IMPLEMENTATION OF LEVEL-INDEX ARITHMETIC USING PARTIAL TABLE LOOK-UP

F. W. J. OLVER and P. R. TURNER

IPST,
University of Maryland,
College Park, MD20742, U.S.A.

Department of Mathematics,
University of Lancaster,
Lancaster LA1 4YL, U.K.

Abstract

This paper is concerned with finding fast efficient algorithms for performing level-index arithmetic. The approach used combines the advantages of parallel processing with the use of table look-up. The latter is used only for short words and the result is a potential implementation with ℓi operation times comparable with floating-point long multiplications.

Introduction

In this paper we explore an alternative approach to the implementation of ℓi and $s\ell i$ arithmetic ([2],[3]and[4]) to that used in [9]. The objective remains the same, namely the provision of fast algorithms which will make level-index arithmetic a feasible, practical computing facility. For simplicity of description, we again consider the case of ℓi arithmetic operations, the extensions for the $s\ell i$ case being straightforward.

A conclusive comparison between the approach described here and that in [9] is not possible without detailed analysis of the relative efficiencies of the hardware components required. However, it seems likely that any efficient implementation will make use of aspects of both these approaches - and probably others as well.

The level-index addition/subtraction algorithm is a technique for obtaining $z = n+h$ such that

$$\phi(z) = \phi(x) \pm \phi(y) \qquad (1.1)$$

where $x = \ell+f \geq y = m+g$; ℓ,m,n are nonnegative integers; $f,g,h \in [0,1)$ and the ℓi representation function ϕ is given by

$$\phi(x) = \exp(\exp(\ldots(\exp f)\ldots)), \qquad (1.2)$$

the exponentiation being carried out ℓ times. This requires the computation of the (finite) sequences defined by

$$a_j = 1/\phi(x-j), \quad b_j = \phi(y-j)/\phi(x-j) \qquad (1.3)$$

and

$$c_j = \phi(z-j)/\phi(x-j). \qquad (1.4)$$

(For a full description of the algorithm see [3]).

In the next section we describe a partial table look-up approach to the computation of $\{a_j\}$ and $\{b_j\}$ based on breaking a number up into packets of 6 bits. This idea extends into the calculation of $\{c_j\}$ which we discuss in section 3.

Throughout the description we make considerable use of parallelism both in the vector sense and at a bit-by-bit level based on the use of the Carry Saver Adder (CSA) and the storage of internal quantities as "double numbers". (See [9] and, for a full description of the CSA and "double number" philosophy,[7] and [10]). The benefit here is the limited use of the Carry Propagate (CPA) or Carry Look-ahead Adders which are necessarily much more time consuming.

2. Computation of $\{a_j\}$,$\{b_j\}$ using partial table look-up.

The sequence $\{a_j\}$ is generated by the relations

$$a_{\ell-1} = e^{-f}, \quad a_{j-1} = \exp(-1/a_j) \qquad (j = \ell-1,\ldots,1) \qquad (2.1)$$

while for $\{b_j\}$ we have

$$b_{m-1} = a_{m-1}e^{g}, \quad b_{j-1} = \exp((b_j-1)/a_j) \qquad (j = m-1,\ldots,1,). \qquad (2.2)$$

The required absolute working precisions for these quantities for single-length ℓi arithmetic are typically 2^{-37} for $\{a_j\}$ and 2^{-32} for $\{b_j\}$. (Note that for the case $m=0$ we also require $b_0 = a_0 g$.)

Consider first the calculation of a_{j-1} for $j<\ell$. The basic strategy is to form the reciprocal

$$t = 1/a_j, \qquad (2.3)$$

say, and then use table look-up for subwords of the binary representation of t combined with parallel multiplications to obtain a_{j-1}.

In order to obtain a_{j-1} with absolute precision (ap) 2^{-37} we require t to this same precision.

Also, if $a_j \leq 2^{-5}$ then

$$a_{j-1} \leq e^{-32} \leq 2^{-46} = 0; \text{ ap } (2^{-37}) . \quad (2.4)$$

Thus if $a_j \leq 2^{-5}$ we return the value $a_{j-1} = 0$. Otherwise the (temporary) storage of t requires 42 bits (including 5 before the binary point). We write

$$t = t_1 + t_2 + \ldots + t_7 \quad (2.5)$$

where each t_i is simply the i^{th} block of 6 bits from this 42-bit word. The values of $\exp(-t_i)$ $(i=1,2,\ldots,7)$ can be obtained by <u>simultaneous</u> table look-up since the possible ranges of values of the t_i are disjoint. The required value is then the product of these quantities.

