AN IMPROVED ALGORITHM FOR HIGH-SPEED FLOATING-POINT ADDITION

Nhon T. Quach and Michael J. Flynn

Technical Report: CSL-TR-90-442

August 1990

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

AN IMPROVED ALGORITHM FOR HIGH-SPEED FLOATING-POINT ADDITION

by

Nhon T. Quach and Michael J. Flynn

Technical Report: CSL-TR-90-442

August 1990

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305-4055

Abstract

This paper describes an improved, IEEE conforming floating-point addition algorithm. This algorithm has only one addition step involving the significand in the worst-case path, hence offering a considerable speed advantage over the existing algorithms, which typically require two to three addition steps.

Key Words and Phrases: Improved floating-point addition algorithm, floating-point hardware design, IEEE rounding

Contents

List of Figures

List of Tables

1 Introduction

Floating-point (FP) addition is one of the most frequent arithmetic operations in scientific computing. Despite its conceptual simplicity, FP addition in most high-speed arithmetic units today has roughly the same latency as FP multiplication. This is largely because most existing FP addition algorithms require two to three addition steps involving the significand (as explained below), a relatively time-consuming operation. In this paper, we describe a new FP addition algorithm. The algorithm has only one significand addition step in the worst case path, hence offering a considerable speed advantage over earlier algorithms. We briefly review these (existing) FP addition algorithms in Section 2 and present ours in Section 3. Concluding remarks are given in Section 4. Appendix A is a collected review of the notation used in this paper.

2 A Brief Review of FP Addition Algorithm

An FP addition operation consists of the following steps [1]:

1. Exponent subtraction (*ES*): Subtract the exponents and denote the difference $|E_a - E_b| = d$.
2. Alignment (*Align*): Right shift the significand of the smaller operand by d bits. Denote the larger exponent E_f.
3. Significand addition (*SA*): Perform addition or subtraction according to the effective operation, E_o, which is the arithmetic operation actually carried out by the adder in the FP unit.
4. Conversion (*Conv*): Convert the result to sign-magnitude representation if the result is negative. The conversion is done with an addition step. Denote the result S_f.
5. Leading one detection (*LOD*): Compute the amount of left or right shift needed and denote it E_n. E_n is positive for a right shift and negative otherwise.
6. Normalization (*Norm*): Normalize the significand by shifting E_n bits and add E_n to E_f.
7. Rounding (*Round*): Perform IEEE rounding [2] by adding "1" when necessary to the LSB of S_f. This step may cause an overflow, requiring a right shift. The exponent, E_f, in this case has to be incremented by 1.

The above algorithm (Algorithm A1) is slow because the composing steps in the addition operation are essentially performed serially. We can improve the algorithm in the following ways:

1. In Algorithm A1, the *Conv* step is only needed when the result is negative and can be avoided by swapping the significands. By examining the sign of the result of the ES step, we can swap the significands accordingly so that the smaller significand is

subtracted from the larger one. In the case of equal exponents, the result may still be negative and requires a conversion. But no rounding is needed in this case. Hence, rounding and conversion are made mutually exclusive by the swapping step, allowing us to combine them. Note that an associated advantage of swapping is that only a shifter is now needed.

2. The *LOD* step can be performed in parallel with the *SA* step, removing it from the critical path. This optimization is important when a massive left shift is required as a result of significand cancellation in the case of an effective subtraction.

3. So far, we have been able to reduce the number of steps down to: *ES*, *Swap*, *Align*, *SA* || *LOD*, *Conv* || *Round*, and *Norm* (the symbol "||" indicates that the steps can be executed in parallel). Algorithm A1 can be further optimized by recognizing that the *Align* and the *Norm* steps are mutually exclusive. Normalization requiring a large number of left shifts is needed only when $d \leq 1$. Conversely, alignment requiring a large number of right shifts is needed only when $d > 1$. By distinguishing these two cases, only one full length shift, either the alignment or the normalization one, is in the critical path [3].

The steps in Algorithm A1 and this improved algorithm (Algorithm A2) are summarized in Table 1. In Algorithm A2, the *Pred* step in the $d \leq 1$ path predicts whether a one-bit right shift is needed to align the significands. Note that Algorithm A2 increases the speed by executing more steps in parallel, requiring therefore more hardware.

Table 1: Steps in Conventional FP Addition Algorithms.

Algorithm A1	Algorithm A2	
	$d \leq 1$ and Effective Subtraction	Others
ES	*Pred* + *Swap*	*ES* + *Swap*
Align		*Align*
SA	*SA* || *LOD*	*SA*
Conv	*Conv* || *Round*	*Round*
LOD		
Norm	*Norm*	
Round	*select*	*select*

Algorithm A2 is commonly used, in one form or another, in today's high-performance FP arithmetic units [4, 5, 6]. From the above discussion, we see that Algorithm A2 requires 2 addition steps involving the significands in the critical paths in both the $d \leq 1$ and the $d > 1$ paths (*SA* and *Round*).

