A comparison of three rounding algorithms for IEEE floating-point multiplication

Guy Even
Tel-Aviv University
Dept. of Electrical Engineering Systems
Tel-Aviv 69978, Israel
guy@eng.tau.ac.il [*]

Peter-M. Seidel
University of Saarland
Computer Science Department
D-66041 Saarbruecken, Germany
pmseidel@cs.uni-sb.de [†]

Abstract

A new IEEE compliant floating-point rounding algorithm for computing the rounded product from a carry-save representation of the product is presented. The new rounding algorithm is compared with the rounding algorithms of Yu and Zyner [23] and of Quach et al. *[18]. For each rounding algorithm, a logical description and a block diagram is given and the latency is analyzed.*

We conclude that the new rounding algorithm is the fastest rounding algorithm, provided that an injection (which depends only on the rounding mode and the sign) can be added in during the reduction of the partial products into a carry-save encoded digit string. In double precision the latency of the new rounding algorithm is 12 *logic levels compared to* 14 *logic levels in the algorithm of Quach* et al.*, and* 16 *logic levels in the algorithm of Yu and Zyner.*

1. Introduction

Every modern microprocessor includes a floating-point (FP) multiplier that complies with the IEEE 754 Standard [9]. The latency of the FP multiplier is critical to the floating-point performance since a large portion of the FP instructions consists of FP multiplications (37% percent in benchmark applications [13]). A lot of research has been devoted to optimizing the latency of adding the partial products to produce the product, e.g. [1, 5, 11, 12, 14, 15, 16, 22]. More recently, work on rounding the product according to the IEEE 754 Standard has been published [3, 4, 6, 17, 18, 19, 20, 23, 25]. Assuming that the partial product reduction of the multiplier outputs a carry-save encoded representation the exact product, the following natural question arises: What is the fastest method to compute the rounded product given the carry-save encoded representation of the exact product ?

We consider and compare three rounding algorithms: (a) the algorithm of Quach *et al.* [18], which we denote by the QTF algorithm; (b) the algorithm of Yu and Zyner [23], which we denote by the YZ algorithm; and (c) a new algorithm that is based on injection based rounding [6], which we denote by the ES algorithm. We provide block diagrams of these rounding algorithms, optimized for speed. We measure the latency of the algorithms in logic levels to enable technology independent comparisons. The main building blocks of these algorithms are similar, and consist of a compound adder and the computation of a sticky and carry bits. Thus, the costs of the three algorithms are similar, and the paper deals with finding the fastest algorithm.

We focus on double precision multiplication in which each significand is represented by 53 bits. The algorithms assume normalized significands in the range $[1, 2)$, and therefore, their product is in the range $[1, 4)$. We do not consider the cases that deal with denormal or special values since supporting denormal values can be obtained by using an extended exponent range [10, 20, 24] and the computation on special values can be done in parallel [8].

The paper is organized as follows. Section 2 contains some preliminary issues. In Sections 3-5, the three different rounding algorithms are described, and analyzed. In Section 6, a summary and conclusion is given. Due to space limitations some sections and the correctness proofs are omitted and can be found in the full version [7].

2. Preliminaries

Notation Let $x[z_1 : z_2] = x_{z_1} x_{z_1+1} \cdots x_{z_2} \in \{0, 1\}^*$ denote a binary string. Since we deal with fractions, we index binary encoded bit strings by $x_0.x_1x_2\ldots$, so that x_i is associated with the weight 2^{-i}. The value encoded by $x[z_1 : z_2]$ is denoted by $|x[z_1 : z_2]| = \sum_{i=z_1}^{z_2} x_i \cdot 2^{-i}$.

[*] Supported in part by Intel Israel LTD. under grants awarded in 97/98.

[†] Supported by the DFG Graduiertenkolleg “Effizienz und Komplexitaet von Algorithmen und Rechenanlagen”, Universitaet des Saarlandes.

IEEE rounding The IEEE-754-1985 Standard [9] defines four rounding modes: round toward 0, round toward $+\infty$, round toward $\Leftrightarrow\infty$, and round to nearest (even). Based on the sign of the number these rounding modes can be reduced to the three rounding modes: RZ (round to zero), RNE (round to nearest-even), and RI (round to infinity) [18]. Furthermore, Quach et al. [18] suggested to implement RNE by RNU (round to nearest-up). The reason that RNE can be implemented by RNU is that $r_{RNU}(x) \neq r_{RNE}(x)$ iff $x = (y_1 + y_2)/2$ and the least significant bit (LSB) of the binary encoding of y_2 is 1. Therefore, obtaining $r_{RNE}(x)$ from $r_{RNU}(x)$ can be accomplished by "pulling down" the LSB, if $x = (y_1 + y_2)/2$.

3. The ES rounding algorithm

In this section we review injection based rounding [6], and we present and analyze the new ES rounding algorithm.