To estimate the time for this operation we begin with the reciprocation. We first shift a_j to the interval $[\frac{1}{2},1)$ and subtract the result from 1 so that we require $1/(1-\delta)$ for some $\delta \in (0,\frac{1}{2}]$. This is given to the necessary accuracy by

$$1 + \delta + \ldots + \delta^{41} .$$

Using the same notation as in [9] we suppose such a shift takes b time units (t.u.) and that a further b t.u. will be required to form the one's complement to yield δ. We also assume that a single CSA operation takes a t.u.

We next form δ^2 as a double number. Using the CSA we reduce the 42 terms to 2 in 8a+b t.u. Next, the powers $\delta^{2^k+1}, \ldots, \delta^{2^{k+1}}$ can be formed in parallel for $k = 1,2,\ldots,5$ as "double x double" products. (This requires the facility to "broadcast" δ^{2^k} as an input to several multipliers simultaneously). Each such product generates 168 terms which can be reduced to 2 in 12a + b t.u.

There are now 40 double and 2 single numbers to be added to form a single number. This takes a further 10a + b + c t.u., where c t.u. is the time required for the CPA (carry propagate adder) operation. Finally to obtain t itself this result must be shifted to compensate for the initial shift of a_j. The total time for the reciprocation is therefore

$$78a+10b+c \text{ t.u.}$$

Separating t into the subwords $t_1,\ldots,t_7$ is another shift operation. This is to be followed by the simultaneous table look-up after which we have seven single 42-bit numbers to be multiplied together. Six of these factors can be multiplied in pairs leaving the results as double numbers. Continuing as in the reciprocation process and then using the CPA to express the result a_{j-1} as a single number we find that this part of the operation requires a total of

$$b+8a+b+2(12a+b)+c = 32a+4b+c \text{ t.u.} \quad (2.6)$$

The overall time to produce a_{j-1} from a_j is therefore

$$110a+14b+2c+e \text{ t.u.}$$

where e t.u. is the time needed for a 6-bit table look-up. The calculation of $a_{\ell-1}$ does not require an initial reciprocation and therefore takes only

$$32a + 4b + c + e \text{ t.u.}$$

(The fact that there are now only 6 factors in the product does not yield any saving in the multiplication time).

The sequence $\{b_j\}$ can be computed in parallel with $\{a_j\}$. (We must use the alternative definitions $b_{m-1} = \exp(g-f)$ for the case $\ell=m$ and $b_{m-1} = \exp(g-1/a_m)$ for $m < \ell$). Firstly we can form $1-b_j$ while the reciprocation stage above is being used to obtain $1/a_j$ as a double number. These results must then be multiplied to produce $t' = (b_j - 1)/a_j$. This multiplication takes 10a+b t.u. The remainder of the calculation of b_{j-1} follows precisely that of a_{j-1}.

This time-lag of 10a+b t.u. is no penalty since the table look-ups for t and t' cannot be performed simultaneously unless the table is duplicated. However, since the table requires $2^6 \times 7 \times 42 = 18,816$ bits, it is unrealistic to suppose that it can be repeated in several places on a "level-index chip". (There will be a similar delay in the "parallel" computation of a_{m-1} and b_{m-1} in the case $m=\ell$ since a full subtraction is needed to evaluate f-g. This is accounted for in the timings below. For the case $m<\ell$ the computation of b_{m-1} can be achieved in the same time as a_{m-1} and so there is a delay of e t.u. caused solely by the non-duplication of the table). It is likely therefore that the fetching and carrying times of these table look-ups would be such that $e > 10a+b$.

Combining these results we see that the overall time for obtaining a_{j-1}, b_{j-1} from a_j, b_j is

$$T_1 := 110a + 14b + 2c + 2e \text{ t.u.} \quad (2.7)$$

while that for computing $a_{\ell-1}$ (and $b_{\ell-1}$ when $m=\ell$) is bounded by

$$T_2 := 32a + 4b + c + 2e + \max(c,e) \text{ t.u.} \quad (2.8)$$

It is worth noting here that retention of t,t' as double numbers would be unlikely to result in any saving since it would remove the advantage of simultaneous table look-up. The comparison of these timings with those obtained in [9] depends critically on the relative timings of e t.u. for the table look-up and d t.u. for obtaining the sign of a double number. The corresponding value

for T_1 in [9] is 140a+50b+41d. With the assumption that a = b ≃ c/32 the comparison reduces, approximately, to a comparison between e and 20d which would probably depend on architectural details.

