Applied and Computational Mathematics Division

NISTIR 5660

Computing and Applied Mathematics Laboratory

Parallel and Serial Implementations of SLI Arithmetic

Daniel W. Lozier
Peter R. Turner

June 1995

U.S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards and Technology
Gaithersburg, MD 20899

NIST

NISTIR 5660

Parallel and Serial Implementations of SLI Arithmetic

Daniel W. Lozier
Peter R. Turner

U.S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards
and Technology
Applied and Computational Mathematics Division
Computing and Applied Mathematics Laboratory
Gaithersburg, MD 20899

June 1995

U.S. DEPARTMENT OF COMMERCE
Ronald H. Brown, Secretary

TECHNOLOGY ADMINISTRATION
Mary L. Good, Under Secretary for Technology

NATIONAL INSTITUTE OF STANDARDS
AND TECHNOLOGY
Arati Prabhakar, Director

PREPRINT

This paper has been submitted to *Theoretical Computer Science*, a technical journal published by Elsevier Science, Amsterdam, The Netherlands, for a planned special issue on computing with real numbers. The paper is subject to revision for compliance with the recommendations and requirements of referees and editors.

PARALLEL AND SERIAL IMPLEMENTATIONS OF SLI ARITHMETIC

DANIEL W. LOZIER AND PETER R. TURNER

May 5, 1995

ABSTRACT. This paper describes the various algorithms and software implementations of the Level-Index LI and Symmetric Level-Index SLI arithmetic schemes. After a brief introduction to the number representations and the arithmetic algorithms, we describe the original precompiler for including LI and SLI variables and their arithmetic in a Fortran 77 program. The Turbo Pascal unit for SLI arithmetic includes several extended real operations and, also, complex operations using a modulus-argument representation. The Fortran 90 implementation avoids the need for a precompiler and includes extensive testing of the basic algorithms. Finally, we describe the parallel SIMD implementation of SLI arithmetic on a MasPar MP-1 using a massively parallel version of C.

1. INTRODUCTION

This paper is concerned with the software implementations of the level-index LI and symmetric level-index SLI arithmetic schemes which are available for experimental computing on various architectures. Earlier versions of some of these have been described elsewhere, and in those cases only a summary of their features is included here. Before describing the paper in more detail and introducing the LI and SLI systems, we discuss very briefly some of the motivation for this and other arithmetic systems.

One of the primary drawbacks of the binary floating-point system for real-number representation and arithmetic in a general computational environment is its susceptibility to overflow and underflow. Several suggestions have been made and studied to either alleviate or overcome these difficulties, and to address the related problem of the "spacing" between successive representable numbers in the floating-point system. In binary floating-point arithmetic the difference between successive machine numbers is a step function which doubles with every unit increase in the exponent value. The systems which attempt to overcome the overflow problem necessarily address this problem too since greater range within a given wordlength can only be achieved by allowing this spacing to erode gradually.

There have been several suggestions for extended floating-point systems which are mostly modifications of the tapered floating-point proposal of Morris [26]. Proposals for eventual hardware systems have included Matsui and Iri's [25] which uses not only variable lengths for its mantissa and exponent but also allows the

possibility of extending to further "levels" once the basic representable range is exceeded. However these additional levels were not implemented in their work. The system introduced by Hamada [15] and extended in [16] and [39] modified the scheme of Matsui and Iri in using only its level 0 and changing the way in which the "pointer" (the indicator as to how many bits are used by the exponent) is stored. This pointer becomes part of the exponent information. A different modification of the floating-point scheme was proposed by Hull and his coworkers (see [17], for example) using a scheme which allows variable range and precision for its decimal floating-point arithmetic.

In all of these schemes overflow remains a possibility, although a much reduced one. The step-function nature of the spacing between successive numbers remains although the relative precision of the representation is eroded monotonically as the range increases for all except for Hull's scheme, in which precision is increased with the allowable range.

One other approach to the overflow problem is logarithmic arithmetic. Algorithms for this have been discussed in, for example, [2] and [3], error analysis in this system is introduced in [4], and potential hardware designs are described in [18, 19] and [20]. The basic principle of the logarithmic scheme is that a real number is represented by the logarithm of its absolute value relative to some fixed base. A second sign is used for the sign of the original real number. Thus $X > 0$ is represented by E_X such that

$$X = r^{E_X}$$

where r is the fixed radix (usually 2) and E_X is a fixed-point exponent. The number of fractional bits in the representation of this exponent dictates the (constant) relative spacing between successive numbers in this system.

The level-index LI system of number representation and computer arithmetic was first introduced by Clenshaw and Olver in [9]. The representation is based on the use of a *generalized logarithm* function. In that sense the system can be viewed as a natural extension of the ideas of the logarithmic number systems. It is also a generalization of the Matsui-Iri representation in using even more "levels" than they envisage for extending the binary floating-point system. The details of the system and its symmetric counterpart SLI are detailed in subsequent sections but an introduction to the basic idea and some of the notation will be helpful here.

The representation of a large positive number can be achieved by taking its natural logarithm repeatedly until a result is obtained in the interval [0, 1). The number of times the logarithm has been taken is called the *level* while the final value in [0, 1) is called the *index*. Since the level is an integer, no ambiguity is created by representing the original large number by the sum of its level and its index. This representation function is called a generalized logarithm and is denoted here by ψ. Its inverse function is called a *generalized exponential* function and is denoted by ϕ. We refer to *a* generalized logarithm and *a* generalized exponential function since there are many such functions. They have been studied extensively in [8] while their properties for computer arithmetic have been the subject of a series of papers several of which are cited in the references. A good introduction to the subject can be found in [11].

In the level-index scheme, a real number is represented by its sign together with the LI representation of its absolute value. Thus X is represented by $\pm x$ where the

sign is the sign of X and

$$x = \psi(|X|), \tag{1}$$

or equivalently,

$$|X| = \phi(x). \tag{2}$$

The generalized exponential function most commonly used for the LI arithmetic system is defined by

$$\phi(x) = \begin{cases} x & \text{if } 0 \le x < 1, \\ e^{\phi(x-1)} & \text{if } x \ge 1. \end{cases} \tag{3}$$

The corresponding generalized logarithm is given by

$$\psi(X) = \begin{cases} X & \text{if } 0 \le X < 1, \\ 1 + \psi(\ln(X)) & \text{if } X \ge 1. \end{cases} \tag{4}$$

Thus, if $x = l + f$ where l is the integer part of x is the LI representation of $X > 0$, it follows that

$$X = \phi(x) = e^{e^{\cdot^{\cdot^{e^f}}}}, \quad x = \psi(X) = l + \ln(\ln(\ldots \ln(X)))$$

where the exponentiation or logarithm is performed l times.

The SLI system [12] uses a similar representation function for "large" quantities, that is $|X| \ge 1$. However for "small" numbers the LI representation of its reciprocal is used; this reciprocal need not be formed, this is just a convenient way to describe the representation. Thus a real number X is represented in the SLI system by

$$X = \pm\phi(x)^{\pm 1} \tag{5}$$

where $x \ge 1$. The principal sign is the sign of X and the reciprocation sign is $+1$ if $|X| \ge 1$ and -1 otherwise. It follows that for the SLI representation

$$x = 1 + \psi(|\ln|X||). \tag{6}$$

The basic properties of the LI and SLI systems including representation, arithmetic and analysis have been extensively discussed in [9, 10, 11, 12, 22], and [27]. Possible hardware algorithms were the subject of [28] and [33]. Applications using this system have been detailed in [13, 21, 23], and [24].

