Rounding Algorithms for IEEE Multipliers

Mark R. Santoro **Gary Bewick** **Mark A. Horowitz**

Stanford University
Stanford, CA 94305

Abstract

Several technology independent rounding algorithms for multiplying normalized numbers are presented. The first is a simple rounding algorithm suitable for software simulation or moderate performance hardware multipliers. The next two algorithms are parallel addition schemes suitable for high performance VLSI multipliers. The latter algorithm eliminates the carry produced by the lower order bits from the critical path. Several methods for computing the sticky bit are are also presented. Included is a new fast and efficient technique for computing the sticky bit directly from the carry save form without undergoing the expense of a carry propagate addition.

1. Introduction

Many applications exist in which IEEE floating-point multiplication is not required. However, to be widely accepted, current and future floating-point coprocessors must adhere to IEEE standard 754 for binary floating-point arithmetic [4]. The standard can be implemented in software, hardware, or a combination of the two [2]. The performance of modern digital systems demands direct hardware implementations of floating-point multipliers. To match the performance of the hardware multipliers, the rounding modes must also be implemented in hardware.

Three algorithms will be presented for implementing round to nearest/up. It will then be shown how the round to nearest/up result can be adjusted to produce the correct IEEE rounded result. Finally, three methods will be presented for computing the sticky bit. All of the rounding algorithms and sticky methods presented are technology independent and can be used with several types of multiplier architectures.

2. Round to Nearest

The IEEE standard 754 default rounding mode is **round to nearest**. The standard states that "in this mode the representable value nearest to the infinitely precise result shall be delivered; if the two nearest representable values are equally near, the one with its least significant bit zero shall be delivered. " Round to nearest as defined in IEEE 754 is actually **round to nearest/even**. This means always round to nearest, and in the case of a tie round to even.

A conventional rounding system , **round to nearest/up**, adds 1/2 to the least significant bit (LSB) of the desired result and then truncates by removing the bits to the right of the LSB. Round to nearest/up produces exactly the same result as round to nearest/even in all cases except when a tie occurs. If the even result were the smaller value, round to nearest/up would incorrectly round up. Dealing with the tie case before rounding makes round to nearest/even more complex and slower than round to nearest/up. For this reason, the rounding algorithms developed in this paper will produce a round to nearest/up result. At the end of the paper the so-called "sticky" bit, which identifies the tie case, will be introduced. It will then be shown how the correct round to nearest/even result (IEEE round to nearest) can be obtained from the round to nearest/up result by simply forcing the LSB to a 0 in the case of a tie.

3. Simple Round to Nearest/up Algorithm

Most high performance VLSI multipliers use some sort of array or tree structure to sum the partial products in the mantissa portion of a floating-point multiply [7]. Figure 1 shows a flow diagram for the mantissa handling section of a floating-point multiply unit. This simple round to nearest/up scheme will be referred to as Algorithm 1.

