
| Shifter + zero detect (106 bits) | 7.2 | 9.0 |
| Shifter (106 bits) | 6.9 | 8.8 |
| Exponent add/subtract (11 bits) | 0.4 | 4.0 |
| Latches | 2.4 | 2.0 |
| Control logic, global routin, pads, etc | 37.8 | * |
| Total | 100.8 | * |

Table 2. Estimates for the IEEE double-precision processor

to specify the precision of the computation and recompute the result with higher precision when the required accuracy is not achieved. The processor can also be used to evaluate the accuracy of programs before running them on a general purpose processor, or can be used to select between various programs based on their accuracy for given inputs. By providing hardware support for variable-precision, interval arithmetic, a substantial speedup over existing software methods is achieved. Area and delay estimates demonstrate the efficiency of the design. The hardware design and arithmetic algorithms have been simulated in C++. Further testing, hardware simulation, and performance evaluation are required before the design is complete.

# Acknowledgements

Special thanks are extended to Professor Jeff Ely at Louis and Clark University who provided the variable-precision, interval arithmetic package which was modified to simulate the hardware design presented in this paper.

# References

[1] Adams, E. and Kulisch, U. *Scientific computing with automatic result verification.* Academic Press, 1993, pp. 1–12.

[2] Anderson, F. S. et al. *The IBM System/360 Model 91: floating point execution unit.* IBM Journal of Research and Development **11** (1967), pp. 24–53.

[3] Bickerstaff, K. C., Schulte, M. J., and Swartzlander, E. E. *Reduced area multipliers.* In: "Proceedings 1993 Application Specific Array Processors", 1993, pp. 478–489.

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

[5] Brent, R. P. and Kung, H. Y. *A regular layout for parallel adders.* IEEE Transactions on Computers **C–31** (1982), pp. 260–264.

[6] Carter, T. *Cascade: hardware for high/variable precision arithmetic.* In: "Ninth Symposium on Computer Arithmetic", 1989, pp. 184–191.

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

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

[9] Fike, C. T. *Computer evaluation of mathematical functions.* Prentice Hall, Englewood Cliffs, NJ, 1968.

[10] Flynn, M. J. *On division by functional iterations.* IEEE Transactions on Computers **C–19** (1970) pp. 702–706.

[11] Hammer, R., Neaga, M., and Ratz, D. *Pascal–XSC new concepts for scientific computation and numerical data processing.* In: Adams, E. and Kulisch, U. (eds) "Scientific Computing with Automatic Result Verification", Academic Press, 1993, pp. 15–44.

[12] Hansen, E. *Global optimization using interval analysis.* Marcel Dekker, New York, NY, 1992.

[13] Knofel, A. *Fast hardware units for the computation of accurate dot products.* In: "Tenth Symposium on Computer Arithmetic", 1991, pp. 70–75.

[14] Knofel, A. *Hardware kernel for scientific/engineering compuations.* In: Adams, E. and Kulisch, U. (eds) "Scientific Computing with Automatic Result Verification", Academic Press, 1993, pp. 549–570.

[15] Koren, I. *Computer arithmetic and algorithms.* Prentice Hall, Englewood Cliffs, NJ, 1993.

[16] 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.

[17] Kulisch, U. W. and Miranker, W. L. *Computer arithmetic in theory and in practice.* Academic Press, New York, NY, 1981.

[18] *IEEE Standard 754 for binary floating point arithmetic.* American National Standards Institute, Washington, DC, 1985.

[19] Lawo, C. *C–XSC new concepts for scientific computation and numerical data processing.* In: Adams, E. and Kulisch, U. (eds) "Scientific Computing with Automatic Result Verification", Academic Press, 1993, pp. 71–86.

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

[21] Markstein, P. W. *Computation of elementary functions on the IBM RISC System/6000 processor.* IBM Journal of Research and Development **34** (1990), pp. 111–119.

[22] Mathews, J. H. *Numerical methods for computer science, engineering and mathematics.* Prentice Hall, Englewood Cliffs, NJ, 1987.

[23] Moore, R. E. *Interval analysis.* Prentice Hall, Englewood Cliffs, NJ, 1966.

[24] Ramamoorthy, C. V., Goodman, J. R., and Kim, K. H. *Properties of iterative square-rooting methods using high-speed multiplication.* IEEE Transactions on Computers **C–21** (1972), pp. 837–847.

[25] Reuter, E. K. et al. *Some experiments using interval arithmetic.* In: "Fourth Symposium on Computer Arithmetic", 1978, pp. 75–81.

[26] Schulte, M. J. and Swartzlander, E. E. *Exact rounding of certain elementary functions.* In: "Eleventh Symposium on Computer Arithmetic", 1993, pp. 128–145.

[27] Schulte, M. J. and Swartzlander, E. E. *Parallel hardware designs for correctly rounded elementary functions.* Interval Computations 4 (1993), pp. 65–88.

[28] Schulte, M. J., Omar, J., and Swartzlander, E. E. *Optimal initial approximations for the Newton-Raphson division algorithm.* Submitted to Computing, 1994.

[29] 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.

Received: February 25, 1994
Revised version: November 23, 1994

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