

Correcting the Normalization Shift of Redundant Binary Representations

Peter Kornerup, *Member, IEEE*

Abstract—An important problem in the realization of floating point subtraction is the identification of the position of the first non-zero digit in a radix represented number, since the significand usually is to be represented left normalized in the part of the word(s) allocated for representing its value. There are well-known log-time algorithms for determining this position for numbers in non-redundant representations, which may also be applied to suitably (linear-time) transformed redundant representations. However, due to the redundancy in the latter case the position thus determined may need a correction by one. When determination of the shift amount is to be performed in parallel with conversion to non-redundant representation (the subtraction), it must be performed on the redundant representation. This is also the case when the significand is to be retained in a redundant representation until the final rounding. This paper presents an improved algorithm for determining the need for a correction of the normalization shift amount, which can be run in parallel with the algorithm finding the “approximate” position.

Index Terms—Floating-point addition, normalization, leading-one determination (LOD), leading-zero anticipation (LZA)

I. INTRODUCTION

In the realization of subtraction in a floating-point (FP) unit, it is necessary to normalize the result when cancellation of significant digits has occurred. For an FP subtraction with cancellation, where the result is to be delivered in the same accuracy as the operands, the result will be exact and no rounding will be needed. However, in the implementation of a fused multiply-add instruction, $a \times b + c$, or when the result is to be delivered in a lower precision, the result will in general not be exact, hence it may be advantageous to keep the resulting difference in a redundant representation until the very last moment before rounding and conversion to non-redundant representations to take place. In both cases a negation may be needed when a sign-magnitude result is required.

Traditionally, obtaining a normalized result of a subtraction was a three-step process. First perform the subtraction in a carry-completing adder, then determine the number of leading zeroes, and finally perform the normalization shifts, all these steps taking at best logarithmic time in the number of bits of the operands ($\Theta(\log(n))$). The shift amount may be determined by an algorithm, e.g., [7], usually denoted LZA (for Leading Zero Anticipation) or LOD (for Leading One Determination).

The problem of determining the normalization shift amount for a redundantly represented value was apparently first analyzed in [5], [9] provides an overview of the general problem of leading zero anticipation, with an extensive bibliography. In

[2] it was shown that it is possible in constant time to convert a number represented in borrow-save/signed-digit representation (i.e., before the subtraction) over the digit set $\{-1, 0, 1\}$, into a bitstring having the same or one less leading zero valued digits as the non-redundantly represented value. To obtain a properly normalized representation of the value, it may thus be necessary with a one-bit left shift.

The authors then proposed an algorithm which in logarithmic time ($\Theta(\log_2(n))$) can determine the need for a correction, intended to be run in parallel with the determination (by a LZA/LOD algorithm) of the “approximate” number of leading zeroes, and with the adder performing conversion to non-redundant form (the subtraction). Their algorithm is to be realized as two binary trees running in parallel. One tree determining the need for a correction under the assumption that the value is positive, and the other if it is negative. The nodes of each of the trees take 4 bits of input from each of two sub-trees, and return 4 bit of output, thus essentially performing a 16-to-8 (~ 2 -to-1) reduction.

In [10] another algorithm was proposed, running in $\Theta(\log_{1.5}(n))$ -time, where each node takes 2 bits of input from each of three sub-trees, producing two times two bits, thus a 6-to-4 (~ 3 -to-2) reduction.

Using a quite similar approach to that taken in [2] we shall here show that a single binary tree is sufficient, where the nodes take 3 bits of input from each of two sub-trees, and produce 3 bits of output, i.e., 6-to-3 (~ 2 -to-1) reduction. Thus it is a significant simplification compared to the solution proposed in [2], and in fewer levels than proposed in [10].