3 The New Algorithm

3.1 General Ideas

The key ideas behind our approach can be summarized as follows.

- In Algorithm A2, the SA step requires one of the significands to be 2's complemented in the case of an effective subtraction. We observed that this complementation step and the rounding one are mutually exclusive and can therefore be combined.
- In the IEEE *round to nearest* (*RTN*) mode, computing $A+B$ and $A+B+1$ is sufficient to account for all the normalization possibilities to be discussed below.[1] By selecting the results using C_{in} computed based on the lower order bits of the significands, complementation and rounding can be done simultaneously, saving one addition step.

Table 2: Steps in the Present FP Addition Algorithm.

The New Algorithm	
$d \leq 1$ and Effective Subtraction	Others
$Pred + Swap$	$ES + Swap$
$SA \parallel Conv \parallel Round \parallel LOD$	$Align$
$Norm$	$SA \parallel Round$
$select$	$select$

Hence, the challenge is in deriving the equation for C_{in}. Since in FP addition, normalization of the result may require a one-bit right shift, no shift, or a left shift which may be of as many bits as the length of the significand, C_{in} needs to account for all these normalization possibilities, such that the final selected result will appear to be rounded properly.

Because the significand is 53-bit and because a right shift of up to 52 bits may be needed during alignment, a 105-bit adder is potentially needed. Since we are only concerned with the higher order 53-bit, we use a 53-bit adder in the interest of hardware efficiency. In the case of complementation, a "1" needs to be added at the bit position 105. How far left into the higher order bit this complementing "1" bit, C_c, propagates and whether it reaches the adder, clearly depends on the lower order bits of the shifted significand. When C_c does reach the adder, it is added to the "L" bit position. The rounding "1" bit C_r, on the other hand, is always added to the rounding bit position "R" (Fig. 1).

When the bit pattern of the shifted significand is such that C_c reaches the real adder, the guard bit G, the round bit R, and the sticky bit s, must all be zero; therefore, no rounding is required. Hence, complementation and rounding, as far as the adder is concerned, are

[1] In the *round to positive* and *negative infinity* (*RTPI* and *RTNI*) modes, it is necessary to compute not only $A+B$ and $A+B+1$, but also $A+B+2$, making it harder than the *RTN* mode. A row of half adder has to be used to add "2" to $A+B$. This case is discussed in more detail in the appendix.

mutually exclusive and can be combined. Table 2 lists the steps in the new algorithm. The number of significand addition step in both paths have been reduced to one.

While it is clear that this argument holds for the cases when the result of the SA step needs a left shift and needs no shift, it is less so for the case when the result needs a one-bit right shift, because rounding in this case requires adding "2", not "1", to $A + B$. The explanation lies in the definition of the *RTN* mode.[2] In the case of a one-bit right shift, it is only necessary to add "2" to $A + B$ when the L bit of $A + B$ is "1" because after the right shift, the L becomes the G bit. Hence, adding "1" to the L of $A+B$ causes the carry into the N (next to LSB) bit, to be true, equivalent to adding "2" to $A + B$.

Implementing the *round to zero* mode is easy because a simple truncation suffices in this case and no rounding is needed. For the *RTPI* and *RTNI* modes, the situation is more complicated and we treat them in Appendix B to avoid digression. The logic equation for C_{in} for the *RTN* mode, which is the default IEEE rounding mode, will be derived below.

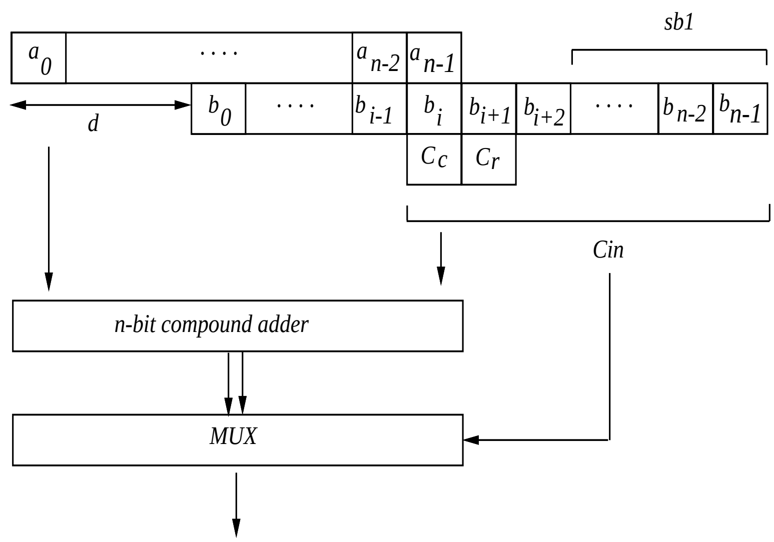

Figure 1: Explanation of the General Approach for the *RTN* Mode.

