Leading-One Prediction Scheme for Latency Improvement in Single Datapath Floating–Point Adders*

Javier D. Bruguera
Dept. of Electronic and Computer Eng.
University of Santiago de Compostela
15706 Santiago de Compostela, Spain
bruguera@dec.usc.es

Tomas Lang
Dept. Electrical and Computer Eng.
University of California at Irvine
Irvine CA 92697, USA
tlang@uci.edu

Abstract

This paper describes the design of a Leading–one Predictor (LOP) for floating–point addition, with an exact determination of the shift amount required. Previous LOP proposals produce a shift amount which might be in error by one position, so that this error has to be corrected after the addition terminates, increasing the critical path. Our design incorporates a concurrent detection of this error so that the amount of shift is corrected before the actual shift, without increasing the latency. The scheme presented here is applicable to the common case of a single datapath floating-point addition in which the output of the adder is always positive. We estimate the reduction in the critical path and the increase in area.

1. Introduction

Leading-one prediction is used in floating-point adders to eliminate the delay of the encoding of the leading–one position from the critical path. This encoding 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 and is incorporated in several floating-point unit designs and commercial processors [4, 5, 6, 7, 10]. We develop here an enhancement of the LOP to reduce the latency even further.

The direct way to perform the normalization is illustrated in Figure 1a. Once the result of the addition has been computed, the Leading–One Detector (LOD) determines the position of the leading one and then the result is left shifted. Alternatively, as shown in Figure 1b, the shift amount can be determined in parallel with the significands addition. The Leading–one Predictor (LOP) anticipates the amount of the shift for normalization from the input operands.

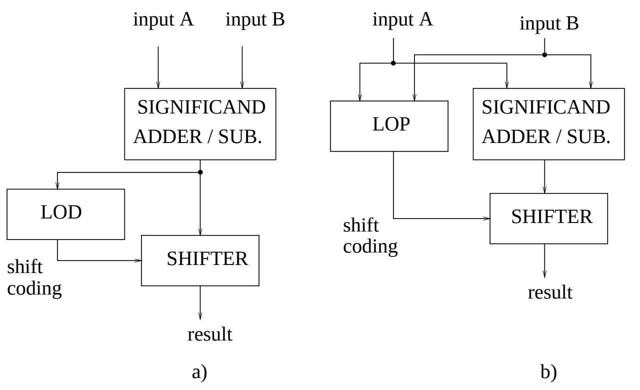


Figure 1. Magnitude Add/Sub and Normalize.

Several LOPs [1] have been recently proposed in the literature [3, 9, 10]. In [3, 9] the relative magnitudes of the adder operands is unknown, whereas in [10] it is known which is larger (Figure 2). In the latter, the required comparison is performed in parallel with the alignment, so it might not increase the critical path. Because the result of subtraction is positive, the adder and the LOP are simpler, resulting in a smaller critical path [10]. We consider this LOP as our starting point. Moreover, we consider operands represented in sign-and-magnitude since this is the representation used by the IEEE standard [1].

It is worth noticing that the scheme proposed in [10] is not applicable for the case in which the floating-point adder is designed with two parallel paths, close and far, depending on the exponent difference. In this case, the

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

[1]The LOP is also called LZA (Leading Zero Anticipator)

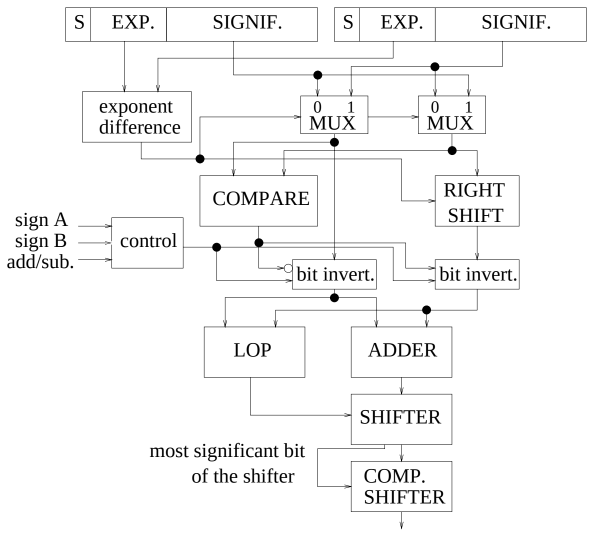


Figure 2. Portion of floating–point adder with comparator (adapted from [10]).