In [4] a very different approach was suggested. Based on the non-redundant result of the subtraction in the adder it is determined by a binary tree if the leading non-zero bit is in an even or an odd position. This is compared to the even or oddness of the shift amount determined by the LZA circuit, thus determining whether a correction is needed. Note that the log-time tree calculation then must take place in parallel with the initial “coarse” shifting, the correction information assumed to be available before a possible final left or right shift of a single position. The authors claim this was possible in a specific single precision FP-implementation. A similar approach was reported in [1], where the correction information also is determined in parallel with the “coarse” shifting.

As background let us briefly recall the problem of determining the number of leading zeroes, $\text{lza}(P_m)$ of a given non-redundant fixed point number, $P_m = \sum_0^{m-1} d_i \beta^i$. It is easily described recursively by a “divide and conquer” algorithm [7] when m is a power of 2, say $m = 2^k$.

Algorithm 1 (Leading Zeroes Anticipation, LZA):

Stimulus: An integer k and a fixed point number $P_{2^k} = \sum_{i=0}^{2^k-1} d_i \beta^i$ where the digits d_i are members of a non-redundant digit set for base β .

Response: The value of $\text{lza}(k, P_{2^k})$, the number of leading zero-valued digits in P_{2^k} .

Method: **if** $k = 0$
then if $d_0 = 0$ **then** $t := 1$
else $t := 0$;
return t ;
else $L := \sum_{i=0}^{2^{k-1}-1} d_{i+2^{k-1}} \beta^i$; $R := \sum_{i=0}^{2^{k-1}-1} d_i \beta^i$;
 $l := \text{lza}(k-1, L)$; $r := \text{lza}(k-1, R)$;
if $l = 2^{k-1}$ **then return** $l + r$
else return l
end;

The algorithm can directly be implemented as a tree structure where each node receives the number of leading zeroes from two sibling nodes, possibly together with a bit signaling whether the sub-string is non-zero, to simplify the test $l = 2^{k-1}$. When the length is calculated in binary, note that the left signaling bit inverted can then just be concatenated (by \circ) as the new most-significant bit to either the value of r or l . The new signaling bit is then the OR of the two incoming bits, where at the leaf nodes $z \equiv (d \neq 0)$, d being the digit value and $t = ' '$ is the empty string.

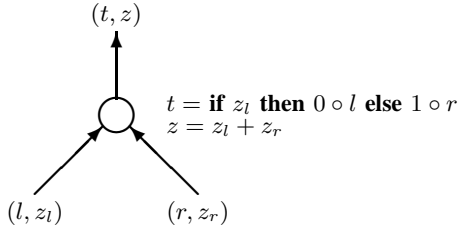


Fig. 1. LZA-node

Next Section II considers the equivalent problem of determining the shift amount to normalize a redundantly represented number, taking insignificant non-zero digits into account. As in [2] the proposed algorithm is intended to be run in parallel with conversion from redundant to non-redundant representation, i.e., a carry look-ahead adder. Subsequently, the actual normalization may then be performed either on the redundant or on the non-redundant result of the conversion. In Section III conclusions are drawn, in particular it compares the proposed solution with those of [2], [10] and [4].

II. NORMALIZATION OF REDUNDANTLY REPRESENTED NUMBERS

To normalize a number given in a redundant representation, it is necessary first to eliminate a possible leading string of non-zero digits of no significance. For the two important redundant binary representations, it turns out to be possible by a simple constant time conversion to obtain a bit pattern having either the same or one less leading zero than the non-redundant binary representation of the value. This makes it possible to “quasi-normalize” a redundant number, say in binary to be in the interval $[1; 4)$.

For borrow-save representations, employing the N and P -mappings defined in [3] (realizable by arrays of half-adders), the N -mapping is capable of transforming a digit string $01\bar{1} \dots \bar{1}$ into the string $000 \dots 1$, and the P -mapping will transform a string $0\bar{1}1 \dots 1$ into $000 \dots \bar{1}$. If the number is in 2’s complement carry-save, a similar Q -mapping will convert the representation into borrow-save. Hence if the sign of the number is known we have the following

Observation 2: If the sign of a binary radix polynomial in borrow-save or carry-save representation is known, then it is possible in constant time to transform it into a digit string having the same or one less leading zero than the value in non-redundant representation. For borrow-save this is obtained by applying the N -mapping to a positive number, and the P -mapping on a negative. A carry-save represented polynomial is first to be transformed by a Q -mapping, and the result in borrow-save is to be mapped as before.