3.2 Logic Equation for C_{in} for the *RTN* Mode:

To derive the logic equation for C_{in}, we differentiate 3 cases. Case 1 is when E_o is addition, case 2 when E_o is subtraction and $d > 1$ and case 3 when E_o is subtraction and $d \leq 1$. In the actual implementation to be described in the following section, cases 1 and 2 are merged to form one path, controlled by g_{in} (i.e., g_{in} playing the role of C_{in}). Case 3 forms another path controlled by l_{in} (i.e., l_{in} playing the role of C_{in}). In what follows, we first derive a generic equation for C_{in} and then apply it to the three cases to arrive at the l_{in} and g_{in} equations.

[2]The *RTN* mode rounds up a number in all cases except a tie, whence it rounds up when the LSB is odd and truncates when the LSB is even.

3.2.1 General Equation for C_{in}:

As defined in Appendix A, C_r is the bit needed to perform rounding. It can be determined from Table 3. From the table,

$$C_r = S \vee L \tag{1}$$

Table 3: Truth Table for the *Round to Nearest* Mode

L	G	S	C_r
0	0	0	x
0	0	1	x
0	1	0	0
0	1	1	1
1	0	0	x
1	0	1	x
1	1	0	1
1	1	1	1

The equation for C_c can be written as

$$C_c = \overline{G}\,\overline{R}\,\overline{s}$$

This is because only when the G bit, the R bit, and the s bit are all zero does C_c reach the adder. Note that in the case of an effective subtraction, the equations for G, R, and s have to be written with complementation taken into account.

The equation for C_{in} is simply

$$C_{in} = GC_r \vee C_c$$

3.2.2 Applying C_{in} to the Three Cases

1. **Case 1: E_o is addition** Since E_o is addition, no complementation of either operand is needed; therefore, $C_c = 0$. We further differentiate two cases: result needing no right shift (i.e., no normalization is needed) and result needing a one-bit right shift. In the case of addition, only these two cases are possible because the value of a normalized significand ranges between [1,2).

   (a) **Result needing no right shift (NRS)** This case occurs when there is no carry-out from the g adder with carry-in=0. The logic equation is

   $$NRS = \overline{E}_o \overline{g}^0_{out}$$

   The logic equations for L, G, and S are

   $$L = a_{n-1} \oplus b_{n-1}$$

$$S = R \vee s = b_{n+1} \vee s$$

$$G = b_n$$

From Eq. 1, we have

$$C_r = S \vee L = b_{n+1} \vee s \vee (a_{n-1} \oplus b_{n-1})$$

C_{in} in this case is

$$C_{in} = GC_r = b_n \left[b_{n+1} \vee s \vee (a_{n-1} \oplus b_{n-1}) \right]$$

(b) **Result needing one bit right shift (ORS)** This case occurs when E_o is addition and there is a carry-out from the adder. The logic equation is

$$ORS = \overline{E}_o g^0_{out}$$

Because the result needs a one-bit right shift, the equations for L, G, and S must take it into account. We have:

$$L = a_{n-2} \oplus b_{n-2} \oplus a_{n-1} b_{n-1}$$

$$S = b_n \vee b_{n+1} \vee s$$

$$G = a_{n-1} \oplus b_{n-1}$$

So that C_r becomes

$$\begin{aligned} C_r &= S \vee L \\ &= b_n \vee b_{n+1} \vee s \vee (a_{n-2} \oplus b_{n-2} \oplus a_{n-1} b_{n-1}) \end{aligned}$$

and

$$\begin{aligned} C_{in} &= GC_r \\ &= (a_{n-1} \oplus b_{n-1}) \left[b_n \vee b_{n+1} \vee s \vee (a_{n-2} \oplus b_{n-2} \oplus a_{n-1} b_{n-1}) \right] \end{aligned}$$

Using the identity $(x \oplus y)(z \oplus xy) = (x \oplus y)z$ and simplifying, we get

$$C_{in} = (a_{n-1} \oplus b_{n-1})(b_n \vee b_{n+1} \vee s \vee a_{n-2} \oplus b_{n-2})$$

2. **Case 2: E_o is subtraction and $d > 1$:** We again differentiate two cases: result needing no left shift and result needing one left shift. Only these two cases are possible because $d > 1$. The following example clarifies this point.

Example 1:

$$\begin{array}{lr} & 1.10000000000 \\ - & 0.01111111111 \\ \hline & \downarrow \\ & 1.10000000000 \\ & 1.10000000000 \\ + & 1 \\ \hline & 11.00000000001 \end{array} \qquad \begin{array}{lr} & 1.00000000000 \\ - & 0.01111111111 \\ \hline & \downarrow \\ & 1.00000000000 \\ & 1.10000000000 \\ + & 1 \\ \hline & 10.10000000001 \end{array}$$

(a) Result needs no left shift (b) Result needs one left shift

Examples 1(a) and 1(b) show what can happen to the significands. The exponents differ by 2 and the significand of the smaller operand has to be right shifted by 2 bits. In both examples, the overflow bit is ignored. The significand in Example 1(b) needs a one-bit left shift to normalize the result while that in Example 1(a) does not.

