Leading-One Prediction with Concurrent Position Correction

Javier D. Bruguera, *Member*, *IEEE*, and Tomás Lang, *Member*, *IEEE Computer Society*

Abstract—This paper describes the design of a leading-one prediction (LOP) logic for floating-point addition with an exact determination of the shift amount for normalization of the adder result. Leading-one prediction is a technique to calculate the number of leading zeros of the result in parallel with the addition. However, the prediction might be in error by one bit and previous schemes to correct this error result in a delay increase. The design presented here incorporates a concurrent position correction logic, operating in parallel with the LOP, to detect the presence of that error and produce the correct shift amount. We describe the error detection as part of the overall LOP, perform estimates of its delay and complexity, and compare with previous schemes.

Index Terms—Floating-point addition, normalization, leading-one prediction.

1 Introduction

Leading-one prediction is used in floating-point adders to eliminate the delay of the determination of the leading-one position of the adder output from the critical path. This determination is needed to perform the normalization of the result. Since the latency of floating-point addition is significant in many applications, this prediction might be of practical importance.

The direct way to perform the normalization is illustrated in Fig. 1a. Once the result has been computed, the Leading-One Detector (LOD)[1] counts and codes the number of leading zeros and, then, the result is left shifted. However, this procedure can be too slow since it is necessary to wait until the result is computed to determine the shift amount. Alternatively, as shown in Fig. 1b, the normalization shift can be determined in parallel with the significands addition. The Leading-One Predictor (LOP)[2] anticipates the amount of the shift for normalization from the operands. Once the result of the addition is obtained, the normalization shift can be performed since the shift has been already determined. This approach has been used in some recent floating-point unit designs and commercial processors [3], [7], [8], [9], [11], [17], [18].

As described below, the basic schemes developed for the leading-one predictor give the position with a possible error of one bit. Because of this, a second step consists of detecting and correcting this error, but this step increases the overall delay. To avoid this delay increase, we propose a correction procedure which detects the error in parallel with the determination of the position so that the correction can be performed concurrently with the first stage of the shifter. The evaluation and comparison presented show that it is plausible that this can be achieved in a specific implementation, both for the single datapath and the double datapath cases.

1. The LOD is also called LZD (Leading Zero Detector).
2. The LOP is also called LZA (Leading Zero Anticipator).

• *J.D. Bruguera is with the Department of Electronic and Computer Engineering, University of Santiago de Compostela, 15706 Santiago de Compostela, Spain. E-mail: bruguera@dec.usc.es.*
• *T. Lang is with the Department of Electrical and Computer Engineering, University of California at Irvine, Irvine, CA 92697. E-mail: tomas@ece.uci.edu.*

Manuscript received 29 June 1998; revised 7 Dec. 1998.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 107085.

1.1 Previous Work

Several LOPs have been proposed in the literature [4], [16], [18]. They consist of two parts: a basic LOP that determines the position of the leading one with an error of one bit and a module to detect the error and correct it. This detection and correction can be done after the massive left shift or concurrently. Moreover, the basic LOP can be for the case in which the result of the subtraction is always positive or for the more general case of arbitrary sign. We describe first a scheme that has no concurrent correction and is only applicable to the positive case and then discuss two more general proposals.

The LOP described in [18] has the general structure shown in Fig. 2a. As described in detail in Section 2, the pre-encoding examines pairs of bits of the operands and produces a string of zeros and ones, with the leading one in the position corresponding to the leading one of the addition result. This string is used by the LOD to produce an encoding of the leading-one position. Because of the characteristics of the pre-encoding, the resulting leading-one position might have an error of one bit. Therefore, it is necessary to correct the error by an additional one bit left shift, called the compensate shift and performed after the basic normalization shift. This compensate shift increases the delay of the floating-point addition. The design in [18] is performed for the case in which, during the alignment step, the operands are compared and swapped so that the result of the subtraction is always positive. This simplifies the implementation of the adder and of the LOP.

In [4] and [16], LOPs for positive or negative result are described. As in the previous scheme, these LOPs have the possibility of a wrong prediction by one position. They include a concurrent correction based on carry checking,

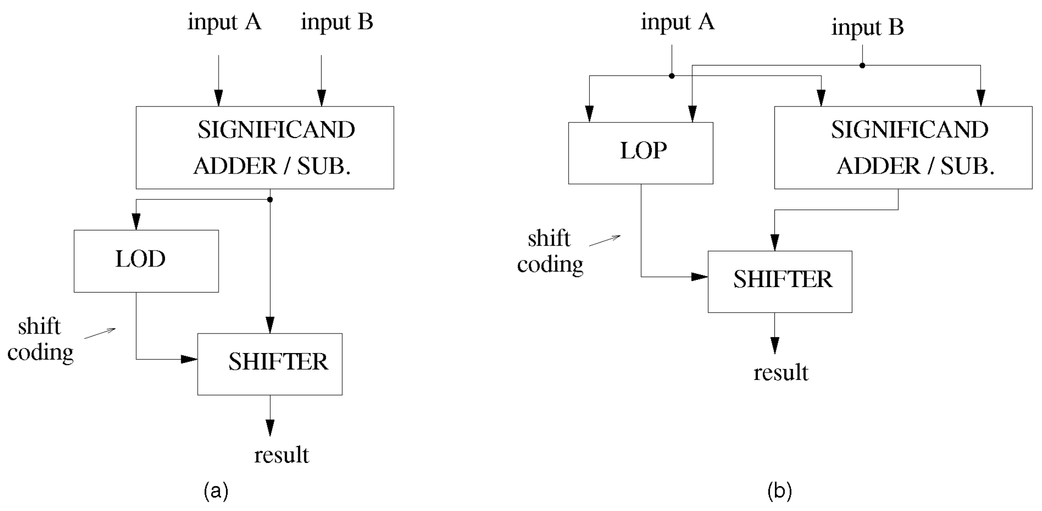


Fig. 1. Magnitude addition and normalization for a floating-point adder unit. (a) Without LOP. (b) With LOP.

whose general structure is shown in Fig. 2b. To perform the correction, the carry in the adder going into the anticipated leading-one position is checked and the position is corrected according to the carry value. The correction is done in the last stage of the normalization shift. Therefore, in principle, the correction does not increase the delay. However, as we show in Section 5, the carry selection is slow so that it introduces an additional delay in the floating-point addition.

1.2 Contribution

The main contribution of this paper is to propose and evaluate a method to perform the correction to the one position error of the basic LOP during the normalization shift without producing any delay degradation. This is achieved by detecting the error condition concurrently with the basic LOP (and, therefore, with the significands adder). We describe the development of the detection and correction scheme in a systematic way. Since this description has

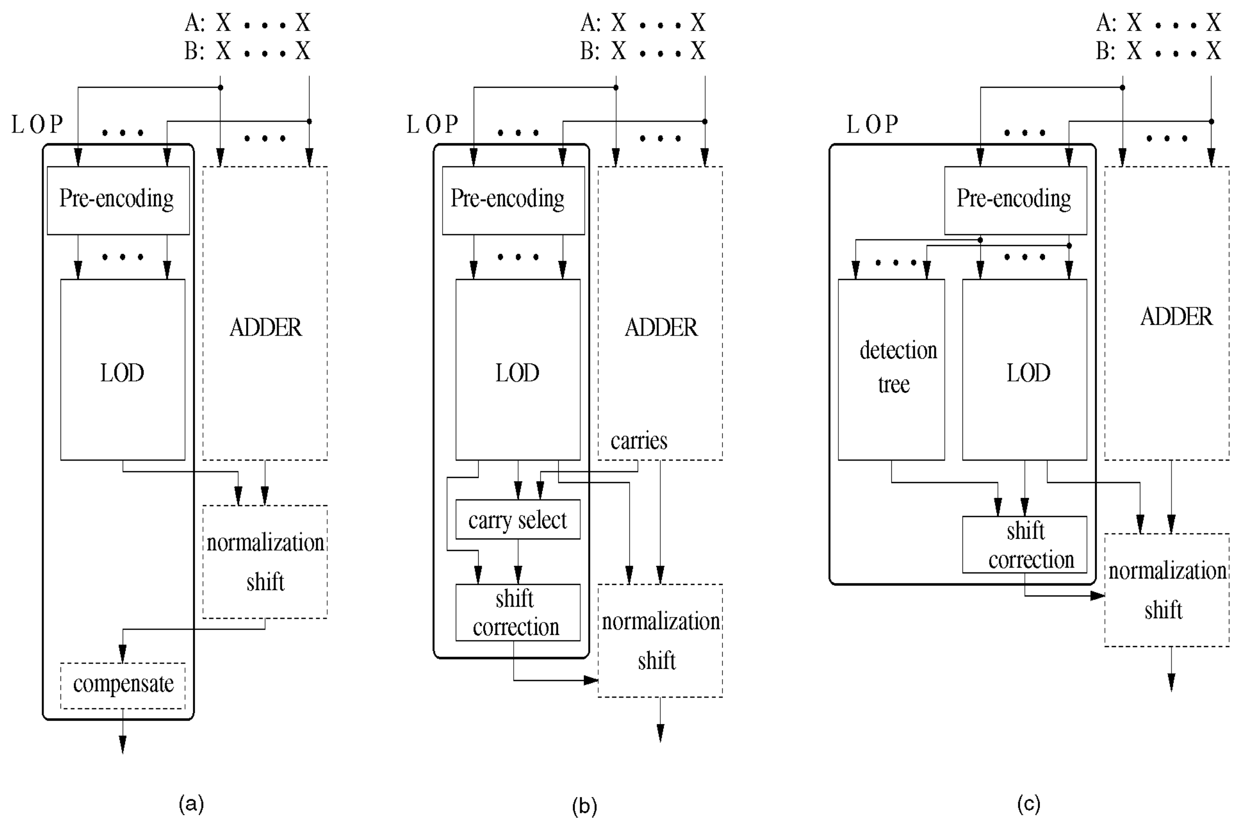


Fig. 2. LOP architectures. (a) Without concurrent correction. (b) With concurrent correction based on carry checking. (c) With concurrent correction based on parallel detection tree.

much in common with the description of the basic LOP, we also include the latter.

Our approach (Fig. 2c) to the basic LOP is similar to that of [18], extended to the case in which the output of the adder can be positive or negative (a version for the case in which the operands to the adder are previously compared so that the result of the subtraction is always positive is described in [2]). It is based on the location of some bit patterns producing the leading-one and the binary coding of its position by means of a binary tree. Moreover, we include another pre-encoding and trees to detect the occurrence of an error in the basic LOP. The output of these trees is then used to correct the output of the LOD so that the correct shift is performed. Since the detection and correction can be performed before the last stage of the normalization shift, the delay of the addition is not increased.