3.1. Injection based rounding

Rounding by injection reduces the rounding modes RI and RNU to RZ [6]. The reduction is based on adding an injection that depends only on the rounding mode:

$$\text{INJ} = \begin{cases} 0 & \text{if RZ} \\ 2^{-53} & \text{if RNU} \\ 2^{-52} \Leftrightarrow 2^{-104} & \text{if RI.} \end{cases}$$

For $mode \in \{RZ, RNU, RI\}$, the effect of adding INJ is summarized in the following equation:

$$x \in [1,2) \quad \Rightarrow \quad r_{mode}(x) = r_{RZ}(x + \text{INJ}). \qquad (1)$$

If the exact product, denoted by EXACT, is in the range $[2,4)$, then the injection must be fixed in order to make the reduction to RZ correct. The correction amount, denoted by INJ_COR, is defined by

$$\text{INJ\_COR} = \begin{cases} 0 = 0 \Leftrightarrow 0 & \text{if RZ} \\ 2^{-53} = 2^{-52} \Leftrightarrow 2^{-53} & \text{if RNU} \\ 2^{-52} = 2^{-51} \Leftrightarrow 2^{-104} \Leftrightarrow (2^{-52} \Leftrightarrow 2^{-104}) & \text{if RI} \end{cases}$$

Therefore, if x is in the range $[2,4)$ the effect of adding the injection and the correction amount is summarized in the following equation for $mode \in \{RZ, RNU, RI\}$:

$$x \in [2,4) \Longrightarrow r_{mode}\left(\frac{x}{2}\right) = r_{RZ}\left(\frac{x + \text{INJ} + \text{INJ\_COR}}{2}\right). \qquad (2)$$

Our assumption is that INJ is added in the multiplier adder array, and therefore $|\text{SUM}| + |\text{CARRY}| = \text{EXACT} + \text{INJ}$.

3.2. Description

The following description of the ES rounding algorithm works under the assumption that the SUM and CARRY-strings already include the injection INJ (but not INJ_COR). The corresponding block diagram is depicted in Fig. 1.

1. The SUM and CARRY-strings are divided into a high part (positions $[\Leftrightarrow 1:52]$) and a low part (positions $[53:104]$).

2. From the low part, we compute the carry-bit $C[52] = \sigma_{52}$, the round-bit $R = \sigma_{53}$ and the sticky-bit $S = \text{OR}(\sigma_{54}, \sigma_{55}, \ldots, \sigma_{104})$, where $\sigma_{52} \cdots \sigma_{104}$ is the binary string that satisfies:
$|\sigma_{52} \cdots \sigma_{104}| = |\text{SUM}[53:104]| + |\text{CARRY}[53:104]|$.

3. The high part is input to a line of Half Adders and produces the output $(X_{sum}[\Leftrightarrow 1:51], L_X)$ and $X_{carry}[\Leftrightarrow 1:51]$. Note that the bit L_X is in position 52, and that no carry is generated to position $\Leftrightarrow 2$, because the exact product is less than 4 (even after adding the injection).

4. A Compound adder computes from $X_{sum}[\Leftrightarrow 1:51]$ and $X_{carry}[\Leftrightarrow 1:51]$ the sum $|Y0[\Leftrightarrow 1:51]|$ and the incremented sum $|Y1[\Leftrightarrow 1:51]| = |Y0[\Leftrightarrow 1:51]| + 2^{-51}$.

5. The Increment Decision box receives the round-bit (R), the carry-bit ($C[52]$), the LSB (L_X), the MSB ($Y0[\Leftrightarrow 1]$) and the rounding modes (RN, RI). The output signal is the increment decision INC.

6. The most significant bits $Y0[\Leftrightarrow 1]$ and $Y1[\Leftrightarrow 1]$ indicate whether $Y0$ and $Y1$ are in the range $[2,4)$. Depending on these bits, $Y0$ and $Y1$ are normalized:
$$Z0[0:51] = \begin{cases} Y0[0:51] & \text{if Y0[-1]=0} \\ sft\_right(Y0[\Leftrightarrow 1:50]) & \text{if Y0[-1]=1} \end{cases}$$
$$Z1[0:51] = \begin{cases} Y1[0:51] & \text{if Y1[-1]=0} \\ sft\_right(Y1[\Leftrightarrow 1:50]) & \text{if Y1[-1]=1} \end{cases}$$

7. The rounded result (except for the least significant bit) is selected between $Z0$ and $Z1$ according to the increment decision INC, as follows:
$$\text{RESULT}[0:51] = \begin{cases} Z0[0:51] & \text{if INC=0} \\ Z1[0:51] & \text{if INC=1} \end{cases}$$

8. In the case of RNE, the least significant bit needs to be corrected since RNE and RNU do not always result with the same least significant bit. The correction of the least significant bit is computed by two parallel paths; one path ("overflow" path) working under the assumption that the rounded result overflows (i.e. greater than or equal to 2), and the other path ("no-overflow" path) working under the assumption that the rounded result does not overflow.

   The "no-overflow" path is implemented by the box called "fix L (novf)", that gets as input the round bit R, the sticky bit S, and a signal RNE indicating whether the rounding mode is round to nearest even. If the output *not(pd)* equals zero, then the LSB should be pulled down.

