Implementation and Analysis of Extended SLI Operations

Peter R Turner

Mathematics Department, US Naval Academy, Annapolis, MD 21402

Abstract

This paper is concerned with extended arithmetic operations, such as forming scalar products, in symmetric level index, SLI, arithmetic. Schemes for the implementation of such algorithms are described and analysed both in terms of comparative timings for these operations and their floating-point counterparts and in terms of the control of errors in the computation. It is seen that with sufficient parallelism available in the SLI processor the computation can be as fast as for floating-point operations. Also we see that the SLI operation can be modified to produce just a single rounding error from extended operations very economically.

1 Introduction

The level-index number system for computer arithmetic was first suggested in Clenshaw and Olver [1],[2]. The scheme was extended to the symmetric level index, SLI, representation in [4] and has been studied in several further papers in the last few years. Much of the earlier work is summarized in the introductory survey [3]. The primary virtue of SLI arithmetic is its freedom from overflow and underflow and the consequent ease of algorithm development available to the scientific software designer. This is not the only arithmetic system that has been proposed with this aim; for example the work of Matsui and Iri [10] and Hamada [6] suggested modified floating-point systems which share some of the properties of level-index.

Possible hardware implementations of SLI arithmetic were discussed in [13] and [14] while a software implementation incorporating some extended arithmetic was described in [16]. The error analysis of SLI arithmetic is discussed in [2] and [4] and is extended in [8], [11] and [12]. Applications and software engineering aspects of the level-index system have been discussed in [5],[9] and [15].

In this paper, we concentrate on the extended arithmetic operations, such as forming scalar products of two vectors, which are of fundamental importance in scientific computing. At the heart of any such operation is the extended addition of a finite series of terms. This operation was studied in some detail for interval arithmetic [7] and a greatly extended accumulator was designed to return the result with just a single roundoff error. In this paper we consider the implementation of extended summation algorithms for the SLI system and their analysis. We see that the time penalty associated with SLI arithmetic is reduced so sharply that, with sufficient parallelism available in the SLI processor, the summation could even be faster than conventional floating-point arithmetic. Moreover the analysis demonstrates that the objective of a single rounding error for the extended summation can be achieved at very modest cost of increasing the precision of the internal computation very slightly.

We begin with a brief review of the SLI representation and its arithmetic. In Section 2, we describe the algorithms for extended summation and their implementation. Section 3 is concerned with the timings of these algorithms and their comparison with floating-point systems. In Section 4, we present the error analysis of these operations and its implications for the control of the overall error.

The symmetric level index representation of a real number X is given by

$$X = s_X \phi(x)^{r_X} \tag{1}$$

where the two signs s_X and r_X are ± 1 and the *generalized exponential* function is defined for $x \geq 0$ by

$$\phi(x) = \begin{cases} x & 0 \leq x < 1, \\ \exp(\phi(x-1)) & x > 1. \end{cases} \tag{2}$$

It follows that for $X>1$,

$$X = \exp(\exp(\ldots(\exp f)\ldots)) \tag{3}$$

where the exponentiation is performed $l=[x]$ times and $x=l+f$. The integer part, l of x is called the *level* and the

CH3015-5/91/0000/0118$01.00 © 1991 IEEE

fractional part, f is called the *index*. The freedom of this system from over- and underflow results from the fact that, working to a precision of no more than 5,500,000 decimal places in the index, the system is closed under the four basic arithmetic operations apart from division by zero. This is discussed briefly in [1], [4] and considered in some detail in [8].

The appropriate error measure for computation in the level index system is no longer relative error (which corresponds approximately to absolute precision in the mantissa of floating-point numbers) but *generalized precision* which corresponds to absolute precision in the index. This error measure is introduced in [1].

In order to study the extended SLI operations which are at the heart of the present work, it is desirable to remind the reader of the fundamentals of the basic arithmetic algorithms for the SLI system. The properties of the natural logarithm function reduce all such operations to addition or subtraction. The algorithms are described in some detail in [4].

The basic problem is that of finding the SLI representation $s_Z\phi(z)^{r_Z}$ of $Z=X\pm Y$ where X, Y are also given by their SLI representations. Without loss of generality, we may assume that $X\geq Y>0$ so that $s_Z=+1$. The computation entails the calculation of the members of three short sequences which vary according to the particular circumstances. In every case, the sequence defined by

$$a_j = \frac{1}{\phi(x-j)} \qquad (j = l-1, l-2, \ldots, 0) \tag{4}$$

where $l=[x]$, is computed using the recurrence relation

$$a_{j-1} = \exp(-1/a_j); \qquad a_{l-1} = e^{-f}. \tag{5}$$

Depending on the values of r_X, r_Y and r_Z, the other sequences that may be required aregiven, for appropriate starting values, by

$$b_j = \frac{\phi(y-j)}{\phi(x-j)}, \quad \beta_j = \frac{\phi(x-j)}{\phi(y-j)}, \quad \alpha_j = \frac{1}{\phi(y-j)} \tag{6}$$
$$c_j = \frac{\phi(z-j)}{\phi(x-j)}, \quad h_j = \phi(z-j).$$

The ranges of their values are always suitably bounded and the terms are computed according to recurrence relations similar to (5) using exponentials and logarithms of special arguments. See [4] for full details of the algorithms and [13], [14] for possible schemes for hardware implementation. The particular implementation of extended operations presented in Section 2 is based on the modified CORDIC approach of [14].