The next section of this paper is devoted to a brief introduction to the SLI arithmetic algorithms. This is followed in §3 by a description of the first software implementation using a Fortran 77 precompiler. This implementation uses an older version of the arithmetic algorithms but has the virtue of allowing existing Fortran 77 code to be run with just a change of variable declarations. In §4, a Turbo Pascal implementation is described. This was the first implementation to use the algorithms described in §2. It also has been augmented with various extended real operations, complex SLI arithmetic, and mixed operations for integer-SLI arithmetic. The representation and arithmetic adopt the internal wordlengths obtained in the various error analyses. The implementation described in §5 relates to development of Fortran 90 code which eliminates the need for a precompiler. This code also allows the internal wordlengths to be varied, making experimentation with the

arithmetic algorithms easier. The last implementation is written in MPL, a massively parallel version of ANSI C, for a MasPar MP-1 system. This takes advantage of the parallelism to reduce the time-penalties associated with any software arithmetic. It also allows exploration of potential parallelism in any future hardware designs.

2. LI and SLI Arithmetic Algorithms

Algorithms for the basic arithmetic were first presented in [10] and [12]. In the LI algorithms, there are special cases to be treated when one or more of the arguments has level 0. Because these special cases do not arise in SLI arithmetic, they are omitted from the following description; the missing details can be found in [10]. The resulting LI algorithms are simplified, and they form the basis of the fundamental SLI arithmetic operations.

LI multiplication and division can be achieved by taking logarithms (or decrementing the levels) of the arguments and using the addition or subtraction algorithms. We concentrate therefore on LI addition and subtraction in which we seek the LI representation $\phi(z)$:

$$\phi(z) = Z = X \pm Y = \phi(x) \pm \phi(y) \tag{7}$$

where we shall assume that

$$x \geq y \geq 1 \quad \text{and} \quad z \geq 1.$$

Denote the levels and indices of these representations by

$$x = l + f, \quad y = m + g, \quad \text{and} \quad z = n + h$$

Now, dividing (7) by the larger argument, we obtain

$$c_0 = \frac{\phi(z)}{\phi(x)} = 1 \pm \frac{\phi(y)}{\phi(x)} = 1 \pm b_0. \tag{8}$$

The LI algorithms are based on computing b_0 by using a (short) recursive sequence and then obtaining z from c_0 using another recursive sequence. One characterization of this algorithm is that $\phi(z)$ is being computed as a relative perturbation of the larger operand $\phi(x)$.

To compute b_0, we use a recurrence relation for

$$b_j = \frac{\phi(y-j)}{\phi(x-j)} \quad \text{for } j = m-1, m-2, \ldots, 0. \tag{9}$$

The starting value for this recurrence is

$$b_{m-1} = \frac{\phi(1+g)}{\phi(x-m+1)} = \exp(g - \phi(x-m)) = \begin{cases} e^{g-f} & \text{if } m = l, \\ e^{g-1/a_m} & \text{if } m < l \end{cases}$$

where $a_m = 1/\phi(x-m)$ is obtained from another recurrence relation for

$$a_j = \frac{1}{\phi(x-j)} \quad \text{for } j = l-1, l-2, \ldots, 0. \tag{10}$$

The recurrence for this a-sequence begins with $a_{l-1} = 1/\phi(1+f) = e^{-f}$ and is then given by

$$a_{j-1} = \exp\left(\frac{-1}{a_j}\right) \quad \text{for } j = l-1, l-2, ..., 1. \tag{11}$$

The corresponding recurrence for the b-sequence is

$$b_{j-1} = \exp\left(\frac{-(1-b_j)}{a_j}\right) \quad \text{for } j = m-1, m-2, \ldots, 1. \tag{12}$$

Both of these are easily verified using the definitions. For example, for $0 < j < l$,

$$a_{j-1} = \frac{1}{\phi(x-j+1)} = \frac{1}{\exp\left(\phi(x-j)\right)} = \exp\left(-\phi(x-j)\right) = \exp\left(\frac{-1}{a_j}\right).$$

It is apparent from (11) and (12) that $0 < a_j, b_j \leq 1$ for every j. (Only b_j can equal 1 and that only if $x = y$.) We observe that, from $j = m-1$ onwards, these two recurrence relations can be computed in parallel on a machine with even minimal parallelism.

Before completing the description of the LI algorithm, we discuss briefly the modifications that are needed for SLI arithmetic. Full details of these algorithms and their analysis are included in [12]. The most important difference is that one or both of the operands may be "small", that is, in reciprocal form. In the case of "mixed" arithmetic we seek

$$\phi(z) = Z = X \pm Y = \phi(x) \pm \phi(y)^{-1} \tag{13}$$

and, dividing by the larger argument, we get

$$c_0 = 1 \pm \frac{1}{\phi(x)\phi(y)} = 1 \pm a_0\alpha_0 \tag{14}$$

where $a_0 = 1/\phi(x)$ is computed as before and $\alpha_0 = 1/\phi(y)$ is also computed using an a-sequence but with y in place of x.

In the "small" case, we require

$$\phi(z)^{-1} = Z = X \pm Y = \phi(x)^{-1} \pm \phi(y)^{-1} \tag{15}$$

and dividing by $1/\phi(x)$ yields

$$c_0^{-1} = 1 \pm \frac{\phi(x)}{\phi(y)} = 1 \pm b_0^{-1} = 1 \pm \beta_0 \tag{16}$$

where the quantity β_0 can be computed by a recurrence similar to, but slightly more complex than, (12).

In all cases, we conclude the first phase of the algorithm by computing either c_0 or its reciprocal. These are essentially equivalent for the purpose of the SLI algorithm. There are exceptional cases for the remaining part of the algorithm which are detailed in [12] but are not discussed here. These relate to the possibility that the difference between two "large" numbers (or between a large one and a small one) may be "small" or the sum of two small numbers may be large. These *flipover* cases can be treated by minor variations of the algorithm described here.

Otherwise the large and mixed operations are identical from this point. We define members of the c-sequence by

$$c_j = \frac{\phi(z-j)}{\phi(x-j)} \tag{17}$$

for $j = 0, 1, ..., \min(n, l)$. Now the condition $c_j < a_j$ is equivalent to $\phi(z-j) < 1$, and so $n = j$ and $h = \phi(z-j) = c_j/a_j$ which will complete the computation. There is a simple recurrence for most of this c-sequence which is essentially the reversal of the direction of (12):

$$c_{j+1} = 1 + a_{j+1} \ln c_j \tag{18}$$

for j not greater than $l-2$, with the recursion terminating if the above condition is satisfied. If necessary, the final member of this sequence is replaced by

$$h_l = f + \ln c_{l-1}$$

and at most one further logarithm is needed in the eventuality that $h_l \geq 1$. (One additional logarithm is always sufficient since $2\phi(x) < \exp(\phi(x)) = \phi(x+1)$ so that a sum can never have a level that exceeds that of the larger argument by more than 1.)

The completion of the small arithmetic algorithm is similar. The biggest difference arises from the fact that we do not have c_0 but c_0^{-1}. It follows that (18) must be modified so that

$$c_1 = 1 - a_1 \ln c_0^{-1}$$

after which the rest of this sequence can be computed using (18). The only other difference is that more than one additional logarithm may be necessary in order to obtain the final h-value. (The difference of two small numbers can be *much* smaller than either operand and so its (reciprocal) level can exceed those of the operands by more than 1.)

The various quantities involved in (all the variants of) this algorithm are uniformly bounded and can be computed to fixed absolute precisions. These working precisions are examined in [10] and [12].

