LEADING ONE PREDICTION — IMPLEMENTATION, GENERALIZATION, AND APPLICATION

Nhon Quach
Michael J. Flynn

Technical Report: CSL-TR-91-463

March 1991

This work was supported by NSF contract No. MIP88-22961.

Leading One Prediction — Implementation, Generalization, and Application

by

Nhon Quach and Michael Flynn

Technical Report: CSL-TR-91-463

March 1991

Computer Systems Laboratory

Departments of Electrical Engineering and Computer Science

St anford University

St anford, California 94305

Abstract

This paper presents the concept of leading-one prediction (LOP) used in most high-speed floating-point adders in greater detail and describes two existing implementations. The first one is similar to that used in the TBM RS /6000 processor. The second is a distributed version of the first, consuming less hardware when multiple patterns need to be detected. We show how to modify these circuits for sign-magnitude numbers as dictated by the IEEE standard.

We then point out that (1) LOP and carry lookahead in parallel addition belong to the same class of problem, that of bit pattern detection. Such a recognition allows techniques developed for parallel addition to be borrowed for bit pattern detection. And (2) LOP can be applied to compute the sticky bit needed for binary multipliers to perform IEEE rounding.

Key Words and Phrases: Leading **one** prediction, high-speed floating-point adders, parallel addition, IEEE rounding, sticky bit computation, parallel implementation.

Contents

List of Figures

List of Tables

1 Introduction

In floating-point (FP) addition, the result of a subtraction may require a massive left shift during the normalization step [1]. To normalize, the straightforward way is to wait for the arrival of the result to be normalized and then perform a leading one[1] detection (LOD) by counting the number of preceding zero's or one's in the result. This number is then used to drive a shifter to produce the final normalized result. LOD is slow because detection of the leading one or zero cannot begin until the arrival of the result.

Leading one/zero prediction (LOP) is a technique in which the number of preceding zero's or one's in the result can be predicted directly from the input operands to within an error of one bit, in parallel with the addition/subtraction step. The error comes from the possible carry in. The amount of prediction will later be fine tuned when the carry into that bit position becomes available. Fig. 1 illustrates the difference between LOD and LOP. Many high speed FP units employ LOP [2, 3, 4, 5]. With the exception of Hokenek and Montoye [ii]. the description of LOP contained in these references, however, is at best sketchy.

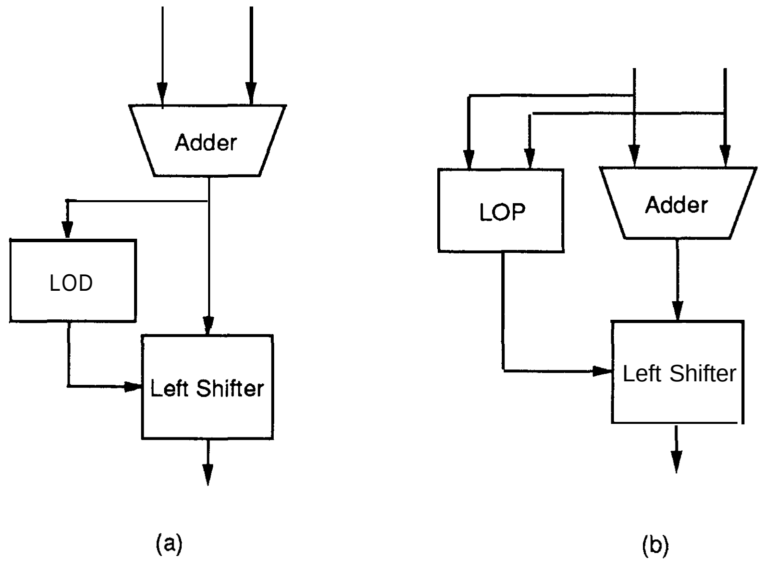

Figure 1: How leading one prediction works: (a) Leading one detection and (b) Leading prediction.