Since almost all the floating-point processors [6], [10], [13] use the IEEE standard [1], we consider the case of sign-and-magnitude representation of the operands. The paper is organized as follows: In Section 2, the structure of the LOP is presented. After that, the different modules of the LOP are described: the leading-one position encoding in Section 3 and the concurrent position correction in Section 4. Then, in Section 5, our design is evaluated and compared with other LOPs. Finally, in Section 6, the effect on the floating-point addition latency is discussed.

2 Leading-One Prediction with Concurrent Correction: General Structure

We now give an overview of the structure of the leading-one predictor we propose. Then, in the following sections, we consider individual modules. As stated in the introduction, the two significands are in sign-and-magnitude and the LOP is only applicable and needed when the effective operation is a subtraction. As shown in Fig. 1b, the LOP predicts the position of the leading-one in the result, in parallel with the subtraction of significands.

The LOP operates on the significands after alignment. We denote by $A = a_0 \dots a_{m-1}$ and $B = b_0 \dots b_{m-1}$ these (positive) significands, a_0 and b_0 being the most significant bits ($A = \sum_{i=0}^{m-1} a_i \cdot 2^{-i}$ and $B = \sum_{i=0}^{m-1} b_i \cdot 2^{-i}$), and the operation to be performed by the magnitude adder is $|A - B|$.[3] We develop an LOP for the general case in which either $A \geq B$ or $A < B$.

As shown in Fig. 3, the LOP is divided into two main parts: the **encoding** of the leading-one position and the **correction** of this position. Moreover, these parts are composed of the following components:

Encoding

- A *pre-encoding module* that provides a string of zeros and ones, with the most-significant 1 defining the leading-one position. After this leading one, it is immaterial what the rest of the string is.
- An *encoding tree* (also called leading-one detector (LOD)) to encode the position of the most-significant 1 to drive the shifter for normalization. In addition, the bit V indicates when the result is 0.

Correction

- A *pre-encoding module* providing a string of symbols that is used to determine whether a correction is needed. As indicated in Fig. 3, there is significant commonality between both pre-encoding modules.
- A *detection tree* to determine whether the position indicated by the encoding tree has to be corrected (incremented by one).
- A *correction module* that performs the correction, if necessary, in parallel with the operation of the normalization shifter.

In Sections 3 and 4, we describe these two parts and the corresponding modules and trees.

3 Position Encoding

3.1 Pre-Encoding Module

This module produces a string of zeros and ones. As a first step in the production of this string, we perform a radix-2 signed-digit subtraction of the significands. We obtain

$$W = A - B.$$

This operation is done on each bit slice (without carry propagation), that is,

$$w_i = a_i - b_i \quad with \;\; w_i \in \{-1, 0, 1\}.$$

For clarity, the -1 will be represented as $\bar{1}$.

We now consider the string W to determine the position of the leading one. Formally, the determination of this position requires a conversion of the signed-digit representation into a conventional binary representation. However, as we see now, this conversion is not actually required. To simplify the discussion, we consider separately the cases $W > 0$, $W < 0$, and $W = 0$. The notation used throughout the paper is the following: x denotes an arbitrary substring and 0^k, 1^k, and $\bar{1}^k$ denote strings of k 0s, 1s, and $\bar{1}$s, respectively, with $k \geq 0$.

Case $W > 0$. Since $W > 0$, the first digit w_i different from 0 has to be equal to 1. Therefore, the w string is $0^k 1(x)$. For the substring (x), two situations can be identified as follows:

S1 The digit of W following the first 1 is either 0 or 1. In this case, the leading one is located in position $k+1$ or $k+2$. This is shown by considering two cases:

1. The digit following the first 1 is 1. That is,

$$W = 0^k 11(x).$$

Clearly, the conversion of W to conventional representation has a 1 in position $k+1$, the position of the first 1, since any borrow produced by a negative x is absorbed by the 1 at position $k+2$. In this situation, a leading one in position i is identified by the substring

3. Note that we consider only the positive aligned significands and do not consider the signs of the floating-point operands.

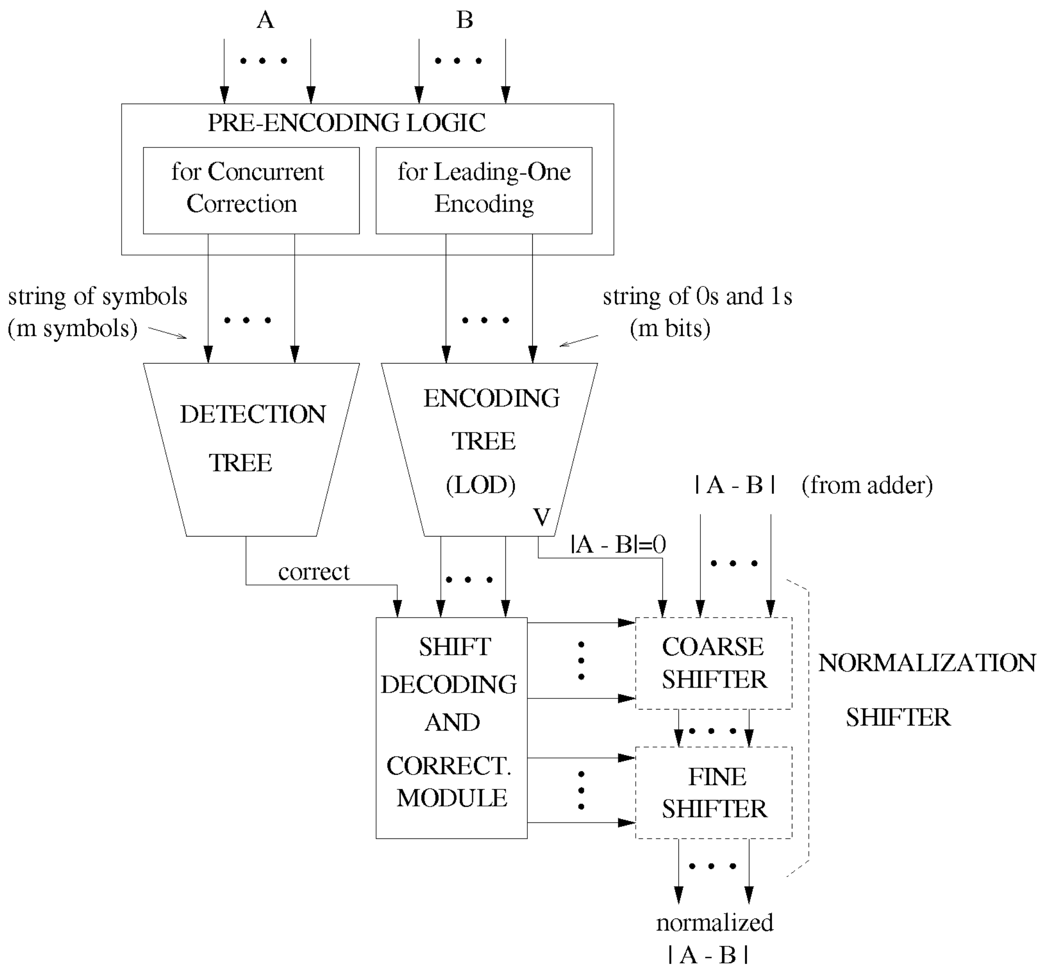


Fig. 3. General structure of the proposed LOP.

$$w_i w_{i+1} = 11.$$

2. The digit following the first 1 is 0. That is,

$$W = 0^k 10(x).$$

Now, two possibilities exist with respect to x, namely,

- (x) is positive (or zero). That is,

$$W = 0...010000...1...$$

The position of the leading one is $k+1$ since there is no borrow from (x).

- (x) is negative. The position of the leading one is $k+2$ because of the borrow produced by (x). That is,

$$\begin{aligned} W &= 0...010000...\bar{1}... \\ &= 0...001111...1... \end{aligned}$$

The problem with this situation is that it is not possible to detect it by inspecting a few digits of W since it depends on the number of zeros before the $\bar{1}$. Consequently, we assume that the position is $k+1$ and correct later.

The leading 1 in position i is identified by the substring

$$w_i w_{i+1} = 10.$$

In summary, for **S1**, the leading one in position i is identified by the substrings

$$w_i w_{i+1} = 11 \textit{ or } 10 = 1(\textit{not } \bar{1}).$$

S2 The first 1 of W is followed by a string of $\bar{1}$s. That is,

$$W = 0^k 1 \bar{1}^j (0,1)(x)$$

which, after conversion, results in

$$W = 0 \dots 0 \dots 01(0,1)(x)$$

If the string of $\bar{1}$s is of length j, the position of the leading 1 is $k+j+1$ or $k+j+2$, depending on a similar situation as in **S1**. Consequently, using the same approach, we assume that the position is $k+j+1$ and correct later. A leading one in position i is identified by substrings

$$w_i w_{i+1} = \bar{1}1 \textit{ or } \bar{1}0 = \bar{1}(\textit{not } \bar{1}).$$

This discussion is summarized in Table 1. Those are all the possible patterns. Other patterns are not possible

TABLE 1
Leading-One Position for $W > 0$

Bit Pattern	Leading-one Position		Substring
$0^k11(x)$	First 1	$k+1$	11
$0^k10\{\text{positive or zero}\}$	First 1	$k+1$	10
$0^k10^j\{\text{negative}\}$	First 0 after the 1	$k+2$	10*
$0^k1\bar{1}^j1(x)$	Last $\bar{1}$ of the string	$k+j+1$	$\bar{1}1$
$0^k1\bar{1}^j0\{\text{positive or zero}\}$	Last $\bar{1}$ of the string	$k+j+1$	$\bar{1}0$
$0^k1\bar{1}^j0^t\{\text{negative}\}$	First 0 after the last $\bar{1}$	$k+j+2$	$\bar{1}0$*

* correction needed

because of the positive result. By combining the S1 and S2 cases, the leading-one position in determined by the substrings

$$w_iw_{i+1} = 1(not\ \bar{1})\ or\ \bar{1}(not\ \bar{1}).$$

Case $W < 0$. The same analysis can be extended to determine the leading one position when $W < 0$. This is achieved by exchanging the role of 1 and $\bar{1}$ in the $W > 0$ case. Therefore, the leading-one position is identified by the following substrings:

$$w_iw_{i+1} = \bar{1}(not\ 1)\ or\ 1(not\ 1).$$

Case $W = 0$. In this case, there is no leading one. The encoding tree will provide a signal indicating this situation. Therefore, it is immaterial what the encoding is.