2 Extended Algorithms and their Implementation

In this section, we first describe the mathematical algorithm for extended summation and then consider how this algorithm may be implemented in an efficient manner on a machine with a high degree of parallelism available in its (hypothetical) SLI processor. The operation of summing the elements of a sequence or the components of a vector plays a central role in much scientific computing especially in the formation of scalar products and in quadrature.

The specific task then is to find the SLI representation $s_Z\phi(z)^{r_Z}$ of $Z = \sum_{i=0}^{N} X_i$ where each component X_i is given by its SLI representation. Thus we seek s_Z, z, r_Z such that

$$s_z\phi(z)^{r_z} = \sum_{i=0}^{N} s_i\phi(x_i)^{r_i} \tag{7}$$

where we shall assume that $X_0>0$ is the largest component in the sum. (Of course, our algorithm must identify this maximum element and the time for this operation must be included in our timing comparisons in the next section.)

In all cases, the computation begins with the calculation of the terms of the sequence

$$a_j = \frac{1}{\phi(x_0-j)}$$

using the recurrence relation (5). The rest of the algorithm is divided into two cases depending on whether $X_0\geq 1$. If $X_0\geq 1$, then the b-sequence is computed for each term X_i ($i\geq 1$) for which $r_i=+1$ to yield $b_{i,0}$. The α-sequence is computed for terms with $r_i=-1$ and the product $a_0\alpha_{i,0}$ is formed. The important observation here is that, given sufficient parallelism in the processor, all of these sequences may be computed simultaneously.

The results of these calculations can be combined to yield

$$c_0 = 1 + \sum_{r_i=+1} s_i b_{i,0} + \sum_{r_i=-1} s_i a_0 \alpha_{i,0} \tag{8}$$

from which the remaining terms of the c-sequence may be computed as usual, together with any of the h_j that may be necessary. For the case where $X_0<1$, in which case all terms in the sum are "small" similar computation with the β-sequences is used to obtain

$$c_0' = \frac{1}{c_0} = 1 + \sum_{i=1}^{N} s_i \beta_{i,0}. \tag{9}$$

The recurrence relations for the c- and h-sequences are

included in the following description of the complete algorithm. Those for the b- and β-sequences are included in the discussion of possible hardware implementations of the algorithm. The detailed derivations of these sequences can be found in [7] and [9].

Algorithm 1 Extended SLI summation

Input: $s_i\phi(x_i)^{r_i}$ $(i=0,1,...,N)$

with $s_0=1$ and $\phi(x_0)^{r_0} > \phi(x_i)^{r_i}$ $(i \geq 1)$

$x_0 = [x_0] + f_0$

Compute: $a_j = 1/\phi(x_0-j)$ $(j=[x]-1,...,0)$ using (4)

If $r_0=+1$ (*Case 1*) **then**

for i=1 to N

if $r_i=+1$ compute $b_{i,0} = \phi(x_i)/\phi(x_0)$

if $r_i=-1$ compute $a_0\alpha_{i,0} = a_0/\phi(x_i) = 1/\phi(x_0)\phi(x_i)$

$r_Z = +1$; $c_0 = 1 + \sum_{r_i=+1} s_i b_{i,0} + \sum_{r_i=-1} s_i a_0 \alpha_{i,0}$

If $r_0=-1$ (*Case 2*) **then**

for i=1 to N compute $\beta_{i,0} = \phi(x_0)/\phi(x_i)$

$r_Z = -1$; $c = 1 + \sum_{i=1}^{N} s_i \beta_{i,0}$

If $c=0$ **then** $Z=0$; return

else $s_Z = \mathrm{sgn}(c)$; $c = |c|$, $l_Z = 1$

Case 1: if $c<a_0$ **then** $r_Z = -1$; $h = -\ln(c/a_0)$

compute h-sequence:

repeat

$h=\ln h$; $l_Z = l_Z+1$

until $h<1$

$z = l_Z + h$; return

Case 2: if $ca_0>1$ **then** $r_Z = +1$; $h = \ln ca_0$

compute h-sequence;

$z = l_Z + h$; return

If $[x]=1$ **then** $h = f_0 + r_0 \ln c$;

compute h-sequence;

$z = l_Z + h$; return

$c = 1 + r_0 a_1 \ln c$

For i=1 to $[x]$-2

if $c<a_i$ **then** $z = l_Z + c/a_i$; return

else $l_Z = l_Z + 1$; $c = 1 + a_{l_Z} \ln c$

If $c_{[x]-1}<a_{[x]-1}$ **then** $z = l_Z + c/a_{[x]-1}$; return

else $l_Z=[x]$; $h = f_0 + \ln c$

compute h-sequence

$z = l_Z + h$; return.

This is the algorithm which is implemented in the software implementation of SLI arithmetic described in [16] although it was convenient there to implement separate c-sequences for the two cases. Our primary concern here though is with potential hardware implementations of the SLI system including the extended summation operation.