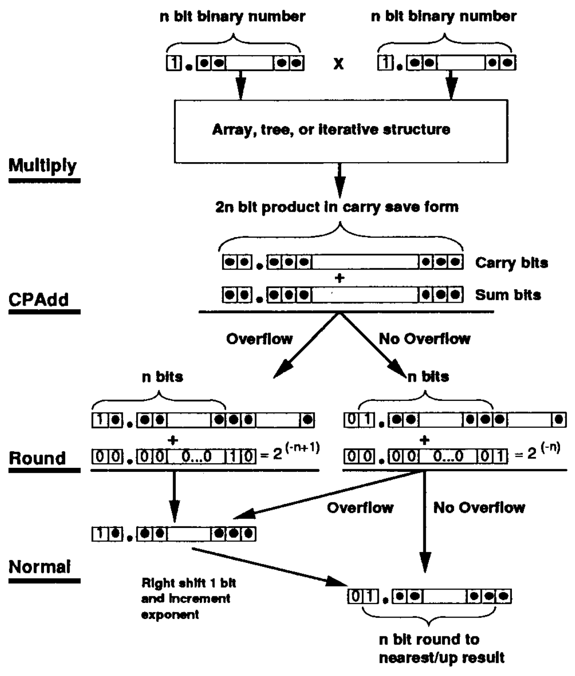

Figure 1. Algorithm 1 Data Flow

The top section (**Multiply**) accepts two normalized mantissas and uses some type of reduction structure which produces the product in carry save form (two 2n bit numbers). These two numbers are then added in the **CPAdd** section to produce a complete 2n bit product. There are two possible rounding operations which then occur, depending on the most significant bit (MSB) of this product. If the resulting product is in the range $2 \leq$ product < 4 (overflow), the constant $2^{(-n+1)}$ is added to the product and the result is truncated to n-2 bits to the right of the decimal point. A normalization shift (**Normal**) of 1 to the right is then necessary to restore the rounded product to the range $1 \leq$ rounded product < 2, with an appropriate adjustment of the exponent. If the original 2n bit product was in the range $1 \leq$ product < 2 (no overflow), then the constant $2^{(-n)}$ is added. In most cases this rounded product will be less than 2 and the rounding operation is finished. However, it is possible that the addition of $2^{(-n)}$ could cause the rounded product to be equal to 2, in which case a normalization shift of 1 and an exponent adjustment is necessary (as in the left branch).

The low order n-2 bits from the **CPAdd** section of Figure 1 are not used in any of the following steps. The only effect that these bits have on the final result is due to the carry they generate into the most significant n+2 bits. Thus, the carry propagate adder need never actually compute the sum of the least significant n-2 bits. The 2n bit carry propagate adder can be replaced by an n+2 bit carry propagate adder, with an input carry, and some auxiliary hardware which computes the carry from the least significant n-2 bits. The smaller adder is clearly an advantage where a hardware implementation is concerned.

Algorithm 1 requires two carry propagate additions in series. Algorithm 2 concentrates on computing these additions in parallel, which significantly increases performance. Finally, Algorithm 3 moves the carry from the least significant bits out of the critical path.

4. Parallel Addition Schemes

If an n+2 bit carry propagate adder is used in the **CPAdd** section of Figure 1, then the carry from the lower bits (Cin) will be added at the $2^{(-n)}$ bit position. Assuming that no overflow occurred, an additional $2^{(-n)}$ will be added to the result in the **Round** section. The $2^{(-n)}$ bit position will thus be called the round bit position or R bit. The 1 that always gets added to the R bit position for rounding will be identified as Rin. If no overflow occurs, adding Cin and Rin to the R bit position will produce the correct round to nearest/up result.

Now consider the overflow case. The MSB, known as the overflow bit (V), is a 1. By assuming that no overflow would occur, $2^{(-n)}$ was added for rounding. If an overflow did occur, then $2^{(-n+1)}$ should have been added for rounding. The difference of $2^{(-n)}$ must be added to correct the rounding. This can be done by defining a new bit that is added to the $2^{(-n)}$ bit position in the case of an overflow. This bit will be called the overflow rounding bit (Rv). The correct rounding can thus be obtained by simply adding the carry from the lower order bits (Cin), the rounding bit (Rin), and the overflow rounding bit (Rv), to the R bit position. These bits are shown in Figure 2.

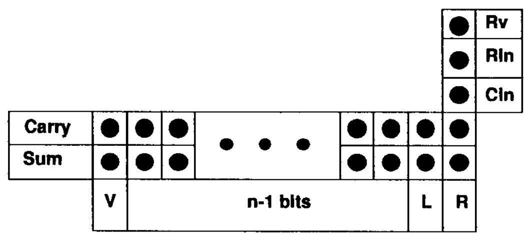

Figure 2. Bits to be Summed for Correct Round to Nearest/up

Fast and effective implementations for summing the bits in Figure 2 must overcome two problems. First, the value of Rv is not known until the sum of all of the other bits have been computed. Second, an adder with 5 input slots at the LSB is required.

The first problem can be overcome by computing two carry propagate additions in parallel. The first, assuming Rv=0, and the second, assuming Rv=1. When the overflow condition is known, the correct sum can then be selected using a multiplexor. These two additions are related, as the first is simply one larger than the second. This provides many possibilities for the designer. An efficient technique is to simply merge the two carry propagate adders into one. A conditional sum adder (**CSAdd**), or carry select adder as it is often known, computes two possible outputs [5]. The first assumes the input carry is a 0, and the second assumes the input carry is a 1. When the input carry is known the correct output is picked. This compound adder requires much less hardware than two separate adders since only the carry chain need be duplicated. In the more general sense a conditional sum type adder produces two results in the form A+B and A+B+1. A signal, not necessarily the carry, is then used to select the desired output.