(a) **Result needing no left shift (NLS)** This case is similar to the addition case with no right shift. There is a complication, however. The complication arises from the complementation of the smaller significand because E_o is subtraction. The equation for G, R, and s can be written with the help of Table 4. The columns in the table shows the b_n, b_{n+1}, and s bits after the alignment step, the 1's complement step, and the 2's complement step. The 1's complement and the 2's complement steps are required because E_o is subtraction. The meaning of *prop* and *kill* are as follows. Recall the definition of the s bit, which is the ORing of all the shifted b_i bits; a "0" means that all the shifted bits are zero. After 1's complementation, $\overline{s}$ means first complementing the b_i bits and then ORing them. The case of $s = 0$ after 1's complementation allows a carry-in to be propagated and is denoted *prop*. Similarly, an $s = 1$ means that at least one of the bit must be 1. After 1's complementation, this means that at least one of the bits must be 0. This bit pattern does not allow a "1" to be propagated and is denoted *kill* in the table. The "after 2's complement" column is obtained by adding the complementing "1" to the "after 1's complement" column, creating an extra column (C_c) in the process. During the 2's complement process, adding "1" to a sticky bit turns it from a *prop* condition into a "0" and a *kill* condition into a "1".

From Table 4, we know that C_c is only true when b_n, b_{n+1}, and s are all zero. Hence,

$$L = C_c \oplus a_{n-1} \oplus b_{n-1} = \overline{b}_n \overline{b}_{n+1} \overline{s} \oplus a_{n-1} \oplus b_{n-1}$$

The equations for S and G can be written based on the same table. From the table,

$$S = R \vee s = b_{n+1} \vee s$$

Table 4: Truth Table for Determining Guard, Round, and Sticky Bits in Case 2a.

After Shifting			After 1's Complement			After 2's Complement			
b_n	b_{n+1}	s	$\overline{b}_n$	$\overline{b}_{n+1}$	$\overline{s}$	C_c	G	R	s
0	0	0	1	1	prop	1	0	0	0
0	0	1	1	1	kill	0	1	1	1
0	1	0	1	0	prop	0	1	1	0
0	1	1	1	0	kill	0	1	0	1
1	0	0	0	1	prop	0	1	0	0
1	0	1	0	1	kill	0	0	1	1
1	1	0	0	0	prop	0	0	1	0
1	1	1	0	0	kill	0	0	0	1

$$G = \overline{b}_n \oplus \overline{b}_{n+1}\overline{s}$$

From Eq. 1,

$$\begin{aligned} C_r &= S \vee L \\ &= b_{n+1} \vee s \vee \overline{b}_n\overline{b}_{n+1}\overline{s} \oplus a_{n-1} \oplus b_{n-1} \end{aligned}$$

and

$$\begin{aligned} C_{in} &= GC_r \vee C_c \\ &= (\overline{b}_n \oplus \overline{b}_{n+1}\overline{s})(b_{n+1} \vee s \vee \overline{b}_n\overline{b}_{n+1}\overline{s} \oplus a_{n-1} \oplus b_{n-1}) \vee \overline{b}_n\overline{b}_{n+1}\overline{s} \\ &= \overline{b}_n(b_{n+1} \vee s) \vee b_n\overline{b}_{n+1}\overline{s}(a_{n-1} \oplus b_{n-1}) \vee \overline{b}_n\overline{b}_{n+1}\overline{s} \end{aligned}$$

Simplifying, we have

$$C_{in} = \overline{b}_n \vee b_n\overline{b}_{n+1}\overline{s}(a_{n-1} \oplus b_{n-1}) \tag{2}$$

This case occurs when $NLS = E_o\left[(\overline{b_n \vee b_{n+1} \vee s})g_0^1 \vee (b_n \vee b_{n+1} \vee s)g_0^0\right]$. This is because when the complementing "1" does not reach the adder (i.e., if $b_n \vee b_{n+1} \vee s$ is true), then g_0^0 (the MSB of the result of the adder with carry-in=0) should be examined; otherwise, g_0^1 (the adder with carry-in=1) should be examined.