The implementation described here is an extension of the ideas of [14] using modified CORDIC algorithms for the computation of the various sequences.

The preliminary step is the determination of the maximal element in the sum. We recall from [16] that the electronic representation of the SLI system is such that the order relations are precisely those of the same bit-strings compared as 2's complement integers.

It follows that this preliminary step can be achieved with a simple tree of "integer-discriminators". This tree will have $\log_2(N+1)$ stages assuming sufficient parallelism that at least $(N+1)/2$ such discriminators can operate simultaneously.

It is clearly important here that an efficient integer discrimination algorithm is used here. The comparison of integers is essentially the same operation as the determination of the sign of a "double number" which was a critical operation in the modified CORDIC approach to SLI arithmetic outlined in [14]. We digress briefly to consider this operation and to present an efficient low-level algorithm for integer comparison.

Consider then the operation of determining which is the larger of two 2's complement integers p and q. The algorithm proceeds as though both representations were standard binary integers and then reverses the outcome if the sign bit of p is a 1. It suffices therefore that our algorithm determines the larger of two aligned **positive** binary numbers of $n+1$ bits.

The first stage of the algorithm replaces each pair of bits (p_i,q_i) with the pair (d_i,p_i) $(i=0,1,...,n)$ where

$$d_i = p_i \text{ XOR } q_i = p_i + q_i \bmod 2 = \begin{cases} 1 & p_i \neq q_i \\ 0 & p_i = q_i \end{cases} \quad (10)$$

Now if $d_i=1$ and $d_j=0$ for $j<i$ then $p>q$ if $p_i=1$ while $p<q$ if $p_i=0$. (If there is no $d_i=1$ then $p=q$.) The task is therefore to find the first such d_i and the corresponding p_i. This is achieved through a tree of single bit tests as follows.

For simplicity, we assume here that $n=2^k-1$ so that the word length is a power of 2. (Clearly any shorter words can be extended with 0's.) The second phase is a k-stage process in each stage of which we consider successive

pairs (d_{2i},p_{2i}) and (d_{2i+1},p_{2i+1}). The first of these is retained if $d_{2i}=1$ while the second is selected otherwise. These single-bit tests can be effected simultaneously for each pair. The number of such pairs is thus halved at each step so that, after k such steps, there is only one pair (D,P) say remaining. If $D=0$ then $p=q$, whereas if $D=1$, then $p>q$ if and only if $P=1$. For 2's complement representations, the final decision is reversed if $p_0=1$.

The algorithm is summarized for positive numbers below. Clearly the arguments could be any aligned representations though we describe the process for integers.

Algorithm 2 Fast discriminator

Input: Positive binary integers p, q of 2^k bits.

Initialize:

For $i=0$ to 2^k-1 set $(d_i^{(0)},p_i^{(0)}) = (p_i \vee q_i, p_i)$ (Parallel)

Loop: For $j=1$ to k (Serial)

for $i=0$ to $2^{k-j}-1$ (Parallel)

$$(d_i^{(j+1)},p_i^{(j+1)}) = \begin{cases} (d_{2i}^{(j)},p_{2i}^{(j)}) & \text{if } d_{2i}^{(j)}=1 \\ (d_{2i+1}^{(j)},p_{2i+1}^{(j)}) & \text{if } d_{2i}^{(j)}=0 \end{cases}$$

Output: **If** $d_0^{(k)}=0$ **then** $p=q$

else $p>q \Leftrightarrow p_0^{(k)}=1$

The process is illustrated in the following examples.

Example 1

We consider two examples using 8-bit words.

	(a)		(b)	
	p = 1 1 0 0 1 0 1 0		p = 1 0 1 0 1 0 1 0	
	q = 1 0 1 1 1 1 1 0		q = 1 0 1 1 1 1 1 0	
Initialization:				
	d = 0 **1 1** 1 0 1 0 **0**		d = 0 **0 0** 1 0 1 0 **0**	
	p = 1 **1 0** 0 1 0 1 **0**		p = 1 **0** 1 **0** 1 **0** 1 **0**	
j=1	\/ \/ \ /\ /		\/ \ / \/ \/	
	d = **1** 1 **1** 0		d = 0 **1** **1** 0	
	p = **1** 0 **0** 0		p = 0 **0** **0** 0	
j=2	\ / \ /		\ / \ /	
	d = **1** **1**		d = **1** **1**	
	p = **1** 0		p = **0** 0	
j=3	\ /		\ /	
	d = **1**		d = **1**	
	p = **1**		p = **0**	
Output:	$p > q$		$p < q$	

In both cases, $p_0=1$ and so the conclusion would be reversed if these were 2's complement integers. Note how the first (i.e left most) pair in $(d^{(0)},p^{(0)})$ for which $d_i^{(0)}=1$ is preserved throughout the process to yield the required result. If $d_{2i}^{(j)}=d_{2i+1}^{(j)}=0$ then there is no difference between p and q in the appropriate positions. The fact that it is the second pair which are transmitted to the next phase in this situation is an arbitrary decision - but this is choice which allows the simple one-bit tests to be performed in the (parallel) inner loop.