3. Fortran 77 Implementation

An unpublished implementation in Fortran 77 was developed in the mid-1980's by D. W. Lozier. Its purpose was to provide a test vehicle which would allow Fortran programs to be re-interpreted and re-executed in LI or SLI arithmetic. This was accomplished by embedding into the language new data types for LI and SLI variables, and extending to these most of the standard operations and functions for data of type `REAL`. This implementation was used, for example, in [21] to solve a graphics problem that arose in a model of turbulent combustion; see [29, 30] for the physical and chemical details.

3.1. Approach. A very important potential audience for LI and SLI arithmetic is in scientific computing. Particular emphasis was placed on Fortran in this implementation because it seemed clear in the early and mid 1980's that it was the predominant language of scientific computing. Since then other languages and systems have gained importance but the choice of Fortran as a vehicle for an implementation of LI and SLI arithmetic is still justified.

The main argument against Fortran 77 was that it offers no direct support for the introduction of data types (this objection is removed in Fortran 90; see §5 below). The same objection had been raised and overcome in the 1970's with respect to other nonstandard arithmetic systems. For example, multiple-precision systems, which extend floating-point arithmetic to arbitrary precision, are described in [5] and [38]. The software described in the first of these references is known as *MP* (Multiple Precision), and in the remaining reference as *SP* (Super Precision). The new SP and MP data types are introduced indirectly by means of a precompiler, i.e. a language compiler that accepts a Fortran-like program as input and generates a standard Fortran program as output. This output is then compiled, linked and executed just like any other Fortran program.

A precompiler is included with SP as a necessary part of a general multiple-precision computing facility. Originally MP had no precompiler but later it was linked to Augment [14]. This linkage is described in [7]. All SP and MP operations, including arithmetic, assignment, comparison, function evaluation, input, output and type conversion, are performed by ordinary Fortran subprograms of either subroutine or function type. The task of the precompiler is to recognize declarations of nonstandard variables and to connect each operation that involves them to the appropriate subprogram. The precompilers operate by reading an input file that details all of these connections, then reading and translating the input Fortran-like program.

Since the precompiler approach had been successful for multiple-precision arithmetic, it was decided the same approach should be used to develop a Fortran-like LI and SLI facility.

3.2. Slitran. A Fortran-like language, identified in this paper as Slitran, was implemented using Augment [14] to provide three new numerical data types:

```
LEVEL INDEX
SYMMETRIC LEVEL INDEX
FLOATING POINT
```

Variables of these new types are carried in standard variables of type `DOUBLE PRECISION`, and operations with them are simulated by algorithms using double-precision floating-point arithmetic operations.

Operators and functions are provided for each new type:

Unary operators + and -
Arithmetic operators +, -, *, /, and **
Logical operators .EQ., .NE., .GT., .GE., .LT., and .LE.
Assignment =
Absolute value and square root functions ABS and SQRT
Exponential and logarithmic functions EXP and LOG
Maximum and minimum functions MAX and MIN

Where an operator or function has more than one operand or argument, all must be of the same type. With the obvious exception of the logical operators, all the operators and functions, including assignments, produce a result of the same type as the operands. Explicit functions are provided for type conversion among all combinations of these types and the three Fortran types `INTEGER`, `REAL` and `DOUBLE PRECISION`. The remaining three Fortran types, `COMPLEX`, `LOGICAL` and `CHARACTER` are recognized and processed by Slitran but they do not interact with the new types.

The new type `FLOATING POINT` was introduced for two reasons. First, it provides a simulation of IEEE-standard floating-point formats and arithmetic operations on non-IEEE computers. This was felt to be desirable to enable comparisons to be made of new arithmetic systems against the best available system for general scientific computing. It is usually conceded that the IEEE standard is a close-to-optimal specification of floating-point arithmetic, at least for a 32-bit binary format. Second, the new type incorporates symbols for underflow, overflow, and "infinite" and "indefinite" numbers, allowing computation to proceed in the face of floating-point exceptions in a manner analogous to computing with "Not-a-Numbers", or NaNs, in IEEE arithmetic. This is important because comparisons will be wanted between IEEE arithmetic and the new arithmetic systems in problems where IEEE arithmetic leads to underflow or overflow.

Slitran usage is similar, up to a point, to ordinary Fortran usage. Variables are declared as in Fortran, and arithmetic expressions and function invocations are coded in the usual fashion, but input and output of variables of the new types require special coding. No special facilities other than type conversion functions are provided for input, since input will almost always be within the normal floating-point range. Input is read into variables of Fortran type, then converted explicitly to one of the new types.

Ordinary Fortran output normally requires a `WRITE` statement with a unit specifier, a format designator, and an output list:

```
WRITE(unit,format) expr_1, expr_2, ..., expr_n
```

A new statement is provided for usage when new types are among the variables to be output:

```
XWRITE(unit) = expr_1 & expr_2 & ...  & expr_n
```

where each expression in the output list can have any one of the nine supported types. The symbol `&` is an "operator" defined such that Augment can construct a character string from the output list, and = is an "assignment" that causes Augment to write the string to the `unit`. Technically, `XWRITE` is a special form of Fortran function, called a "field function", which can be used on the left side of an assignment. In Fortran, the format designator either points to or is itself a character string. A different method, using *global parameters*, is used in Slitran.

There are some 67 global parameters in Slitran that can be used for fine control of its exact behavior. Each has an alphanumeric name, represented by a character string, and a corresponding integer value. They are manipulated with a field function named SYS (for *system*). For example, let `I` be a variable of type integer. Then

```
I = SYS('SWIDTH')
```

sets I equal to the current integer value of SWIDTH (the output field width in characters for variables of type SYMMETRIC LEVEL INDEX), and

```
SYS('SWIDTH') = I
```

changes the current value of SWIDTH to the value of I. Statements like

```
SYS('SWIDTH') = SYS('SWIDTH') + 10
```

are valid; this one increases the current value of SWIDTH by 10.

The global parameters associated with output operations specify the *field width*, the *number of decimals after the decimal point*, and, in the case of floating-point types, the *number of decimals in the exponent*. All have default values, e.g., 16, 8 and 2. Numbers are right-justified in the output field, and the field is blank-filled on the left. Numbers of type REAL, DOUBLE PRECISION, or FLOATING POINT are expressed in Fortran E format with one digit before the decimal point. When a number of type FLOATING POINT is generated that is outside the representable range, Slitran classifies it as underflow, overflow or the result of an indefinite operation (such as division by zero). In these cases, an appropriate alphabetic string is placed in the output field. Numbers of type LEVEL INDEX or SYMMETRIC LEVEL INDEX are expressed in Fortran F format with one digit in the integer part to represent the level. The numbers are enclosed in square brackets, with the sign of the number in front of the opening bracket. The reciprocation indicator in the case of SLI variables is placed behind the opening bracket.

The direct output of LI and SLI variables using XWRITE displays the level and index in decimal. Usually output will be wanted in a more familiar format. The type conversion function FLP is useful in this regard. It takes an argument of any type and converts it to type FLOATING POINT. If a number underflows or overflows, a special bit pattern is stored, and this is recognized during output processing.

The internal format of the new types is also subject to control by global parameters. The default wordlength is 32 bits, and the default floating-point parameters are those of the IEEE standard. Because these three types are carried internally in variables of type DOUBLE PRECISION, the new types cannot be simulated to more than 48 bits or so.

Three modes of *abbreviation* are supported for the new types: *rounding*, *chopping* and *unabbreviated*. Rounding is done by adding a half-unit in the last bit position of the significand or index, then truncating and storing the result. Chopping is done by simply truncating and storing the result. The unabbreviated mode does not modify the double-precision result before storing it. Rounding is the default mode; the other modes can be selected by setting the appropriate global parameter. For example, coding like

```
MODE = SYS('ABBREV')
SYS('ABBREV') = SYS('UNABBR')
code to be executed to full available precision
SYS('ABBREV') = MODE
```

can be useful in computations that require guard digits.