There is a potential confusion in the logic equations involving the b_{n-1} term because of the complementation. If the $a_{n-1} \oplus b_{n-1}$ term is obtained from the adder circuitry, then there is no need to invert b_{n-1} because the adder control logic has done so for us. If, on the other hand, the $a_{n-1} \oplus b_{n-1}$ term is to be implemented locally, then all b_{n-1} and b_{n-2} terms in the logic equations need to be inverted. Throughout this paper, we assume that the b_{n-1} and b_{n-2} terms are obtained from the adder circuitry. To make it explicit, we denote $S_{g1} = a_{n-1} \oplus b_{n-1}$ and $S_{g2} = a_{n-2} \oplus b_{n-2}$. The subscript g indicates that the signal is obtained from the g adder (and subscript l from the l adder). Hence, Eq. 2 becomes

$$C_{in} = \overline{b}_n \vee b_n \overline{b}_{n+1} \overline{s} S_{g1}$$

(b) **Result needing one bit left shift (OLS)** This case occurs when

$$OLS = E_o \left[(\overline{b_n \vee b_{n+1} \vee s}) \overline{g}_0^1 \vee (b_n \vee b_{n+1} \vee s) \overline{g}_0^0 \right]$$

Table 5: Truth Table for Determining Guard, Round, and Sticky Bits in Case 2b.

After Shifting			After 2's Complement				C_r	$LC_r \vee C_c$	$L \oplus C_r$
b_n	b_{n+1}	s	C_c	$G(>> L)$	$R(>> G)$	$s(>> S)$		C_{in}	q
0	0	0	1	0	0	0	0	1	0
0	0	1	0	1	1	1	1	1	0
0	1	0	0	1	1	0	1	1	0
0	1	1	0	1	0	1	0	0	1
1	0	0	0	1	0	0	0	0	1
1	0	1	0	0	1	1	1	0	1
1	1	0	0	0	1	0	0	0	0
1	1	1	0	0	0	1	0	0	0

The equation in this case can be derived as in the previous case, but is conceptually more complicated because of the one-bit left shift. We develop the equation for C_{in} using Table 5. The "After 2's Complement" column is obtained in the same manner as that in Table 4. After a one-bit left shift, the G bit becomes the L bit, the R bit becomes the G bit, and the s bit becomes the final S bit.

To perform rounding, C_r is obtained based on the values of the G and the S bits (the original R and s bits). Recalling the definition of C_{in}, which is equal to $GC_r \vee C_c$, after the one-bit normalization shift, it becomes $LC_r \vee C_c$.

$$C_{in} = LC_r \vee C_c = \overline{b}_n(\overline{b}_{n+1} \vee \overline{s})$$

The q bit is the bit that needs to be shifted in, in case of a left shift because we have an adder of only 53-bit. Its logic equation is simply the mod 2 sum of the C_r bit and the L bit (the original G bit).

$$q = L \oplus C_r = \overline{b}_n b_{n+1} s \vee b_n \overline{b}_{n+1}$$

3. **Case 3: E_o is subtraction and $d \leq 1$:** We again have 2 cases: result needing no left shift and result needing many left shifts. The following example illustrates these two cases.

Example 2:

$$\begin{array}{rr} & 1.11111111110 \\ - & 0.11111100000 \\ \hline \end{array} \quad\quad \begin{array}{rr} & 1.00000000000 \\ - & 0.11111111111 \\ \hline \end{array}$$

$$\downarrow \quad\quad\quad\quad \downarrow$$

$$\begin{array}{rr} & 1.11111111110 \\ & 1.00000011111 \\ + & 1 \\ \hline & 11.00000011110 \end{array} \quad\quad \begin{array}{rr} & 1.00000000000 \\ & 1.00000000000 \\ + & 1 \\ \hline & 10.00000000001 \end{array}$$

(a) Result needs no left shift (b) Result needs many left shifts

In both examples, the exponents differ by 1 and a one-bit right shift is needed to align the significands. The overflow "1" in both cases is simply discarded. In Example 2(a), no normalization is needed while a 11-bit left shift is required for normalization in Example 2(b), which shows that it is possible for a many-bit left shift to occur even when there is a one-bit right shift during significand alignment.

(a) **Result needing no left shift** This case is basically the same as the second case (Case 2a) but can be simplified because both b_{n+1} and s equal zero.

$$L = \overline{b}_n \oplus S_{l1}$$

$$S = 0$$

$$G = b_n l_0^0$$

$$\begin{aligned} C_r &= S \vee L \\ &= \overline{b}_n \oplus S_{l1} \end{aligned}$$

and

$$\begin{aligned} C_{in} &= GC_r \vee C_c \\ &= l_0^0 b_n S_{l1} \vee \overline{b}_n \end{aligned}$$

(b) **Result needing many bits of left shift (MLS)** MLS may occur in two cases, as illustrated in Example 2. Case 1 is when the exponents are equal, which can be predicted by examining the LSBs of the exponents ($E_{a\langle 0\rangle}$ and $E_{b\langle 0\rangle}$). Case 2 is when the exponents differ by 1 and the significand of the smaller operand needs to be right shifted by one bit. The right shifted bit is called the b_n bit. But for Case 2, the C_c bit always reaches the adder independent of the b_n bit. This is because when $b_n = 0$, C_c will reach the adder and when $b_n = 1$, C_c reaches the b_n bit, causing the carry into adder to be true. Hence, all we need is to detect the first case.

$$C_{in} = \overline{E}_{a\langle 0\rangle} \oplus E_{b\langle 0\rangle}$$

We still need to derive the equation for the bit to be shifted in. The equation is simply

$$q = b_n$$

For an n-bit adder and a d-bit left shift, this bit will occupy the bit position $n-d$ (recall that our numbering convention starts from 0, so that the LSB is at bit $n-1$).