Now for the second problem. Rcarry and Rsum use the carry and sum slots. Rv uses the input carry slot to the CSAdder. This leaves no empty slots for Rin and Cin to be added to the R bit position. Two algorithms will be proposed to fix this problem. Both involve adding Cin and Rin to the R bit position without propagating the carry before computing the carry propagate result.

The data flow of Algorithm 2A is shown in Figure 3. A row of half adders is used to partly sum the carry and sum bits.[1] This leaves a hole in the carry propagate increment adder at Rcarry. The Cin from the lower order bits can be placed into this hole. Rin must still be added to the R bit position. An additional row of half adders could be used as on Cin but there are more economical techniques. Array multipliers typically have empty slots. A 1 can often be injected into the array, or corresponding structure, in the appropriate place such that the effect is to add Rin to the R bit. An iterative multiplier could also have Rin injected into the accumulator. Once Rin and Cin have been added to the R bit position and the CSAdd has completed, the corrected result can be picked based upon the overflow bit from the A+B result. The V bit from the A+B result is used, because the overflow bit must be checked before Rv has been added in. The A+B+1 result has already added Rv to the sum, potentially corrupting the V bit. If the V bit from the A+B result is a 0, the A+B result is chosen. If the V bit is a 1 the A+B+1 result is picked. In this case, since an overflow has occurred, the result must be normalized and the exponent adjusted.

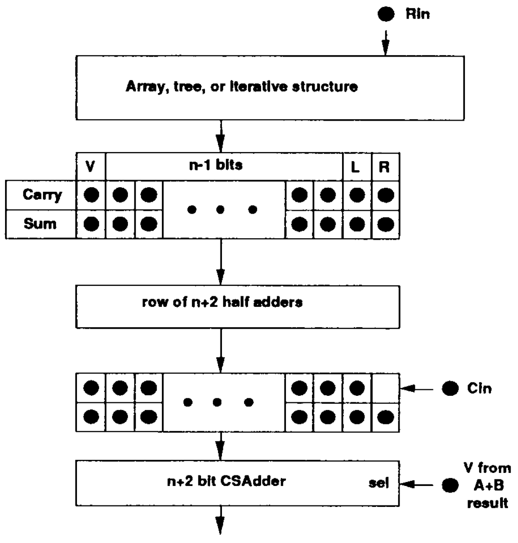

Figure 3. Algorithm 2A Data Flow

In some cases a slot may not exist, or it may be difficult to inject Rin into the multiplier array or accumulator. Figure 4 shows the data flow for Algorithm 2B. This algorithm is similar to Algorithm 2A except that Rin is not injected into the array. Instead, the two least significant half adders are replaced with carry save adders, providing two additional slots at the L and R bit positions.[2] Rin, which is always a 1, can be combined with Cin and placed into these empty slots. If Cin equals 0, then a 1 from Rin should be added to the R bit. If Cin equals 1, then 2 should be added to the R bit position; one from Rin and one from Cin. Adding a 2 to the R bit position is equivalent to adding a 1 to the R+1 (L) bit position. The output of the half adder/CSA row may then be fed to the CSAdder as in Algorithm 2A.

[1]Since developing algorithm 2A it has been learned that half adders have been used in some commercial parts including the Weitek WTL 1164 [9], and the Intel i860 (N10) [3].

[2]For simplicity, an entire row of CSA cells could be used, with the unused inputs set to 0.