In this paper, we attempt to supply "the missing pieces", presenting LOP in greater detail and describe two actual implementations. Our presentation is based on a more unified

[1]In this paper, a leading one refers to the leading one that follows a beginning string of zeros in a positive result to be normalized. Because this may be confused with the string of leading one's preceding the leading zero in a negative result, we will refer to the latter as the preceding one's.

framework — that of bit pattern detection (BPD). Contrary to Hokenek and Montoye [5], we believe that parallel addition and LOP belong to the same class of problem. Sticky bit computation, which is needed for IEEE rounding in FP multipliers, also falls into this category.

The remainder of the paper is organized as follows. In Section 2, we present LOP and describe its implementations. In Section 3, we explain why LOP can be generalized to the problem of carry-lookahead and in Section 4 how it can be applied to sticky bit computation. We then summarize in Section 5.

In this paper, X^i denotes a string of X's of length i. X^* denotes a string of any number of X's (including the empty string). X^*YZ^*, for example, denotes a string that begins with any number of X's, followed by a single Y, and ended with any number of Z's. a_i and b_i are the ith bits of the input operands A and B, respectively. $T_i = a_i \oplus b_i$ (exclusive OR), $Z_i = \overline{a_i \vee b_i}$ (NOR), $G_i = a_i b_i$ (AND), and $P_i = a_i \vee b_i$ (OR). Our numbering convention starts with zero and with the most significant bit (MSB) first; that is, the MSB is numbered as bit zero. Unless otherwise stated, a string always starts from the MSB of the operands A and B. Examples of strings follow: for A = 11000 and B = 10001, the string is $GTZZT$ and for A = 1111110000 and B = 0000010000, is T^5GZ^4.

2 Leading One Prediction

2.1 Theory

Given the operands A and B represented in a 2's complement form, how do we predict the number of preceding zero's or one's in the result? The key to this problem is to realize that there are only a finite number of bit patterns that need to be detected for predicting the position of the leading one or zero. Specifically, only the following two bit patterns will produce a string of preceding zero's:

$$Z^*$$

and

$$T^*GZ^*$$

The first case can only happen when denormalized numbers are allowed. The reader can convince himself or herself by trying a couple of cases. Similarly, to detect a string of preceding one's, only the following two bit patterns need to be detected.

$$G^*$$

and

$$T^*ZG^*$$

Some FP units, such as the Intel i860 processor [6], use a magnitude comparator to make sure that the result is always positive. For these units, only the bit pattern T^*GZ^* need to be detected with an accompanying reduction in hardware. The following discussion assumes the detection of all four bit patterns mentioned above.

LOP works as follows. At each bit position i, it detects and outputs a shift signal $SH_i = 1$ if any of the patterns are found. The logic equation for SH_i is:

$$SH_i = Z^i \vee T^j G Z^k \vee G^i \vee T^j Z G^k$$

where j and k are integer $\in [0, i-1]$ and $j + k = i - 1$. For N-bit operands A and B, the LOP output is an N-bit vector — the SH_i array — consisting of a possible one's string followed by a string of zero's. The transition from one to zero indicates the location of the leading one or zero. In some implementations, the SH_i array may contain all zeros when none of the patterns are found. In others, such as the RS/6000, SH_0 is always equal to 1; hence, the array contains at least a 1. Alternatively, the output of LOP may be represented in a l-of-N code, with the single one or zero indicating the location of the leading one or zero. These two representations are easily interchangeable and is therefore a relatively minor issue. In both cases, the shift amount contained in the SH_i array may be off by one bit because of the possible carry-in. Depending on the implementation of the exponent logic, the total shift amount may be represented as

$$SH_{total} = SH_{coarse} + C_{fine}$$

or as

$$SH_{total} = SH_{coarse} - C_{fine}$$

where SH_{coarse} is the number of one's in the SH_i array and C_{fine} equals one if a fine adjustment is needed. Note that in the first case, we always over-predict and in the second case, always under-predict. This is achieved by appropriately wiring the input operands.

Developing an equation for C_{fine} involves a case-by-case analysis of the bit patterns of the string and is highly implementation dependent. We postpone its description until the following subsection.