It should be observed that this integer discriminator could easily be incorporated into floating-point hardware too.

The motivation for this discriminator in the present context was the need to detect the sign of a "double number" quickly. That is we have a quantity x represented by the sum of two aligned 2's complement words p and q; we wish to identify the sign of x. This can be achieved using the same idea. If the sign bits of the two components are the same then clearly x has this same sign. Otherwise we must identify the value of the first bit of p for which the corresponding bit of q is the **same**. This is achieved by simply reversing the test in the inner loop in Algorithm 1. If no such bit exists then p and q are 1's complements and so their sum is negative; otherwise x is positive if and only if the appropriate $p_i=1$.

It follows that the determination of the sign of a double number can be achieved in $\lceil \log_2 n \rceil$ one-bit discrimination steps for a wordlength of n bits.

Returning to the main objective of this section, we are now in position to start Algorithm 1. The a-sequence for this largest component can be computed exactly as in [14] which is based on the CORDIC-like algorithm for $e^{-1/x}$ defined by

$$\begin{aligned} &v_1 = 1,\ u_1 = K \\ &\delta_k = -\mathrm{sgn}(v_k) \\ &u_{k+1} = u_k + u_k \delta_k 2^{-k} \\ &v_{k+1} = v_k + \delta_k \xi_k \end{aligned} \tag{11}$$

where

$$\xi_k = x\varepsilon_k,\ \varepsilon_k = \tanh^{-1}2^{-k}, \quad K = \prod \cosh \varepsilon_k \tag{12}$$

with the usual repetitions in the sequence ε_k and the product K. Two additional steps using $\varepsilon_1=\tanh^{-1}1/2$ are also included in order to extend the range of convergence of the algorithm. Of course, the same algorithm is used to generate any α-sequences which are needed in Case 1.

The b- and β-sequences are computed using similar relations:

$$\begin{aligned} b_j &= \exp\left((b_{j+1}-1)/a_{j+1}\right) \\ \beta_j &= \exp\left((\beta_{j+1}-1)/a_{j+1}\beta_{j+1}\right) \end{aligned} \tag{13}$$

with appropriate starting values. Minor modifications of

the above algorithm can be used to compute these terms.

One important observation here is that all of these sequences can be computed simultaneously which has an obviously beneficial effect on the speed of extended summation. We shall also see later that compressing the operation in this manner reduces the overall rounding error inherent in the operation to a significant extent.

The next stage is the accumulation of c_0. This is the result of summation of 1 and N fixed-point fractions. (Recall that all the internal arithmetic required for the SLI system can be performed to **fixed** absolute precisions.)

One aspect of the implementation proposed in [14] was the use of a "double-number" format throughout the internal arithmetic. This facility is retained by the extended operation so that the formation of c_0 is reduced to the summation of $2N+1$ fixed point fractions using a tree of Carry Save Adders to obtain the double-number representation of c_0.

The remainder of Algorithm 1 can be implemented just as it would be for a single SLI arithmetic operation. The terms of the c-sequence can again be computed using a modified CORDIC algorithm for the function $a \ln c$ in which the constants used are

$$\alpha_k = a\,\varepsilon_k = a \tanh^{-1} 2^{-k}. \qquad (14)$$

These, like the ξ_k of (12), can be computed in parallel. The standard CORDIC logarithm algorithm can be used for any terms of the h-sequence which are needed. The reader is referred to [14] for a full description of these modified algorithms in which again "double number" representations are used throughout the internal computation.

3. Timing Estimates and Comparisons

We begin this section with estimated timings for the extended SLI summation algorithm. We shall also make comparisons of these estimates with serial and parallel floating-point implementations. Throughout this section we shall assume that the prospective SLI hardware unit has "sufficient" parallelism that any operations which can, in principle, be performed simultaneously, can be so performed in practice.

We list below our notation and assumptions regarding the relative timings of the underlying basic operations measured in some base "time unit", t.u. The basis of our comparisons will be single precision SLI against single precision floating-point operations, which we shall assume take approximately 2c t.u.

Operation	Time in t.u.s
3 to 2 Carry Save Adder operation	a
Shift left or right	$b \approx a/3$
32-bit Carry propagate adder	$c \approx 32a$
32-bit aligned discrimination	$d \approx 5e \approx a$
One-bit logical test or negation	$e \approx a/5$

On a serial machine therefore, the floating-point summation of the terms $X_0,\ldots,X_N$ will require approximately $2Nc$ t.u. For a machine with sufficient parallelism to perform this summation using a typical reduction algorithm, the number of separate reduction stages required is $\lceil \log_2 N+1 \rceil$ so that the overall timing is then approximately $2c \log_2 N+1$. These are the two quantities with which we make our comparisons later.

We note that the time taken for the SLI operation will vary according to the levels of the maximum element and the final result. As in [14] we shall make a "worst case" estimate in which both of these quantities are at level 5 and a "typical" estimate using level 3. It should be noted at the outset that the worst case comparison is especially unfair to the SLI system since the **corresponding floating-point computation could NOT be performed** by any straightforward means.