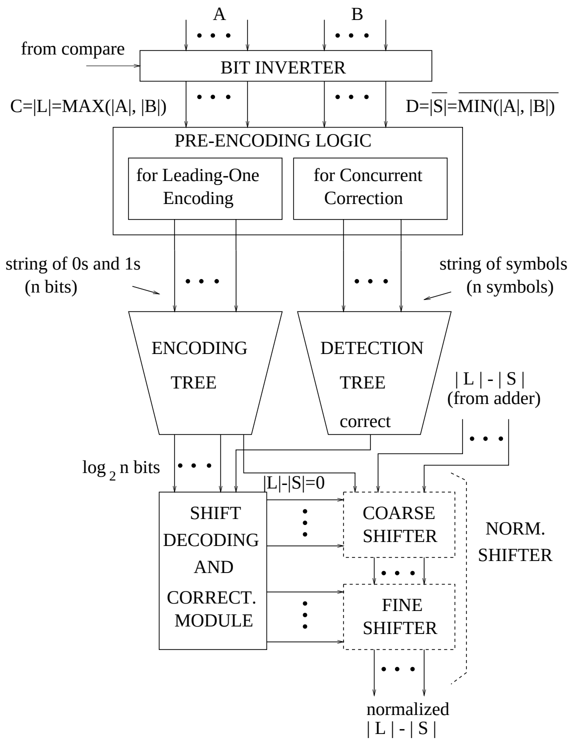


Figure 3. Structure of the proposed LOP

left-shift normalization is performed only in the close path, but this path does not include a right shift, so that the comparator delay increases the critical path. For this case, one of the schemes for unknown relative magnitudes should be used and in [2] we consider a generalization of the approach described here.

In all the previously reported LOP schemes, there is the possibility of a wrong prediction by one position. The need for this correction is detected late so that the correction increases the critical path. For instance, as shown in Figure 2, in [10] this detection is performed after the shifter and a compensation shift is required. This correction accounts for 12.5% of the delay of addition plus shifting [10]. On the other hand, the LOP proposed in [3] checks the carries in the adder to determine if a correction is needed. As shown in [10], the delay introduced by this checking results in a larger cycle time. The main contribution of this paper is a modification of the LOP to detect the error concurrently with the position encoding, resulting in the corresponding reduction in the critical path. The reduction of the overall latency of a pipelined implementation depends on other parts of the floating-point adder; in Section 5.4 we estimate that the implementation of [10] might be reduced from five to four pipeline stages.

In Section 2 the structure of the LOP we propose is described. Then, in following sections, each part of the LOP is analyzed: the algorithm for the encoding of the leading–position is described in detail in Section 3 and the concurrent position correction in Section 4. Finally, the LOP is compared with other LOP architectures in Section 5 and some conclusions are given.

2. Structure of leading–one prediction

As shown in Figure 3, the LOP we propose is divided into two main parts: the **encoding** of the leading-one position and the **correction** of this position. 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, the rest is not significant.
- An *encoding tree* to encode the position of the most-significant 1 in a way that is used to control the normalization shifter.

Correction

- A *pre-encoding module* providing a string that is used to determine whether a correction is needed. As indicated in Figure 3, there is significant commonality between both pre-encoding modules.

- A *detection tree* to determine if the output of the encoding tree has to be incremented.
- A *correction module* that performs the correction in parallel with the shifter.

3. Position encoding

3.1. Pre–encoding module

This part of the LOP is essentially the same as that presented in [10]. We present it in some detail for completeness and because, in our opinion, in [10] the explanations are not given with sufficient generality, since they are based on a few examples. Moreover, it is the basis for the discussion of the pre-encoding module for correction. This module produces a string of zeros and ones. This is done in two steps:

1. Determination of a string W as a function of the two operands A and B.
2. Determination of the string (called F) as a function of W

3.1.1 Determination of W

The operation to be performed when normalization is needed is an effective subtraction. Moreover, the scheme we are considering compares the magnitudes and complements the smallest, so that the result of the subtraction is positive.