2.2 Implementations

How do we implement a circuit that can detect the above bit patterns efficiently? As shown in Fig. 1, the time for LOP is preferably equal to or less than the adder time. Obviously, if the adder employs a parallel carry lookahead (CLA) scheme, then LOP must employ a similar scheme. In this section, we describe two parallel LOP implementations. The first scheme is similar to the RS/6000 one. The second consumes less hardware when multiple patterns are to be detected at the same time as in the case of LOP. Both schemes have an $O(\log N)$ computation time.

In a parallel implementation, the string is partitioned into groups at the input stage. At the second stage, these groups are in turn partitioned into blocks, and so on. Within each group or block, information is processed independently. In general, the size of the group (and therefore block) is implementation dependent. Fan-in of a logic family or a technology though often dictates this group size, other factors may come into play depending on the specific of an implementation.

The RS/6000 method uses a group (and block) size of four. Each bit position has to supply the Z, T, and G signals, which may be shared with the adder. In this method, detecting the bit pattern Z^* requires ANDing of all Z_i's. Detecting the bit pattern T^*GZ^* requires keeping account of three states: the N (not found) state, indicating that the G bit has not yet been found in all the groups (or blocks) examined so far; the J (just found) state, indicating that the G bit has just been found in the group (or block) being examined; and the F (found) state, indicating that the G bit has already been found. The N and F states correspond to ANDing of all the T_i and Z_i bits, respectively, in a group (or block). The J state corresponding to the following condition:

$$J = GZZZ \vee TGZZ \vee TTGZ \vee TTTG \tag{1}$$

For the subsequent stages, ANDing two N states, NN, produces a (bigger) N state; ANDing an N and a J state, NJ, a (bigger) J state; and ANDing two F states, FF, a (bigger) F state. Any other combination causes SH_i to be false. The logic equation at the block level, for example, is

$$J_{block} = JFFF \vee NJFF \vee NNJF \vee NNNJ \tag{2}$$

Eqn (2) is similar in form to Eqn (1). Its implementation is much like the CLA tree used in parallel adders, but is more hardware intensive because multiple carry trees may be needed. The detection of the bit patterns Z^* and T^*ZG^* can be performed in a similar manner.

In the RS/6000 method, SH_{total} is represented as $SH_{coarse} - C_{fine}$. There are at least two ways to derive an equation for C_{fine}. The first method takes advantage of the fact that at the first bit position where $SH_i = 0$, we only need to know whether we are processing a Z string or a G string. A T string will eventually turn into a Z or a G string, depending on the ending literal. Hence, only a global variable need to be maintained to differentiate these cases. Specifically, $C_{fine} = C_{i-1}$ for a Z string and $C_{fine} = \overline{C}_{i-1}$ for a G string where C_{i-1} is the carry into the $(i-1)$th bit position.

A more intuitive approach would be to argue as follows. How far back in a string do we have to examine at any bit position with $SH_i = 0$ to know that we are processing a Z or a G string? The answer is two bits. So, when $SH_i = 0$, we only need to examine the the $(i-1)$th and the $(i-2)$th bits. Combining the above findings, we have the following equation:

$$C_{fine} = (G_{i-2}G_{i-1} \vee T_{i-2}Z_{i-1} \vee Z_{i-2}G_{i-1})\overline{C}_{i-1} \vee (Z_{i-2}Z_{i-1} \vee T_{i-2}G_{i-1} \vee G_{i-2}Z_{i-1})C_{i-1}$$

Grouping, simplifying, and regconizing that we can substitute A_{i-1} for G_{i-1} and $\overline{A}_{i-1}$ for Z_{i-1}, we obtain

$$C_{fine} = \overline{C}_{i-1} \oplus (\overline{T_{i-2} \oplus A_{i-1}})$$

or equivalently

$$C_{fine} = C_{i-1} \oplus T_{i-2} \oplus A_{i-1} \tag{3}$$

In a sense, the RS/6000 method requires the maintenance of a global state. For SH_{coarse}, for example, we need to know which states (the N, J, and F states) we are in and for C_{fine}, whether we are processing a Z or a G string. But there is a more distributed way. Given that we have to detect the above patterns, how many bits do we have to examine before we can declare a pattern found? Upon a moment's reflection, we can conclude that only three bits need to be examined in a group. To detect the above four patterns, for example, we need to examine the following 3-bit patterns at each bit position.