3.1.1 String to Identify the Leading-One Position

We now produce the string of zeros and ones which has as first one the leading-one. We call this string $F = f_0 \dots f_{m-1}$. The corresponding bit of F is obtained by combining the substrings described before. To simplify the description of F, the values of digit w_i equal to 1, 0, or $\bar{1}$, are now called g_i, e_i, or s_i, respectively. That is, for each bit position of the input operands, the following functions are defined:

$$\begin{aligned} e_i &= 1\ if\ a_i = b_i\ (w_i = 0)\\ g_i &= 1\ if\ a_i > b_i\ (w_i = 1)\\ s_i &= 1\ if\ a_i < b_i\ (w_i = \bar{1}) \end{aligned} \quad (1)$$

with $0 \le i \le m-1$. With this notation the substrings are

$$f_i(pos) = g_i\bar{s}_{i+1} + s_i\bar{s}_{i+1} \quad \text{for } W > 0 \quad (2)$$

$$f_i(neg) = s_i\bar{g}_{i+1} + g_i\bar{g}_{i+1} \quad \text{for } W < 0. \quad (3)$$

Figs. 4a and 4b show examples of the computation of $F(pos)$ and $F(neg)$ according to (2) and (3). It would be possible now to use both strings $F(pos)$ and $F(neg)$ to encode the position of the leading one in separate LODs and to choose between them when the sign is known. However, it is more efficient to combine both strings and have a single LOD.[4] The simplest way to combine them would be to OR the two expressions. However, this produces an incorrect result because, for instance, a 1 of $f_i(neg)$ can signal a leading-one position that is not correct for a positive W. An sample of this is given in Fig. 4c.

Because of the above-mentioned problem, we also use w_{i-1} in the substrings that are ORed to produce the combined F. From Section 3.1, we see that the substring $w_iw_{i+1} = 1(not\ \bar{1})$ identifies the leading one only when $w_{i-1} = 0$. Similarly, the substring $w_iw_{i+1} = \bar{1}(not\ \bar{1})$ identifies the position when $w_{i-1} \neq 0$. Consequently, the extended expression is

$$e_{i-1}g_i\bar{s}_{i+1} + \bar{e}_{i-1}s_i\bar{s}_{i+1}.$$

Similarly, for the negative string,

$$e_{i-1}s_i\bar{g}_{i+1} + \bar{e}_{i-1}g_i\bar{g}_{i+1}.$$

Combining both equations, we obtain

$$f_i = e_{i-1}(g_i\bar{s}_{i+1} + s_i\bar{g}_{i+1}) + \bar{e}_{i-1}(s_i\bar{s}_{i+1} + g_i\bar{g}_{i+1}). \quad (4)$$

An example of the calculation of string F is given in Fig. 4d. Note that, for the case $W = 0$, we obtain all $f_i = 0$.

We postpone the description of the implementation of this module until we have also discussed the pre-encoding for the concurrent correction since these modules share components.

3.2 Encoding Tree

Once the string F has been obtained, the position of the leading one of F has to be encoded by means of a LOD tree [14]. The LOD provides two outputs: a bit V indicating if there is some 1 in F and the binary encoding P of the leading-one position. As shown later, the position is decoded to feed the normalization shifter. In case $W = 0$ ($F = 0$), we obtain the final $V = 0$. This indicates that the result of the addition is 0.

4 Concurrent Position Correction

As explained in Section 3, the position of the leading one predicted from the input operands has one bit error for the following patterns of W:

1. $0^k10^j\bar{1}(x)$ and $0^k1\bar{1}^j0^t\bar{1}(x)$ for $W > 0$.
2. $0^k\bar{1}0^j1(x)$ and $0^k\bar{1}1^j0^t1(x)$ for $W < 0$.

In these cases, the position has to be corrected by adding 1 to the encoding calculated in the tree. Therefore, the concurrent position correction has two steps: 1) detection of when it is necessary to correct and 2) correction of the

4. In contrast, as will be discussed in Section 4, we will use two separate strings for the detection of the pattern for correction.

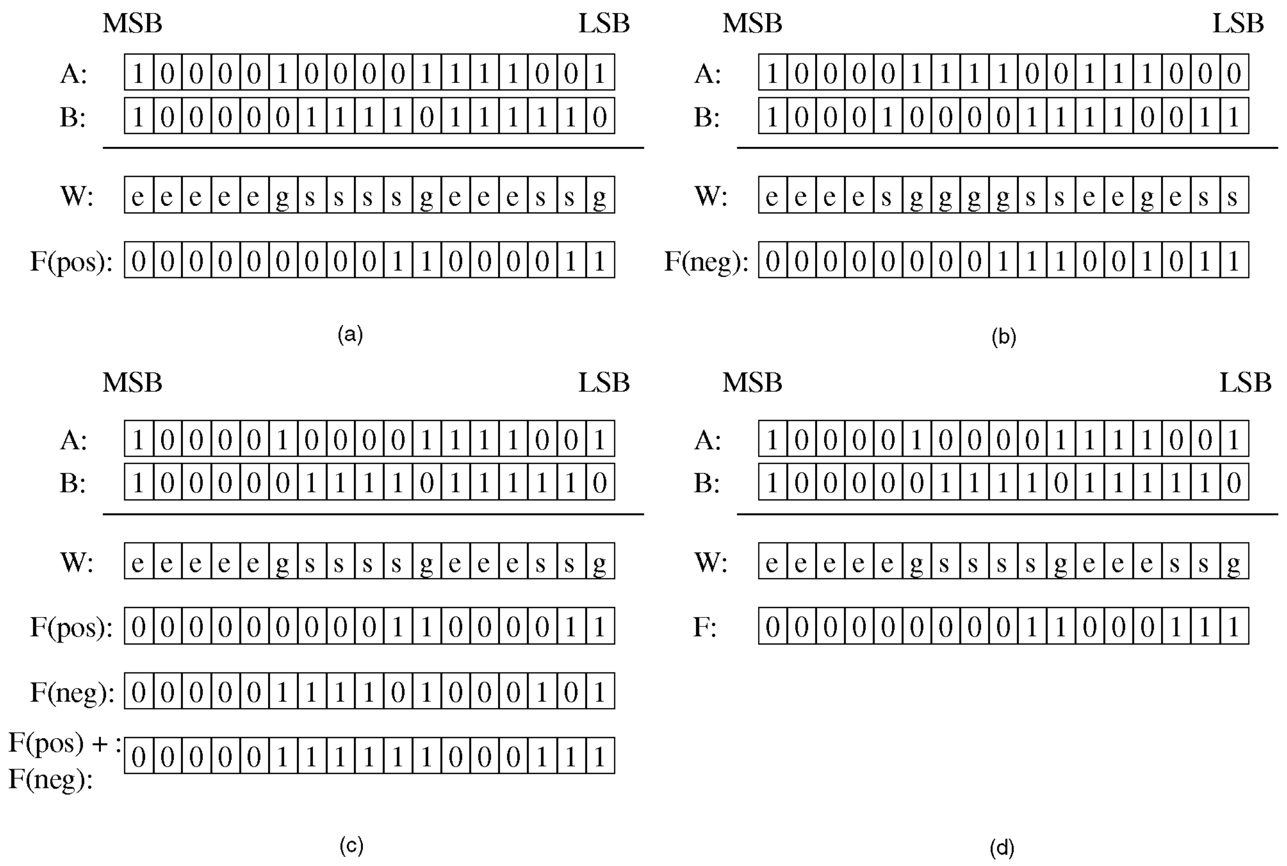

MSB LSB
A: 1 0 0 0 0 1 0 0 0 0 1 1 1 1 0 0 1
B: 1 0 0 0 0 0 1 1 1 1 0 1 1 1 1 1 0
W: e e e e e g s s s s g e e e s s g
F(pos): 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 1 1
(a)
MSB LSB
A: 1 0 0 0 0 1 1 1 1 0 0 1 1 1 0 0 0
B: 1 0 0 0 1 0 0 0 0 1 1 1 1 0 0 1 1
W: e e e e s g g g g s s e e g e s s
F(neg): 0 0 0 0 0 0 0 0 1 1 1 0 0 1 0 1 1
(b)
MSB LSB
A: 1 0 0 0 0 1 0 0 0 0 1 1 1 1 0 0 1
B: 1 0 0 0 0 0 1 1 1 1 0 1 1 1 1 1 0
W: e e e e e g s s s s g e e e s s g
F(pos): 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 1 1
F(neg): 0 0 0 0 0 1 1 1 1 0 1 0 0 0 1 0 1
F(pos) + F(neg): 0 0 0 0 0 1 1 1 1 1 1 0 0 0 1 1 1
(c)
MSB LSB
A: 1 0 0 0 0 1 0 0 0 0 1 1 1 1 0 0 1
B: 1 0 0 0 0 0 1 1 1 1 0 1 1 1 1 1 0
W: e e e e e g s s s s g e e e s s g
F: 0 0 0 0 0 0 0 0 0 1 1 0 0 0 1 1 1
(d)

Fig. 4. Computation of the intermediate encoding: (a) F(pos), (b) F(neg), (c) F(pos) + F(neg), (d) Combined F.

position encoding. The first step is carried out in parallel with the leading-one encoding and the second one with the normalization shifting.

4.1 Detection

Fig. 5 shows the general scheme for the detection of a correction pattern. As explained before, the detection is performed by two modules: a pre-encoding module and a detection tree. In the pre-encoding logic, two different strings are obtained: G_p is used to detect the presence of a positive correction pattern (case $W > 0$) and G_n to detect a negative correction pattern (case $W < 0$). As done for the position encoding of the previous section, it is possible to combine both strings and have one tree to detect both types of patterns. However, we have found that this substantially complicates the tree so that we have opted to use two different trees: G_p is processed by the positive tree and G_n by the negative tree. We now describe these modules.

4.1.1 Pre-Encoding Module

For this pre-encoding, we use the W string obtained before. Two new encodings are constructed to carry out the detection, G_p to detect a positive correction pattern and G_n to detect a negative one. In both cases, it is necessary to distinguish between the digit values 1 and $\bar{1}$. Therefore, digits in the string G_p and G_n can take values $\{-1, 0, 1\}$. To simplify the notation, we use n, z, and p for $\bar{1}$, 0, and 1, respectively.