Calling L the largest and S the smallest we obtain

$$W = |L| - |S|, \quad W > 0$$

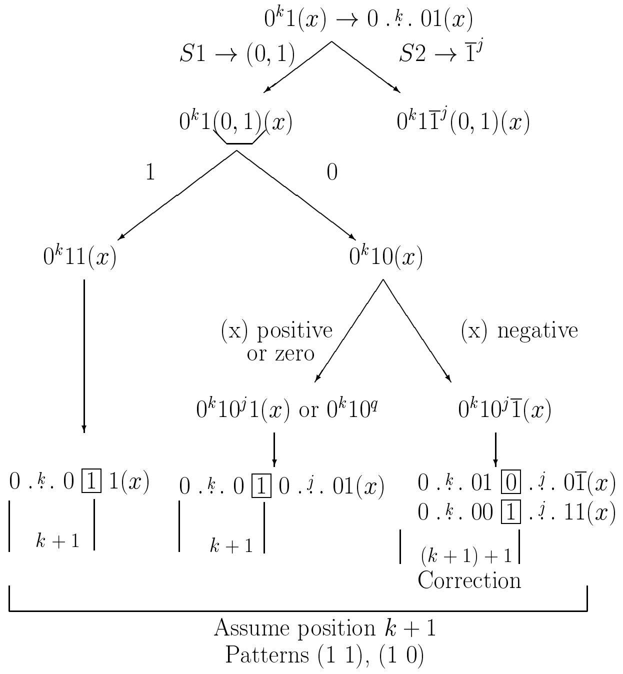
The adder computes this value in conventional representation. However, for the detection of the leading-one position this representation is not necessary. Therefore, we perform this operation without borrow propagation, by allowing the output to be in a radix–2 signed-digit representation. That is, this operation is done on each bit slice[2], so that

$$w_i = l_i - s_i \quad \textit{with } w_i \in \{-1, 0, 1\}$$

where w_i, l_i and s_i are the i-th bits of the respective bit vectors. For clarity, in the sequel -1 is represented as $\bar{1}$. Calling p, z and n the values 1,0, and $\bar{1}$ of w_i we obtain the following switching expressions:

$$p_i = l_i \bar{s}_i \qquad z_i = \overline{(l_i \oplus s_i)} \qquad n_i = \bar{l}_i s_i$$

[2]Note that this is a particular case of signed-digit addition where both operands are in non-redundant representation and the subtraction does not require a borrow.



$0^k1(x) \to 0 \overset{k}{\ldots} 01(x)$
$S1 \to (0,1)$
$S2 \to \bar{1}^j$
$0^k1(0,1)(x)$
$0^k1\bar{1}^j(0,1)(x)$
1
0
$0^k11(x)$
$0^k10(x)$
(x) positive or zero
(x) negative
$0^k10^j1(x)$ or 0^k10^q
$0^k10^j\bar{1}(x)$
$0 \overset{k}{\ldots} 0 \boxed{1} 1(x)$
$0 \overset{k}{\ldots} 0 \boxed{1} 0 \overset{j}{\ldots} 01(x)$
$0 \overset{k}{\ldots} 01 \boxed{0} \overset{j}{\ldots} 0\bar{1}(x)$
$0 \overset{k}{\ldots} 00 \boxed{1} \overset{j}{\ldots} 11(x)$
$k+1$
$k+1$
$(k+1)+1$
Correction
Assume position $k+1$
Patterns (1 1), (1 0)

Figure 4. Patterns for leading–one detection

These expressions are now modified because, as shown in Figure 3, we obtain the inputs for the encoding module from the output of the conditional inverters. We call the corresponding bits vectors C (for L) and D (for S). Because of the comparator we have $c_i = l_i$ and $d_i = \bar{s}_i$. Replacing in the previous expressions

$$p_i = c_i d_i \qquad z_i = (c_i \oplus d_i) \qquad n_i = \bar{c}_i \bar{d}_i$$

Since these expressions are symmetric with respect to c_i and d_i, they are valid for both possible cases, namely $C = |A|$ or $C = |B|$.

3.1.2 Determination of F

We now consider the string W to determine the position of the leading one. Formally, this requires a conversion of the signed-digit representation into a conventional representation. However, as we see now this conversion is not actually required

The alternative situations for the location of the leading one are described in the diagram of Figure 4. The notation used is the following: (x) denotes an arbitrary substring and 0^k, 1^k and $\bar{1}^k$ denote strings of k 0's, 1's and $\bar{1}$'s, respectively, with $k \geq 0$.

Since $W > 0$, the first digit w_i different from 0 has to be equal 1, as shown on top of the diagram. For the substring (x) two situations can be identified,

S1 The first digit of (x) 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 (see diagram):

1. The digit is 1. That is, $W = 0...011(x)$. Clearly, the conversion of W to conventional representation has a 1 in position $k+1$ 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 $w_i w_{i+1} = 11$
2. The digit is 0. That is, $W = 0...010(x)$. Now, two possibilities exist,
   - (x) is positive. 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

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

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

$$\begin{aligned} W &= 0...01\bar{1}...\bar{1}(0,1)(x) \\ &= 0...000...1(0,1)(x) \end{aligned}$$

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

$$w_i w_{i+1} = \bar{1}1 \textit{ or } \bar{1}0 = \bar{1}(\textit{not } \bar{1}) \qquad (2)$$

In the particular case with $W = 0$, there is no leading one. This is detected by the encoding tree.