With the use of the algorithm of the previous section for the signs of the double number representations (and a slightly improved "squarer") it follows that the timings given in [14] for single SLI operations are overestimated. If we also modify the computation of the starting value for the b-sequence so that this sequence is computed entirely in parallel with the a-sequence, the "typical" and "worst case" times quoted there can be reduced to 30c and 49c t.u. respectively.

This represents a slowdown by a factor of about 15 compared to the same algorithm executed in floating-point on a fast serial machine. Much of this loss may well be recovered by the fact SLI arithmetic allows much simpler program structures to be used since it does not need protecting from possible overflow or underflow problems.

The principal finding of this section is that by using the extended Algorithm 1, this loss of speed is significantly reduced or even removed.

The precomputation of the largest component requires $\log_2(N+1)$ steps each of which is an integer discriminator implemented according to Algorithm 2 in 5e t.u.

The remainder of the algorithm splits into two principal cases depending on whether $X_0 \geq 1$. We shall concentrate first on the "large" case, that is $X_0 \geq 1$ or, equivalently, $r_0 = +1$. We shall subdivide this case further but the computation proceeds the same way at least as far as the computation of c_0.

The computation of the a-sequence for X_0 is common to all cases and is achieved as in [14] in a total of

78a+39b+39d + ([x]-1)(132a+50b+41d) t.u.

It was observed in [14] that the b-sequence can be computed in parallel with the a-sequence provided that the starting value is defined appropriately. The less severe precision requirement for terms of this sequence also implies that it can be computed alongside the a-sequence at no extra cost in time.

For terms of magnitude less than 1, the α-sequence is required. The computation is just the same as for the a-sequence and so can also be computed in the same time. For each such term we then need the multiplication of the two representations of a_0 and α_0. These are both 37-bit double numbers and so 148 terms need to be summed with a CSA-tree. This multiplication requires 12a+b t.u.

It follows that the total time required to produce all the components of the sum which forms c_0 is

90a+40b+39d + (l_{max}-1)(132a+50b+41d) t.u.

where l_{max} is the maximum level of any component of the sum. (This could be greater than [x] if very small quantities with reciprocals greater than $|X_0|$ are present.)

The next stage is the summation of N double numbers and 1. This can be achieved with another CSA-tree in approximately $\log_{3/2}(2N+1) \approx 1.7(1+\log_2 N)$ steps. This requires $1.7(1+\log_2 N)$a t.u.

There are now three possibilities: namely $|Z| \geq |X_0|$, $1 \leq |Z| < |X_0|$ or the "flip-down" case where $|Z| < 1$ for which the final result must be represented in reciprocal form. In each of the first two of these cases, the remaining parts of the algorithm are performed as in the proposed implementation in [14]. The only difference of any significance is the possibility that a longer h-sequence is needed. In ordinary SLI addition at most one term of this sequence is needed. Here there is a possibility of needing up to 4 such terms if the series is **very** long and all terms are close to unity. However the total length of the c-sequence and h-sequence will still not exceed 5 for nontrivial summation. Since the terms of the h-sequence are computed by a straightforward logarithm routine, which is quicker than the modified CORDIC algorithm for the c-sequence, we obtain an upper bound for the time by taking each step (except possibly the final one) to be a computation of the form $1 + a \ln c$. The time obtained for this operation in [14] is 83a+37b+34d t.u. per step. The final step in the case where $[z] \geq [x_0]$ is necessarily just a logarithm (of a double number) which can be achieved in 70a+35b+34d t.u. Finally, we must convert the final result to "single" form using the Carry Propagate (or Carry Look-Ahead) adder.

For the "flip-down"case where $|X_0| \geq 1 > |Z|$, the computation of a c-sequence is replaced with an h-sequence which as we have already observed is a quicker calculation.

Summarizing, in the case of "large" extended summation the total time for the "worst case" where $[x_0]=[z]=5$ is given by

Find X_0	$\log_2(N+1)$*5e
a-,b-,α-sequences	90a+40b+39d + 4(132a+50b+41d)
c_0	$1.7(1+\log_2 N)$a
c-,h-sequences	70a+35b+34d + 4(83a+37b+34d)
Final CPA	c
TOTAL	$1022a+423b+c+373d+\lceil\log_2(N+1)\rceil(5e+1.7a)$
	$\approx 49c + 2.7a\lceil\log_2(N+1)\rceil$
	$\leq$ 50c t.u for $N \leq 2^{11}$.

We see immediately that this operation can be expected to be faster than the floating-point calculation would be for $N \geq 25$ on a serial machine - but of course the floating-point computation would be impossible.

For a more typical estimate, we consider the case where no component of the sum or the result is at a level exceeding 3. The corresponding total is now $592a+249b+c+223d+2.7a\lceil\log_2(N+1)\rceil \leq 30c$ t.u. for the same range of N. In this case the extended summation - even of 2000 terms - compares favorably with the serial floating-point summation of just 15 terms.

We see that the fact that the summation is compressed into the formation of c_0 has the effect of making the extended aspect of the operation virtually free. This statement remains valid for the case of the extended summation of small quantities. The only important difference from the point of view of timing the operation is the replacement of the b-sequence with the β-sequence. Comparing the two equations in (13), we see that the essential difference is the extra multiplication $a_{j+1}\beta_{j+1}$.