As was seen in §1, the generalized exponential function is determined by its definition on the unit interval. Three different choices may be selected by setting the appropriate global parameter. The simplest definition, and this is the default, arises from the identity function; see eq. (3). It is continuous and continuously

differentiable but higher derivatives are discontinuous. The alternative choices are much smoother. They were provided for experimentation with alternative SLI arithmetic algorithms based on surface fitting.

The remaining global parameters will not be described here. Most are used to control the interception and processing of error conditions.

3.3. Algorithms. The algorithms used to perform arithmetic operations in Slitran were a precursor to the ones described in §2. They are less efficient in that they employ a doubly recursive computing process. They are described here as a necessary part of the description of Slitran, and also because they afford an independent check on the newer algorithms.

We begin with LI subtraction. It is sufficient to consider the equation

$$\phi(z) = \phi(x) - \phi(y) \quad (x \geq y \geq 0). \tag{19}$$

As in §2, we use the notation

$$x = l + f, \quad y = m + g, \quad z = n + h \tag{20}$$

where l, m and n are the levels, respectively, of $\phi(x), \phi(y)$ and $\phi(z)$. Our task is to compute z, i.e. n and h, given l, f, m, and g. Toward this end, we introduce the further equations

$$\phi(z_i) = \phi(x_i) - \phi(y_i) \quad (i = 0, 1, \ldots, m) \tag{21}$$

where

$$x_i = l - m + i + f, \quad y_i = i + g, \quad z_i = n_i + h_i. \tag{22}$$

An outer recursion generates the sequence $z_0, z_1, \ldots, z_m = z$. The ith term of this sequence satisfies the equation

$$\phi(z_i) = c_0^{(i)}/a_{m-i} \tag{23}$$

where $a_{m-i} = 1/\phi(x_i)$ is a term of the a-sequence defined in eq. (10) and

$$c_0^{(i)} = 1 - \phi(y_i)/\phi(x_i). \tag{24}$$

Now, if $c_0^{(i)} < a_{m-i}$, then $n_i = 0$ and $h_i = c_0^{(i)}/a_{m-i}$. Otherwise, logarithms of $\phi(z_i)$ must be computed repeatedly until the result is in the interval $[0, 1)$. This constitutes the inner recursion. Assuming that we have

$$\ln^j \phi(z_i) = c_j^{(i)}/a_{m-i+j}, \tag{25}$$

then we can write

$$\ln^{j+1} \phi(z_i) = c_{j+1}^{(i)}/a_{m-i+j+1} \tag{26}$$

where

$$c_{j+1}^{(i)} = 1 + a_{m-i+j+1} \ln c_j^{(i)}. \tag{27}$$

The inner process ends with $n_i = j$ and $h_i = c_j^{(i)}/a_{m-i+j}$ when $c_j^{(i)} < a_{m-i+j}$. The c-sequence and c-recurrence of this section are analogous to the ones found in §2; cf. eqs. (17) and (18).

The outer recursion hinges on the determination of the starting value (24) for the c-sequence. When $i = 0$, we have

$$c_0^{(0)} = 1 - a_m g.$$

When $i \geq 1$, we can write

$$\frac{\phi(y_i)}{\phi(x_i)} = e^{\phi(y_i - 1) - \phi(x_i - 1)} = e^{-\phi(z_{i-1})} = \frac{1}{\phi(1 + z_{i-1})};$$

cf. eq. (21). The rightmost member of this equation is computed as an a-sequence with index h_{i-1} in place of f; cf. eqs. (10) and (11). Then it is substituted into eq. (24).

Turning now to LI addition, we continue with the assumption that $x \geq y \geq 0$. Define

$$\ln^j \phi(z) = \phi(x - j) + d_j \quad (j = 0, 1, \ldots, l)$$

Then either $n = l$ and $h = f + d_l$ (if $f + d_l < 1$), or $n = l + 1$ and $h = \ln(f + d_l)$. When $l = m = 0$, $d_0 = g$. Otherwise, the d-sequence is computed from the recurrence

$$d_r = \ln(1 + a_{j-1} d_{j-1})$$

starting from

$$d_1 = \begin{cases} \ln(1 + a_0 g) & \text{if } m = 0, \\ \ln(1 + 1/\phi(t + 1)) & \text{if } m > 0 \end{cases}$$

where

$$\phi(t + 1) = e^{\phi(t)} = e^{\phi(x-1) - \phi(y-1)}.$$

The subtraction procedure is called to compute t, and the term $1/\phi(t + 1)$ is computed by an a-sequence with the appropriate index.

Except for special cases when the levels of one or both operands are zero, LI multiplication and division rely on the identities

$$\frac{\phi(x)}{\phi(y)} = e^{\phi(x-1) - \phi(y-1)}, \quad \phi(x)\phi(y) = e^{\phi(x-1) + \phi(y-1)},$$

the right sides of which are easily obtained with the aid of the LI addition and subtraction procedures.

The SLI representation is slightly different in Slitran than in other implementations. An explicit reciprocation bit, as in eq. (5), is not used. Instead, a *positive* number X is represented by the *signed* number

$$x = \Psi(X) = \begin{cases} \psi(x) - 1 & \text{if } X \geq 1, \\ 1 - \psi(1/X) & \text{if } 0 < X < 1 \end{cases} \tag{28}$$

where ψ is the LI mapping (4). The inverse, in terms of the inverse LI mapping (3), is

$$X = \Phi(x) = \begin{cases} \phi(1 + x) & \text{if } x \geq 0, \\ 1/\phi(1 - x) & \text{if } x < 0. \end{cases}$$

The identity

$$\Phi(x)\Phi(-x) = 1$$

shows that changing the sign of an internal number corresponds to reciprocating the (positive) external number that it represents. The actual sign of the external number is carried separately, as it is in the other SLI implementations discussed in this paper.

It is worth noting that two signs are used in the definition of floating-point numbers. The familiar device of biasing (adding a constant to) the exponent, so as to avoid having to represent an explicit sign, is available also in SLI representation, and indeed it is used (with a bias constant of 8) in Slitran.

The efficiency of Slitran suffers in comparison to the other implementations because (i) SLI operations are done indirectly by calling on LI operations; (ii) LI operations do not take advantage of later algorithmic developments, such as the singly recursive algorithms for addition and subtraction; and (iii) global parameter control adds to the total run time. However, it is a close parallel to conventional Fortran and it offers the user considerable control over the algorithmic details of the simulated computer arithmetic. Furthermore, the inefficiencies are not inherent; they could be reduced by re-coding.

4. Turbo Pascal Implementation

The original version of this implementation of SLI arithmetic was described in [34]. Since then it has undergone a number of modifications, improvements and extensions. This Turbo Pascal[1] implementation was developed for experimental computation on personal computers. It uses the arithmetic algorithms described in §2 which avoid the doubly recursive aspect of the previous implementation. The algorithms for SLI arithmetic are based directly on the definition of that representation rather than using combinations of LI operations and the rules of algebra.

The SLI representation is mapped into a conventional binary integer representation in an order-preserving manner. Indeed, the Turbo Pascal type `slisingle` is identified with the 32-bit type `longint`. The machine representation consists of the appropriate binary encoding of the two signs and of x which, as before, has an integer part, the *level*, of three bits. The order-preserving nature of this representation is achieved by using a ones complement form for negatives, and complementing the level and index of quantities in reciprocal form. The packing algorithm is detailed in [34]

For this implementation, the precisions forecast by the error analyses of [10] and [12] are the precisions used in the internal computation. The built-in exponential and logarithmic functions are used internally. Proposals have been made for other internal algorithms based on table look-up or modified CORDIC algorithms [28, 33] but that is not the topic under present discussion.