We now produce F. The corresponding bit of F is obtained by combining (1) and (2),

$$f_i = p_i \bar{n}_{i+1} + n_i \bar{n}_{i+1} = (p_i + n_i)\bar{n}_{i+1} \qquad (3)$$

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 (a priority encoder) [8, 10]. The LOD has a tree structure, where each level is composed of a two input multiplexer. It also detects the case $F = 0$.

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,

$$0^k 1 0^j \bar{1}(x) \quad \textit{and} \quad 0^k 1 \bar{1}^j 0^t \bar{1}(x)$$

In these cases the position has to be corrected by adding 1 to the encoding calculated in the tree. We now describe how to perform this correction without increasing the critical path. The proposed concurrent position correction has two steps: (1) detection of when it is necessary to correct and (2) correction of the position encoding. The first step is carried out in parallel with the encoding and the second as part of the normalization shifting.

4.1. Detection

4.1.1 Pre-encoding module

For this pre-encoding we use the W string. To detect the patterns for correction, we need to distinguish between the digit values 1 and $\bar{1}$. Consequently, instead of producing a string F which contains only ones and zeros, we produce a string G with 1, 0, and $\bar{1}$.

For the two patterns of W needing correction, the corresponding string F is as follows:

W	$0\ldots010\ldots0\bar{1}(x)$	$0\ldots01\bar{1}\ldots\bar{1}\bar{1}0\ldots0\bar{1}(x)$
F	$0\ldots010\ldots00(x)$	$0\ldots000\ldots010\ldots00(x)$

Therefore, making G equal to F in all bits except in the last $\bar{1}$, where we introduce the value $\bar{1}$ instead of 0, we obtain the pattern $0^k 1 0^j \bar{1}$, which permits to determine when a correction is needed. Figure 5 shows that this pattern only appears in G when the predicted position has to be corrected. Using b (beginning of pattern), m (middle), and e (end) for 1, 0 and $\bar{1}$ in G we obtain

$$b_i = f_i \qquad e_i = z_{i-1} n_i \qquad m_i = \overline{b_i + e_i}$$

Note that for substring $w_{i-1} w_i w_{i+1} = 0\bar{1}0$ or $w_{i-1} w_i w_{i+1} = 0\bar{1}1$ both e_i and b_i of G are set. In this case, the value ($e_i = 1$, $b_i = 1$) is interpreted as $e_i = 1$, as discussed further when presenting the detection tree. Figure 6 shows the implementation of the pre–encoding logic to compute strings F and G.

$$\begin{array}{lll} W & & W \,/\, G \end{array}$$

$$0^k\;1^j\;(x) \longrightarrow \begin{array}{l} 0^k\;1^j \ldots \\ m^k\;b^j\;\ldots \end{array}$$

$$0^k\;1\;0^j\;1\;(x) \begin{array}{l} \xrightarrow{(0,1)} \begin{array}{l} 0^k\;1\;0^j\;1\;(0,1)\;\ldots \\ m^k\;b\;m^j\;b\;\ldots \end{array} \\ \xrightarrow[\overline{1}]{} \begin{array}{l} 0^k\;1\;\;0^j\;\;1\;\;\overline{1}\;\;\overline{1}\;\;(0,1)\;\ldots \\ m^k\;b\;m^j\;\;m\;m\;b\;\ldots \end{array} \end{array}$$

$$0^k\;1\;0^j\;\overline{1}\;(x) \longrightarrow \begin{array}{l} 0^k\;\;1\;\;0^j\;\;\overline{1}\;\ldots \\ m^k\;b\;m^j\;\;e\;\ldots \\ \textbf{correction needed} \end{array}$$

$$0^k\;1\;\overline{1}^j\;1\;(x) \begin{array}{l} \xrightarrow{(0,1)} \begin{array}{l} 0^k\;\;1\;\;\overline{1}^{j-1}\;\;\overline{1}\;\;1\;\;(0,1)\;\ldots \\ m^k\;\;m\;m^{j-1}\;b\;b\;\ldots \end{array} \\ \xrightarrow[\overline{1}]{} \begin{array}{l} 0^k\;\;1\;\;\overline{1}^{j-1}\;\;\overline{1}\;\;1\;\;\overline{1}\,\overline{1}\,(0,1)\;\ldots \\ m^k\;\;m\;m^{j-1}\;b\;m\;m\;b\;\ldots \end{array} \end{array}$$