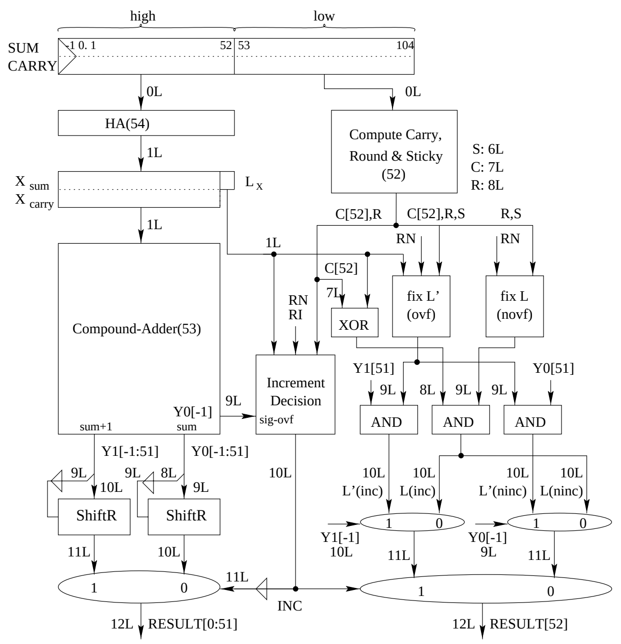

Figure 1. Block diagram of the ES rounding algorithm annotated with timing estimates

The "overflow" path is implemented by the box called "fix L (ovf)", that gets as input the L_X bit, the carry-bit $C[52]$, the round bit R, the sticky bit S, and the signal RNE. If the output *not(pd')* equals zero, then the LSB should be pulled down.

Note, that the pull down signals are inactive if the rounding mode is not RNE.

9. The least significant bit of the rounded result before fixing the LSB (in case of a discrepancy between RNE and RNU) equals one of three values:

   (a) if the rounded result does not overflow, then the LSB equals $L_X \oplus C[52]$;

   (b) if the rounded result overflows and the increment decision is not to increment, then the LSB equals $Y0[51]$; and

   (c) if the rounded result overflows and the increment decision is to increment, then the LSB equals $Y1[51]$.

   The fixing of the LSB is implemented by combining (using AND-gates) the pull-down signals with the corresponding candidates for the LSB signals.

   The outputs of the 3 AND-gates are denoted by: $L'(inc)$, $L'(ninc)$, and $L(inc)$. For the sake of clarity, we introduce $L(ninc)$, which equals $L(inc)$.

10. The LSB of the rounded result equals $L(ninc)$ if no overflow occurred and no increment took place. The LSB of the rounded result equals $L(inc)$ if no overflow occurred and an increment took place. The LSB of the rounded result equals $L'(ninc)$ if an overflow occurred and no increment took place. The LSB of the rounded result equals $L'(inc)$ if an overflow occurred and an increment took place.

    According to the 4 cases, the LSB of the rounded result is selected depending on the overflow signals and the increment decision

3.3. Details

In this section we describe the functionality of three boxes in figure 1, that have not been fully described yet.

Fix L (novf). This box belongs to the path that assumes that the product is in the range $[1, 2)$. Recall that there might be a discrepancy between RNE and RNU when a tie occurs, namely, when the exact product equals the midpoint between two successive representable numbers. Let EXACT denote the value of the exact product, the "Fix L (novf)" generates a signal *not(pd)* that satisfies:

$$\text{EXACT} \in [1, 2) \Rightarrow (\textit{not(pd)} = 0 \Leftrightarrow \text{ a tie occurs in } RNE)$$

When a tie occurs there are two possibilities: (a) If RNU and RNE agree, then both yield a rounded result with a LSB equal to zero. Pulling down the LSB in this case is not required, but causes no damage. (b) If RNU and RNE disagree, then the LSB of the RNU result must be pulled down.

Without the addition of the injection, a tie occurs when $R = 1$ and $S = 0$. Since an injection of 2^{-53} is already included, a tie occurs when $R = 0$ and $S = 0$. Therefore the *not(pd)* signal is defined by:

$$\textit{not(pd)} = \text{OR}(R, S, not(RNE))$$

Fix L' (ovf). This box belongs to the path that assumes that the product is in the range $[2, 4)$. The "Fix L' (ovf)" generates a signal *not(pd')* that satisfies:

$$\text{EXACT} \in [2, 4) \Rightarrow (\textit{not(pd')} = 0 \Leftrightarrow \text{ a tie occurs in } RNE)$$

The difference between *not(pd')* and *not(pd)* is that *not(pd')* is used under the assumption that the product is greater than or equal to 2. Without the addition of the injection, a tie occurs (in case of overflow) when $L_X \oplus C[52] = 1$, $R = 0$ and $S = 0$. Since an injection of 2^{-53} is already included, a tie occurs when $L_X \oplus C[52] = 1$, $R = 1$ and $S = 0$. Therefore the *not(pd')* signal is defined by:

$$\textit{not(pd')} = \text{OR}(not(L_X \oplus C[52]), not(R), S, not(RNE))$$

Increment Decision. The increment decision box has two paths, depending on whether an overflow occurs. The "no-overflow"-path (i.e. $Y0[\Leftrightarrow 1] = 0$) produces an increment decision, if $(L_X + C[52] = 2)$. The "overflow"-path working under the assumption that an overflow occurs (i.e. $Y0[\Leftrightarrow 1] = 1$) needs to take into account the correction of the injection, denoted by INJ_COR. It produces an increment decision, if $L_X + C[52] + \text{INJ\_COR} \cdot 2^{52} + R/2 \geq 2$. Therefore the INC signal is defined by:

$$\text{INC} = \begin{cases} L_X \wedge C[52] & \text{if } Y0[\Leftrightarrow 1] = 0 \text{ or } RZ \\ L_X \vee C[52] & \text{if } Y0[\Leftrightarrow 1] = 1 \text{ and } RI \\ R + L_X + C[52] \geq 2 & \text{if } Y0[\Leftrightarrow 1] = 1 \text{ and } RNE \end{cases}$$

3.4. Delay analysis

In the section we present a delay analysis of the rounding algorithm depicted in Fig.1. Our analysis is based on the following assumptions:

1. Consider a carry look-ahead adder, and let d_{CLA} denote the delay of the 53-bit adder measured in logic levels. We assume, that the MSB of the sum has a delay of at most $d_{CLA} \Leftrightarrow 1$ logic levels. This assumption is easy to satisfy for the carry look-ahead adder of Brent and Kung [2]. Otherwise, satisfying this assumption may require arranging the parallel-prefix network, so that the MSB is ready one logic level earlier.