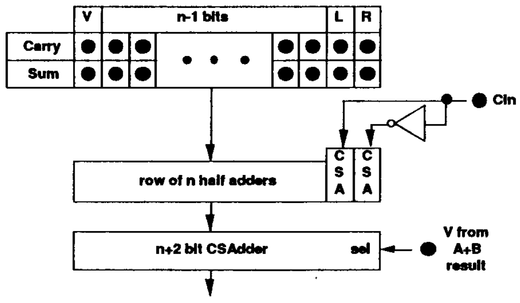

Figure 4. Algorithm 2B Data Flow

Σ3	Σ5	R to L Carry
1	1-3	{0,1}
2	2-4	{1,2}
3	3-5	{1,2}

Where:

$\Sigma 3$ = Rsum + Rcarry + Rin
$\Sigma 5$ = Rsum + Rcarry + Rin + Rv + Cin
R to L carry = Possible carry from R to L

Table 1. Carry Propagation From R to L

In the case of a conventional array, the carry from the lower order bits (Cin) may be determined soon after the carry save bits, so waiting for Cin before doing the half or full additions may not be a problem. Other multipliers may require additional time to determine the carry from the lower order bits. As an example, iterating multipliers may require one or more additional cycles to determine Cin [6]. The next section develops a rounding algorithm which eliminates Cin from the critical path.

5. Removing Cin From the Critical Path

Referring back to Figure 2, five bits must be added at the R bit position. They are Rv, Rin, Cin, Rcarry, and Rsum. Since Rin is always a 1 the sum of these five bits can range from 1 to 5. The resulting carry from the R bit to the L bit will be equal to 0, 1, or 2. Since the CSAdder can only propagate a carry of 0 or 1 in parallel the situation may appear hopeless, but this is not the case.

Knowing that Rin is always a 1 narrowed the range of possible sums from 0-5 to 1-5. It is possible to further narrow the range of possible sums, and thus narrow the range of possible carries, by summing some of the bits before the others are known. Rv is not known until the carry propagate addition is completed and the V bit is examined. It was also stated that the goal of this section was to start the carry propagate additions before Cin is known. Rcarry and Rsum are both known before Rv and Cin. In fact, they are known at the same time or before all of the other carry/save bits, and the carry propagate additions cannot be started until these bits are known. By looking at Rin, Rcarry, and Rsum, the possible sum, and possible resulting carries, of the 5 bits at the R bit position can be further narrowed as shown in Table 1. For example: if Rcarry = 1, and Rsum = 0 then, the sum of Rsum, Rcarry, Rin, Rv, and Cin must be in the range 2-4. The possible carry from the R to the L bit will then be in the set {1,2}.

From Table 1 it can be seen that summing Rin, Rsum, and Rcarry limits the possible carries into the L bit to one of two sets: {0,1} or {1,2}. Within each set the carry differs by exactly 1; therefore, the set of possible rounded results from the L bit up can differ by at most 1. This is important because the CSAdder computes results in the form A+B and A+B+1. In addition, since the R bit is not part of the final correctly rounded result it need not be included in the carry propagate addition. Knowing the correct carry set yields one of two possible cases:

Case 1: The carry set is {0,1}

In this case either a 0 or a 1 must be added to the L bit position. Since the CSAdder directly computes results in the form A+B and A+B+1 both possible correct results are computed. If the actual carry is a 0 the A+B result is selected, with the A+B+1 result selected if the carry is a 1.

Case 2: The carry set is {1,2}.

In this case either a 1 or a 2 must be added to the L bit position. Since a CSAdder cannot compute A+B+2, a row of half adders should be used providing a slot to add 1 to the L bit position. This leaves either a 0 or a 1 to be added to the L bit position. This is precisely case 1 and should be handled as such.

Referring to Table 1 the carry set {0,1} (Case 1) is chosen only if $\Sigma 3$ = 1. Since Rin is always 1, a logical OR on Rcarry and Rsum can be used to differentiate between Case 1 and Case 2. Placing the output of this OR gate into the empty slot created by a row of half adders will correctly add a 1 to the L bit for Case 2 and a 0 for Case 1 (see Figure 5).

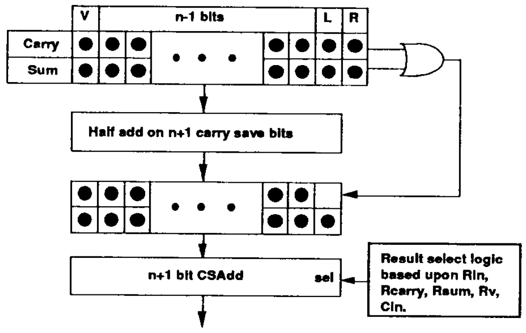
V
n-1 bits
L
R
Carry
Sum
Half add on n+1 carry save bits
n+1 bit CSAdd
sel
Result select logic based upon Rin, Rcarry, Rsum, Rv, Cin.