$$0^k\;1\;\overline{1}^j\;0^t\;1\;(x) \begin{array}{l} \xrightarrow{(0,1)} \begin{array}{l} 0^k\;\;1\;\;\overline{1}^{j-1}\;\;\overline{1}\;\;0^t\;\;1\;\;(0,1)\;\ldots \\ m^k\;\;m\;m^{j-1}\;b\;m^t\;\;b\;\ldots \end{array} \\ \xrightarrow[\overline{1}]{} \begin{array}{l} 0^k\;\;1\;\;\overline{1}^{j-1}\;\;\overline{1}\;\;0^t\;\;1\;\;\overline{1}\;\;(0,1)\;\ldots \\ m^k\;\;m\;m^{j-1}\;b\;m^t\;\;m\;b\;\ldots \end{array} \end{array}$$

$$0^k\;1\;\overline{1}^j\;0^t\;\overline{1}\;(x) \longrightarrow \begin{array}{l} 0^k\;\;1\;\;\overline{1}^{j-1}\;\;\overline{1}\;\;0^t\;\;\overline{1}\;\ldots \\ m^k\;\;m\;m^{j-1}\;b\;m^t\;\;e\;\ldots \\ \textbf{correction needed} \end{array}$$

Figure 5. Patterns in string G

4.1.2 Detection tree

To determine if the correction has to be carried out, it is necessary to detect pattern $m^j b m^k e(x)$ in G. We use a binary tree to detect this pattern. For each node of the tree, five values, M, B, E, Y, and U are needed, representing the strings shown in the last column of Table 1. Y indicates detection of the pattern, and U a string incompatible with the pattern.

Each node of the tree receives as input the output from two nodes of the preceding level (Figure 7a)). Table 1 shows the function for a node of the tree.

For a simple implementation we encode the five values with four variables and assign the code 0000 to state U. Then, the logic equations are,

$$M = M^l M^r \qquad B = M^l B^r + B^l M^r \qquad (4)$$
$$E = M^l E^r + E^l \qquad Y = Y^l + M^l Y^r + B^l E^r$$

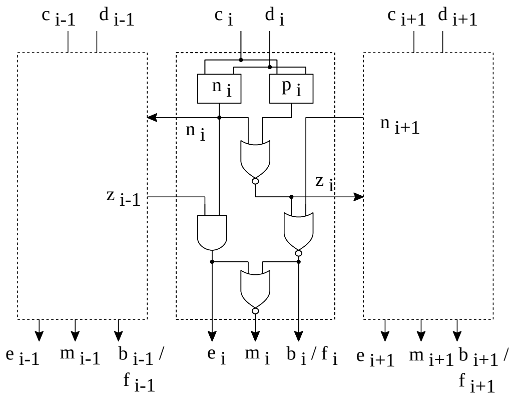


Figure 6. Implementation of the pre–encoding

Table 1. Function of a Tree Node

Left input	Right input					string
	M	B	E	Y	U	
M	M	B	E	Y	U	m^j
B	B	U	Y	U	U	$m^j b m^k$
E	E	E	E	E	E	$m^k e\ (x)$
Y	Y	Y	Y	Y	Y	$m^j b m^k e\ (x)$
U	U	U	U	U	U	other strings

where (M^l, B^l, E^l, Y^l) and (M^r, B^r, E^r, Y^r) represent the left input and the right input, respectively. Figure 7b) shows the implementation of a node. As explained in section 4.1.1, there are situations in which both b_i and e_i of G are 1. However, since the output is E whenever $E^l = 1$ this does not affect the rest of the detection tree. The need for correction is obtained by $Y = 1$ in the last level of the tree.