Note that a constant time conversion can only be based on a bounded left and right context of each position. Consider the following digit string in borrow-save representation¹ $A = 01\bar{1}^j 0^k \bar{1}$ for $j \geq 1$ and $k \geq 2$. Let the result of applying the N -mapping be B , and the non-redundant result be C , then

$$\begin{aligned} A &= 01\bar{1} \dots \bar{1}0 \dots 00\bar{1} \\ B &= 000 \dots 10 \dots 0\bar{1}1 \\ C &= 000 \dots 01 \dots 111 \end{aligned}$$

The example shows how the N mapping was able to eliminate a leading string of non-significant non-zero digits. But also how an isolated unit-valued digit followed by a string of the opposite sign introduces an extra leading zero when the string is converted to non-redundant representation, this being the reason for the uncertainty of the LZA algorithm when applied to a redundant representation.

Since it is not possible with a bounded context to identify the sign of the right context, the presence of the extra leading zero in the non-redundant representation cannot be detected in constant time.

For the case where the sign is not known, we note that it does not really matter for the LZA algorithm what happens to any less significant string to the right. We may thus seek transformations which does not even preserve the value. To obtain the approximate number of leading zeroes, it is sufficient to find a bit string having the same or one less leading zeros as the non-redundant representation of the value, provided that we are able to find the necessary correction.

Noting that the N and P -mappings only use a one-digit right context, we will now include left context also, as this may be used to identify the sign when dealing with the most-significant end of the string. Considering borrow-save representation, assume that an extra zero-valued digit is appended to the left, just for the purpose of having a left context in the most significant position, but not to be counted. Similarly append a zero-valued digit on the right as a right context.

For the purpose of the following analysis assume that the thus extended borrow-save string is $d_n d_{n-1} \dots d_0 d_{-1}$, $d_n =$

¹The notation x^i means i occurrences of the symbol, here $x \in \{\bar{1}, 0, 1\}$.

$d_{-1} = 0$, with $d_i \in \{-1, 0, 1\}$, but is represented as three bit strings

$e_n e_{n-1} \cdots e_0 e_{-1}, p_n p_{n-1} \cdots p_0 p_{-1}$ and $m_n m_{n-1} \cdots m_0 m_{-1}$, where

$$e_i \equiv (d_i = 0) \quad p_i \equiv (d_i = 1) \quad \text{and} \quad m_i \equiv (d_i = -1). \quad (1)$$

This will allow us to use a complement notation where e.g., $\bar{e}_i = p_i + m_i$, is a shorthand for d_i being either 1 or -1 .

Using techniques similar to those employed in [2], but here without distinguishing between positive and negative valued input, we want to generate other bit strings, identifying ‘‘interesting’’ positions, e.g., in particular a leading position containing a unit digit, as this will be needed for the LZA algorithm. But in particular we want to determine if this is an isolated unit, because if it is followed (after a non-empty sequence of zeroes) by a string of the opposite sign, this is precisely the situation when a correction is needed.

Hence we will try to identify situations with isolated units like $\cdots 010 \cdots$, i.e., in the notation above, such values of i for which $e_{i+1} p_i e_{i-1}$ holds. Or alternatively strings of the form $\cdots 01\bar{1}0 \cdots$ which can be identified by $p_{i+1} m_i e_{i-1}$, or more generally $\cdots 01\bar{1} \cdots \bar{1}0 \cdots$, that is other strings which reduce to isolated units. Note that the resulting unit will occur in position i for which $m_{i+1} m_i e_{i-1}$ holds, provided that the leading part gets eliminated. However, if the latter pattern is found isolated, i.e., in the context $e_{i+2} m_{i+1} m_i e_{i-1}$ with a leading zero, it will be identified as just some string of negative sign (see t_i below). Thus we can express positions where isolated positive units occur, as positions i where the following expression holds true:

$$\begin{aligned} u_i &= e_{i+1} p_i e_{i-1} + p_{i+1} m_i e_{i-1} + m_{i+1} m_i e_{i-1} \\ &= (e_{i+1} p_i + \bar{e}_{i+1} m_i) e_{i-1}. \end{aligned}$$