2. The compound adder is implemented so that the delay of the sum is d_{CLA} and the delay of the incremented sum is $d_{CLA}+1$. This can be obtained by ORing the carry-generate and carry-propagate signals [21].

3. Consider the box in which the carry, round and sticky bits are computed. According to the first assumption, since the widths of this box and the compound adder are similar, the delay of the carry bit is $d_{CLA} \Leftrightarrow 1$ logic levels and the delay of the round bit is d_{CLA} logic levels. The delay of the sticky bit is estimated to be $d_{CLA} \Leftrightarrow 2$ logic levels, based on the fast sticky bit computation presented in [23].

4. We assume that the delay associated with buffering a fan-out of 53 is one logic level.

Figure 1 depicts the block diagram of the injection based ES rounding algorithm annotated with timing estimates. We assigned d_{CLA} the value of 8 logic levels. This implies that the sticky bit is valid after 6 logic levels, the carry-bit $C[52]$ is valid after 7 logic levels, and the round-bit is valid after 8 logic levels. Similarly, the sum $Y0$ is valid after 9 logic levels, the MSB $Y0[\Leftrightarrow 1]$ is valid after 8 logic levels, the incremented sum $Y1$ is valid after 10 logic levels, and the MSB $Y1[\Leftrightarrow 1]$ is valid after 9 logic levels. In this way we obtain the estimated delay of 12 logic levels for the rounded product.

4. The YZ rounding algorithm

In this section we review and analyze the rounding algorithm of Yu and Zyner, which was reported to have been implemented in the ULTRASparc RISC microprocessor [23]. We refer to this algorithm as the YZ rounding algorithm.

4.1. Description

Figure 2 depicts a block diagram of the YZ rounding algorithm. This description differs from the description in [23] in two ways:

1. In [23] the sum output by the 3-bit adder has only three bits. We believe that this is a mistake, and that the sum should have four bits (denoted by $Z[50:53]$).

2. The sum and the incremented sum in [23] is fed to a $4:1$-mux, which selects one of them either shifted to the right or not. We propose to normalize the sum and the incremented sum before the selection takes place. This early normalization helps reduce the delay of the rounding circuit at the cost of one additional shifter.

The algorithm is described below:

1. The SUM and CARRY-strings are divided into a high part (positions $[\Leftrightarrow 1:53]$) and a low part (positions $[54:104]$).

2. From the low part the carry bit $C[53] = \sigma_{53}$ and the sticky bit $S = \text{OR}(\sigma_{54}, \sigma_{55}, \ldots, \sigma_{104})$ are computed, where $\sigma_{53} \cdots \sigma_{104}$ is the binary string that satisfies: $|\sigma_{53} \cdots \sigma_{104}| = |\text{SUM}[54:104]| + |\text{CARRY}[54:104]|$.

3. The higher part is input to a line of Half Adders, that outputs $X_{sum}[\Leftrightarrow 1:53]$ and $X_{carry}[\Leftrightarrow 1:52]$. Note that no carry is generated to position $[\Leftrightarrow 2]$, because the exact product is less than 4.

4. The high part $X_{sum}[\Leftrightarrow 1:53]$ and $X_{carry}[\Leftrightarrow 1:52]$ is divided into two parts. Positions $[\Leftrightarrow 1:50]$ are fed into the Compound Adder, that outputs the sum $Y0[\Leftrightarrow 1:50]$ and the incremented sum $|Y1[\Leftrightarrow 1:50]| = |Y0[\Leftrightarrow 1:50]| + 2^{-50}$. Positions $[51:53]$ are added with the carry bit $C[53]$ to produce the sum $Z[50:53]$.

5. The processing of $Z[50:53]$ is split into two paths; one working under the assumption that the rounded product will not overflow (i.e. less than 2), and the other path working under the assumption that the rounded product will overflow.

   The "no-overflow" path computes a rounding decision, $rd[52]$, in the *round dec.(novf)* box. The rounding decision $rd[52]$ is added with $Z[50:52]$ in the

Σ-*novf* box to produce the sum $Z_{novf}[50:52]$. In [7] we prove that this 3 bit addition does not produce a carry bit in position 49. The sum $Z_{novf}[50:52]$ has two roles: positions $[51:52]$ are the result bits in positions $[51:52]$ if no overflow occurs, and position $[50]$ is used to detect if a carry is generated in position $[50]$ if no overflow occurs. The bit $Z_{novf}[50]$ decides whether the upper sum $Y0[0:50]$ or the incremented sum $Y1[0:50]$ should be selected in the no-overflow case.