This operation entails forming the product of two double numbers of which β has the shorter wordlength. We find that we must sum 128 terms; using a CSA tree this is

reduced to just two in 11a+b t.u. This together with the standard b-sequence calculation still implies that the computation of the next term of the β-sequence is quicker than that of the a-sequence. It follows that the bound on the time taken to produce c_0 for the large case remains valid here.

The final stages of the "Case 2" algorithm are identical with those of the large case above. The only time loss is derived from the slightly slower test of $ca_0>1$ rather than $c<a_0$. This entails a further "double × double" multiplication which adds 11a+b t.u. to the overall time. The approximate timings quoted above remain valid.

The comparisons made above were not entirely fair since we assumed considerable parallelism in the SLI computation but only a serial architecture for the floating-point summation. Of course this is not entirely unfair either as it points to a possible trade-off to be considered in deciding how faster technology is best utilized.

We conclude this section with a comparison of projected times for our parallel SLI implementation with floating-point computation using similar parallelism. The summation of $N+1$ terms on a sufficiently parallel floating-point architecture will require $\lceil \log_2(N+1) \rceil$ steps of a typical reduction algorithm which takes, under our assumptions, $2c\lceil \log_2(N+1) \rceil$ t.u. For the "typical" range calculation, the SLI summation takes $29c + 2.7a\lceil \log_2(N+1) \rceil$ t.u. compared with $49c + 1.7a\lceil \log_2(N+1) \rceil$ t.u. for the "worst case". In Table 1, below, we show the relative timings for different vector lengths. Again recall that floating-point would fail for the worst case computation.

Table 1 *Comparison of extended summation times for floating-point and SLI arithmetic on parallel architectures*

# terms	Flp time	"Typical" SLI time	SLI:flp	"Worst case" SLI time
16	8c	30c	4:1	50c
64	12c	30c	2.5:1	50c
256	16c	30c	2:1	50c
1024	20c	30c	1.5:1	50c
4096	24c	30c	1.25:1	50c

It is clear from the table that for even moderate length sums, the time-penalty incurred by the more robust SLI arithmetic is very small while for massively parallel machines it may be almost undetectable.

From the entries in Table 1, we can also deduce relative timings for scalar products of two vectors. On our "sufficiently parallel" machine, the floating-point calculation requires just one more parallel multiplication increasing each of the quoted times by 2c t.u. The corresponding SLI multiplications can also be performed simultaneously - but more slowly.

Since the multiplication of two SLI quantities is equivalent to the addition of their logarithms the effective levels of all quantities are reduced by one. It follows that the "worst case" and "typical" timings are reduced to 40c and 20c t.u. respectively. The corresponding overall times for the scalar product are therefore 90c and 50c t.u. The slowdown ratios for scalar products of vectors of the same lengths as in Table 1 thus vary from 5:1 down to 2:1.

4. Error Analysis and Control

In much the same way as for the consideration of timings, in order to obtain error bounds for these extended operations, we must consider separate cases. We shall again treat one of the main cases in some detail and then describe the differences and summarize the results for the others. In the spirit of [4], the aim of the analysis is to demonstrate that the rounding errors involved in Algorithm 1 can be restricted to the order of the inherent error in the operation. As in [4], we find that working to fixed absolute precisions in the various stages of the internal computation achieves the desired control. We begin by studying the case of "large" arithmetic with a "large" result; that is, $|X_0|, |Z| \geq 1$. Note here that we do **NOT** assume that all terms are of the same sign, nor even that the signs of X_0 and Z are the same.

In this case, the inherent error can be approximated by linear perturbation theory to obtain

$$|\delta z| \leq \left(\left| \frac{\phi'(x_0)}{\phi'(z)} \right| + \sum_{r_i=+1} \left| \frac{\phi'(x_i)}{\phi'(z)} \right| + \sum_{r_i=-1} \left| \frac{\phi'(x_i)}{\phi^2(x_i)\phi'(z)} \right| \right) \gamma_0 \qquad (15)$$

where each component is correct to the accuracy, γ_0, of the representation, so that $|\delta x_i| \leq \gamma_0$. Also using the facts that ϕ' is an increasing function, $\phi^2(x)/\phi'(x) = \phi(x)/\phi'(x-1)$ and $|\phi(x)/\phi'(x-1)| \geq 1$ for $x \geq 1$ (see Lemma 3.1 of [4]) it follows for the case where $z \geq x_0$ that this inherent error is bounded by $(N+1)\gamma_0$. If, on the other hand, $z<x_0$, then the appropriate bound is $(N+1)\gamma_0\phi'(x_0)/\phi'(z)$. These are the bounds which our algorithm should be designed to achieve by choosing appropriate working precisions.