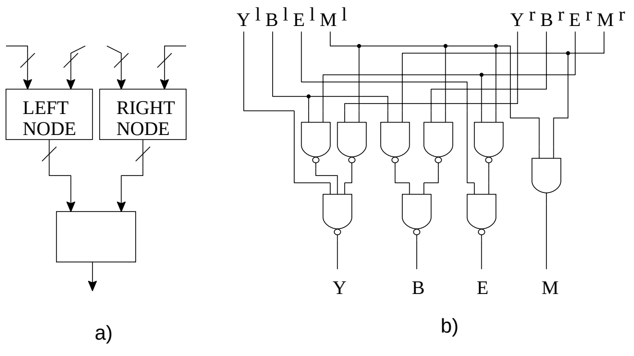


Figure 7. Detection tree node implementation

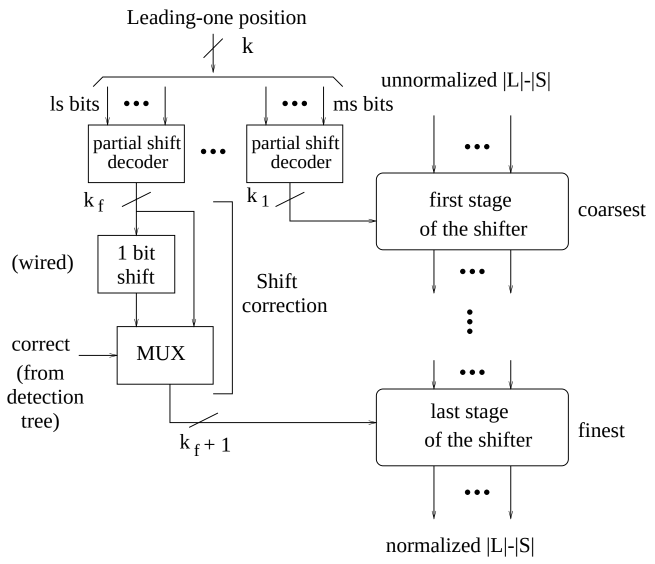


Figure 8. Concurrent correction

4.2. Normalization shift correction

The last step on the LOP we propose is the correction of the leading–one position. The correction is done by incrementing the shift amount.

To reduce the delay of the shifter it is convenient to decode the shift amount in parallel with the adder [10]. Moreover, because of implementation constraints, the shifter has more than one stage. As shown in Figure 8, 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. We perform the correction at this last stage, so that the shifter has to be modified to shift from 0 to $k_f + 1$ positions. The selection between correction and 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 and compared with the LOP described in [10] in terms of delay of the critical path and added hardware. There are many design decisions that depend on the overall constraints and on the specific technology used. To make the comparison with [10] more meaningful, in the modules that are not part of the LOP, such as the adder and the shifter, we follow the design decisions of [10], which correspond to a $0.5\mu m$ CMOS implementation of a 64 bits floating-point adder. For the common blocks we perform an estimate and compare with the values obtained from [10] and for the new blocks we perform compatible estimates.

Table 2. Delay of the basic components

ELEMENT	OUR ESTIM. (t_{nand})	FROM [10] ns	FROM [10] t_{nand}
Inverter	0.5		
2–input MUX	3.0		
Pre–encoding F	3.0	0.6	3.0
Pre–encoding G *	4.0		
LOD	16.0	3.5	17.5
Detection tree *	12.0		
Shift decoder + buffer	4.0	0.8	4.0
Correction + buffer *	4.0		
Adder		5.0	25.0
Shifter (1st level)		1.1	5.5
Shifter (2nd level)		0.9	4.5
Compensate shifter		1.0	5.0

* modules added by our proposal

5.1. Timing Analysis

We now determine the critical path of the addition and normalization shift. For the adder and the shifter we use the delay numbers of [10], as shown in Table 2. For the other blocks, we make an estimate using as delay unit the delay of a simple gate (2-input NAND). Finally, to combine these values, we translate the delays in nanoseconds into gate delays by assigning a delay of 0.2 nsec per gate, resulting in the last column of the Table. There is a good correspondence between the estimates and the values obtained from [10].

The delay of the pre–encoding logic corresponds to the hardware implementation of Figure 6. To compute the delay of the detection tree and LOD we have considered trees with six levels. The first level of the LOD is composed only of one level of NOR gates, whereas the delay of the remaining levels corresponds to a 2–input multiplexer. On the other hand, the delay of each level of the detection tree is determined by a two-level NAND network (see Figure 7). Note that the M output has a load of four gates; however, the load of the slowest path in the node, the Y output, is of only one gate. Therefore, the load of M does not affect the global delay of the node.

The shifter in [10] is formed by two stages, which allows us to perform the correction during the first stage and to apply it in the second. For the delay of the correction mux and the shift decoders the effect of the large load imposed on them by the shifter has been included. Thus, we have considered a delay of 4 NAND gates for each, which is comparable with the delay indicated in [10] for the decoder.