Let us consider $W > 0$ first. Fig. 6 shows all the possible patterns for W. To detect the two patterns of W, we construct the string G_p, in such a way that the pattern

$$z^k p z^q n \ (x)$$

appears in G_p when correction is needed (Figs. 6c and 6f). Since what we need to detect is the pattern consisting of the leading one (encoded in $F(p)$) followed by zeros and terminating in a $\bar{1}$, we do as follows:

- Use the $F(p)$ string described by (2) of Section 3. This will give us the leading one followed by zeros.
- For the combination $w_{i-1}w_i = 0\bar{1}$, make $G_i = n$. This will give us the position of the $\bar{1}$.

The resulting relation between substrings of W and digits of G_p is shown in Table 2a. Note that, for substrings $w_{i-1}w_iw_{i+1} = 0\bar{1}0$ or $w_{i-1}w_iw_{i+1} = 0\bar{1}1$, both p and n of G_p are set. According to the previous discussion, these cases have to be interpreted as n. This interpretation is performed by the positive detection tree. Fig. 6 indicates, for each pattern of W, the corresponding string G_p. As can be seen, G_p has the pattern $z^k p z^q n$ only for the cases in which correction is needed.

Similarly, we construct G_n in such a way that pattern $z^k n z^q p \ (x)$ appears when correction is needed. Table 2b shows the relation between W and the digits of G_n.

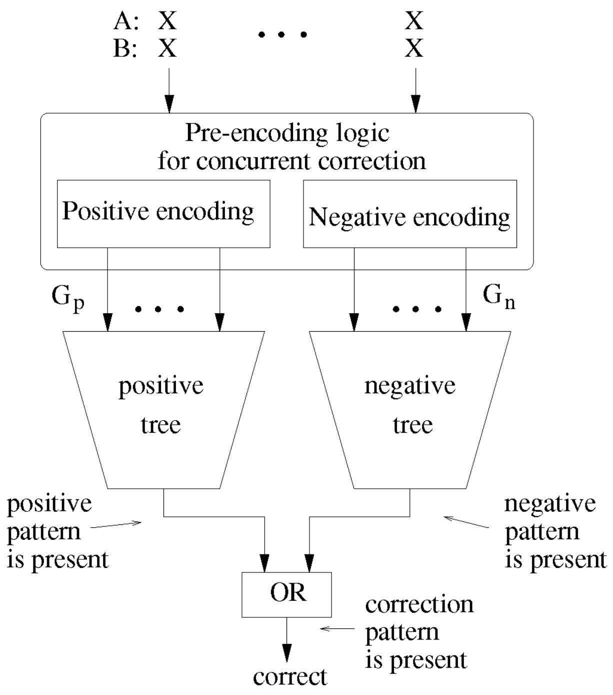

A: X
B: X
X
X
Pre-encoding logic for concurrent correction
Positive encoding
Negative encoding
Gp
Gn
positive tree
negative tree
positive pattern is present
negative pattern is present
OR
correction pattern is present
correct

Fig. 5. Detection of a correction pattern.

4.1.2 *Implementation*

The implementation of the pre-encoding module is shown in Fig. 7. It implements the expressions for F, G_p, and G_n, namely,

- For F $f_i = e_{i-1}(g_i\bar{s}_{i+1} + s_i\bar{g}_{i+1}) + \bar{e}_{i-1}(s_i\bar{s}_{i+1} + g_i\bar{g}_{i+1})$
- For G_p $p_i = (g_i + s_i)\bar{s}_{i+1}$ $n_i = e_{i-1}s_i$ $z_i = \overline{p_i + n_i}$
- For G_n $n_i = (g_i + s_i)\bar{g}_{i+1}$ $p_i = e_{i-1}g_i$ $z_i = \overline{p_i + n_i}$

Note that $(g_i + s_i)$ is used instead of $\bar{e}_i$ to reduce the delay of the pre-encoding logic.

4.1.3 *Detection Tree*

To detect if one of the two patterns for correction is present, a binary tree can be used, the input to the tree being the intermediate encoding G. However, if a single tree is used to detect the *positive* pattern (pz^kn) and the *negative* one (nz^kp), the number of values of each node of the tree would be large, resulting in a complex and slow hardware implementation. Therefore, we propose to use two different trees, one to detect the positive pattern (positive tree) and the other to detect the negative pattern (negative tree). As shown in Fig. 5, these two trees operate in parallel, but if one of the patterns is present, only the corresponding tree will detect it.

4.1.4 *Positive Tree*

The positive tree receives as input the string G_P and has to detect if the pattern $z^kpz^qn(x)$ is present. A node of the tree has five possible values, Z, P, N, Y, and U, representing the following substrings:

$$\begin{aligned} &Z \Rightarrow z^j \\ &P \Rightarrow z^jpz^t \\ &N \Rightarrow z^jn(x) \\ &Y \Rightarrow z^jpz^kn(x) \\ &U \Rightarrow \text{other strings,} \end{aligned} \tag{5}$$

where P and N indicate, respectively, that the first and the last part of the pattern have been found, Y indicates that the whole pattern has been detected, and U indicates a string incompatible with the pattern.

Each node of the tree receives as input the output from two nodes of the preceding level and produces the combined value. Fig. 8a illustrates how the nodes of different level are combined and Table 3a shows the function table of a node of the tree. The left input of the node is represented in the first column of the table and the right input in the first row. Output Y is the result of the combination of a left P and a right N value. Once the Y value has been set, the combination with any other right input results in Y since, once the string has been found in the most-significant digits of the string, the least-significant digits have no effect. Fig. 8b shows an example of the detection of the pattern.

Note that, if the first digit different from z in G_p is n, that is, we are examining a negative W string with the positive tree, then the value obtained as output in the last level of the tree will be N.

For a simple implementation, we encode the five values with four variables and assign code 0000 to value U. With this encoding, the logic equations are:

$$\begin{aligned} Z &= Z^lZ^r \\ P &= Z^lP^r + P^lZ^r \\ N &= N^l + Z^lN^r \\ Y &= Y^l + Z^lY^r + P^lN^r, \end{aligned} \tag{6}$$

where $T^l = (Z^l, P^l, N^l, Y^l)$ and $T^r = (Z^r, P^r, N^r, Y^r)$ represent the left input and the right input, respectively.

4.1.5 *Negative Tree.*

The negative tree is obtained by exchanging the role of P and N in the positive tree. It receives as input the G_n string. The node function is shown in Table 3b. Similarly to the positive tree, if a positive W string is processed, the final value obtained is P.

Implementation. The hardware implementation of the nodes in the positive tree (6) and the negative tree is shown in Fig. 9.

4.2 Correction of the Normalization Shift

The last step in the LOP we propose is the correction of the leading-one position, which is performed by incrementing by one the shift amount. As done in [18], to reduce the delay of the shifter it is convenient to decode the shift amount in parallel with the adder (if there is sufficient time). Moreover, because of implementation constraints, the shifter has more than one stage. As shown in Fig. 10, the stages are organized from the coarsest to the finest. This last one performs a shift by one of several contiguous positions, say from 0 to k_f binary positions. As indicated in this figure, we