The equation for l_{in} can be obtained by ORing Cases 3a and 3b,

$$l_{in} = l_0^0 b_n S_{l1} \vee \overline{b}_n \vee \overline{E}_{a\langle 0\rangle} \oplus E_{b\langle 0\rangle}$$

3.2.3 Merging Case 1 and Case 2

From the preceding section, the equation for g_{in} is

$$\begin{aligned} g_{in} \quad = \quad & (ORS)S_{g1}(b_n \vee b_{n+1} \vee s \vee S_{g2}) \vee \\ & (NXS)\left[\overline{E}_o b_n(b_{n+1} \vee s \vee S_{g1}) \vee E_o(\overline{b}_n \vee b_n \overline{b}_{n+1}\overline{s}S_{g1})\right] \vee \\ & (OLS)\overline{b}_n(\overline{b}_{n+1} \vee \overline{s}) \end{aligned}$$

where

$$ORS = \overline{E}_o g_{out}^0$$

$$NXS = NRS \vee NLS = \overline{E}_o \overline{g}_{out}^0 \vee E_o\left[(\overline{b_n \vee b_{n+1} \vee s})g_0^1 \vee (b_n \vee b_{n+1} \vee s)g_0^0)\right]$$

and

$$OLS = E_o\left[(\overline{b_n \vee b_{n+1} \vee s})\overline{g}_0^1 \vee (b_n \vee b_{n+1} \vee s)\overline{g}_0^0\right]$$

Substituting and simplifying, we get

$$\begin{aligned} g_{in} \quad = \quad & \overline{E}_o\left[g_{out}^0 S_{g1}(b_n \vee b_{n+1} \vee s \vee S_{g2}) \vee \overline{g}_{out}^0 b_n(b_{n+1} \vee s \vee S_{g1})\right] \vee \\ & E_o\left\{\overline{b}_n\overline{b}_{n+1}\overline{s} \vee g_0^0\left[\overline{b}_n(b_{n+1} \vee s) \vee b_n\overline{b}_{n+1}\overline{s}S_{g1}\right] \vee \overline{g}_0^0\overline{b}_n(b_{n+1} \oplus s)\right\} \end{aligned}$$

The final C_{in}, which selects between the results of the l and the g paths, is true when E_o is subtraction and the true absolute difference of the exponent d is less than or equal to 1. (C_{in} is true when selecting the l path.)

4 Summary

A new floating-point addition algorithm has been presented. This algorithm has only one addition step involving the significand in the critical path while performing full IEEE rounding. In floating-point addition, 2's complementation of one of the significands is needed in the case of an effective subtraction. The key ideas presented in this paper are: first, complementation and rounding are mutually exclusive and can be combined. Second, for the *round to nearest* mode, pre-computing $A+B$ and $A+B+1$ is enough to account for all the normalization possibilities (*see* Section 3). *round to infinity* modes are more difficult to speed up than the *round to nearest* mode; they requires an extra row of half-adder to perform rounding correctly (*see* Appendix B).

5 Acknowledgement

The authors would like to thank Naofumi Takagi for his valuable comments on an early version of this paper.

References

[1] S. Waser and M. J. Flynn, *Introduction to Arithmetic for Digital Systems Designers.* New-York: Holts, Rinehart and Winston, 1982.

[2] The Institute of Electrical and Electronics Engineers, Inc., 345 East 47th Street, New York, NY 10017, USA, *ANSI/IEEE Std 754-1985: IEEE Standard for Binary Floating-Point Arithmetic*, 1985.

[3] M. P. Farmwald, *On the Design of High Perfromance Digital Arithmetic Units.* PhD thesis, Stanford University, Aug. 1981.

[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] P. Y. Lu, A. Jain, J. Kung, and P. H. Ang, "A 32-MFLOP 32b CMOS Floating-Point Processor," in *In Proc. of the IEEE International Solid-State Circuit Conference*, pp. 28–29, 1988.

[6] M. Birman, A. Samuels, G. Chu, T. Chuk, L. Hu, J. McLeod, and J. Barns, "Developing the WTL3170/3171 Sparc Floating-Point Coprocessor," *IEEE Micro*, pp. 55–64, Feb. 1990.