The "overflow" path computes a rounding decision, $rd'[51]$, in the *round dec.(ovf)* box. The rounding decision $rd'[51]$ is added with $Z[50:51]$ in the Σ-*ovf* box to produce the sum $Z_{ovf}[50:51]$. In [7] we prove that this 2 bit addition does not produce a carry bit in position 49. The sum $Z_{ovf}[50:51]$ has two roles: position $[51]$ serves as the result bit in position $[52]$ if overflow occurs, and position $[50]$ is used to decide whether an increment should take place in the upper part.

6. The decision which path should be chosen is made by the *select decision* box. First, an overflow signal ovf is computed as follows:

$$ovf = Y0[\Leftrightarrow 1] \vee (Z_{novf}[50] \wedge Y1[\Leftrightarrow 1]). \qquad (3)$$

This overflow signal determines whether $Z_{ovf}[50]$ or $Z_{novf}[50]$ is chosen as the carry-bit that effects position $[50]$. Thus, the increment decision inc is computed by the selection:

$$inc = \begin{cases} Z_{ovf}[50] & \text{if } ovf = 1 \\ Z_{novf}[50] & \text{if } ovf = 0 \end{cases} \qquad (4)$$

7. The two least significand bits of the rounded product are computed as follows: If no overflow occurs ($ovf = 0$), then

$$result[51:52] = Z_{novf}[51:52].$$

If an overflow occurs ($ovf = 1$), then $result[52] = Z_{ovf}[51]$. The bit $result[51]$ depends on whether an increment takes place or not:

$$result[51] = \begin{cases} Y1[50] & \text{if } inc = 1 \text{ and } ovf \\ Y0[50] & \text{if } inc = 0 \text{ and } ovf. \end{cases}$$

Note that $inc = Z_{ovf}[50]$ if $ovf = 1$. Since the signal $Z_{ovf}[50]$ is ready earlier than inc, we use $Z_{ovf}[50]$ to control the selection:

$$result[51] = \begin{cases} Y1[50] & \text{if } Z_{ovf}[50] = 1 \text{ and } ovf \\ Y0[50] & \text{if } Z_{ovf}[50] = 0 \text{ and } ovf. \end{cases}$$

The selection between $Y1[50]$ and $Y0[50]$ is done by the sel multiplexer in Fig.2. The *lower mux* in Fig.2 implements the selection of $result[51:52]$.

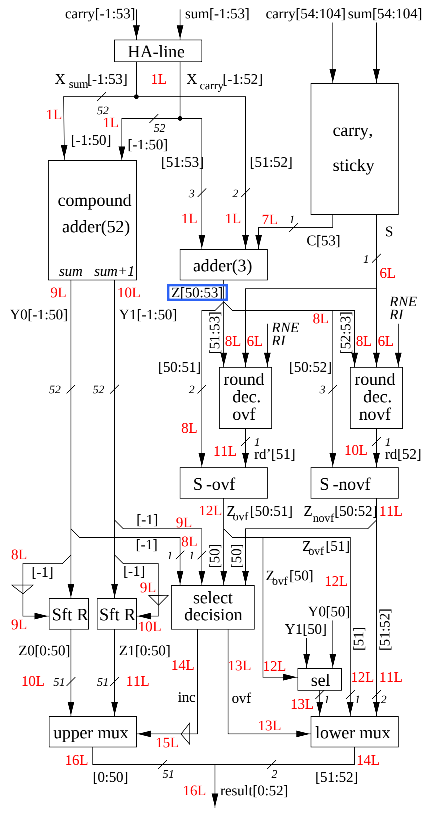

Figure 2. Block diagram of the YZ rounding algorithm annotated with timing estimates

8. The most significant bits $Y0[\Leftrightarrow 1]$ and $Y0[\Leftrightarrow 1]$ indicate whether $Y0$ and $Y1$ are in the range $[2,4)$. Depending on these bits, $Y0$ and $Y1$ are normalized:

$$Z0[0\!:\!50] = \begin{cases} Y0[0:50] & \text{if Y0[-1]=0} \\ sft\_right(Y0[\Leftarrow 1\!:\!49]) & \text{if Y0[-1]=1} \end{cases}$$

$$Z1[0\!:\!50] = \begin{cases} Y1[0:50] & \text{if Y1[-1]=0} \\ sft\_right(Y1[\Leftarrow 1\!:\!49]) & \text{if Y1[-1]=1} \end{cases}$$

9. The upper part of the rounded result is selected between $Z0$ and $Z1$ according to the increment decision:

$$\text{RESULT}[0:50] = \begin{cases} Z0[0:50] & \text{if } inc = 0 \\ Z1[0:50] & \text{if } inc = 1. \end{cases}$$

4.2. Delay analysis

Figure 2 depicts the YZ rounding algorithm annotated with timing estimates. We use the same assumptions on signal delays that are used in Sec.3.4. We argue that at least 16 logic levels are required. The path in which the sum and incremented sum are computed does not lie on the critical path. The critical path consists of the carry-bit computation, the 3-bit adder, the *round dec. (novf)* box, the Σ-*novf* box, the *select decision* box, a driver, and the *upper mux*.

We considered the following optimizations to minimize delay for a lower bound on the required number of logic levels:

1. The 3-bit adder is implemented by conditional sum adder; the late carry-in bit $C[53]$ selects between the sum and the incremented sum. This is a fast implementation because the bits of X_{carry} and X_{sum} are valid after one logic level and the carry-bit $C[53]$ is valid after 7 logic levels.