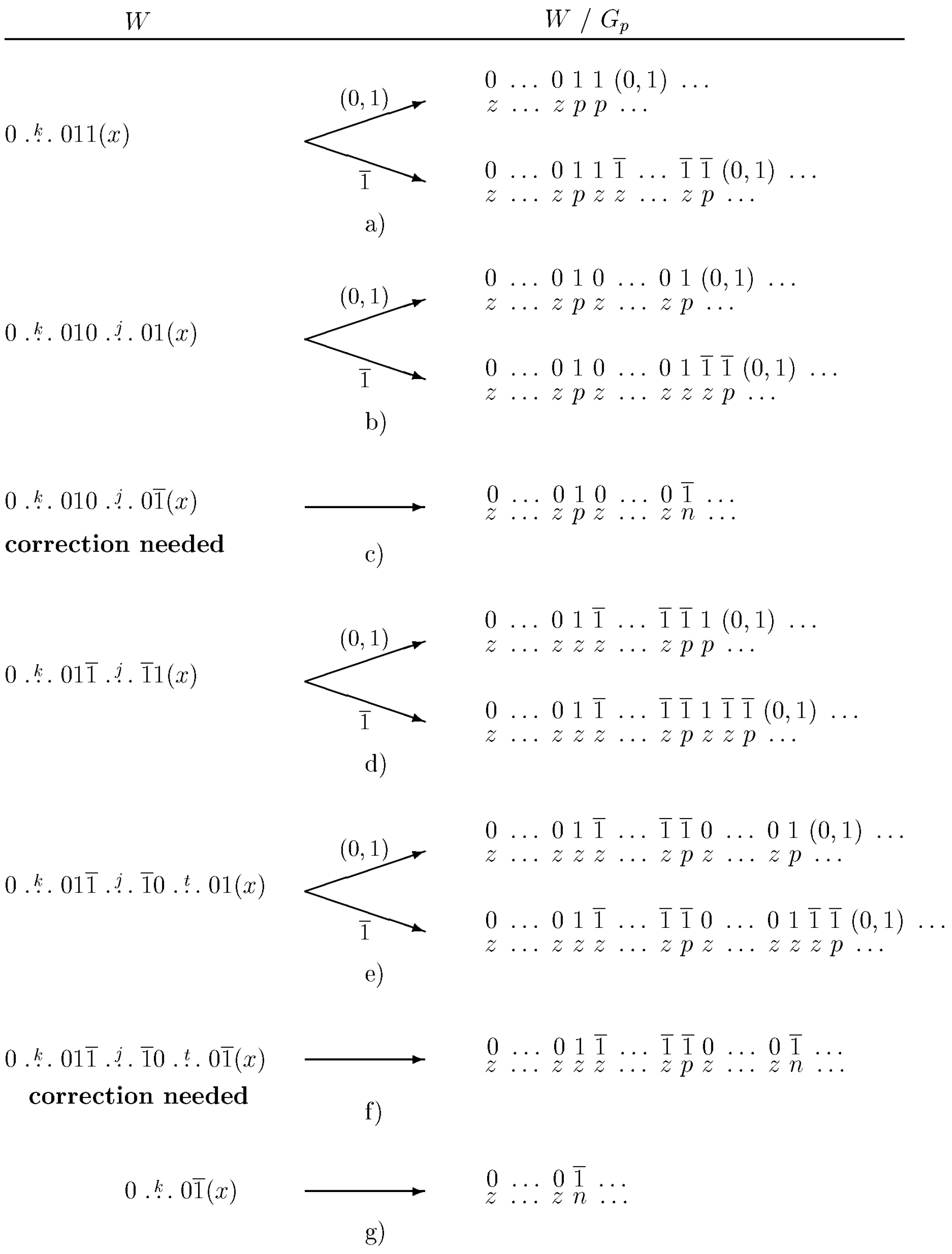

W
$W\ /\ G_p$
$0\ \overset{k}{\ldots}\ 011(x)$
$(0,1)$
$0\ \ldots\ 0\ 1\ 1\ (0,1)\ \ldots$
$z\ \ldots\ z\ p\ p\ \ldots$
$\bar{1}$
$0\ \ldots\ 0\ 1\ 1\ \bar{1}\ \ldots\ \bar{1}\ \bar{1}\ (0,1)\ \ldots$
$z\ \ldots\ z\ p\ z\ z\ \ldots\ z\ p\ \ldots$
a)
$0\ \overset{k}{\ldots}\ 010\ \overset{j}{\ldots}\ 01(x)$
$(0,1)$
$0\ \ldots\ 0\ 1\ 0\ \ldots\ 0\ 1\ (0,1)\ \ldots$
$z\ \ldots\ z\ p\ z\ \ldots\ z\ p\ \ldots$
$\bar{1}$
$0\ \ldots\ 0\ 1\ 0\ \ldots\ 0\ 1\ \bar{1}\ \bar{1}\ (0,1)\ \ldots$
$z\ \ldots\ z\ p\ z\ \ldots\ z\ z\ z\ p\ \ldots$
b)
$0\ \overset{k}{\ldots}\ 010\ \overset{j}{\ldots}\ 0\bar{1}(x)$
correction needed
$0\ \ldots\ 0\ 1\ 0\ \ldots\ 0\ \bar{1}\ \ldots$
$z\ \ldots\ z\ p\ z\ \ldots\ z\ n\ \ldots$
c)
$0\ \overset{k}{\ldots}\ 01\bar{1}\ \overset{j}{\ldots}\ \bar{1}1(x)$
$(0,1)$
$0\ \ldots\ 0\ 1\ \bar{1}\ \ldots\ \bar{1}\ \bar{1}\ 1\ (0,1)\ \ldots$
$z\ \ldots\ z\ z\ z\ \ldots\ z\ p\ p\ \ldots$
$\bar{1}$
$0\ \ldots\ 0\ 1\ \bar{1}\ \ldots\ \bar{1}\ \bar{1}\ 1\ \bar{1}\ \bar{1}\ (0,1)\ \ldots$
$z\ \ldots\ z\ z\ z\ \ldots\ z\ p\ z\ z\ p\ \ldots$
d)
$0\ \overset{k}{\ldots}\ 01\bar{1}\ \overset{j}{\ldots}\ \bar{1}0\ \overset{t}{\ldots}\ 01(x)$
$(0,1)$
$0\ \ldots\ 0\ 1\ \bar{1}\ \ldots\ \bar{1}\ \bar{1}\ 0\ \ldots\ 0\ 1\ (0,1)\ \ldots$
$z\ \ldots\ z\ z\ z\ \ldots\ z\ p\ z\ \ldots\ z\ p\ \ldots$
$\bar{1}$
$0\ \ldots\ 0\ 1\ \bar{1}\ \ldots\ \bar{1}\ \bar{1}\ 0\ \ldots\ 0\ 1\ \bar{1}\ \bar{1}\ (0,1)\ \ldots$
$z\ \ldots\ z\ z\ z\ \ldots\ z\ p\ z\ \ldots\ z\ z\ z\ p\ \ldots$
e)
$0\ \overset{k}{\ldots}\ 01\bar{1}\ \overset{j}{\ldots}\ \bar{1}0\ \overset{t}{\ldots}\ 0\bar{1}(x)$
correction needed
$0\ \ldots\ 0\ 1\ \bar{1}\ \ldots\ \bar{1}\ \bar{1}\ 0\ \ldots\ 0\ \bar{1}\ \ldots$
$z\ \ldots\ z\ z\ z\ \ldots\ z\ p\ z\ \ldots\ z\ n\ \ldots$
f)
$0\ \overset{k}{\ldots}\ 0\bar{1}(x)$
$0\ \ldots\ 0\ \bar{1}\ \ldots$
$z\ \ldots\ z\ n\ \ldots$
g)

Fig. 6. Patterns in the string G_p for $W > 0$ case.

TABLE 2
Relation between W and (a) G_p and (b) G_n

w_{i-1}	w_i	w_{i+1}	G_i
$--$	g	$\bar{s}$	p
$--$	s	$\bar{s}$	p
e	s	$--$	n
otherwise			z

(a)

w_{i-1}	w_i	w_{i+1}	G_i
$--$	s	$\bar{g}$	n
$--$	g	$\bar{g}$	n
e	g	$--$	p
otherwise			z

(b)

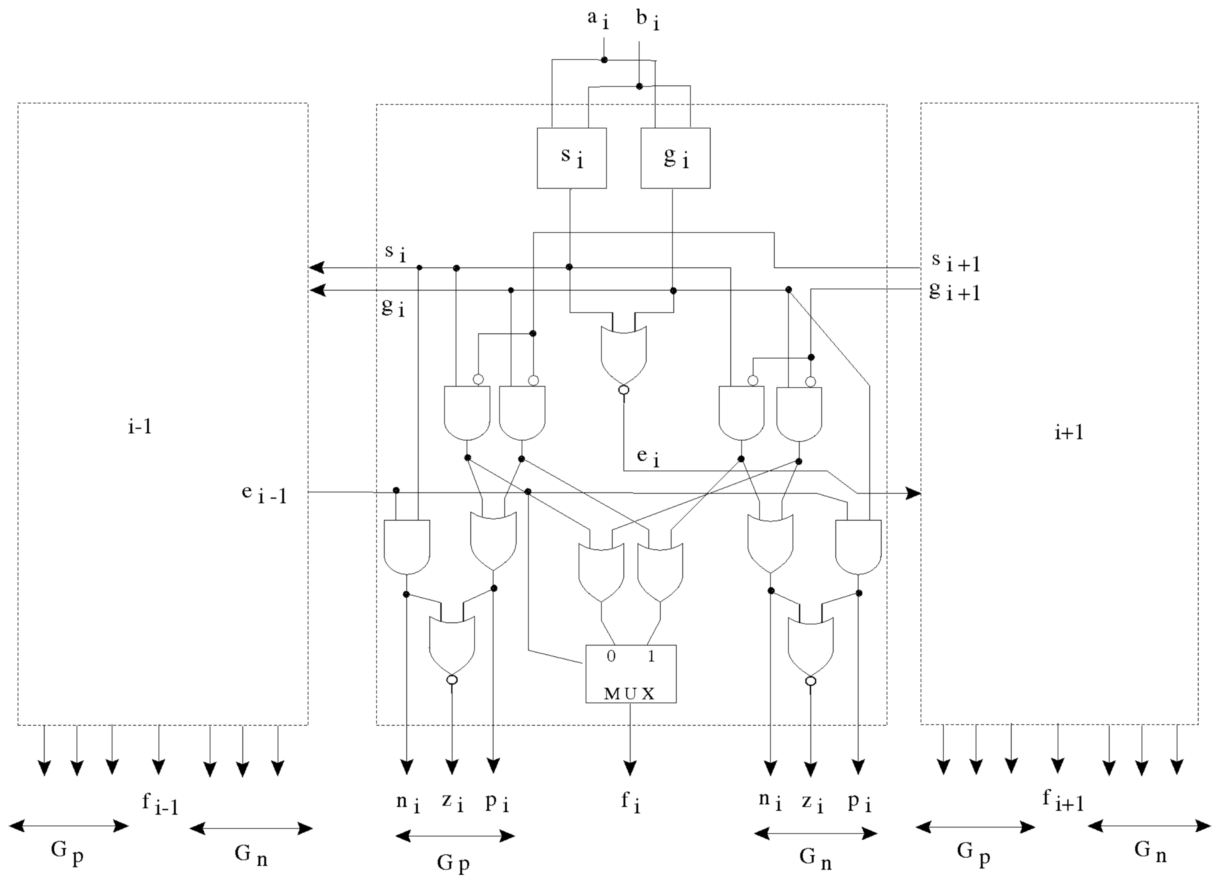
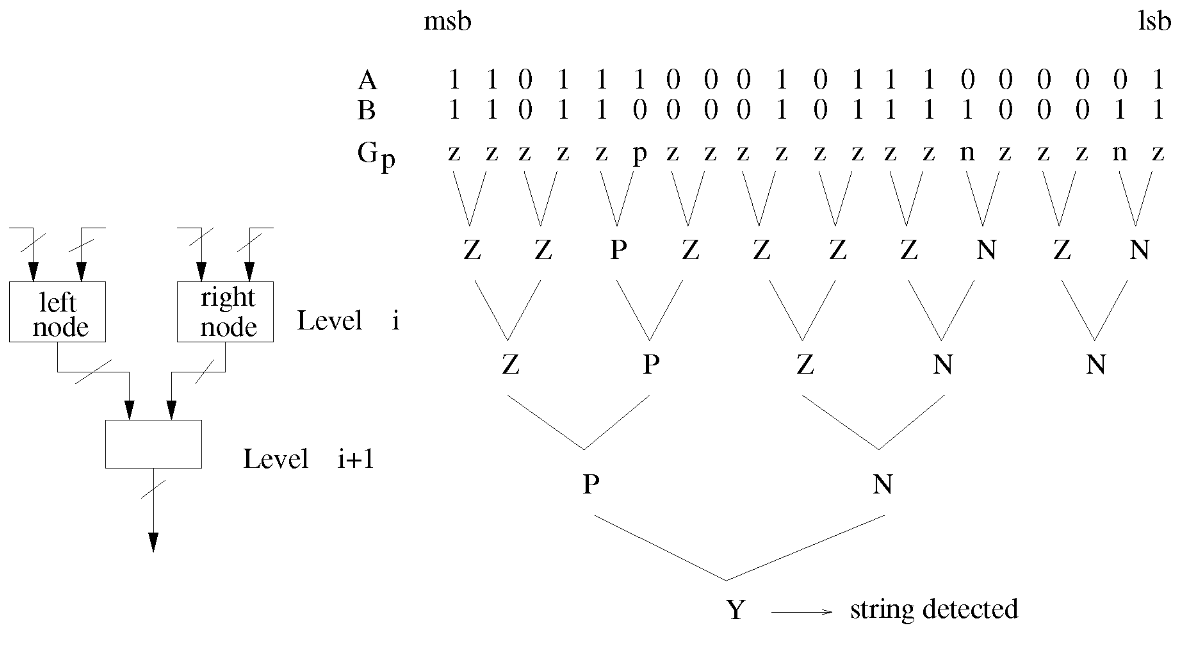