3. The sequence $\{c_j\}$

This sequence is computed using the relation

$$c_{j+1} = 1 + a_{j+1} \ell n\, c_j \qquad (j=0,1,\ldots) \qquad (3.1)$$

where $c_0 = 1 \pm b_0$. Since $b_0 \in [0,1]$, c_0 is obtained either by inserting a one before the binary point or by forming the one's complement of b_0. In both cases this takes b t.u.

To compute $\ell n\, c_j$ we first write

$$c_j = s(1+\sigma) \qquad (3.2)$$

where s comprises the first six (significant)bits of c_j and $\sigma < 2^{-6}$. The first step is to divide c_j by the six-bit number s. As in the previous section we first shift s and form the one's complement so that division is by $1-\delta$ with $0 \leq \delta < \frac{1}{2}$. This takes 2b t.u. We require $\ell n(1+\sigma)$ to ap(2^{-32}) and so σ to this same accuracy. In this case δ is a 6-bit single number from which we can compute simultaneously δ^2 and $c_j\delta$ as 12-bit and 32-bit double quantities respectively in 3a+b t.u. Next $c_j\delta^2, c_j\delta^3$ and δ^4 can be computed in parallel, as double numbers. The remaining terms are computed in three similar stages.

The timings for these operations and their summation using the same approach as in §2 are as follows:

$c_j\delta^2,\ c_j\delta^3,\ \delta^4$	9a+b	t.u.
$c_j\delta^4,\ldots,c_j\delta^7,\delta^8$	11a+b	t.u.
$c_j\delta^8,\ldots,c_j\delta^{15},\delta^{16}$	12a+b	t.u.
$c_j\delta^{16},\ldots,c_j\delta^{31}$	12a+b	t.u.
$\sum_{r=0}^{31} c_j\delta^r$	9a	t.u.

To complete the calculation of σ this sum must be shifted back to compensate for the original shift in s and then unity must be subtracted. This subtraction can be achieved with one further CSA operation (see [6] for example) and so the double number σ is obtained in a total of 57a + 8b t.u.

Since $\sigma < 2^{-6}$ the required precision in $\ell n(1+\sigma)$ is obtained using the approximation

$$\sigma - \frac{\sigma^2}{2} + \frac{\sigma^3}{3} - \frac{\sigma^4}{4} + \frac{\sigma^5}{5} \quad .$$

With 1/3 and 1/5 as stored binary constants we can compute σ^2, $\sigma/3$ and $\sigma/5$ in parallel as double numbers. Only σ^2 requires a "double x double" product; this has 128 terms and so takes 12a+b t.u. Next, $\sigma^3/3$ and σ^4 can be produced in parallel and then $\sigma^5/5$. During the final stage we also perform the shifts required to obtain $\sigma^2/2$ and $\sigma^4/4$. This gives 10 terms to be summed together with ℓn s' and $-p\ell n\ 2$, where s' is the shifted value 2^ps and p is the number of leading zero bits in c_j. (Note that, for addition, $c_j > 1$ so that p=0). The value of ℓn s' is obtained by table look-up while the other stages proceed; similarly $-p\ell n\ 2$ can be computed from the stored constant $\ell n\ 2$ during this time. This summation takes a further 5a t.u. which gives a total time of

$$57a + 8b + 3(12a+b) + 5a = 98a + 11b \text{ t.u.} \qquad (3.3)$$

for the evaluation of $\ell n\, c_j$ as a double number.

To complete the calculation of c_{j+1} the last result must be multiplied by a_{j+1} after which the addition of unity requires a further CSA operation before the CPA is used to produce c_{j+1}. The overall time is therefore

$$98a + 11b + (10a+b) + a + c = 109a + 12b + c \text{ t.u.} \qquad (3.4)$$

The corresponding time for this stage in [9] is 83a+37b+34d t.u. Again the comparison depends critically on the relative values of c and d and of the 6-bit table look-up which we have implicitly assumed here takes no more than about 100a or 3c t.u.

To complete the ℓi operations we have either a division or at most two further logarithms to compute. These final stages can be performed, using the same general approach in at most 2(98a + 11b + c) t.u.