$$U_i = Z_{i-2}Z_{i-1}Z_i \vee T_{i-2}G_{i-1}Z_i \vee G_{i-2}Z_{i-1}Z_i \vee G_{i-2}G_{i-1}G_i \vee T_{i-2}Z_{i-1}G_i \vee Z_{i-2}G_{i-1}G_i$$

Grouping, we have

$$U_i = [(Z_{i-2} \vee G_{i-2})Z_{i-1} \vee T_{i-2}G_{i-1}]\, Z_i \vee [(G_{i-2} \vee Z_{i-2})G_{i-1} \vee T_{i-2}Z_{i-1}]\, G_i$$

or

$$U_i = (\overline{T}_{i-2}\, Z_{i-1} \vee T_{i-2}G_{i-1})\, Z_i \vee (\overline{T}_{i-2}G_{i-1} \vee T_{i-2}\, Z_{i-1})G_i$$

Again substituting A_{i-1} for G_{i-1} and $\overline{A}_{i-1}$ for Z_{i-1}, we obtain

$$U_i = (\overline{T_{i-2} \oplus A_{i-1}})Z_i \vee (T_{i-2} \oplus A_{i-1})G_i$$

In addition, we need to account for the fact that SH_i = 1 whenever T_{i-1} = 1 (as explained below); hence, the equation becomes

$$U_i = T_{i-1} \vee \overline{T}_{i-1}\left[(\overline{T_{i-2} \oplus A_{i-1}})Z_i \vee (T_{i-2} \oplus A_{i-1})G_i\right] \qquad (4)$$

The first occurrence of U_i indicates the location of the leading one or zero. Additional (parallel) means must be provided to detect such an event. Note that as more and more patterns need to be detected, this distributed scheme becomes more and more attractive when compared with the first one.

In this distributed scheme, developing the equation for C_{fine} takes a bit more (conceptual) work. How do we know when we need to adjust? This question can be answered by examining all possible 3-bit patterns shown in Table 1. Patterns with a T in the middle need not be examined because they always produce an SH_i = 1. The reason is as follows. For $Z_{i-2}T_{i-1}X_i$ (X represents don't care), we know that the $Z_{i-2}T_{i-1}$ bit pattern would have already caused SH_{i-1} to be zero. What happens at bit position i really does not matter. The same argument holds for the $G_{i-2}T_{i-1}X_i$ patterns. Finally, $T_{i-2}T_{i-1}X_i$ always produces an SH_i = 1. Consequently, we can conclude that any 3-bit patterns with T_{i-1} = 1 need not be examined for fine adjustment because the coarse adjustment unit can be designed to always output an SH_i = 1 upon detecting these patterns.

Returning to the table, the first column contains the bit patterns of the string at the $(i-2)$th to the ith bit positions. The location of the leading one is (arbitrarily) assumed to be at bit position $i-1$.[2] The second and third columns are the sums of the bit pattern with the carry into the ith bit equal to one and zero, respectively. The fourth column indicates the condition of C_i under which adjustment is needed.

In ROW 1, we know we never need to adjust because $Z_{i-2}Z_{i-1}\, Z_i$ can not produce an $SH_i = 0$. In the second row, when C_i = 0, the most significant non-sign bit is actually at

[2]This **is possible by properly wiring the** T_i, Z_i, **and** G_i **terms.**

$X_{i-2}X_{i-1}X_i$	Sum ($C_i = 1$)	Sum ($C_i = 0$)	Adjustment
ZZZ	001	000	0
ZZT	010	001	$\overline{C_i}$
ZZG	011	010	0
ZGZ	101	100	0
ZGT	110	101	C_i
ZGG	111	110	0
TZZ	101	100	0
TZG	111	110	0
TZT	110	101	C_i
TGZ	001	000	0
TGT	010	001	$\overline{C_i}$
TGG	101	100	0
GZZ	001	000	0
GZT	010	001	$\overline{C_i}$
GZG	011	010	0
GGZ	101	100	0
GGT	110	101	C_i
GGG	111	110	0