Figure 5. Algorithm 3 Data Flow

Once both possible results have been computed, the correct one must be picked. Cin should now be known; Rv, however, is not. The V bit from the A+B output is not automatically the correct value to use for Rv as it was in Algorithm 2. The reason is that the other bits may have already determined that the A+B+1 result should be chosen regardless of the value of the V bit. To determine the correct V bit to use for Rv, a preliminary A+B or A+B+1 value should be chosen based upon the $\Sigma 5$ column of Table 2, assuming that Rv=0. The V bit of the selected output will be the correct V bit to assign to Rv. The value of $\Sigma 5$ should then be recalculated using the actual Rv, and the correct A+B or A+B+1 result selected. The normalization and exponent adjustment is the same as in Algorithm 2.

$\Sigma 3$	$\Sigma 5$	R to L Carry	Output	
1	1	0	A+B	
1	2	1	A+B+1	Case 1
1	3	1	A+B+1	
2	2	1	A+B	
2,3	3	1	A+B	Case 2
2,3	4	2	A+B+1	
2,3	5	2	A+B+1	

where:

$\Sigma 3$ = Rsum + Rcarry + Rin
$\Sigma 5$ = Rsum + Rcarry + Rin + Rv + Cin
R to L carry = Possible carry from R to L
Output = The CSAdder output to be selected

Table 2. CSAdder Output Selection

6. IEEE Rounding Modes

6.1 Round Toward +∞, -∞, and Zero

In addition to round to nearest, the default rounding mode, IEEE standard 754 defines three other optional rounding modes. These "directed" rounding modes are round toward +∞, round toward -∞ and round toward zero. Once round to nearest has been implemented the other rounding modes are relatively simple. To begin with, consider round toward zero. This is simply a truncation. All of the previous algorithms will work except the Rin and Rv bits will now be 0.

Now lets look at round toward +∞. The standard states that "when rounding toward +∞ the result shall be the format's value (possible +∞) closest to and no less than the infinitely precise result." Basically what this says is that in the case of a positive result if all the bits to the right of the LSB of the desired result are 0 then the result is correct. If any of these bits are a 1, (i.e. R=1 or sticky=1) then a 1 should be added to the LSB of the result. If the result is negative it should be truncated. When rounding toward -∞ the exact opposite holds. The direct rounding algorithms can be summarized as follows:

```
Round Toward Zero
    Truncate

Round Toward +∞
    if sign = positive
        if any bits to the right of the result LSB = 1
            Add 1 to result
        else
            Truncate at LSB
    if sign = negative
        truncate at LSB

Round Toward -∞
    if sign = negative
        if any bits to the right of the result LSB = 1
            Add 1 to result
        else
            Truncate at LSB
    if sign = positive
        truncate at LSB
```

6.2 Obtaining The IEEE Round to Nearest Result

It was stated earlier that round to nearest/up produces exactly the same result as round to nearest/even except when a tie occurs. A tie can only occur when the result is exactly halfway between two numbers of representable precision. For example:

37.25XXX	Raw number to be rounded
+0.05	Add 0.05 to round to nearest/up
37.30	Sum
37.3	Truncated - Final rounded Result

If the X's are all zeros then 37.25000 is exactly half way between 37.2 and 37.3. In this case round to nearest/up produces a different result than round to nearest/even. There are only two cases to be considered. Either all of the X's are 0, or they are not. The bit which distinguishes between these cases is referred to as the sticky bit. This bit is a 0 if all of the X's are 0, and 1 if any of the X's are non-zero.

To produce the correct round to nearest/even result from the unrounded result, a 1 is potentially added to the round bit. The bit (E) to be added to the round bit (R) for correct IEEE round to nearest is based upon the L, R, and sticky (S) bits as shown in Table 3.