Fig. 7. Implementation of the pre-encoding logic.



left node
right node
Level i
Level i+1
msb
lsb
A 1 1 0 1 1 1 0 0 0 1 0 1 1 1 0 0 0 0 0 1
B 1 1 0 1 1 0 0 0 0 1 0 1 1 1 1 0 0 0 1 1
Gp z z z z z p z z z z z z z z n z z z n z
Z Z P Z Z Z Z N Z N
Z P Z N N
P N
Y string detected
(a)
(b)

Fig. 8. Binary tree to detect the correction pattern. (a) Connection between levels. (b) Example.

perform the correction at this last stage so that the shifter has to be modified to shift from 0 to $k_f + 1$ positions. This should have a negligible effect on the delay of the last stage. The relation between k_f and m, the length of the significands, depends on how the shifter is implemented. Notice that the selection between correction and

TABLE 3
Node Functions (a) for the Positive Tree and (b) for the Negative Tree

	Z	P	N	Y	U
Z	Z	P	N	Y	U
P	P	U	Y	U	U
N	N	N	N	N	N
Y	Y	Y	Y	Y	Y
U	U	U	U	U	U

(a)

	Z	P	N	Y	U
Z	Z	P	N	Y	U
P	P	P	P	P	P
N	N	Y	U	U	U
Y	Y	Y	Y	Y	Y
U	U	U	U	U	U

(b)

no-correction can be made in parallel with the previous stages of the shifter.

5 Evaluation and Comparison

In this section, the LOP architecture we propose is evaluated in terms of delay of the critical path and added hardware complexity. Then, we compare it with implementations of the two schemes discussed in Section 1.1, namely the LOP without concurrent correction and the LOP with concurrent correction based on carry checking.

The evaluation of the LOP and the comparison is performed at a *qualitative* level. A more detailed and accurate evaluation depends on a large number of technological factors and on the floating-point unit requirements.

5.1 Evaluation

Fig. 11a shows the general structure of the proposed LOP. The critical path and the hardware complexity of the added modules can be obtained from this block diagram and the implementations shown in the previous sections. From those implementations, we estimate that the delay of the adder is larger than the delay of the LOP (pre-encoding logic and LOD tree) and of the detection tree with its pre-encoding logic. With these considerations, we have estimated that the slowest path is the one going through the adder, coarse shifter and fine shifter, as shown in the figure.

5.2 Comparison

In this section, we compare our LOP architecture with concurrent correction with two other LOP alternatives: an LOP without concurrent correction and an LOP with a correction scheme based on the utilization of the addition carries. Their main characteristics are the following:

1. **LOP without concurrent correction** (Fig. 11b). This is an extension of the one described in [18] to the case in which the output of the adder can be either positive or negative. Since the LOD determines the position of the leading-one within an error of one bit, a compensation shift is included once the normalization has been performed.

   The only difference, in terms of critical path delay, with the proposed LOP with concurrent correction is that the compensation shifter is included in the critical path. The delay of this compensation shift is estimated in [18] to be 13 percent of the delay of addition plus shift. Consequently, this percentage is saved by the inclusion of the concurrent correction.

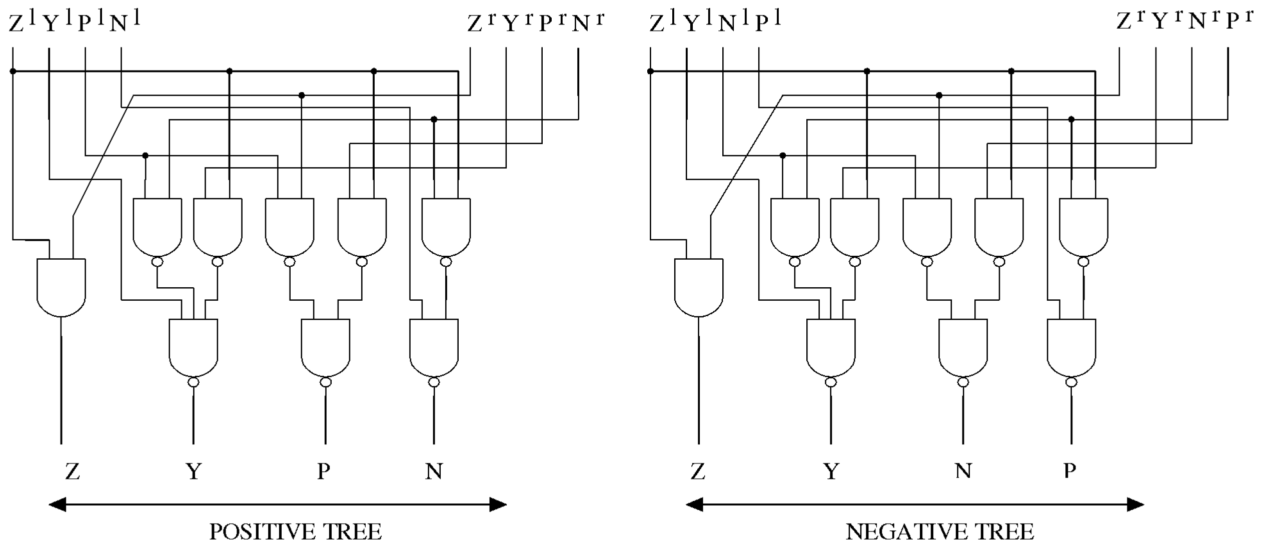


Fig. 9. Hardware implementation of a tree node.

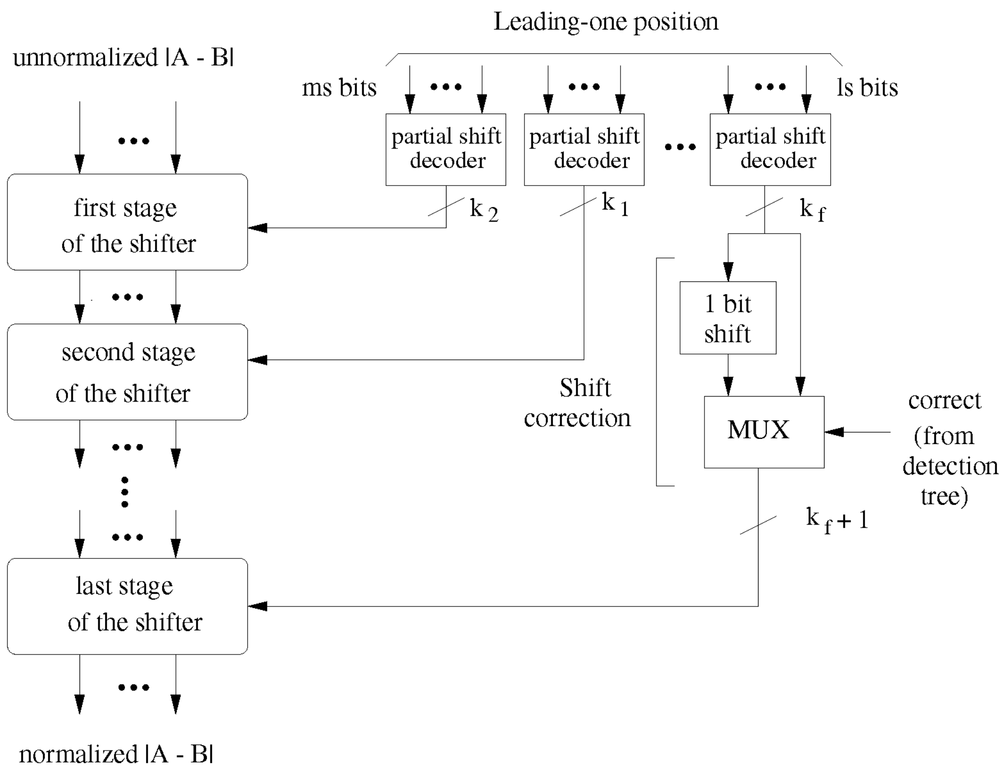


Fig. 10. Concurrent correction of the leading-one position.

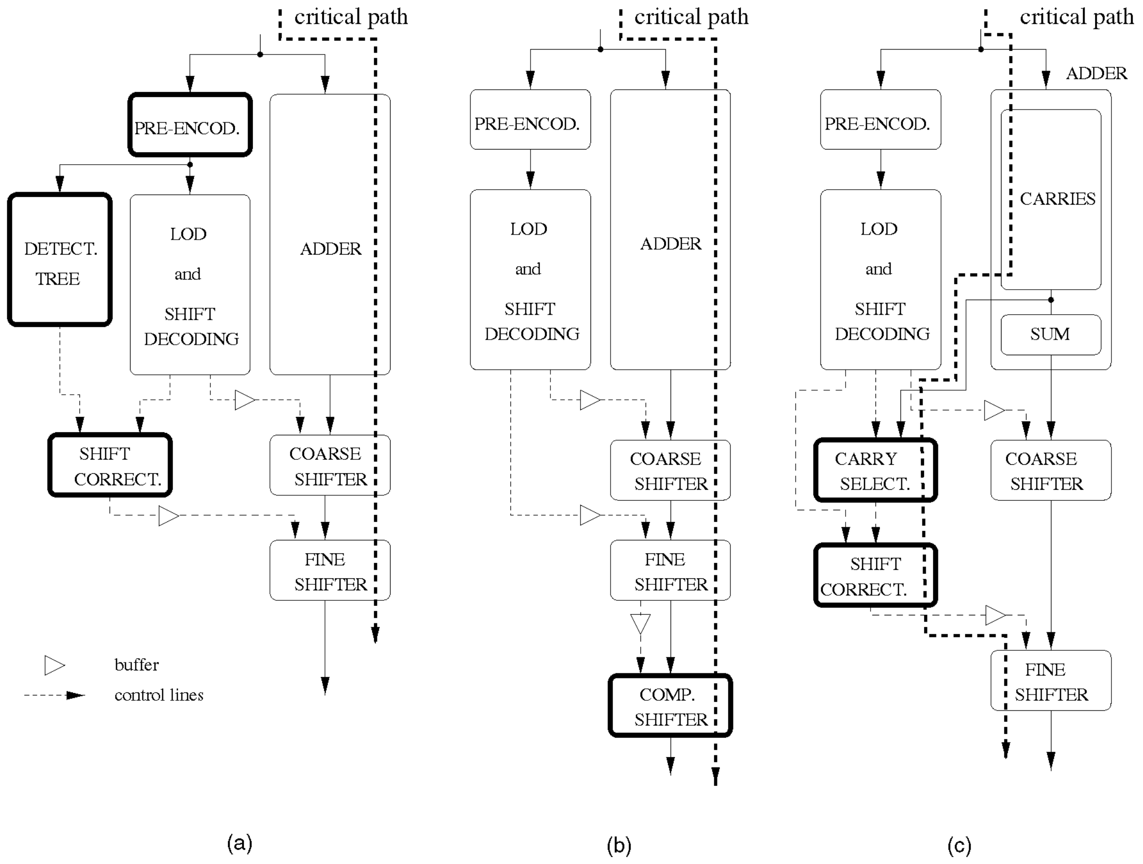


Fig. 11. General structure and critical path delay of (a) LOP with concurrent correction based on detection tree, (b) LOP without concurrent correction, and (c) LOP with concurrent correction based on carries checking.