The following subsections discuss some of the extensions that have been incorporated into the Turbo Pascal SLIUNIT. These include implementations of algorithms for mixed integer-SLI arithmetic using the integer exactly; computation of the elementary functions; extended arithmetic operations such as summation, scalar products, vector norms and polynomial evaluation; and complex SLI arithmetic using the polar representation of complex numbers. All of these work directly with the SLI representations of the various arguments. Some of these are described or

[1] Turbo Pascal is a trademark of Borland International, Inc.

outlined in [34, 35, 36, 37] although many of those descriptions have since been improved upon. We simply summarize some of these features here.

The extended sums and related operations are performed using algorithms which exhibit a natural parallelism, and they reduce the normal linear time-penalty which is expected for serial computation. Some of these ideas re-emerge in the parallel implementations — especially the massively parallel C implementation described in §6.

4.1. Mixed Integer-SLI Arithmetic. One of the benefits of the mixed algorithms lies in the fact that integers are used exactly. (Like any computing environment, integer variables must be stored, represented and operated upon without incurring error.) Apart from the basic arithmetic operations of integer-SLI addition, subtraction, multiplication and division, the integer power function and integer-roots of any order are easily incorporated into the SLI arithmetic framework.

We illustrate this by considering the operation of forming integer powers of SLI variables. This has further application in the evaluation of polynomial functions using a technique similar to that employed for the extended arithmetic operations below.

We thus require z and the associated signs such that

$$Z = \pm\phi(z)^{\pm 1} = \left(\pm\phi(x)^{\pm 1}\right)^N = X^N$$

where N is an integer. The two signs are easily resolved:

$$\begin{aligned} Z < 0 \quad & \text{iff } (X < 0) \text{ and } (N \mod 2 = 1), \\ |Z| \geq 1 \quad & \text{iff } (N = 0) \text{ or } (|X| \geq 1,\ N > 0) \text{ or } (|X| < 1,\ N < 0). \end{aligned}$$

The problem thus reduces to forming positive integer powers of positive quantities greater than unity:

$$\phi(z) = \phi(x)^N$$

with $N > 0$. Taking natural logarithms, we obtain

$$\phi(z-1) = N\phi(x-1)$$

from which it follows that

$$c_1 = \frac{\phi(z-1)}{\phi(x-1)} = N. \tag{29}$$

The algorithm is completed by generating the c-sequence as was done in §2; cf. eqs. (17) and (18).

The modifications for the other mixed operations are mostly even simpler than this. They all share the property that the integer is used exactly rather than being first converted (promoted) to its SLI representation. Observe that this property is *not* shared by other arithmetic systems - except perhaps for this power operation if repeated multiplication is used.

4.2. SLI Algorithms for Elementary Functions. Algorithms for some of the elementary functions are necessarily simpler in SLI than in other arithmetic formats. Others, of course, are more complicated. Among the ones which are simplified, not surprisingly, are the natural logarithmic and exponential functions. By definition,

$$\ln\left(\phi(x)^{\pm 1}\right) = \pm\phi(x-1)$$

and the only complication arises when $x < 2$ so that $\phi(x-1)$ must be converted back from its fixed-point fraction to its SLI representation by forming (perhaps repeated) logarithms of this result. Thus, for example, for $\phi(x) = 2 = \phi(1+\ln 2)$ we get

$$\ln\left(\phi(x)\right) = \ln 2 = \left(\frac{1}{\ln 2}\right)^{-1} = (e^{-\ln\ln 2})^{-1} = \phi(1-\ln\ln 2)^{-1} = \phi(1.3665...)^{-1}.$$

Functions such as absolute value, reciprocation, and negation are very simple bitwise operations on the `slisingle` representation of the variable. Indeed, all of these are implicit in the sign determination process for the basic arithmetic algorithms.

From the integer power operation above, it should be apparent that evaluation of monomial terms is also straightforward. Taking logarithms as in (29),

$$\phi(z) = \phi(y) * \phi(x)^N$$

yields

$$c_1 = \frac{\phi(z-1)}{\phi(x-1)} = N + \frac{\phi(y-1)}{\phi(x-1)} = N + b_1 \tag{30}$$

This can be made into an efficient algorithm for the evaluation of general polynomials by using the extended operations described later.

The other elementary functions which are incorporated into the Turbo Pascal implementation are the basic trigonometric functions. The only ones built into Turbo Pascal itself are `sin`, `cos` and `arctan`. The SLIUNIT is restricted to these same functions. The first two make no direct use of the SLI representation of their arguments, but use instead conversion between floating-point and SLI representations. One reason for this choice is that for sufficiently large arguments, the real interval represented by a specific floating-point or SLI number covers more than a period of the function. There is, therefore, nothing to be gained by trying to evaluate these functions to high accuracy for SLI numbers outside the range of the floating-point system.

By contrast, the arctangent function does have a legitimate domain over the whole real line. By using the identities

$$\arctan|X| = \frac{\pi}{2} - \arctan\frac{1}{|X|}, \tag{31}$$

$$\arctan(-X) = -\arctan(X), \tag{32}$$

the SLI arctangent algorithm is reduced to evaluation of $\arctan(a_0)$ where, as usual, $a_0 = 1/\phi(x)$.

4.3. Complex SLI Arithmetic. Complex SLI arithmetic is incorporated into the Turbo Pascal SLIUNIT using the polar representation

$$Z = Re^{i\theta} = \phi(r)^{\pm 1} e^{i\theta}, \tag{33}$$

where the modulus is represented in standard SLI form and the argument θ is stored as a *fixed-point* fraction in $[-1, 1)$ of π. The two parts can be packed conveniently into the 64-bit integer Turbo Pascal type comp to facilitate their use in "in-line" function and arithmetic calls.

The complex SLI arithmetic algorithms can be achieved without the need to convert to real and imaginary parts, and to make multiple calls to the underlying real algorithms, by using the cosine rule for the appropriate "triangle" in the complex plane. The algorithm for "large" addition proceeds much as for real arithmetic using

$$c_0^2 = 1 + b_0^2 + 2b_0 \cos\theta \tag{34}$$

and then

$$c_1 = 1 + \frac{1}{2} a_0 \ln\left(c_0^2\right) \tag{35}$$

where θ is the angle between the two "position vectors". Of course, since b_0 is computed by evaluating the exponential function, b_0^2 can be computed with no additional difficulty. The computation of the modulus of the result can be completed using the usual SLI algorithm. Modifications for the "mixed" and "small" cases are similar.

Similarly, it turns out that $\Delta\theta$, the difference between the argument of the result and that of the larger operand θ_0, is obtainable from

$$\Delta\theta = \arctan\left(\frac{b_0 \sin\theta_0}{1 + b_0 \cos\theta_0}\right) \tag{36}$$

The derivation of these equations (34), (35) and (36) are given in [37], which also includes an error analysis and observations on the implementation of the special steps which are required by these equations. The important finding is that it remains true that fixed-point internal computation is sufficient, with similar working precisions to those required for the conventional real SLI algorithms.

Of course, the use of the polar form for complex arithmetic has the effect of making complex multiplication equivalent to a single real multiplication for the modulus and a fixed-point addition for the argument. This compares with the usual six real operations for a conventional complex product. There is no compensating additional cost since the algorithm just outlined is also cheaper than two real SLI additions.

4.4. Extended SLI Operations. The potential for parallelism in the SLI extended operation algorithms is discussed in [23], and in [36] in which the algorithm for extended summation is also detailed and analyzed. We content ourselves here with an outline of the algorithm for extended summation and a description of its uses in the Turbo Pascal implementation to compute scalar products and vector norms.