Many of the details are similar to the analysis used in [4]. The error bounds for the computation of individual a-, b-

or α-sequences are unchanged for the extended algorithm from those obtained in [2]. Thus, we have

$$|\delta a_j|,\ |\delta \alpha_j| \le \lambda\gamma_1 \qquad (16)$$
$$|\delta b_{i,0}| \le \phi'(x-1)\rho(\gamma_2 + \lambda\gamma_1/e)$$

where γ_1, γ_2 are the working precisions to which terms of the a- and b-sequences are computed and

$$\lambda = \frac{4+e^2}{e^e} + 1 \approx 1.75, \quad \rho = \sum_{j=0}^{\infty} \frac{1}{\phi'(j)} < 2.4. \qquad (17)$$

For the extended sum under present consideration, this allows us to bound the error in c_0:

$$|\delta c_0| \le N\rho(\gamma_2 + e^{-1}\lambda\gamma_1)\phi'(x-1) \qquad (18)$$

which is just N times the error bound at this stage in the simple large addition algorithm.

The improvement in the error control for the extended sum derives from the fact that only one c- or h-sequence is needed so that the propagation of this error is minimized. In the situation where $|Z| \ge |X_0|$, the analysis continues precisely as for the addition case in [2] except that additional terms of the h-sequence may be needed. The propagation of the error through the calculation of the remainder of the c-sequence (with working precision γ_2) yields the bound

$$|\delta c| \le \rho\left(N(\gamma_2 + \lambda\gamma_1/e) + \gamma_2 + \lambda\gamma_1 \ln(1/\gamma_2)\right) \qquad (19)$$

The computation of the necessary terms of the h-sequence adds at most a further $\rho\gamma_2$ to the overall error. (See [4], for details.) It follows that the final error is bounded by

$$|\delta z| \le \rho\left(\gamma_2(N+2) + \lambda\gamma_1(N/e + \ln(1/\gamma_2))\right) \qquad (20)$$

The main term in the bound for a single addition is $2(\rho+1)\gamma_2$ since $\gamma_1 \ll \gamma_2$. It follows that the error in this extended sum of $N+1$ terms is less than $N/2$ times that for the single operation.

For the situation where $z<x_0$, the inherent error is bounded by $(N+1)\gamma_0\phi'(x_0)/\phi'(z)$. This is the situation where some cancellation has occurred but it has not been so severe as to cause the result to flip down to reciprocal form. The algorithm, and its analysis, are unchanged for the computation of c_0. Since $|Z| \ge 1$, it follows that

$$c_0 = \phi(z)/\phi(x_0) \ge 1/\phi(x_0) = a_0 \ge \gamma_1. \qquad (21)$$

Just as with the large subtraction algorithm considered in [2], the c-sequence is increasing and so we have

$$|\ln c_j| \le \ln 1/\gamma_1$$

for every j for which the sequence is to be computed.

The obvious modifications to the analysis of [2] can now be used to obtain the final error bound

$$|\delta z| \le \frac{\phi'(x)}{\phi'(z)}\{\gamma_2(1+\rho(N+1)) + \lambda\gamma_1(1+\rho(N/e+\ln 1/\gamma_1))\} \qquad (22)$$

in which we again see that the dominant term is less than $N/2$ times that for the single subtraction algorithm.

The last case we consider in any detail is the one which might be expected to be most troublesome. This is the situation where $|X_0| \ge 1 > |Z|$ which is the case of severe cancellation. Specifically then we have

$$s_z\phi(z)^{-1} = \phi(x_0) + \sum_{i=1}^{N} s_i\phi(x_i)^{r_i} \qquad (23)$$

which has the inherent error

$$|\delta z| \le \phi^2(z)\left\{\left|\frac{\phi'(x_0)}{\phi'(z)}\right| + \sum_{r_i=+1}\left|\frac{\phi'(x_i)}{\phi'(z)}\right| + \sum_{r_i=-1}\left|\frac{\phi'(x_i)}{\phi^2(x_i)\phi'(z)}\right|\right\}\gamma_0$$

which, by a similar argument, we may bound by

$$|\delta z| \le (N+1)\gamma_0\frac{\phi'(x_0)\phi(z)}{\phi'(z-1)}. \qquad (24)$$

Yet again the analysis is unchanged as far as the calculation of c_0. At this stage, for this case we have $c_0<a_0$ and we form $h_1 = -\ln(c_0/a_0)$ and proceed to compute further terms of the h-sequence as necessary. The analysis of [4] for the case of large subtraction with cancellation can now be used to obtain

$$|\delta z| \le \frac{\phi'(x_0)\phi(z)}{\phi'(z-1)}\{\gamma_2(1+\rho(N+1)) + \lambda\gamma_1(1+N\rho/e)\}. \qquad (25)$$

Again the rounding error is magnified by less than $N/2$. In the various cases of extended "small" sums, in which $r_i=-1$ for every i, the inherent error is bounded by

$$|\delta z| \le (N+1)\gamma_0\frac{\phi'(x-1)\phi(z)}{\phi(x)\phi'(z-1)} \qquad (26)$$

for $|Z| \le 1$ and by

$$|\delta z| \le (N+1)\gamma_0\frac{\phi'(x-1)}{\phi(x)\phi'(z)} \qquad (27)$$

for the "flip-up" case where $|Z|>1$. Extensions of the analysis of [4] similar to those used above yields an error bound for the cancellation case (which for small arithmetic corresponds to $z>x_0$) which shows an increase by a factor of about $7N/12$ over the single small subtraction algorithm. Similar results apply to the remaining cases so that in all cases the roundoff error of the extended operations is increased by only about half the expected factor. It follows that with the same working

precisions as suggested in [4] - and used in the software implementation in [16] - the roundoff errors of extended summation are only about half of the inherent error bounds.