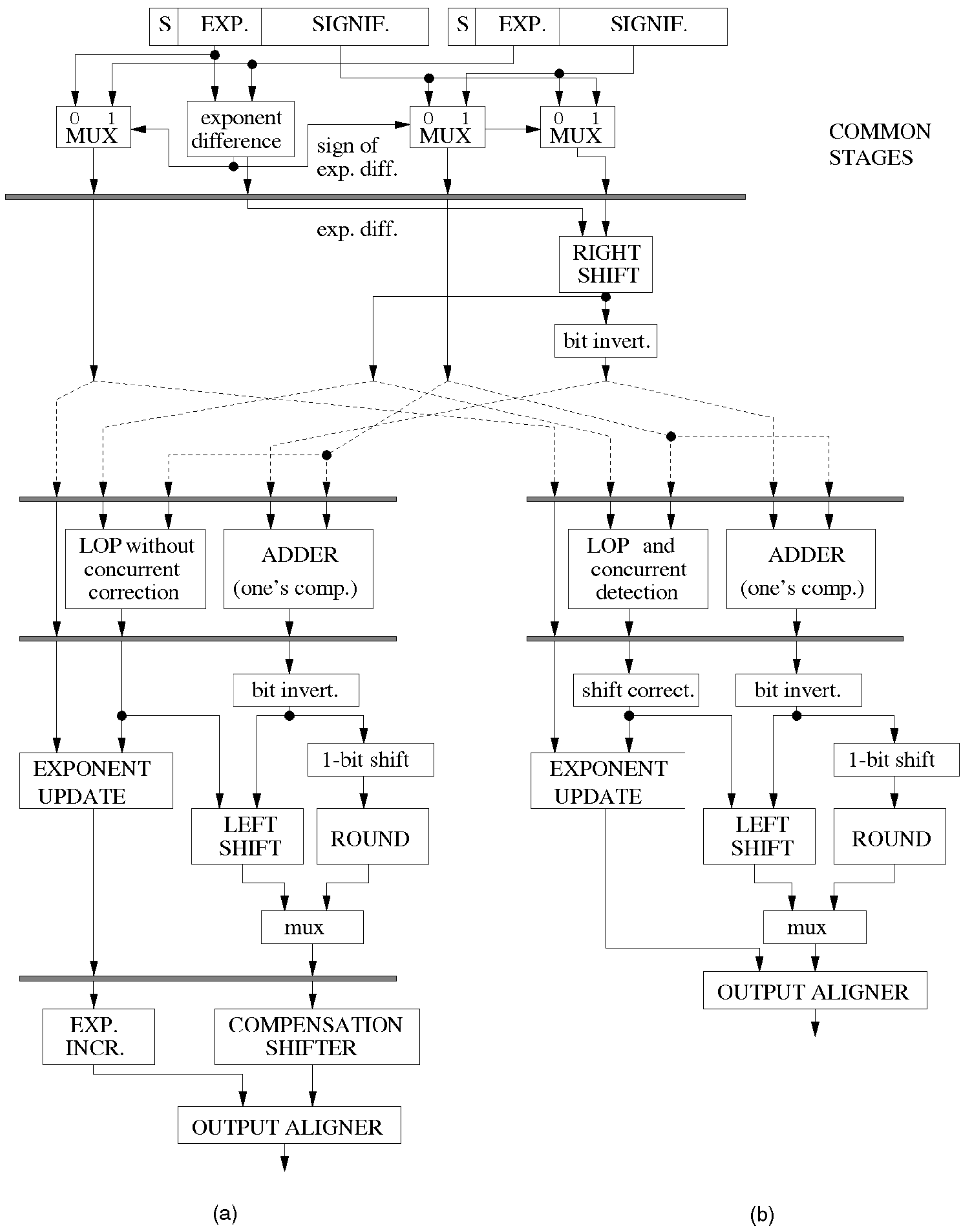


Fig. 12. (a) Single datapath floating-point adder without comparison (we have included concurrent rounding). (b) Reduced latency single datapath floating-point adder without comparison (with concurrent rounding and LOP with concurrent correction).

In terms of hardware complexity, our LOP adds the components for concurrent correction (pre-encoding logic, detection tree, and shift correction) but eliminates the compensate shifter. A rough estimate, performed at the gate level, indicates that the proposed LOP has an area of about 80 percent more than the LOP without concurrent correction.

2. **LOP with concurrent correction based on carries** (Fig. 11c). As discussed in [4], [16], the error in the leading-one position can be detected by checking the carry in the corresponding position. Therefore, for this scheme, the carries from the adder have to be calculated explicitly and the corresponding carry selected according to the output of the LOD. To accomplish this selection, it is preferable that the LOD output consist of a string of 1s followed by 0s, which is achieved by implementing the LOD as a prefix tree. Moreover, since the carry selection and shift correction is performed partially overlapped with the coarse shifter, it might be convenient to implement the shifter so as to reduce the delay of the fine shifter.

In this case, the compensation shifter is not needed, but the carry selection introduces additional delays in the critical path. We have estimated that the critical path, as shown in the figure, is composed of the logic for calculation of the carries (in the adder), carry selection, shift correction, and fine shift. Consequently, the difference in delay with respect to the LOP proposed here corresponds to the

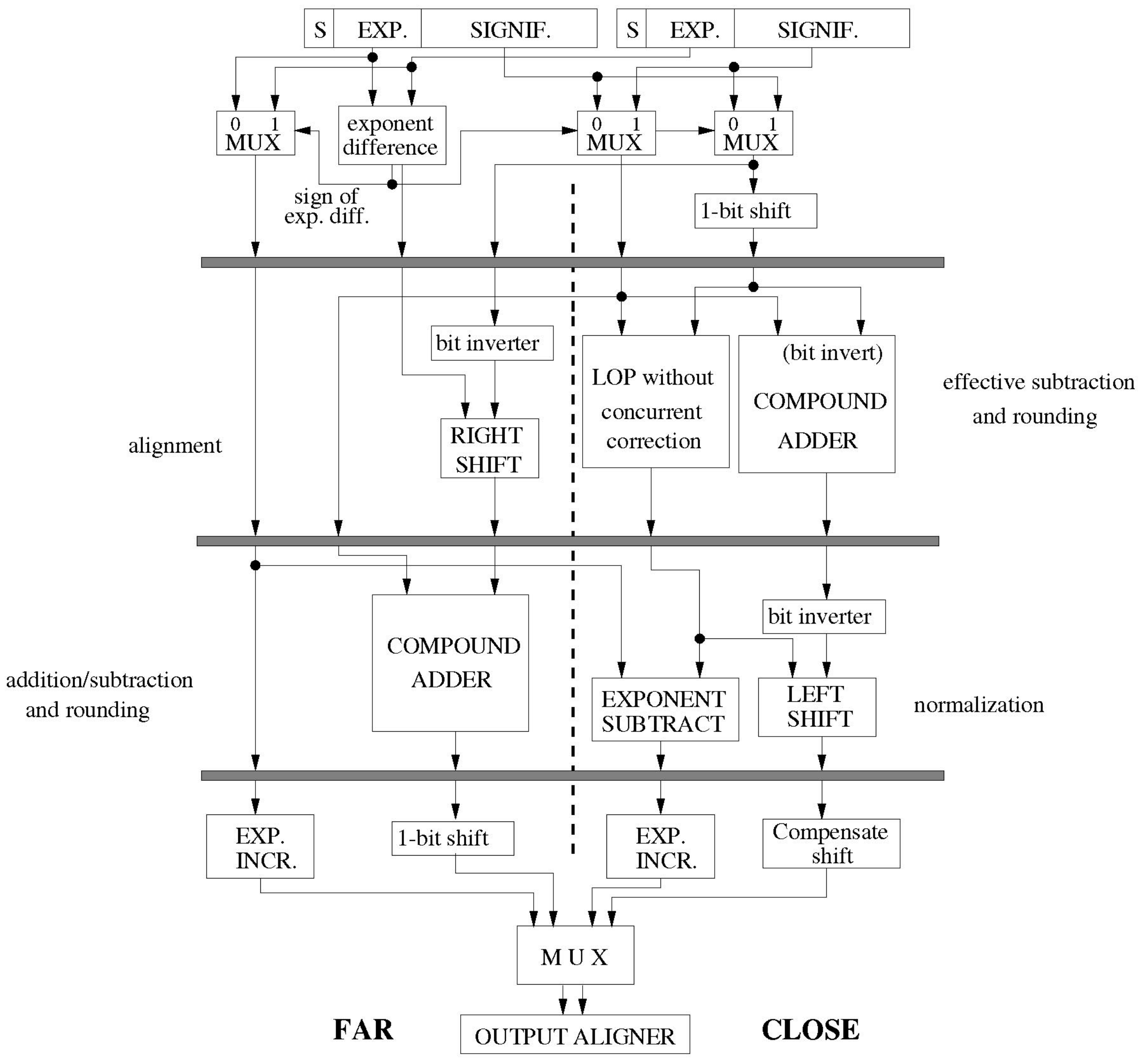


Fig. 13. Latency of the double datapath floating-point adder.

difference between the delay of the carry select plus shift correction and the delay of the coarse shifter. A rough estimation at the gate level indicates that this difference is about 10 percent of the delay of the adder plus shifter. Then, our LOP architecture also improves the critical path delay of this LOP.

In terms of hardware complexity, the added modules in this LOP are the carry selection logic and the correction shifter, but the LOD tree has a slightly more complex structure since it is a prefix tree. We have performed a rough estimation of the hardware complexity of those modules and we have concluded that the area of the carry selection logic is about 10 percent smaller than the area devoted to the concurrent correction in the proposed LOP.