Equivalently we can by symmetry express positions where isolated negative units occur by:

$$\begin{aligned} v_i &= e_{i+1} m_i e_{i-1} + m_{i+1} p_i e_{i-1} + p_{i+1} p_i e_{i-1} \\ &= (e_{i+1} m_i + \bar{e}_{i+1} p_i) e_{i-1}. \end{aligned}$$

Obviously, there are other positions of positive value, they can similarly be expressed as:

$$\begin{aligned} s_i &= e_{i+1} p_i p_{i-1} + p_{i+1} m_i p_{i-1} + m_{i+1} m_i p_{i-1} \\ &= (e_{i+1} p_i + \bar{e}_{i+1} m_i) p_{i-1}, \end{aligned}$$

where the term $m_{i+1} m_i p_{i-1}$ is the termination of a string of the form $\cdots 01\bar{1} \cdots \bar{1}1 \cdots$, provided that the leading part is eliminated. Similarly for positions of negative value we find:

$$\begin{aligned} t_i &= e_{i+1} m_i m_{i-1} + m_{i+1} p_i m_{i-1} + p_{i+1} p_i m_{i-1} \\ &= (e_{i+1} m_i + \bar{e}_{i+1} p_i) m_{i-1}. \end{aligned}$$

As these expressions are mutually exclusive we can express positions where zeroes occur by:

$$z_i = 1 - (u_i + s_i + v_i + t_i)$$

whose complement $w_i = u_i + s_i + v_i + t_i$ defines a bit pattern to which the LZA algorithm can be applied. This is summarized in the following well known result:

Observation 3: Given a number x in borrow-save representation $d_{n-1} \cdots d_0$, then the bit-string $w_{n-1} \cdots w_0$, $w_i = e_{i+1}(p_i \bar{m}_{i-1} + m_i p_{i-1}) + \bar{e}_{i+1}(m_i \bar{m}_{i-1} + p_i p_{i-1})$, has the same or one less leading zero as the non-redundant representation of x .

Since the bit patterns defined above are mutually exclusive, for any position i we can thus encode these using three bits in sign-magnitude

$$\begin{aligned} s_i &\sim 001 \\ u_i &\sim 010 \\ x_i &\sim 011 \\ z_i &\sim 100 \\ y_i &\sim 101 \\ v_i &\sim 110 \\ t_i &\sim 111 \end{aligned} \quad (2)$$

where two new truth values x_i and y_i have been added to be used later, and 000 is not used.

For an implementation note that the 3-bit encoding in position i is a function only of the four bits u_i, s_i, t_i, v_i , which are defined in terms of the 9 bits e_j, p_j, m_j for $j = i+1, i, i-1$, which in turn are defined by the 6 bits encoding the 3 digit values d_{i+1}, d_i, d_{i-1} . Thus a 6-bit-in, 3-bit-out PLA structure as in Fig. 2 is capable of forming the encoding in a few logic levels. This PLA can simultaneously also define w_i to be used in Observation 3.