2. The rounding decision boxes are implemented by cascading two levels of multiplexers that are controlled by $Z[52:53]$ in the no-overflow path and by $Z[51:52]$ in the overflow path. In the overflow path, $Z[53]$ is combined with sticky-bit, and hence the rounding decision requires 3 logic levels. In the no-overflow path, only 2 logic levels are required.

3. The addition of the rounding decision bit required only one logic level using a conditional sum adder.

4. The inc signal is valid after 3 more logic levels, due to the need to compute the signal ovf in two logic levels (see Eq. 3), and one selection according to Eq. 4.

5. The inc signal passes through a driver due to the large fanout. This driver incurs a delay of one logic level, and controls the *upper mux* to output the result after 16 logic levels.

5. The QTF rounding algorithm

Quach *et al.* [18] presented methods for IEEE compliant rounding. Their technique is a generalization of the rounding algorithm of Santoro *et al.* [19]. In this section we present a rounding algorithm that is based on the method of Quach *et al.* while aiming for minimum delay.

Apart from reducing the rounding modes to RZ, RNU and RI, the key idea used in the methods of Quach *et al.* and Santoro *et al.* is to inject a prediction bit that is based on the rounding mode and the values of SUM[53] and CARRY[53]. The injection of the prediction bit reduces the number of possibilities of the rounded result.

In this section we deviate from Quach *et al.* [18] in the following points:

1. The presentation in the paper of Quach *et al.* is separated according to the rounding mode. Since we are investigating rounding algorithms that support all the rounding modes, we integrated the rounding modes into one algorithm.

2. Quach *et al.* suggest several options for the choice of the prediction logic in RNU. Only one possibility was suggested in modes RZ and RI. Since the prediction logic lies on the critical path, we chose to simplify the prediction logic as much as possible by defining:

$$pred = \begin{cases} 1 & \text{if } RI \wedge (\text{SUM}[53] \vee \text{CARRY}[53]) \\ 0 & \text{otherwise} \end{cases}$$

3. Quach *et al.* separate the rounding decision and the compound adder. They use a 3-way compound adder that computes $sum, sum+1$ and $sum+2$. The correct sum is selected by the control logic. We are interested in a faster design, and therefore, we break the 3-way adder into a Half-Adder line, a 2-way compound adder, and a mux. The control logic uses an output of the 2-way compound adder, and the LSB (in case of no overflow) is generated by the control logic as well as the increment decision.

5.1. Description

Figure 3 depicts a block diagram of a rounding algorithm that we suggest based on Quach *et al.* [18]. There are many similarities between the rounding algorithm based on injection rounding and the rounding algorithm based on Quach *et al.*, so we point out the differences and the new notations.

Before being input to the compound adder, the high part of the SUM and CARRY pass through two lines of Half-Adders. The first line makes room for the prediction bit. The second pass enables separating the bit $L_{X'}$ in position $[52]$ (this is, in fact, part of a 3-way compound adder). The increment decision has two paths: one for overflow and the other for no-overflow. The MSB $Y0[\Leftrightarrow 1]$ selects which path outputs the increment decision inc. In addition the increment decision computes the LSB (before fixing for RNE) in case an overflow does not occur.

5.2. Details

In this section we describe the details of the *increment decision* box and the *LSB-fix for RNE* box.

Increment decision box. The outputs of the *increment decision* box are the increment decision inc and the bit L that equals the LSB of the rounded product before fixing in the case that no overflow occurs. The increment decision is partitioned into two paths. One for the case that an overflow occurs which computes the signal inc_{ovf}, and the other path for the case that no overflow occurs which computes the signal inc_{novf}. The increment signal inc is then selected from these two paths according to the overflow detection by bit $Y0[\Leftrightarrow 1]$. The following equations define the signals inc_{ovf}, inc_{novf}, and inc.

$$inc_{novf} = \begin{cases} (L_{X'} + C[52] \geq 2) & \text{if RZ} \\ (R + L_{X'} + C[52] \geq 2) & \text{if RNU} \\ ((S \vee R) + L_{X'} \geq 2) & \text{if RI and } C[52] = pred \\ 0 & \text{if RI and } C[52] \neq pred \end{cases}$$

$$inc_{ovf} = \begin{cases} (L_{X'} + C[52] \geq 2) & \text{if RZ} \\ (L_{X'} + C[52] \geq 1) & \text{if RNU} \\ (S \vee R \vee L_{X'}) & \text{if RI and } C[52] = pred \\ L_{X'} & \text{if RI and } C[52] \neq pred \end{cases}$$

$$inc = \begin{cases} inc_{novf} & \text{if } Y0[\Leftrightarrow 1] = 0 \\ inc_{ovf} & \text{if } Y0[\Leftrightarrow 1] = 1 \end{cases}$$

The bit L, which equals the LSB of the rounded product (before fixing) in case no overflow occurs, is defined by:

$$L = \begin{cases} L_{X'} \oplus C[52] & \text{if RZ} \\ R \oplus L_{X'} \oplus C[52] & \text{if RNU} \\ (S \vee R) \oplus L_{X'} \oplus C[52] \oplus pred & \text{if RI} \end{cases}$$

Note that the case of RI is complicated due to the possibility that $pred \neq C[52]$. If $pred = C[52]$, then $L = (S \vee R) \oplus L_{X'}$, but if $pred \neq C[52]$, the the effect of the wrong prediction is reversed by $(S \vee R) \oplus L_{X'} \oplus pred \oplus C[52]$.