As a conclusion, in our rough estimate, the LOP proposed has the smallest critical path delay and the largest hardware complexity of the three LOP schemes.

6 Floating-Point Addition Latency Reduction

We now consider the effect of the proposed concurrent correction on the overall latency of the floating-point addition. This depends on the delay of the other parts of this adder and, in a pipelined implementation, on the processor cycle time.

6.1 Latency Reduction in Single-Datapath Floating-Point Adders

Fig. 12a shows the block diagram of a single-datapath floating-point adder using an LOP without concurrent position correction [10], [13]. The adder has been pipelined into five stages, as done in [18]. The significand addition is performed in a one's complement form [5]. The adder with the LOP with concurrent correction is similar except that the compensation shifter and the exponent incrementer in the fifth stage are eliminated. This might permit to merge the two last stages, as shown in Fig. 12b. Then, the number of stages of the pipeline can be reduced to four, while maintaining the same critical path delay.

A similar scheme and latency reduction is obtained for the single datapath floating-point adder with comparison

[6], [18]. In this case, a comparator is included in the second stage to assure that, in the case of an effective subtraction, the smaller operand is subtracted from the larger one, and, therefore, the result is always positive. The latency reduction obtained for this kind of adder is analyzed in [2].

6.2 Latency Reduction in Double-Datapath Floating-Point Adders

Fig. 13 shows a double-datapath architecture [11], [12]. In it, the *FAR* datapath computes effective subtractions with exponent differences larger than one, as well as all effective additions. On the other hand, effective subtractions with an exponent difference equal or smaller than one are computed in the *CLOSE* datapath. In both paths the addition/subtraction is combined with the rounding in the compound adder [11], [15]. This adder computes $A + B$ and $A + B + 1$ and selects one of these to perform the rounding. An array of half adders is included in the FAR datapath to compute $A + B + 2$, required for the rounding to infinity when there is a significand overflow.

In the *FAR* datapath, to assure a positive result of the adder, the smaller operand is complemented for effective subtraction. Moreover, in that path, no full-length normalization is required since the maximum left shift of the result is one bit.

With respect to the *CLOSE* datapath, the maximum alignment shift is one, so no complete alignment shifter is required. On the other hand, a full-length normalization shift may be required. Therefore, the LOP is needed only in this datapath. Note that, as the effective operation is always subtraction, bit inverters for the smaller operand have been included inside the compound adder. In the case of equal exponents, the result can be negative but the conversion to a sign-and-magnitude representation is reduced to a bitwise inversion of the $A + B$ output of the compound adder [11].

The pipelining of the adder has been derived by transforming the pipelined single datapath floating-point adder of Fig. 12. To determine the influence of the LOP with concurrent correction on the latency, we analyze the total delay of the two paths using LOPs with and without concurrent correction. Considering the blocks in the CLOSE and in the FAR paths, we see that both paths have almost the same modules:

CLOSE. Exponent difference and operand swapping, 1-bit right shifter (alignment), compound adder, bit inverter, normalization left shifter, compensate shifter, mux, and output aligner.

FAR. Exponent difference and operand swapping, alignment right shifter, bit inverter, compound adder, 1-bit left shifter(normalization), mux, and output aligner.

Then, the only difference between the two datapaths is the Compensate shift in the *CLOSE* datapath. Consequently, if concurrent correction is used and the Compensate shift is eliminated, both paths have the same delay, allowing a pipelining into four stages with a smaller stage delay. Then, as there is some flexibility to pipeline the two paths into several stages, it might be even possible to obtain a three stage pipeline.

7 Conclusions

Leading-One Prediction (LOP) is a technique that permits the reduction of the delay of the floating-point adder since, as the LOP is operating in parallel with the significand adder, the amount of normalization shift is known before the result of the significand addition. However, the predicted leading-one position can have an error of one bit. Two alternatives to correct this error were previously proposed: LOP without concurrent correction and LOP with concurrent correction based on carry checking. The first alternative includes a compensation shift after the normalization shift, whereas the second one includes the logic necessary to correct the normalization shift concurrently with the operation of the significand adder and the normalization shifter, avoiding the use of a compensation shifter. Both alternatives are general since they can operate with adders in which the result can be positive or negative.

We have presented an alternative general LOP algorithm and its implementation to obtain an exact prediction of the normalization shift. Our approach includes the logic necessary to concurrently detect when the prediction will be wrong and to correct the normalization shift, without introducing any additional delay to the adder. This improves the performance with respect to LOPs with concurrent correction based on the checking of the carries of the significand adder, where the logic necessary to perform the checking introduces an additional delay. Our rough estimates indicate a reduction of about 10 percent in the delay of the significant addition and normalization shifter using our LOP algorithm with respect to both LOP without concurrent correction and LOP with concurrent correction based on carry checking. This improvement can be used to reduce the latency of a pipelined floating-point adder.

Acknowledgments

This work was supported in part by the Ministry of Education and Science (CICYT) of Spain under contract TIC96-1125-C03. This work was performed in part while J.D. Bruguera was with the University of California at Irvine.

References

[1] Am. Nat'l Standard Inst. and Inst. of Electrical and Electronic Engineers, *IEEE Standard for Binary Floating-Point Arithmetic,* ANSI/IEEE Standard, std. 745-1985, 1985.

[2] J.D. Bruguera and T. Lang, "Leading-One Prediction Scheme for Latency Improvement in Single Datapath Floating-Point Adders," *Proc. Int'l Conf. Computer Design (ICCD'98),* pp. 298-305, 1998.

[3] D. Greenley et al., "UltraSparc: The Next Generation Superscalar 64-bit Sparc," *Proc. COMPCON'95,* pp. 442-451, 1995.

[4] E. Hokenek and R.K. Montoye, "Leading-Zero Anticipator (LZA) in the IBM RISC System/6000 Floating-Point Execution Unit," *IBM J. Research and Development,* vol. 34, no. 1, pp. 71-77, 1990.

[5] E. Hokenek, R.K. Montoye, and P.W. Cook, "Second-Generation RISC Floating Point with Multiply-Add Fused," *IEEE J. Solid-State Circuits,* vol. 25, no. 5, pp. 1,207-1,213, 1990.

[6] N. Ide, H. Fukuhisa, Y. Kondo, T. Yoshida, M. Nagamatsu, J. Mori, I. Yamazaki, and K. Ueno, "A 320-Mflops CMOS Floating-Point Processing Unit for Superscalar Processors," *IEEE J. Solid-State Circuits,* vol. 28, no. 3, pp. 352-361, 1993.

[7] L. Kohn and S.W. Fu, "A 1,000,000 Transistor Microprocessor," *Proc. IEEE Int'l Solid-State Circuits Conf.,* pp. 54-55, 1989.

[8] J.A. Kowaleski et al., “A Dual-Execution Pipelined Floating-Point CMOS Processor,” *Proc. IEEE Int'l Solid-State Circuits Conf.*, p. 287, 1996.

[9] R.K. Montoye, E. Hokenek, and S.L. Runyon, “Design of the IBM RISC System/6000 Floating-Point Execution Unit,” *IBM J. Research and Development*, vol. 34, no. 1, pp. 59-70, 1990.

[10] H. Nakano, M. Nakajima, Y. Nakakura, T. Yoshida, Y. Goi, Y. Nakai, R. Segawa, T. Kishida, and H. Kadota, “An 80-Mflops (Peak) 64-b Microprocessor for Parallel Computer,” *IEEE J. Solid–State Circuits*, vol. 27, no. 3, pp. 365-372, 1992.

[11] S.F. Oberman, H. Al–Twaijry, and M.J. Flynn, “The SNAP Project: Design of Floating-Point Arithmetic Units,” *Proc. 13th IEEE Symp Computer Arithmetic*, pp. 156-165, 1997.

[12] S.F. Oberman and M.J. Flynn, “A Variable Latency Pipelined Floating-Point Adder,” Technical Report CSL-TR-96-689, Stanford Univ., 1996.

[13] F. Okamoto, Y. Hagihara, C. Ohkubo, N. Nishi, H. Yamada, and T. Enomoto, “A 200-Mflops 100-MHz 64-b BiCMOS Vector-Pipelined Processor (VPP) ULSI,” *IEEE J. Solid-State Circuits*, vol. 26, no. 12, pp. 1,885-1,983, 1991.

[14] V.G. Oklobdzija, “An Algorithmic and Novel Design of a Leading Zero Detector Circuit: Comparison with Logic Synthesis,” *IEEE Trans. VLSI Systems*, vol. 2, no. 1, pp. 124-128, 1994.

[15] N.T. Quach and M.J. Flynn, “An Improved Algorithm for High-Speed Floating-Point Addition, Technical Report CSL-TR-9-442, Stanford Univ., 1990.

[16] N.T. Quach and M.J. Flynn, “Leading-One Prediction: Implementation, Generalization and Application,” Technical Report CSL-TR-91-463, Stanford Univ., 1991.

[17] N.T. Quach and M.J. Flynn, “Design and Implementation of the SNAP Floating Point Adder,” Technical Report CSL-TR-91-501, Stanford Univ., 1991.

[18] H. Suzuki, H. Morinaka, H. Makino, Y. Nakase, K. Mashiko, and T. Sumi, “Leading-Zero Anticipatory Logic for High Speed Floating Point Addition,” *IEEE J. Solid-State Circuits*, vol. 31, no. 8, pp. 1,157-1,164, 1996.

Javier D. Bruguera received the BS degree in physics and the PhD degree from the University of Santiago de Compostela (Spain) in 1984 and 1989, respectively. Currently, he is a professor with the Department of Electronic and Computer Engineering at the University of Santiago de Compostela. His research interests are in the area of computer arithmetic, VLSI design for signal and image processing, and parallel architectures. He is a member of the IEEE.

Tomás Lang received an electrical engineering degree from the Universidad de Chile in 1965, an MS degree from the University of California (Berkeley) in 1966, and the PhD degree from Stanford University in 1974. He is a professor in the Department of Electrical and Computer Engineering at the University of California, Irvine. Previously, he was a professor in the Computer Architecture Department of the Polytechnic University of Catalonia, Spain, and a faculty member of the Computer Science Department at the University of California, Los Angeles. Dr. Lang's primary research and teaching interests are in digital design and computer architecture with current emphasis on high-speed and low-power numerical processors and multiprocessors. He is coauthor of two textbooks on digital systems, two research monographs, one IEEE Tutorial, and author or coauthor of research contributions to scholarly publications and technical conferences. He is a member of the IEEE Computer Society.