The basic problem of extended SLI summation is to find

$$\pm\phi(z)^{\pm 1} = \sum_{i=0}^{N} \pm\phi(x_i)^{\pm 1}$$

where we shall assume that $X_0 = \pm\phi(x_0)^{\pm 1}$ is the largest magnitude term. (Since the binary representation is consistent with integer-order, identifying this largest argument is a simple operation.) For the (usual) case where $|X_0| \geq 1$, the algorithm can proceed as above after the computation of

$$c_0 = 1 + \sum_{r_i=+1} s_i b_0^{(i)} + \sum_{r_i=-1} s_i a_0 \alpha_0^{(i)}$$

where s_i, r_i represent the sign and reciprocation sign of X_i and $b_0^{(i)} = \phi(x_i)/\phi(x_0)$, $a_0 = 1/\phi(x_0)$, $\alpha_0^{(i)} = 1/\phi(x_i)$. It is apparent that only one a-sequence and one c-sequence are needed, together with either an α- or a b-sequence for each of the other terms. This represents a saving of about 67% of the work that would be needed for the usual serial summation of the same terms. For the case where all terms are "small" there is a similar modification.

The formation of scalar products can now be achieved with a two-stage process. The first is the elementwise multiplication of the two SLI-vectors which is followed by this extended summation. This, like the basic summation operation, is incorporated into the Turbo Pascal implementation in just this way.

Similarly, the usual vector norms could be computed by first forming the appropriate powers of each element, then using this summation algorithm, and finally taking the appropriate "root" of the result. However, this can be (and is) improved further by observing that the largest term will also have the largest p-th power. Thus, the definition of c_0 is adjusted so that

$$c_0 = 1 + \sum (X_i/X_0)^p \,,$$

and then its p-th root is formed within the first logarithm of the c-sequence. The details are not important here. The point is that it illustrates both the versatility of the SLI algorithm for modification to more complicated computation and its suitability to parallel implementation.

5. FORTRAN 90 Implementation

An implementation in Fortran 90 was published in the 1993 Ph. D. thesis [31] of I. Reid. Its purpose coincides, at least in part, with that of the earlier Fortran 77 implementation of D. W. Lozier; cf. §3 of this paper. Both provide for re-interpreting and re-executing Fortran programs in SLI arithmetic through the introduction of a new data type. The later implementation does not support level-index variables and operations, but it uses the efficient, singly recursive algorithms of §2. Efficiency is improved further when an operand is of type `INTEGER` by modifications that use the integer representation directly, as was done in the Turbo Pascal implementation. Special emphasis is placed on simulating the exact bit-precision that is called for in theory in the fixed-point generation of the a-, b-, c- and related sequences, thereby providing for the possibility of justifying the theory experimentally. The thesis includes a test program that was used for this purpose.

As was noted in §3, Fortran 90 includes among its improvements over Fortran 77 the capability to introduce new data types, extend to them standard Fortran operators and functions, and supply for them additional operators and functions. Accordingly, there is no need for an approach using a precompiler.

5.1. Structure. Variables of the new data type SLI occupy two locations of type LOGICAL (for the sign and reciprocation bits), a location of type INTEGER (for the level), and a location of type DOUBLE PRECISION (for the index). This internal organization is of no concern to the user, as the declaration

```
TYPE(SLI) var_1, var_2, ..., var_n
```

suffices to identify variables of the new type.

Operators and functions are provided:

Unary minus -
Arithmetic operators +, -, /, *, ** (with implicit type conversion)
Logical operators ==, /=, >, >=, <, <=
Assignment = (with implicit type conversion)
Absolute value and integer root functions SLIABS and INT_ROOT
Exponential and logarithmic functions SLIEXP and SLILN
Generalized distance GD
Reciprocal and set-reciprocal-bit functions RECIP and RECIP_TRUE

With the exception of the logical operators and the generalized distance, the result of all operators and functions is of type SLI. Arithmetic operators and assignment support implicit type conversion, i.e. an operand can be of type INTEGER, REAL or DOUBLE PRECISION. In all other cases, the operands must be of type SLI. Logical operators are supported only in the new-style Fortran 90 notation; the old-style .EQ., etc., cannot be used with operands of type SLI. The set-reciprocal-bit function forces the reciprocation sign to be $+1$.

The generalized distance was introduced to facilitate comparisons in the test program. A problem with any computer arithmetic system is how to measure the difference between computed results. In an LI system, it is customary to use the norm

$$||X - Y|| = |\psi(X) - \psi(Y)|$$

where the generalized logarithm (4) is extended to the negative real axis by odd symmetry. In an SLI system, we cannot just substitute the SLI-mapping (28) because of the singularity at the origin. The function

$$\mathrm{gd}(X, Y) = |\Psi(X) - \Psi(Y)|$$

is a meaningful measure if and only if X and Y are of like sign. Therefore, Reid introduced the expanded definition

$$\mathrm{gd}(X, Y) = \begin{cases} |\Psi(|X|) - \Psi(|Y|)| & \text{if signs same,} \\ |\Psi(|X|) + \Psi(|Y|) + 14| & \text{otherwise} \end{cases} \tag{37}$$

together with the convention that $\Psi(0) = -7$. The constants 7 and 14 arise because of the maximum level that is allowed in Reid's implementation. In effect, Reid replaces the SLI representation of zero (which would most naturally be defined as zero) with the smallest representable SLI number. The expanded definition allows small numbers of opposite sign to be "close" to each other.

5.2. Testing. The expanded gd (generalized distance) function (37) is used to compare a result computed in SLI arithmetic against a more accurate value computed in higher precision. The 48-bit SLI format allocates 43 bits to the index. Accordingly, with γ_0 denoting "machine epsilon", we have

$$\gamma_0 = 2^{-43}$$

for abbreviated SLI arithmetic. The comparison values are computed in double precision wherever possible, and where not possible in unabbreviated SLI. In either case the machine epsilon γ is the same,

$$\gamma = 2^{-53}$$

for IEEE arithmetic.

The test program applies to the binary operations +, -, *, / and **. That is, a function

$$Z = f(X, Y)$$

is being tested, where

$$X = \Psi(x), \quad Y = \Psi(y), \quad Z = \Psi(z).$$

Now fix attention on a pair of operands x and y. Assume these are stored without error in double precision in the test program. The fractional parts are truncated to 43 bits, and the resulting numbers $\tilde{x}$, $\tilde{y}$ are stored in SLI format. Then the result and generalized distance

$$\tilde{Z} = f\left(\Phi(\tilde{x}), \Phi(\tilde{y})\right), \quad \tilde{d} = \mathrm{gd}(Z, \tilde{Z})$$

are computed. For + and *, $\tilde{d} < \gamma_0$ is used as the acceptance criterion for accurate results.

This criterion is too severe for other operations because their algorithms involve subtraction with the attendant possibility of heavy, significance-losing cancellation. Therefore, the inherent error due to a last-bit perturbation[2] of the operands is estimated by computing

$$d = \frac{\mathrm{gd}(Z, Z_1) + \mathrm{gd}(Z, Z_2)}{2}$$

where

$$Z_1 = f\left(\Phi(x + \gamma_0/2), \Phi(y)\right)), \quad Z_2 = f\left(\Phi(x), \Phi(y + \gamma_0/2)\right)).$$

Then the acceptance criterion becomes $\tilde{d} < d$. These computations are done in double precision or unabbreviated SLI, whichever is appropriate.