LSB-fix for RNE. The *LSB-fix for RNE* box outputs two signals: $not(pd)$ is used to pull-down the LSB if a "tie" occurs, but no overflow occurs, and $not(pd')$ is used to pull-down the LSB if a "tie" and an overflow occur. These signals are defined as follows:

In contrast to injection based rounding no injection or prediction is contained in the $L_{X'}$, R and S-bit computation in RNE. If no overflow occurs, a "tie" occurs iff $R = 1$ and $S = 0$, in which case the LSB should be pulled down for RNE. Therefore,

$$not(pd) = \text{OR}\ (not(R), S, not(RNE))$$

If an overflow occurs, a "tie" occurs iff $L_{X'} + C[52] = 1$, $R = 0$ and $S = 0$, in which case the LSB should be pulled down for RNE. Therefore,

$$not(pd') = \text{OR}\ (not(L_{X'} \oplus C[52]), R, S, not(RNE))$$

The full version [7] contains a proof that the selection signal *inc* selects correctly between the incremented sum and the non-incremented sum.

5.3. Delay analysis

Figure 3 depicts the rounding algorithm based on Quach *et al.* [18] with delay annotation. The delay assumptions that are used here are similar to those used in the two previous rounding algorithms. The rounding algorithm depicted in Fig.3 uses a prediction logic which lies on the critical path. The delay of the prediction logic is two logic levels.

Following Quach *et al.*, Fig.3 depicts a non-optimized processing order in which the post-normalization shift takes place after the round selection. The increment decision box is assumed to be organized as follows: The bits S, $C[52]$, R and $Y0[\Leftrightarrow 1]$ are valid after $6, 7, 8, 10$ logic levels, respectively. To minimize delay we implement the rounding equations by 4 levels of multiplexers, so that the results can be

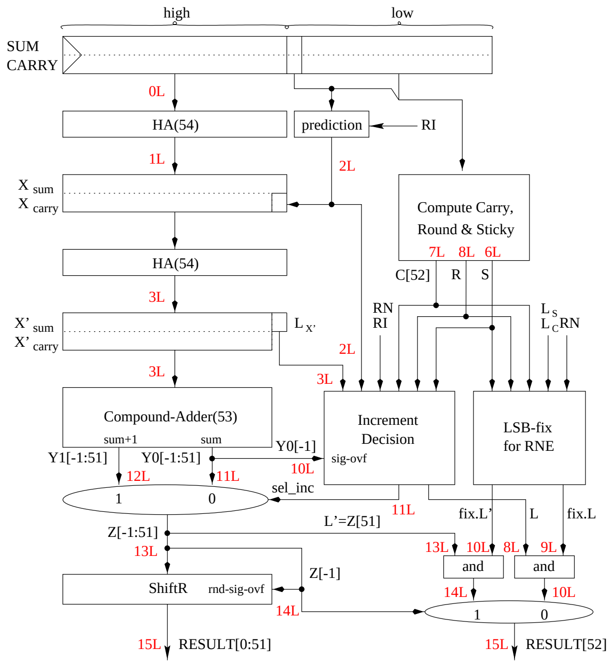

Figure 3. Block diagram of the QTF rounding algorithm annotated with timing estimates

selected conditionally the signals as they arrive. Thus, a total delay of 15 logic levels is obtained. By performing the post-normalization before the round selection, one logic level can be saved to obtain a total delay of 14 logic levels.

6. Summary and Conclusions

A new IEEE compliant floating-point rounding algorithm for computing the rounded product from a carry-save representation of the product is presented. The new rounding algorithm is compared with two previous rounding algorithms. For each rounding algorithm, a logical description and a block diagram is given, and the latency is analyzed. To make the comparison as relevant as possible, we considered optimizations of the previous algorithms which improve the delay.

After the optimizations the main differences between the three rounding algorithms lie in: (a) the rounding decision; and (b) the partitioning of the carry-save encoded product. These differences are discussed below.

(a) The QTF and ES algorithms simplify the rounding decision by an early addition of a value (this value is called the *prediction* in the QTF algorithm and the *injection* in the ES algorithm). In the QTF algorithm, the prediction depends on the rounding mode and on the carry-save digit positioned 53 digits to the right of the radix point. In the ES algorithm, the injection depends only on the rounding mode and we assume that it is added in with the partial products, and thus the product already includes the injection. The rounding decision in the YZ algorithm is based on customary rounding tables.

(b) In the YZ algorithm the lower and upper parts of the carry-save product are separated by a "buffer" of three carry-save digits in positions $[51:53]$, where the position of a digit denotes how many digits it is to the right of the radix point. In the other two algorithms the upper part consists of positions $[\Leftarrow 1:52]$, and the lower part consists of position $[53:104]$.

Supporting all four rounding modes of the IEEE 754 Standard is an error prone task. In the full version [7] we therefore provide correctness proofs of all three algorithms which describe, formalize and clarify the tricky aspects. From this point of view, the YZ algorithm is easiest to prove and the QTF algorithm is the most intricate (especially the rounding decision logic).