Before Rounding			Add to R		L After Rounding	
L	R	S	E	U	L_E	L_U
X	0	0	d	1	X	X
X	0	1	d	1	X	X
0	1	0	0	1	0	1
1	1	0	1	1	0	0
X	1	1	1	1	X	X

where:
E = Bit added for correct round to nearest/even.
U = Bit added for correct round to nearest/up.
L_E = The L bit after round to nearest/even.
L_U = The L bit after round to nearest/up.
d = Don't care. E can not effect L_E.

Table 3. Round to Nearest/even versus Round to Nearest/up

In contrast, round to nearest/up assumes that the bit to be added to the R bit for correct rounding (U) is always a 1. The only case where the round to nearest/up bit (U) will produce a different result from the round to nearest/even bit (E) is shown in row 3 of Table 3, where E=0, and U=1. In this case round to nearest/up changed the L bit from a 0 (L=0) to a 1 (L_E=1), while round to nearest/even left the L bit unchanged (L_U=0). The important thing to notice is that when round to nearest/up changed the L bit to a 1, the 1 was not propagated. As such, only the L bit was effected. This means that the correct round to nearest/even result can be obtained from the round to nearest/up result by restoring the L bit to a 0.

By assuming that the round bit will be a 1, the round to nearest/up algorithms have the advantage over the round to nearest/even methods in that the carry propagate addition can take place before the sticky bit has been computed. This means that the round to nearest/up result can be obtained using any of the methods presented in this paper. The correct IEEE round to nearest/even result can then be obtained by observing only the L, R, and sticky bits, and forcing the L bit to 0 if required. Care should be taken however, as operations such as right shifting in the event of an overflow, and adding Rv and Rin can change the position and/or value of the S, L, and R bits.

7. Computing the Sticky Bit

7.1 A Simple Method to Compute the Sticky Bit

The first method for determining the sticky bit is conceptually the simplest, as it stems from the very definition of the sticky bit. Recall that the sticky bit was defined to be equal to 0 if the value of all of the bits to the right of the round bit is 0. To determine the sticky bit, begin with a carry propagate addition on all of the bits. The sticky bit (S) will be the OR of all of the bits to the right of the R bit. This method is quite simple in concept, and is often used in practice. One drawback is that a full carry propagate addition, followed by a logical OR, must be done on all of the lower order carry save bits.

7.2 Computing Sticky From the Input Operands

The sticky bit may also be computed directly from the inputs to be multiplied, bypassing the multiply array completely. The number of trailing zeros in the binary number X•Y is exactly equal to the number of trailing zeros in X plus the number of trailing zeros in Y.[3] The trailing zeros in X and Y can be counted and summed while the multiply is taking place. If the sum is greater than the sum of bits to the right of the round bit, then the sticky bit is a 0. The advantage of using this method is that the sticky bit can be computed in parallel with the actual multiplication, removing the sticky bit from the critical path.

[3]The number of trailing zeros in the product is exactly equal to the sum of the trailing zeros in the operands for any representation in which the base is prime. This is true because prime numbers cannot be factored. Non-prime bases can be factored; therefore, the number of trailing zeros in the product can be greater than the sum of the trailing zeros in the operands. As an example, in base 10 if the least significant non-zero bits in the operands were 2 and 5 respectively an additional zero would be created, and the number of trailing zeros in the product is larger than the sum of the trailing zeros in the operands.

7.3 Computing Sticky From the Carry Save Bits

The third method depends upon all of the partial products being positive (i.e. no Booth encoding has been used) [1]. Given this assumption, **a simple logical OR on the carry-save form of the bits to the right of the round bit will yield the correct sticky bit.**

This simple ORing of the carry save bits works for the following reason. If the 2n bit carry save result is scanned from right to left, the first non-zero carry/sum pair will contain a single 1. That is, either the carry or the sum will be a 1 but not both. This single one could not generate a carry during a carry propagate addition, and since all of the bits to its right are zero, there is no carry to propagate. This will cause the single 1 to remain in its current position. If this position is to the right of the R bit, the sticky bit will be a 1.

To see why this is true, refer to Figure 6. This figure shows a section of the partial products for the multiplication of A • B. Each row represents a single partial product which will be generated and later summed to form the carry save form of the final product. A0B0 represents the partial product represented by the logical AND of bit A0 with bit B0, and so on.