We have introduced the "machine epsilons" γ_0 and γ corresponding to SLI arithmetic with 43-bit index and "unabbreviated" SLI arithmetic on an IEEE computer. Linearized perturbation theory has been applied to determine the required machine epsilons for the a-, b-, c- and related sequences. After a review, Reid concludes that the choices

$$\gamma_0 = 2^{-43}, \quad \gamma_1 = 2^{-47}, \quad \gamma_2 = 2^{-53}$$

[2]Although the natural perturbation would appear to be γ_0, Reid uses half this value in his thesis.

are compatible with the theory, where the smallest machine epsilon, γ_2, is used for a-sequences and γ_1 is used for all other sequences. His results of extensive testing support the adequacy of these choices.

But Reid also observes that it would be advantageous in a hardware implementation to be able to reduce the wordlength implied by γ_2. Such a reduction for γ_2 and even for γ_1 might be possible, since the linearized theory does not produce sharp bounds. Accordingly, he repeated the tests with

$$\gamma_2 = \gamma_1, \quad \gamma_1 \in \{2^{-47}, 2^{-46}, 2^{-45}\}.$$

The tests passed the acceptance criteria for the two smaller values of γ_1, but not when the largest value was used.

6. Massively Parallel C Implementation

The most recent implementation of SLI arithmetic is as a part of the Computer Arithmetic Laboratory being developed for the MasPar MP-1 system. This project has been outlined in [1]. The architecture of this machine is a rectangular SIMD array of processors with nearest neighbor connections and toroidal wraparound. The particular system being used has 4096 processors in a 64×64 array. There are two languages available on this system: a variant of Fortran 90 with High Performance Fortran extensions, and a parallel extension of ANSI C. The implementation of SLI arithmetic uses the latter. In this section, we begin with a brief description of the computer system and its suitability for this purpose. This is followed by a description of the SLI arithmetic algorithms used which are further modifications of those employed by the earlier implementations. The modified algorithm possesses a greater degree of natural parallelism, especially for extended operations.

6.1. The MasPar MP-1 System. As is stated above, the system being used is a 64×64 SIMD array. The individual processors are just 4-bit processors so that *all* the built-in arithmetic is implemented in microcode. The power of the system is derived from the massive parallelism that is available for appropriate computations. Like all SIMD architectures, at any point in a program all processors are either performing the same instruction (on their individual data) or are inactive. The advantages of using such a system for implementing experimental arithmetic systems arise out of its flexibility.

For example, the arithmetic can be implemented in serial in such a way that the computation is spread across the processor array. This allows the computation to take advantage of the parallelism to reduce the time-penalty which is otherwise incurred by a software implementation. By implementing floating-point arithmetic in a similar manner, reasonably fair comparisons between the execution times of the two systems can be made.

The other great advantage is that minor adjustments in the algorithms can be implemented with relative ease. This alleviates the need for building experimental hardware until after extensive experimentation has been performed. In a similar manner, the area-speed trade-off can be examined by restricting the amount of parallelism allowed in a particular implementation.

The language being used is MPL (*Massively Parallel Language*) which is a parallel extension of C. The primary augmentation of ANSI C comes form the inclusion of *plural* variables of all the standard and user-defined types; all such variables have

an *instance* on each of the processors in the array. Each processor has its own memory so that the system is a *distributed memory* machine. Each processor is connected to its four nearest neighbors (in the North, East, South and West directions) through the *Xnet* while more distant communication is achieved through the *router* hardware and software.

6.2. SLI Algorithm Modification. The underlying representation of SLI variables in this implementation is essentially similar to that of the Turbo Pascal implementation described in §4. The algorithms used are modifications of those of §2 and we highlight only the variations here. The principal modification is for the basic SLI addition and subtraction algorithm. The b- and β-sequences are replaced by making further use of the α-sequence in a way which makes the parallelism of the algorithm more evident.

Consider again the basic SLI arithmetic operation which we can describe as finding z and its signs such that

$$\pm\,\phi(z)^{\pm 1} = Z = X \pm Y = \pm\phi(x)^{\pm 1} \pm \left(\pm\phi(y)^{\pm 1}\right) \tag{38}$$

where we shall assume for simplicity that $X \geq Y > 0$ so that (38) becomes

$$\phi(z)^{\pm 1} = Z = X \pm Y = \phi(x)^{\pm 1} \pm \phi(y)^{\pm 1}.$$

The modified algorithm reduces to

Algorithm: Modified SLI Addition/Subtraction Algorithm

Input	SLI representations $\phi(x)^{r_X}$, $\phi(y)^{r_Y}$ of $X \geq Y > 0$.
Compute	a-sequence: $a_j = 1/\phi(x-j)$,
	α-sequence: $\alpha_j = 1/\phi(y-j)$,
	if $r_X = r_Y = +1$ then $c_0 = 1 \pm a_0/\alpha_0$,
	if $r_X = -r_Y = +1$ then $c_0 = 1 \pm a_0\alpha_0$,
	if $r_X = r_Y = -1$ then $c_0 - 1 = 1 \pm \alpha_0/a_0$.
	Complete the algorithm exactly as described in §2.
Output	SLI representation $\phi(z)^{r_Z}$ of $Z > 0$.

Once this modification is incorporated, the completion of the algorithm is simplified and the corresponding adjustments to the multiplication, division and other operation algorithms are easily obtained.

One of the important aspects to stress here is the great advantage that this yields for a SIMD parallel algorithm. For example, extended summation where the largest operand $X_0 > 1$ now just requires the (simultaneous) computation of the a-sequences for each operand followed by the extended fixed-point summation $c_0 = 1 + \sum_{i=1}^{N} s_i a_0 \left(\alpha_i\right)^{-r_i}$ where s_i is the sign and r_i denotes the reciprocation sign of X_i, and $\alpha_i = 1/\phi(x_i)$. (The use of $(\alpha_i)^{-r_i}$ is purely a notational convenience; it does not imply that this is an appropriate computational procedure.)

Clearly, the parallel array can be used to implement the simultaneous calculation of all these a-sequences. The extended summation can be implemented using the usual recursive doubling algorithm which is already available in MPL as `reduceAddx` where the `x` is used to denote the appropriate variable type for the result. There are several useful reduction algorithms built into the MPL language, including

`reduceMax` which will be of value in identifying the maximal element in the sum at the beginning of this extended algorithm.

MPL also includes a 64-bit integer type `long long` which can be utilized for this extended sum to avoid any complication in the algorithm to cope with the possibility that c_0 requires several bits for its integer part. Indeed, the underlying arithmetic of this implementation uses another aspect of the Computer Arithmetic laboratory, fixed-point fractions of varying lengths. These are all embedded into variables of type `long long`. The lengths used are measured in hexadecimal digits (called "nibbles") because these are the natural units for the 4-bit processors used by the MasPar architecture.

All the other extended operations described previously can be similarly efficiently implemented using the parallel array.

7. Future Developments

In this paper, we have described the current software implementations of LI and SLI arithmetic. The most recent of these, the parallel implementation in MPL on the MasPar MP-1 system, should also provide a good first step towards an eventual hardware implementation. For SLI as for other alternative computer arithmetic systems, the biggest obstacle is the transition from interesting mathematical idea to practical hardware. In [28] and [33], some possible hardware algorithms were explored. The next developments in the MPL implementation will involve further numerical experimentation in a variety of applications and further development of potential hardware designs.

One of the advantages of this massively parallel array is that it makes it possible to simulate the components of a hardware algorithm, and to experiment with the details of this algorithm without the need to build chips until extensive testing has been conducted. Testing of the underlying hardware algorithm can be performed by simulating CORDIC units for special forms of the exponential and natural logarithm functions. Similarly, the use of carry-save-adder trees, look-up tables and other components can be evaluated through simulation. Such a simulated hardware implementation will allow further experimentation on the working precisions that are needed in order to deliver the required accuracy in final results, extending the testing performed in [31]. By restricting the active set of processors, speed-area trade-offs can also be assessed. All of these and other implementation details will be the subject of continued research in this area.