Our conclusion of the delay analysis is that the new ES rounding algorithm is the fastest rounding algorithm, provided that an injection is added in during the reduction of the partial products into a carry-save encoded digit string. With the ES algorithm the rounded product can be computed in 12 logic levels in double precision (i.e. when the significands are 53 bits long). If the injection is not added in during the reduction of the partial products into a carry-save encoded digit string, then an extra step of adding in the injection is required. This step amounts to a carry-save addition, and the latency associated with it is that of a full-adder, namely, 2 logic levels. Thus, if the injection is added in late, then the latency of the ES rounding algorithm is 14 logic levels. The addition of the injection during the reduction of the partial products can be accomplished without a slowdown or with a very small slowdown. The justification for this is: (a) The partial products are usually obtained by Booth recoding and by selecting (e.g. 5:1 multiplexer), and hence, are valid much later than the injection; and (b) The delay of adding the partial products does not increase strictly monotonically as a function of the number of partial products. The delay incurred by adding in the injection, if any, depends on the length of the significands and on the organization of the adder tree.

The other two rounding algorithms do not require an injection, and in double precision, the latency of the QTF rounding algorithm is 14 logic levels. The YZ rounding algorithm ranks as the slowest rounding algorithm, with a latency of 16 logic levels in double precision.

References

[1] H. Al-Twaijry. *Area and Performance Optimized CMOS Multipliers*. PhD thesis, Stanford University, August 1997.

[2] R. Brent and H. Kung. A Regular Layout for Parallel Adders. *IEEE Trans. Comput.*, C-31(3):260–264, 1982.

[3] P. Bromham and R. Carlton. Method for rounding using redundant coded multiply result. U.S. patent 5680339, 1997.

[4] H. Darley. Method and apparatus for rounding in high-speed multipliers. U.S. patent 5170371, 1992.

[5] B. Drerup and E. Swartzlander. Fast multiplier bit-product matrix reduction using bit-ordering and parity generation. *Journal of VLSI Signal Processing*, 7:249–257, 1994.

[6] G. Even, S. Mueller, and P.-M. Seidel. A Dual Mode IEEE multiplier. In *Proc of the 2nd IEEE Int. Conf. on Innovative Systems in Silicon*, pages 282–289. IEEE, 1997.

[7] G. Even and P.-M. Seidel. A comparison of three rounding algorithms for IEEE floating-point multiplication. Technical Report EES1998-8, EES Dep., Tel-Aviv Univ., 1998. http://www.eng.tau.ac.il/Utils/reportlist/reports/repfram.html.

[8] C. Hinds, E. Fiene, D. Marquette, and E. Quintana. Parallel method and apparatus for detecting and completing floating point operations involving special operands. U.S. patent 5339266, 1994.

[9] IEEE standard for binary floating point arithmetic. ANSI/ IEEE754-1985, New York, 1985.

[10] C. Lee. Multistep gradual rounding. *IEEE Transactions on Computers*, 32(4):595–600, April 1989.

[11] C. Martel, V. Oklobdzija, R. Ravi, and P. Stelling. Design strategies for optimal multiplier circuits. *Proceedings 12th Symposium on Computer Arithmetic*, 12:42–49, 1995.

[12] Z.-J. Mou and F. Jutand. Overturned-stairs adder trees and multiplier design. *IEEE Transactions on Computers*, 41(8):940–948, August 1992.

[13] S. Oberman. *Design Issues in High Performance Floating Point Arithmetic Units*. PhD thesis, Stanford, January 1997.

[14] S. Oberman, H. Al-Twaijry, and M. Flynn. The SNAP project: Design of floating point arithmetic units. In *Proceedings of the 13th Symposium on Computer Arithmetic*, volume 13, pages 156–165. IEEE, 1997.

[15] R. Owens, R. Bajwa, and M. Irwin. Reducing the number of counters needed for integer multiplication. *Proceedings 12th Symposium on Computer Arithmetic*, 12:38–41, 1995.

[16] K. Pang, H.-W. Soong, R. Sexton, and P. Ang. Generation of high speed CMOS multiplier-accumulators. *Proceedings IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 217–220, 1988.

[17] W.-C. Park, T.-D. Han, S.-D. Kim, and S.-B. Yang. Floating Point Multiplier Performing IEEE Rounding and Addition in Parallel. *accepted in Journal of System Architecture*, 1998.

[18] N. Quach, N. Takagi, and M. Flynn. On fast IEEE rounding. Technical Report CSL-TR-91-459, Stanford, Jan. 1991.

[19] M. Santoro, G. Bewick, and M. Horowitz. Rounding algorithms for IEEE multipliers. In *Proceedings 9th Symposium on Computer Arithmetic*, pages 176–183, 1989.

[20] P.-M. Seidel. How to half the latency of IEEE compliant floating-point multiplication. In *Proceedings of the 24th Euromicro Conference*, volume 24. IEEE, 1998.

[21] A. Tyagi. A reduced-Area Scheme for Carry-Select Adders. *IEEE Trans. Comput.*, 42(10):1163–1170, 1993.

[22] C. Wallace. A suggestion for parallel multipliers. *IEEE Trans. Electron. Comput.*, EC-13:14–17, 1964.

[23] R. Yu and G. Zyner. 167 MHz Radix-4 floating point multiplier. *Proceedings 12th Symposium on Computer Arithmetic*, 12:149–154, 1995.

[24] R. Yu and G. Zyner. Method and apparatus for partially suporting subnormal operands in floating point multiplication. U.S. patent 5602769, 1997.

[25] G. Zyner. Circuitry for rounding in a floating point multiplier. U.S. patent 5150319, 1992.