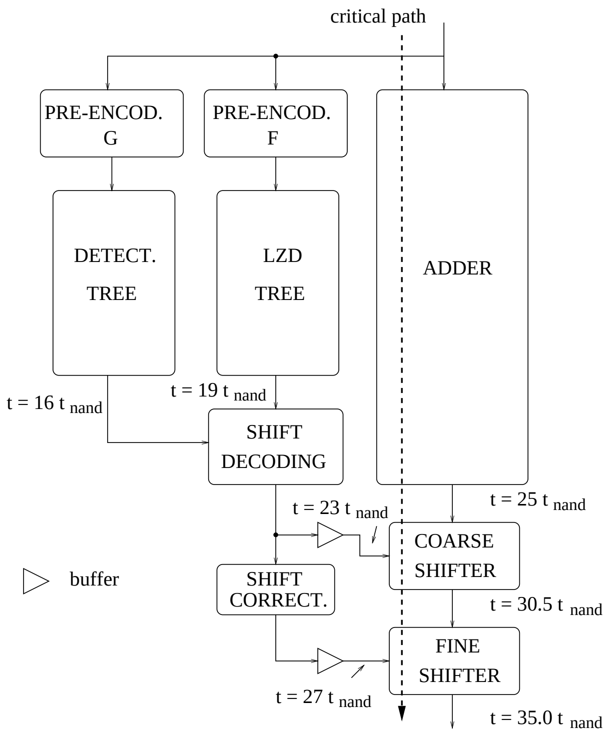


Figure 9. Delay in adder+normalization

Figure 9 shows the delay of each of the parallel paths in the adder, LOP, and shifter. Note that the slowest path is the one going through the adder. This is the same conclusion obtained in [10].

5.2. Components of the LOP

In this estimate, summarized in Table 3, we include only the components of the LOP. This is in contrast with the delay estimate, in which we also included the adder and the shifter. Moreover, as done in [10], the area estimate includes only the active components and not the interconnections.

5.3. Comparison

We now compare with [10], which uses the same approach for the preencoding and the LOD. However, it does not include the concurrent correction but does the correction with a compensate shift. In [10] this compensate shift is evaluated to be a 12.5% of the delay of addition plus shift. Consequently, this percentage is saved by the inclusion of the concurrent correction.

The components of both implementations are the same, except that in our case we have to add the components for concurrent correction and eliminate the components for the compensate shift. An estimation of the number of gates is given in Table 3, which shows an increase in the number of gates from 620 to 1038, corresponding to an increase of 67%. However, note that the area of the components of the LOP should be a small fraction of the total area of the floating-point adder.

Table 3. Gate count of the LOP

	OUR	LOP IN [10]	
	LOP	FROM [10]	OUR EST.
ELEMENT	Gates	Trans. (gates)	Gates
2–input MUX	3		3
Pre–encoding	324	884 (221*)	216
LOD	200	696 (174*)	200
Detection tree	470		
Shift decoder	44	152 (38*)	44
Compen. shift		570 (143*)	160
TOTAL	1038		620

* assuming four transistors per gate

With respect to other LOP architectures, the analysis in [10] compares with the LOP described in [3]. This LOP, that performs the prediction for both positive or negative adder result, uses a concurrent correction scheme based on the checking of the addition carries. That analysis concludes that the LOP in [10] improves the delay time and the hardware complexity by 10% and 45%, respectively. In [2] a detailed comparison for the general LOP case concludes that the solutions using compensation shifter and detection of carries have similar delays. Therefore, our LOP architecture improves both delay time and hardware complexity, with respect to the LOP in [3].

5.4. Floating–point adder latency reduction

The actual reduction in the latency depends on the delay of the other parts of the adder. In relation to the implementation of [10], it is possible that, using the proposed LOP, the latency would be reduced to four stages. To illustrate this reduction, Figure 10a) shows the block diagram of a floating–point adder using a LOP without the concurrent position correction. In contrast, the LOP with concurrent correction would permit to merge the two last stages (Figure 10b)). This merging might not be possible without the concurrent correction because of the delay of the compensate shifter and the exponent incrementer.