References

1. M. A. Anuta, D. W. Lozier, and P. R. Turner, *The MasPar MP-1 as a computer arithmetic laboratory*, Tech. Report NISTIR 5569, Nat. Inst. Standards & Tech., Gaithersburg, MD 20899, 1995.
2. M. G. Arnold, T. A. Bailey, J. R. Cowles, and J. J Cupal, *Redundant logarithmic arithmetic*, IEEE Trans. Comput. **39** (1990), 1077–1086.
3. M. G. Arnold, T. A. Bailey, J. R. Cowles, and M. D. Winkel, *Applying features of IEEE 754 to sign/logarithm arithmetic*, IEEE Trans. Comput. 41 (1992), 1040–1050.
4. J. L. Barlow and E. H. Bareiss, *On roundoff error distributions in floating point and logarithmic arithmetic*, Computing **34** (1985), 325–347.
5. R. P. Brent, *A Fortran multiple-precision arithmetic package*, ACM Trans. Math. Software 4 (1978), 57–70, see also [6].

6. ———, *Algorithm 524. MP, a Fortran multiple-precision arithmetic package*, ACM Trans. Math. Software 4 (1978), 71–81, see remark in same journal vol. 5, no. 4, pp. 518–519, 1979.
7. R. P. Brent, J. A. Hooper, and J. M. Yohe, *An AUGMENT interface for Brent's multiple precision arithmetic package*, ACM Trans. Math. Software **6** (1980), 146–149.
8. C. W. Clenshaw, D. W. Lozier, F. W. J. Olver, and P. R. Turner, *Generalized exponential and logarithmic functions*, Comput. Math. Appl. **12B** (1986), 1091–1101.
9. C. W. Clenshaw and F. W. J. Olver, *Beyond floating point*, J. Assoc. Comput. Mach. **31** (1984), 319–328.
10. ———, *Level-index arithmetic operations*, SIAM J. Numer. Anal. **24** (1987), 470–485.
11. C. W. Clenshaw, F. W. J. Olver, and P. R. Turner, *Level-index arithmetic: An introductory survey*, Numerical Analysis and Parallel Processing, Lecture Notes in Mathematics 1397 (P. R. Turner, ed.), Springer-Verlag, 1989, pp. 95–168.
12. C. W. Clenshaw and P. R. Turner, *The symmetric level-index system*, IMA J. Numer. Anal. 8 (1988), 517–526.
13. ———, *Root squaring using level-index arithmetic*, Computing **43** (1989), 171–185.
14. F. D. Crary, *A versatile precompiler for nonstandard arithmetics*, ACM Trans. Math. Software **5** (1979), 204–217.
15. H. Hamada, *URR: Universal representation of real numbers*, New Gener. Comput. 1 (1983), 205–209.
16. ———, *A new real number representation*, Proc. 8th Symposium on Computer Arithmetic, IEEE Computer Society Press, 1987, pp. 153–157.
17. T. E. Hull, M. S. Cohen, and C. B. Hall, *Specifications for a variable precision arithmetic coprocessor*, Proc. 10th Symposium on Computer Arithmetic, IEEE Computer Society Press, 1991, pp. 127–131.
18. D. M. Lewis, *An architecture for addition and subtraction of long word length numbers in the logarithmic number system*, IEEE Trans. Comput. **39** (1990), 1325–1336.
19. ———, *An accurate lns arithmetic unit using interleaved memory function interpolator*, In Swartzlander Jr. et al. [32], pp. 2–9.
20. D. M. Lewis and L. K. Yu, *Algorithm design for a 30-bit integrated logarithmic processor*, Proc. 9th Symposium on Computer Arithmetic, IEEE Computer Society Press, 1989, pp. 192–199.
21. D. W. Lozier, *An underflow-induced graphics failure solved by sli arithmetic*, In Swartzlander Jr. et al. [32], pp. 10–17.
22. D. W. Lozier and F. W. J. Olver, *Closure and precision in level-index arithmetic*, SIAM J. Numer. Anal. **27** (1990), 1295–1304.
23. D. W. Lozier and P. R. Turner, *Robust parallel computation in floating-point and sli arithmetic*, Computing 48 (1992), 239–257.
24. ———, *Symmetric level index arithmetic in simulation and modeling*, J. Res. Nat. Inst. Standards & Tech. **97** (1992), 471–485.
25. S. Matsui and M. Iri, *An overflow/underflow free floating-point representation of numbers*, J. Inform. Process. 4 (1981), 123–133.
26. R. Morris, *Tapered floating point: A new floating-point representation*, IEEE Trans. Comput. **20** (1971), 1578–1579.
27. F. W. J. Olver, *Rounding errors in algebraic processes - in level-index arithmetic*, Reliable Numerical Computation (M. G. Cox and S. Hammarling, eds.), Oxford University Press, 1990, pp. 197–205.
28. F. W. J. Olver and P. R. Turner, *Implementation of level-index arithmetic using partial table look-up*, Proc. 8th Symposium on Computer Arithmetic, IEEE Computer Society Press, 1987, pp. 144–147.
29. R. G. Rehm, H. R. Baum, H. C. Tang, and D. W. Lozier, *Finite-rate diffusion-controlled reaction in a vortex: A report*, Tech. Report NISTIR 4768, Nat. Inst. Standards & Tech., Gaithersburg, MD 20899, 1992.
30. Ronald G. Rehm, Howard R. Baum, Hai C. Tang, and Daniel W. Lozier, *Finite-rate diffusion-controlled reaction in a vortex*, Combust. Sci. and Tech. **91** (1993), 143–161.
31. I. Reid, *Symmetric level index arithmetic: Towards its integration into the scientific computing environment*, Ph.D. thesis, Lancaster University, Lancaster, U. K., April 1993.
32. E. Swartzlander Jr., M. J. Irwin, and G. Jullien (eds.), *Proceedings, 11th symposium on computer arithmetic*, IEEE Computer Society Press, 1993.

33. P. R. Turner, *Towards a fast implementation of level-index arithmetic*, Bull. Inst. Math. Appl. **22** (1986), 188–191.
34. ———, *A software implementation of sli arithmetic*, Proc. 9th Symposium on Computer Arithmetic, IEEE Computer Society Press, 1989, pp. 18–24.
35. ———, *Algorithms for the elementary functions in level-index arithmetic*, Scientific Software and Systems (M. G. Cox and J. C. Mason, eds.), Chapman & Hall, 1990, pp. 123–134.
36. ———, *Implementation and analysis of extended sli operations*, Proc. 10th Symposium on Computer Arithmetic, IEEE Computer Society Press, 1991, pp. 118–126.
37. ———, *Complex sli arithmetic: Representation, algorithms and analysis*, In Swartzlander Jr. et al. [32], pp. 18–25.
38. W. T. Wyatt Jr., D. W. Lozier, and D. J. Orser, *A portable extended precision arithmetic package and library with Fortran precompiler*, ACM Trans. Math. Software **2** (1976), 209–231.
39. H. Yokoo, *Overflow/underflow-free floating-point number representation with self-delimiting variable length exponent field*, Proc. 10th Symposium on Computer Arithmetic, IEEE Computer Society Press, 1991, pp. 110–117.

Applied and Computational Mathematics Division, National Institute of Standards and Technology, Gaithersburg, MD 20899
E-mail address: dlozier@nist.gov

Mathematics Department, U. S. Naval Academy, Annapolis, MD 21402
E-mail address: prt@sma.usna.navy.mil