Table 1: Developing the logic equation for C_{fine}, the fine adjustment

the ith bit position, requiring therefore a left shift of one more bit. Entries in the other rows can be interpreted similarly. The equation for C_{fine} can be written from the table as

$$C_{find} = (Z_{i-2}G_{i-1} \vee G_{i-2}G_{i-1} \vee T_{i-2}Z_{i-1})T_iC_i \vee (Z_{i-2}Z_{i-1} \vee G_{i-2}Z_{i-1} \vee T_{i-2}G_{i-1})T_i\overline{C}_i$$

which can be rewritten as

$$C_{fine} = [(Z_{i-2} \vee G_{i-2})G_{i-1} \vee T_{i-2}Z_{i-1}]\, T_iC_i \vee [(Z_{i-2} \vee G_{i-2})Z_{i-1} \vee T_{i-2}G_{i-1}]\, T_i\overline{C}_i$$

and as

$$C_{fine} = (\overline{T}_{i-2}G_{i-1} \vee T_{i-2}Z_{i-1})T_iC_i \vee (\overline{T}_{i-2}Z_{i-1} \vee T_{i-2}G_{i-1})T_i\overline{C}_i \tag{5}$$

Further, because patterns with $T_{i-1} = 1$ never occur, we can substitute A_{i-1} for G_{i-1} and $\overline{A}_{i-1}$ for Z_{i-1}, obtaining

$$C_{fine} = (\overline{T}_{i-2}A_{i-1} \vee T_{i-2}\overline{A}_{i-1})T_iC_i \vee (\overline{T}_{i-2}\overline{A}_{i-1} \vee T_{i-2}A_{i-1})T_i\overline{C}_i$$

so that

$$C_{fine} = (T_{i-2} \oplus A_{i-1})T_iC_i \vee (\overline{T_{i-2} \oplus A_{i-1}})T_i\overline{C}_i$$

Hence,

$$C_{fine} = T_i\left[C_i \oplus (\overline{T_{i-2} \oplus A_{i-1}})\right] \tag{6}$$

A possible precharged implementation of Eqns (4) and (6) has been given in Fig. 2. This circuit is obtained from Kershaw et *al.* [7]. In this particular implementation, the output of LOP is represented in a l-of-N code stored in the L_i array. An E signal, which is the logical NOR of all the E_i's, indicates whether a one-bit fine adjustment is needed. The top portion of the circuit is the Manchester carry chain for the adder and is not considered part of the circuit. Initially, F_0 is grounded, discharging F_i depending on the values of the intermediate U_i's. Note that in Eqn (6), T_iC_i can be replaced by C_{i-1}, saving some hardware and obtaining a similar equation as Eqn (3):

$$C_{fine} = \overline{C}_{i-1} \oplus T_{i-2} \oplus A_{i-1} \tag{7}$$

The difference between Eqn (7) and Eqn (3) is important to note. The latter is derived based on the assumption that $SH_{total} = SH_{coarse} - C_{fine}$ and the former on the assumption that $SH_{total} = SH_{coarse} + C_{fine}$. The C_{fine}'s should therefore differ.

2.3 Modification for Sign-Magnitude Numbers

The above presentation of LOP assumes that the input operands are represented in a 2's complement form. For sign-magnitude numbers as dictated by the IEEE 754 standard for the format of the significands, the above schemes do not work. This is because when the result is negative, it must be converted back to a sign-magnitude form with another addition step, possibly changing the position of the leading one or zero.