4. Overall Timings and Conclusions

The worst case for which ℓi arithmetic is not trivial is given by $\ell = m = n = 5$, because $\phi(x) \pm \phi(y)$ is accurately approximated by $\phi(x)$ to every realistic working precision when $x \geq 6$ and $x > y$; see [1], [2] and [8] For this "worst case", only one final logarithm is needed and so we obtain an overall time for finding the sum or difference of two ℓi numbers of

$$(32a+4b+c+e+\max(c,e)) + 4(110a+14b+2c+2e) + b + 4(109a+12b+c) + 98a+11b+c$$

$$= 1006a + 120b + 14c + 9e + \max(c,e) \qquad (4.1)$$

$$\simeq 49c + 9e + \max(c,e) \text{ t.u.} \qquad (\text{for } a=b\simeq c/32).$$

No floating-point system would be able to perform arithmetic on numbers of level 5 magnitude. Indeed most floating-point arithmetic takes place with numbers at levels no more than 3 since $\phi(4.0)\simeq 3.814\times10^6$. For a more realistic comparison - which is still not unfairly favourable to the ℓi system - suppose that $\ell=3$, m=2 and n=4. The overall time is reduced to

$$666a + 79b + 9c + 6e \simeq 32c + 6e \text{ t.u.} \qquad (4.2)$$

The figures quoted above are based on the assumption that $e \geq 11a$ (in the timings for $\{a_j\}$) and $e \leq 100a$ (in those for $\{c_j\}$). In the table below we show comparisons with floating-point long multiplication times (f.p.m.) of approximately 25c for a range of possible values of e.

e	"Realistic"time	"Worst case"time
11a or c/3	34c or 1½ f.p.m.	53c or 2 f.p.m.
16a or c/2	35c or 1½ f.p.m.	55c or 2 f.p.m.
32a or c	37c or 1½ f.p.m.	59c or 2½f.p.m.
64a or 2c	42c or 2 f.p.m.	69c or 2½f.p.m.
96a or 3c	47c or 2 f.p.m.	79c or 3 f.p.m.

It should also be observed that for many calculations even these times could be significantly reduced since, for example, if $\ell = n = 2$ we obtain approximate times of only about 17c + 4e t.u. which for all the above values gives overall times comparable with just one floating-point long multiplication.

Although any firm conclusions must depend on hardware and architectural considerations, the much less sensitive dependence on the table look-up time (compared with the dependence on discrimination time in [9]) suggests that this approach is well worth investigating further as a plausible hardware implementation of ℓi and $s\ell i$ arithmetic. As with the approach in [9] this will be explored in a software simulation using the ICL DAP at Queen Mary College, London.

Of course, the case for using ℓi or $s\ell i$ arithmetic does not depend solely on its implementability. A full mathematical justification in terms of its closedness, smoothness of representation and precision is given in [2],[3],[4] and [8]. A necessary complement to this is a practical justification based on computational experience using this system. This is an on-going project. The early results on applications to binomial probabilities [3] and solving polynomial equations by root-squaring [5] are very encouraging. In both cases the need to scale to avoid over- or under-flow causes severe problems for floating-point arithmetic. Using $s\ell i$ arithmetic results of high precision are obtained from simple programs despite the fact that intermediate calculations have involved numbers from well outside the floating-point range.

Acknowledgement

The authors are pleased to acknowledge several helpful dicussions with and comments on this work from C.W. Clenshaw and D.W. Lozier. The work of one of us (FWJO) has been supported, in part, by the US Army Research Office, Durham under contract DAAG-29-84-K-0022 and the National Science Foundation under Grant DMS-84-19820.

References

[1] C.W. Clenshaw, D.W. Lozier, F.W.J. Olver and P.R. Turner, Generalized exponential and logarithmic functions, Comp. and Maths. with Appls 12B(1986)1091-1101.

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

[3] C.W. Clenshaw and F.W.J. Olver, Level-index arithmetic operations, SIAM J. Num. Anal. (1987).

[4] C.W. Clenshaw and P.R.Turner, The symmetric level-index system, manuscript.

[5] C.W. Clenshaw and P.R.Turner, Root-squaring with $s\ell i$ arithmetic, manuscript.

[6] J.B. Gosling, Design of Arithmetic Units for Digital Computers, MacMillan, London 1980.

[7] K. Hwang and F.A. Briggs, Computer Architecture and Parallel Processing. McGraw-Hill, New York, 1984.

[8] F.W.J. Olver, A closed computer arithmetic, these proceedings.

[9] P.R.Turner, Towards a fast implementation of level-index arithmetic. Bull.IMA.(1986).

[10] S. Waser and M.J. Flynn, Introduction to Arithmetic for Digital Systems Designers, Holt, Rinehart and Winston, New York, 1982.