Although the reduction in latency obtained by the concurrent correction is significant, it might be argued that a larger reduction can be obtained by a two-path architecture [7]. However, this requires complete

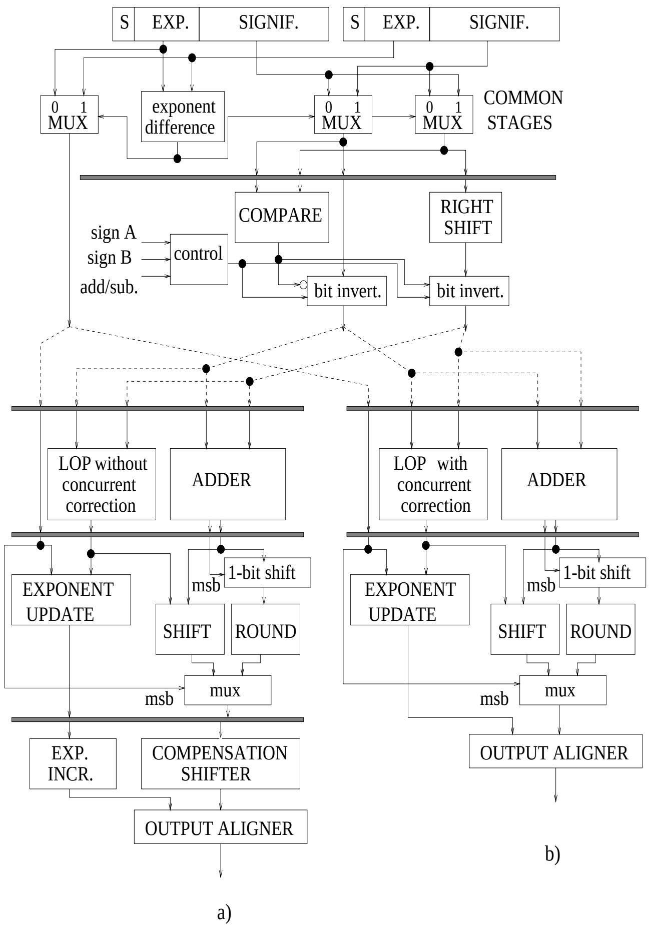


Figure 10. a) Floating-point adder adapted from [10]. b) Reduced latency

adders for each path, resulting in a larger area.

6. Conclusions

We have presented an algorithm and implementation to reduce the critical path of the adder plus normalization shifter portion of a floating-point adder by incorporating a concurrent position correction to the LOP. We showed that the incorporation of the scheme produces a reduction of about 12.5% in the critical path, as compared to the implementation described in [10]. The effect of this reduction on the overall latency of the floating-point adder and on the cycle time depends on many characteristics of the processor, although we have shown that the latency might be reduced from five to four cycles. The added modules increase the number of gates for the LOP by about 70%. However, the effect on the floating-point adder should be small.

We have also described in detail a LOP technique similar to that presented in [10], where the description is only given in terms of a few examples. As far as we know, this is the first description of this type and should help researchers and designers understand the scheme better and allow them to suggest modifications.

References

[1] American National Standard Institute and Institute 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 with Concurrent Position Correction.* Technical Report HPCG–98–008. University of Santiago de Compostela. (1998). Available in *http://www.ac.usc.es.*

[3] E. Hokenek and R.K. Montoye. *Leading–Zero Anticipator (LZA) in the IBM RISC System/6000 Floating–Point Execution Unit.* IBM Journal Research and Development, Vol. 34, No. 1, pp. 71–77. (1990).

[4] E. Hokenek, R.K. Montoye and S.L. Runyon. *Design of the IBM RISC System/6000 Floating–Point Execution Unit.* IBM Journal Research and Development, Vol. 34, No. 1, pp. 59–70. (1990).

[5] D. Greenley et al. *UltraSparc: The Next Generation Superscalar 64–bit Sparc.* Proc. COMPCON'95. pp. 442–451. (1995).

[6] L. Kohn and S.W. Fu. *A 1,000,000 Transistor Microprocessor.* Proc. IEEE Int. Solid–State Circuits Conf. pp.54–55. (1989).

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

[8] V.G. Oklobdzija. *An Algorithmic and Novel Design of a Leading Zero Detector Circuit: Comparison with Logic Synthesis.* IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 2, No. 1, pp. 124–128. (1994).

[9] N.T. Quach and M.J. Flynn. *Leading–one Prediction. Implementation, Generalization and Application.* Technical Report CSL–TR–91–463. Stanford University. (1991).

[10] H. Suzuki, H. Morinaka, H. Makino, Y. Nakase, K. Mashiko and T. Sumi. *Leading–Zero Anticipatory Logic for High Speed Floating Point Addition.* IEEE Journal of Solid–State Circuits, Vol. 31, No. 8, pp. 1157–1164. (1996).