	5	4	3	2	1	0
0	A5B0	A4B0	A3B0	A2B0	A1B0	A0B0
1	A4B1	A3B1	A2B1	A1B1	A0B1	
2	A3B2	A2B2	A1B2	A0B2		
3	A2B3	A1B3	A0B3			
4	A1B4	A0B4				
5	A0B5					

Figure 6. Summation of Partial Products

Assume A2B2 in column 4 is a 1, and column 4 is the first column in which a 1 appears. Since A2 is a 1, B1 and B0 must both be 0, or there would be a 1 in an earlier column formed by A2B1 in column 3 row 1, or A2B0 in column 2 row 0. All products above A2B2 in column 4 contain either a B1 or a B0, and therefore must be 0. Looking across row 2, B2 is a 1. This means A1 and A0 must both be zero or a 1 would exist in columns 3 or 2. All products below A2B2 in column 4 contain either A1 or A0 and thus also must be 0. Therefore A2B2 is the only non-zero partial product in this column. This can easily be generalized to any element in any column, proving that the first column in which a 1 exist will contain a single 1.

8. Conclusions

Several technology independent rounding algorithms suitable for hardware implementations have been presented. Algorithm 1, shown in Figure 1, is a straightforward round to nearest/up algorithm. It demonstrates the basic principles of rounding and is suitable for software simulation, or moderate performance hardware implementation. Algorithms 2 and 3 are better suited for high performance VLSI multipliers. While Algorithm 1 requires two series carry propagate additions Algorithms 2 and 3 use a parallel carry propagate addition scheme. Algorithm 2A (Figure 3) would be a likely choice for most conventional array or full tree multipliers. Algorithm 2B (Figure 4) would be preferable if a blank slot does not exist in the array for summing in the rounding bit (Rin). Though slightly more complex than the other methods Algorithm 3 (Figure 5) is best suited for iterative multipliers, or any multiplier where the carry from the lower order bits is in the critical path. By using the sticky bit, any of the round to nearest/up results can be corrected to comply with IEEE standard 754 rounding. Finally, three methods for determining the sticky bit were presented. The first method originates directly from the definition of the sticky bit. The second method allows the sticky bit to be determined from the input operands in parallel with the actual multiplication. The third method represents a new fast and efficient technique for determining the sticky bit from the carry save bits.

Acknowledgements

The research for this paper was partially supported by the Defense Advanced Project Research Agency (DARPA) under contract N00014-87-K-0828, and by the Center for Integrated Systems (CIS) at Stanford, using facilities provided under NASA contract NAGW 419.

References

[1] A. D. Booth, "A Signed Binary Multiplication Technique", Qt. J. Mech. Appl. Math., Vol. 4, Part 2, 1951.

[2] J. J. Coonen, "An Implementation Guide to a Proposed Standard for Floating-Point Arithmetic", Computer Magazine vol 13, no 1, January 1980.

[3] L. Kohn, S. Fu, "A 1,000,000 Transistor Microprocessor", IEEE Int. Solid-State Circuits conf., February 1989.

[4] "IEEE Standard for Binary Floating-Point Arithmetic", ANSI/IEEE Std 754-1985, New York, The Institute of Electrical and Electronics Engineers, Inc., August 12, 1985.

[5] J. Sklansky, "Conditional Sum Addition Logic", Trans. IRE, Vol. EC-9, No. 2, June 1960, pp. 226-230.

[6] M. Santoro, and M. Horowitz, "A Pipelined 64X64b Iterative Array Multiplier", IEEE Int. Solid-State Circuits conf., pp. 35-36, February 1988.

[7] C. S. Wallace, "A Suggestion for Fast Multipliers", IEEE Transactions on Electronic Computers, Vol. EC-13, pp. 14-17, February 1964.

[8] S. Waser, and M. J. Flynn, "Introduction to Arithmetic for Digital Systems Designers", New York, CBS Publishing, 1982.

[9] Weitek, private correspondence with engineers at Weitek.

[10] J. M. Yohe, "Roundings in Floating-Point Arithmetic", IEEE Transactions on Computers, Vol. C-22 no. 6, pp. 577-586, June 1973.