A Notation

In alphabetical order, this is a list of the symbols used in this paper:

$\vee$ Logical OR. Lowest precedence.

$\oplus$ Exclusive OR (sum mod 2). Precedence higher than $\vee$, but lower than AND.

$Juxtaposition$ AND. Highest precedence.

$A = a_0.a_1a_2\cdots a_{2n-1}$ Significand of the larger (in magnitude) operand. a_0 is the hidden one bit. $a_n = a_{n+1} = a_{n+2} = \cdots = a_{2n-1} = 0$ because no right shift of A is needed.

$Align$ The alignment step in the addition operation.

$B = b_0.b_1b_2\cdots b_{2n-1}$ Significand of the smaller (in magnitude) operand. b_0 is the hidden one bit. $b_n, b_{n+1}, \cdots, b_{2n-1}$ may not be equal to zero after alignment. When E_o is subtraction so that complementation of the smaller operand is needed, b_i (for $i \geq n+2$) is *before* complementation.

C_c A bit needs to be added to the LSB to perform complementation.

C_{in} This signal selects between the results from the g path and the l path.

C_r A bit that needs to be added to the G bit to perform IEEE rounding.

d Magnitude of the exponent difference.

E_o Effective operation. $E_o = 0$ for addition and $E_o = 1$ for subtraction.

E_a Exponent of the larger operand.

$E_{a\langle 0\rangle}$ LSB of E_a.

E_b Exponent of the smaller operand.

$E_{b\langle 0\rangle}$ LSB of E_b.

E_f Exponent of the result operand.

E_n Amount of shifts needed during the alignment step. E_n is positive if the significand needs a right shift and negative otherwise.

ES The significand addition step in the addition operation.

G Guard bit of the result to be rounded.

$g_0.g_1g_2\cdots g_{2n-1}$ The final sum result of the adder in the $d > 1$ path or when E_o is addition. This path is denoted the g path.

$g_0^1.g_1^1g_2^1\cdots g_{2n-1}^1$ The intermediate sum result of the adder with carry-in=1 in the $d > 1$ path and when E_o is addition. This path is denoted the l path.

$g_0^0.g_1^0g_2^0\cdots g_{2n-1}^0$ The intermediate sum result of the adder with carry-in=0 in the g path.

g_{in} This signal selects the result between $g_0^0.g_1^0g_2^0\cdots g_{2n-1}^0$ and $g_0^1.g_1^1g_2^1\cdots g_{2n-1}^1$.

g_{out}^0 Carry-out from the g adder with carry-in=0.

L Least significand bit of the result to be rounded.

$l_0.l_1l_2\cdots l_{2n-1}$ The final sum result of the adder in the $d \leq 1$ path.

$l^0_0.l^0_1l^0_2\cdots l^0_{2n-1}$ The intermediate sum result of the adder with carry-in =0 in the $d \leq 1$ path.

$l^1_0.l^1_1l^1_2\cdots l^1_{2n-1}$ The intermediate sum result of the adder with carry-in=1 in the $d \leq 1$ path.

l_{in} This signal selects between $l^1_0.l^1_1l^1_2\cdots l^1_{2n-1}$ and $l^0_0.l^0_1l^0_2\cdots l^0_{2n-1}$.

LOD The leading one detection step in the addition operation.

LSB Least significant bit.

MLS Indicate that a many-bit left shift is needed in the case of a subtraction.

MSB Most significant bit.

N Next to LSB.

NLS Indicate that no shift of the result is needed in the case of a subtraction.

NRS Indicate that no right shift of the result is needed in the case of an addition.

NXS $= NRS \vee NLS$.

n Length of the significand, equal to 53.

$Norm$ The significand normalization step in the addition operation.

OLS Indicates that a one-bit left shift is needed in case of a subtraction.

ORS Indicates that a one-bit right shift is needed in the case of addition.

$Pred$ The prediction step in the addition operation. This step is used to align the significands in case of $d \leq 1$.

q The bit to be shifted in when a left shift is needed during normalization.

$Round$ The rounding step in the addition operation.

R Round bit (or round bit position).

s $= b_{n+2} \vee b_{n+3} \vee \cdots \vee b_{2n-1}$.

S Final sticky bit of a significand to be rounded.

S_a Significand of the larger (in magnitude) operand.

SA Significand addition.

S_b Significand of the smaller (in magnitude) operand.

S_E Sign of the result.

S_f Significand of the result operand.

S_{g1} $= a_{n-1} \oplus b_{n-1}$ from the g adder.

S_{g2} $= a_{n-2} \oplus b_{n-2}$ from the g adder.

S_{l1} $= a_{n-1} \oplus b_{n-1}$ from the l adder.

S_{l2} $= a_{n-2} \oplus b_{n-2}$ from the l adder.