There are (at least) two ways to solve this problem. The first method uses a one's complement adder. When the result is negative, it is simply bit-inverted. LOP in this case is straightforward because bit inversion does not cause the position of the leading one or

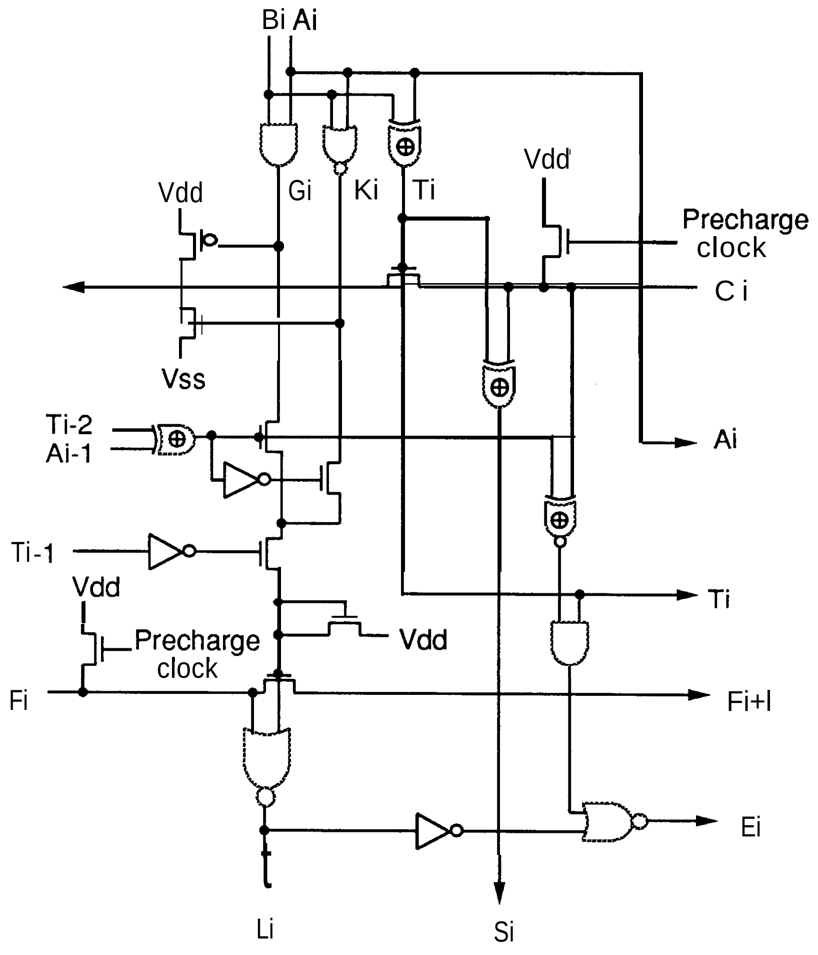

Figure 2: A possible precharged implementation of leading one prediction

zero to change. The LOP unit in this case must be able to detect both leading zero and one. One might think that the use of a one's complement adder will significantly slow down the addition time because the end-around carry must be added. But the farthest distance a carry has to travel in an N-bit addition is at most N-bits; hence, other than the fact that the (end-around) carry has to travel a long distance from the MSB to the LSB, the penalty in addition time is indeed negligible.

The second method is to always subtract the smaller operand from the larger one, ensuring a positive result. This can be done by using a magnitude comparator and a swapper as in the Intel i860. In this case, LOP is simpler because only the pattern T^*GZ^* will produce a string of preceding zero's.[3] The only complication arises from the logic for the fine adjustment at the MSB because the result may have already been normalized, causing the LOP unit to identify an incorrect leading one. The logic for C_{fine} needs to be modified to account for this special case.

For FP adders that perform both $A + B$ and $A + B + 1$ at the same time to avoid the time penalty of the extra rounding step [1], LOP is slightly more complicated. The LOP unit needs to know the correct C_i to examine for fine adjustment. Since LOP is only needed when the operation is a subtraction and when the exponents of the operands differ by at most one, one can observe the LSB of the smaller operand (i.e., the one to be shifted for alignment) to determine the correct C_i to observe. More detail can be found in the SNAP FP adder described in Quach and Flynn [8].

3 Generalization to Parallel Addit ion

Though normally not thought of as such, parallel addition is also a BPD problem. In parallel addition, to compute the final sum in a bit, we need to know whether or not the bit pattern of the lower-order bits will produce a carry in. In other words, we need to detect the following bit pattern:

$$T^*G$$

or more precisely, the pattern

$$T^*GX^*$$

where X represents a don't care. The difference between CLA and LOP is interesting to note. In CLA, the problem is done once a G is found after a string of T's and we don't care about what follows. In LOP, after finding the pattern T^*G, we still have to make sure that the following string is a Z string.