The corresponding bounds in [4] were used to find working precisions which control the roundoff error to be of the order of the inherent error. The same reasoning here allows us the opportunity to control the error of extended summation to give the effect of only a single rounding error - at least for the cases where severe cancellation does not occur. The extra precision needed is clearly dependent on the maximum vector length available in the extended algorithm. For illustrative purposes we shall take this maximum to be $N+1=1024$. Since in [4], we had $\gamma_1 \le 2^{-5}\gamma_2$ it follows that, in the case where $|Z|>|X_0|\ge 1$, we must choose γ_2 so that $2.5\times 2^{10}\gamma_2 \le \gamma_0$. For single length SLI arithmetic, $\gamma_0=2^{-28}$ and so using $\gamma_2=2^{-12}2^{-28}=2^{-40}$ will suffice.

This amounts to adding just 9 bits of precision to the internal computation of the SLI algorithm. This is a very low cost in order to achieve the often sought after goal of a single roundoff error for extended summation.

5. CONCLUSIONS

In this paper, we have described an algorithm for extended arithmetic operations in SLI arithmetic and discussed its possible hardware implementation and error analysis.

The implementation details suggest that any time-penalty associated with the use of SLI arithmetic can be kept to a very small factor on highly parallel computers - perhaps of the order of just 2 or 3 for typical scientific computing programs. Not only do we see that extended operations can be executed in times comparable with single SLI arithmetic operations, the use of this extended algorithm also results in a significant relative reduction in the roundoff error bound. We see too that the error analysis of such extended operations with elements of mixed sign is more straightforward than has sometimes been suggested.

Putting these results together provides us with alternatives for the utilization of improved technology and massive parallelism. One possibility is that any such improvements be used to provide more raw speed. Perhaps a better solution would be to compromise by accepting about half the potential speed-up and using SLI arithmetic instead. A speed-up by a factor of 5 rather than 10, for example, may well be an acceptable "price" for freedom from scaling problems in order to avoid overflow or underflow. We might even conclude that a smaller speed-up would be acceptable in order to perform the internal arithmetic to greater accuracy and so compute vector sums and scalar products with just a single roundoff error.

References

[1] C.W.Clenshaw and F.W.J.Olver, *Beyond floating point*, J. ACM 31 (1984) 319-328.

[2] C.W.Clenshaw and F.W.J.Olver, *Level-index arithmetic operations*, SIAM J Num Anal 24 (1987) 470-485.

[3] C.W.Clenshaw, F.W.J.Olver and P.R.Turner, *Level-index arithmetic: An introductory survey*, Numerical Analysis and Parallel Processing (P.R.Turner Ed.) LNM 1397, Springer Verlag, 1989, pp. 95-168.

[4] C.W.Clenshaw and P.R.Turner, *The symmetric level-index system*, IMA J Num Anal 8 (1988) 517-526.

[5] C.W.Clenshaw and P.R.Turner, *Root-squaring using level-index arithmetic*, Computing 43 (1989) 171-185.

[6] H.Hamada *URR: Universal representation of real numbers*, New Generation Computing, 1 (1983) 205-209.

[7] U.W.Kulisch and W.L.Miranker, *The arithmetic of the digital computer: A new approach*, SIAM Review 28 (1986) 1-40.

[8] D.W.Lozier and F.W.J.Olver, *Closure and precision in level-index arithmetic*, SIAM J Num. Anal, to appear.

[9] D.W.Lozier and P.R.Turner, *Supercomputers need super arithmetic*, NIST Tech Rep, NISTIR 89-4135.

[10] S.Matsui and M.Iri *An overflow/underflow-free floating-point representation of numbers*, J.Inf.Proc. 4 (1981) 123-133

[11] F.W.J.Olver, *A new approach to error arithmetic*, SIAM J Num Anal. 15 (1978) 368-393

[12] F.W.J.Olver, *Rounding errors in algebraic processes - in level-index arithmetic*, Proc. Reliable Numerical Computation (M.G.Cox and S.Hammarling, eds.) Oxford, 1990, pp.197-205.

[13] F.W.J.Olver and P.R.Turner, *Implementation of level-index arithmetic using partial table look-up*, Proc. ARITH8, (M.J.Irwin and R.Stefanelli, Eds.) IEEE Computer Society, Washington, DC, 1987, 144-147.

[14] P.R.Turner, *Towards a fast implementation of level-index arithmetic*, Bull IMA 22 (1986) 188-191.

[15] P.R.Turner, *Algorithms for the elementary functions in level-index arithmetic*, Scientific Software Systems (M.G.Cox and J.C.Mason Eds.) Chapman and Hall, 1990, pp. 123-134.

[16] P.R.Turner, *A software implementation of sli arithmetic*, pp. 18-24, Proc.ARITH9, (M.D.Ercegovac and E.Swartzlander, Eds) IEEE Computer Society, Washington DC, 1989.