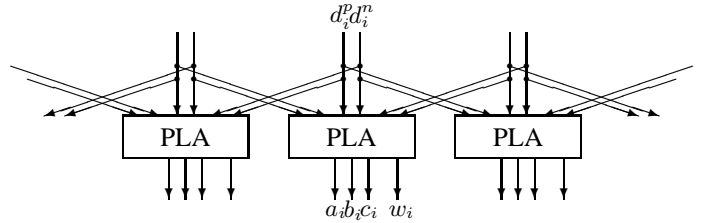


Fig. 2. Encoding of a_i, b_i, c_i and w_i

We are now able to define a log-time search to find unit prefixes followed by strings of the opposite sign, i.e., using notation from formal language theory, strings of the forms

$$X = z^* u z^+ v \sigma \quad \text{or} \quad X = z^* u z^+ t \sigma \quad \text{and} \quad Y = z^* v z^+ u \sigma \quad \text{or} \quad Y = z^* v z^+ t \sigma$$

written as regular expressions², where σ denotes arbitrary strings and a^* and a^+ denote strings of consecutive identical symbols a_i . Note that a correction is needed if and only if strings X or Y are found, so these will be the goals to search for.

To specify the grammar we further define substrings

$$\begin{aligned} U &= z^* u z^* & \text{and} & & S &= z^* s z^* \sigma \\ Z &= z^+ \\ V &= z^* v z^* & \text{and} & & T &= z^* t z^* \sigma \end{aligned}$$

It is now possible to define grammar rules for recursively building these strings in binary tree structures, e.g., $Z \Rightarrow z | Z Z$, $U \Rightarrow u | Z U | U Z$, etc. Using initial values $X = x$, $U = u$, $S = s$, $Z = z$, $T = t$, $V = v$, $Y = y$, it is easier to specify the rules for the nodes as a table:

²Recall that x^* means zero or more occurrences of the symbol x , and x^+ means one or more.

However, selectors can be implemented using complementary pass-transistor logic such that their delay time is shorter than that of a CMOS gate, e.g., see [6]. If implemented this way it is possible to shorten the longest path through a tree node to be less than two gate delays. This applies both to an implementation based on Fig. 3 or on the expressions for a and b above, as these can also be implemented using selectors.

Note that the expression at the root of the tree to decide if a correction is needed is $(a \oplus b)c$ (X or Y has been found), and that the sign of the number is given by a . Finally the expression $\overline{b + c}$ signals that the result is zero.

Observation 5 (Normalizing redundant binary numbers):

The amount of leftshifts necessary to normalize a binary, redundantly represented number can be determined in logarithmic time from a recoded representation of the number. Based on a constant time transformation, one binary tree structure can determine the number of shifts to obtain a quasi-normalized representation, and another tree structure can in parallel find the need for a correction to obtain the proper normalization. The latter tree simultaneously determines the sign of the number.

III. COMPARISONS AND CONCLUSIONS

The LZA-correction node defined above is obviously slower than the LZA-node of Fig. 1, respectively needing two or only one selector. Since the two trees are to run in parallel, it will be the correction tree that determines the overall timing.

Comparing with the two trees proposed in [2], their LZA-correction nodes can be implemented with a two-gate delay. But the two trees together corresponds to a single tree, where each node takes 2 times 8 bits, producing 8 bits in two logic levels, whereas the encoding into 3 bits proposed here significantly simplifies the tree, and reduces its area by an estimated factor 3. However, our nodes have three-gate delays, unless the selectors are implemented with complementary pass-transistor logic. In the latter case the tree will be delay competitive in a much smaller area.

The correction tree of [10] also employs nodes with two-gate delays, but their tree is of the 3-to-2 type, hence it will be taller and thus slower than those of [2], and the tree proposed.

When the LZA-tree and the corrector tree are employed in parallel with conversion to non-redundant representation, preferably it should be the conversion (in the adder) determining the timing. Recall that the nodes of a comparable adder look-ahead tree also need two gate delays.

Timings from an actual implementation of the 3-to-2 tree were reported in [10]. Here the error correction for a 54-bit implementation was reported with a delay of 0.53 ns, the LZA tree of 0.24 ns and the adder of 0.40 ns. A 3-to-2 tree for a 54-bit number requires 10 levels, whereas a binary tree only requires 6 levels. Thus even with three-gate delay nodes, the binary tree presented here should be faster than the 3-to-2 tree with two-gate delay nodes.