B C_{in} for other rounding Modes

For all other rounding modes, the equations for C_{in}, ORS, NXS, and OLS are the same as those for the *RTN* mode. Only the equations for g_{in}, l_{in}, and q differ. In this appendix, we develop the equations for the three other $IEEE$ rounding modes.

B.1 Round to 0

$$g_{in} = E_o(\overline{b_n \vee b_{n+1} \vee s})$$

$$l_{in} = E_o\overline{b}_n$$

In case of a right shift during normalization, the q bit is

$$q = \overline{b}_n(s \vee b_{n+1}) \vee b_n\overline{b}_{n+1}\overline{s}$$

or more concisely,

$$q = b_n \oplus (s \vee b_{n+1})$$

B.2 Round to $+\infty$

Round to infinity is much tougher to speed up than *RTN*. This is because when a right shift is required, as in the case of an effective addition, one has the possibility of adding 0, 1, and 2.[3] A less efficient way to implement this mode is to use three adders, simultaneously computing the three results at the same time.

Referring back to Fig. 1, we recognize that

- One only has a problem when the LSB, L in Fig. 1, is zero because the case of $L = 1$ can be treated in the same way as the *RTN* mode.
- In the case of $L = 0$, pre-computing $A + B$, $A + B + 1$, and $A + B + 2$ is not a problem because the three cases can be reduced down to two: $A + B$ and $A+B+2$. The case of $A+B+1$ can be accounted for by incrementing the $A+B$ case, because $L = 0$ (so there is no carry propagation). To compute $A + B + 2$, one needs to use a row of half adder, as shown in Fig. 2. By modifying the logic equation for g_{in} slightly, the *RTPI* and *RTNI* modes can be implemented in a similar manner as the *RTN* mode.

[3] Round to nearest does not have this problem because its definition affords a simple implementation, as pointed out in the main text, Section 3.2.3.

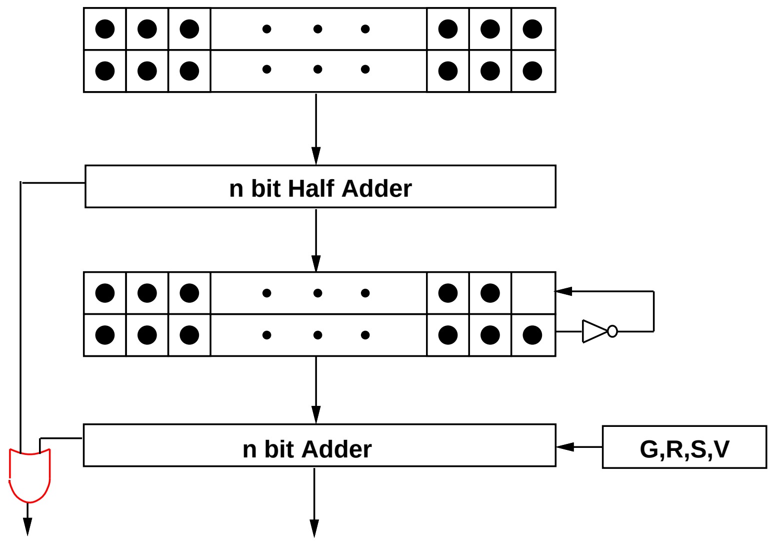

Figure 2: Hardware for Round to Positive Infinity.

Using a similar approach as that for the *RTN* mode and the above argument, we arrived at the following logic equation for g_{in}:

$$\begin{aligned} g_{in} &= (ORS)\overline{S}_E(a_{n-1} \oplus b_{n-1} \vee b_n \vee b_{n+1} \vee s) \vee \\ &\quad (NXS)\left\{E_o(\overline{S}_E \vee \overline{b}_n\overline{b}_{n+1}\overline{s}) \vee \overline{E}_o\left[a_{n-1} \oplus b_{n-1} \vee \overline{S}_E(b_n \vee b_{n+1} \vee s)\right]\right\} \vee \\ &\quad (OLS)(\overline{S}_E\overline{b}_n \vee S_E\overline{b}_n\overline{b}_{n+1}\overline{s}) \end{aligned}$$

where *ORS*, *NXS*, and *OLS* are as defined in Section 3.2.3 in the main text. S_E is the sign of the result. Similarly,

$$l_{in} = \overline{S}_E l_0^0 b_n \vee \overline{b}_n$$

The equation for the q bit is

$$q = \overline{S}_E b_n \vee S_E\left[b_n(b_{n+1} \vee s)\right]$$

For the case of MLS, because $b_{n+1} = s = 0$, the equation reduces to

$$q = b_n$$

B.3 Round to $-\infty$

Because this case is symmetrical about the sign to the case of *RTPI*, all we need to do is to reverse the sign of S_E in the equations for g_{in} and l_{in}.

The equation for the q bit, for example, is

$$q = S_E b_n \vee \overline{S}_E\left[b_n(b_{n+1} \vee s)\right]$$