Techniques developed for parallel addition may now be applied to BPD. The Ling's addition scheme [9, 10, 11, 12], for example, can be borrowed. The sole requirement for using Ling's scheme when detecting a bit pattern $X^*Y Z^*$, is that $Y \supset X$[4] so that $Y_i = X_iY_i$. The advantage of using Ling's scheme comes from the hardware reduction at the input circuitry.[5]

[3]When denormalized numbers are allowed in the input, the pattern Z^* also needs to be detected.

[4] Y implies X.

[5]Here, we assume that the adder does not have to be implemented. When an adder and an LOP unit need to be implemented at the same time, the advantage of Ling's scheme is mostly in speed.

To detect the pattern P^*GZ^*, for example, Ling's scheme allows the implementation of the group generate function as

$$G_{group} = P(GZZZ \vee GZZ \vee PGZ \vee PPG) \tag{8}$$

Eqn (8) is preferable to its un-factored counterpart because the leading P term can be implemented at the lowest stage of the CLA tree as pointed out by Quach and Flynn [12], resulting in a simpler group generate function. In LOP, since we are interested in detecting T^*GZ^* and since G does not $\supset$ G, Ling's scheme can not be applied here. Another optimization often used in CLA adders is to substitute P for T, which is both faster and cheaper (in CMOS). The current generalization reveals that such an optimization is also not possible because in CLA, a PPPGPP string will generate a carry out but will not produce a string of zeros. In general, such a substitution requires that the ending string be a don't care string (i.e., an empty string).

In CLA, the group generate function is

$$G_{group} = G_{i-3} \vee P_{i-3}G_{i-2} \vee P_{i-3}P_{i-2}G_{i-1} \vee P_{i-3}\, P_{i-2}\, P_{i-1}\, G_i$$

Because this is simpler than Eqn (1) and because Ling's scheme can be used to speed up CLA, but not LOP, it is unlikely that LOP can be performed faster than CLA for a given technology using a similar scheme.

It is generally believed that in LOP, information flows from the MSB to LSB and in CLA, from the LSB to MSB. We believe that it is more general to think of information as flowing from the MSB to LSB in both cases. But this could be a moot point.

4 Application to Sticky Bit Computation

In many high-speed multiplication algorithms, the partial products are generated in parallel, followed by a summation step to reduce these partial products into two terms, sum and carry. The lower-order bits (N-l bits for an NxN-bit multiplication) need to be added and then examined to determine the sticky bit. Most implementations first add up these sum and carry terms and then detect for zero. By recognizing that this is a BPD problem, we can detect the trailing zero's directly from the sum and carry terms. Specifically, we need to detect the following bit patterns starting from the (N + 2)th bit position of the sum and carry terms:

$$Z^{N-1}$$

and

$$T^jGZ^k$$

where $j + k = N - 2$ and j, k are integer $\in [0, N-2]$. In addition, we need to detect the bit pattern P^*GX^*, which produces a carry into the higher-order N+1 bits. Note that the first bit pattern produces a carry-out=0 and the second a carry-out=l. The last pattern is basically the carry lookahead chain used in a CLA adder. So, only three chains need to be implemented, a considerable hardware savings over a design that does implement the adder.

One of the advantages of using redundant binary multipliers is in the reduction in its rounding hardware [13, 14, 15]. The application of LOP to sticky bit computation in conventional multipliers will definitely reduce this advantage.

5 Conclusion

In this paper, we have presented the theory of leading one prediction (LOP) in more detail in the framework of bit pattern detection and described two possible implementations. The first implementation is similar to the one used in the RS/6000 processor. The second one is a distributed version of the first, consuming less hardware when multiple patterns are to be detected. We have also shown how to adopt LOP for sign-magnitude numbers.

By treating LOP as a bit pattern detection problem, we show that both carry lookahead in parallel addition and sticky bit computation share the same nature. Most of the materials contained in this paper is not new, but we feel that the subject of LOP needs to be treated more systematically in the framework of bit pattern detection.