However, the implementation of the correction tree proposed here may be realizable with selectors in complementary pass-transistor logic with a delay comparable to that of the adder, such that the correction information can be available when the normalization is to be scheduled.

As mentioned in the introduction, the approach in [4] is very different, as the the correction information (the parity of the MSB position) can only be determined when the result of the adder is available. Hence the method is not comparable to the one presented here. There are other approaches where the error detection is performed in parallel with the normalization, e.g., [1]. For image processing applications [8] allow corrections with some errors, but at a limited rate.

REFERENCES

- [1] F. Arakawa, T. Hayashi, and M. Nishibori. An Exact Leading Non-Zero Detector for a Floating Point Unit. *IEICE Trans. Electron.*, E88-C(4):570–575, 2005.
- [2] J.D. Bruguera and T. Lang. Leading-One Prediction with Concurrent Position Correction. *IEEE Transactions on Computers*, 48(10):1083–1097, October 1999.
- [3] M. Daumas and D.W. Matula. Further Reducing the Redundancy of Notation Over a Minimally Redundant Digit Set. *Journal of VLSI Signal Processing*, 33(1/2):7–18, 2003.
- [4] C.N. Hinds and D.R. Lutz. A Small and Fast Leading One Predictor Corrector Circuit. In *Proc. 39 Asilomar Conference on Signals, Systems and Computers.*, pages 1181–1185. IEEE, 2005.
- [5] E. Hokenek and R.K. Montoye. Leading-Zero Anticipator (LZA) in the IBM RISC System/6000 Floating Point Execution Unit. *IBM Journal of Research and Development*, 34(1):71–77, 1990.
- [6] N. Ohkubo, M. Shinbo, T. Yamanaka, A. Shimizu, K. Sasaki, and Y. Nakagome. A 4.4 ns CMOS 54 × 54-b Multiplier Using Pass Transistor Multiplexer. *IEEE Journal of Solid State Circuits*, 30(3):251–257, 1995.
- [7] V.G. Oklobdzij. An Algorithm and Novel Design of a Leading Zero Detector Circuit: Comparison with Logic Synthesis. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2(1):124–128, March 1999.
- [8] M. Olivieri, F. Pappalardo, S. Smorfa, and G. Visalli. Analysis and Implementation of a Novel Leading Zero Anticipation Algorithm for Floating-Point Arithmetic Units. *IEEE Transactions on Circuits and Systems-II:Express Briefs*, 54(8):685–689, 2007.
- [9] M.S. Schmookler and K.I. Nowka. Leading Zero Anticipation and Detection – A Comparison of Methods. In *Proc. 15th IEEE Symposium on Computer Arithmetic*, pages 7–12. IEEE Computer Society, 2001.
- [10] Ge Zhang, Wei-Wu Hu, and Zi-Chu Qi. Parallel Error Detection for leading Zero Anticipation. *J. Comput. Sci. & Technol.*, 21(6):901–906, 2006.



Peter Kornerup received the Mag. Scient. degree in mathematics from Aarhus University, Denmark, in 1967. After a period with the University Computing Center, from 1969 involved in establishing the computer science curriculum at Aarhus University, he helped found the Computer Science Department there in 1971 and served as its chairman until in 1988, when he became Professor of Computer Science at Odense University, now University of Southern Denmark. He spent a leave during 1975/76 with the University of Southwestern Louisiana, Lafayette, LA; four months in 1979 and shorter stays in many years with Southern Methodist University, Dallas, TX; one month with Universite de Provence in Marseille in 1996, and two months with Ecole Normale Supérieure de Lyon in 2001 and further visits in the following years. Prof. Kornerup has served on program committees for numerous IEEE, ACM and other meetings, in particular he has been on the Program Committees for the 4th through the 19th IEEE Symposium on Computer Arithmetic, and served as Program Co-Chair for these symposia in 1983, 1991, 1999 and 2007. He has been guest editor for a number of journal special issues, and served as an associate editor of these Transactions from 1991 to 1995.