References

[1] N. T. Quach and M. J. Flynn, "An Improved Floating- Point Addition Algorithm ," Tech. Rep. CSL-TR-90-442, Stanford University, Aug. 1990.

[2] F. A. Ware, W. H. McAllister, J. R. Carlson, D. K. Sun, and R. J. Vlach, "64 Bit Monolithic Floating Point Processors," *IEEE Journal of Solid-State Circuit, vol.* SC-17, no. 5, pp. 898–907, Oct. 1982.

[3] W. P. Hays, R. N. Kershaw, L. E. Bays, J. R. Boddie, E. F. Fields, R. L. Freyman, C. J. Garen, J. Hartung, J. J. Klinikowski, C. R. Miller, K. Mondal, H. S. Moscovits, Y. Rotblum, W. A. Stocker, J. Tow, and L. V. Tran, "A 32-bit VLSI Digital Signal Processor," *IEEE Journal **of** Solid-State Circuit,* vol. SC-20, no. 5, pp. 998–1004, Oct. 1985.

[4] B. J. Benschneider, W. J. Bowhill, E. M. Cooper, M. N. Gavrielov, P. E. Gronowski, V. K. Maheshwari, V. Peng, J. D. Pickholtz, and S. Samudrala, "A Pipelined 50-Mhz CMOS 64-bit Floating-Point Arithmetic Processor," *IEEE Transactions on Computers,* vol. 24, no. 5, pp. 1317–1323, Oct. 1989.

[5] E. Hokenek and R. K. Montoye, "Leading-Zero Anticipator (LZA) in the IBM RISC System/6000 Floating-Point Execution Unit," *IBM Journal **of** Res. and Dev., vol. 34,* no. 1, pp. 71–77, Jan. 1990.

[6] H. P. Sit, M. R. Nofai, and S. Kim, "An 80MFLOPS Floating-Point Engine in the i860 Processor ," in *Proc. of International Conference on Computer Design,* pp. 374-379, 1989.

[7] R. N. Kershaw, L. E. Bays, R. L. Freyman, J. Klinikowski, C. R. Miller, K. Mondal, H. S. Moscovits, W. A. Stocker, and L. V. Tran, "A Programmable Digital Signal Processor with 32b Floating-Point Arithmetic," in *In Proc. **of** the IEEE International Solid-State Circuit Conference,* pp. 92-93, 1985.

[8] N. T. Quach and M. J. Flynn, "The SNAP Floating-Point Adder," Tech. Rep. In Preparation, Stanford University, 1991.

[9] H. Ling, "High Speed Binary Adder," *IBM Journal **of** Res. and Dev.,* vol. 25, no. 3, pp. 156-166, May 1981.

[10] G. Bewick, P. Song, G. DeMicheli, and M. J. Flynn, "Approaching a Nanosecond: A 32-Bit Adder," in *Proc. **of** International Conference on Computer Design,* pp. 221–224, 1988.

[11] R. W. Doran, "Variants of an Improved Carry Lookahead Adder," *IEEE Transactions on Computers,* vol. C-37, no. 9, pp. 1110-1113, Sep. 1988.

[12] N. T. Quach and M. J. Flynn, "High-Speed Addition in CMOS," Tech. Rep. CSL-TR-90-415, Stanford University, Fab. 1990.

[13] N. T. Quach, N. Takagi, and M. J. Flynn, "On Fast IEEE Rounding," Tech. Rep. CSL-TR-91-459, Stanford University, Jan. 1991.

[14] N. Takagi, H. Yasuura, and S. Yajima, "High-Speed VLSI Multiplication Algorithm with a Redundant Binary Addition Tree," *IEEE Transactions on Computers, vol. C-34,* no. 9, pp. 789-796, Sep. 1985.

[15] S. Kuninobu, T. Nishiyama, H. Edamatsu, T. Taniguchi, and N. Takagi, "Design of High Speed MOS Multiplier Using Redundant Binary Representation," in *Proc. **of** the* 8^{th} *Symposium on Computer Arithmetic,* pp. 80–86, 1987.