ELSEVIER

INTEGRATION, the VLSI journal 29 (2000) 167–180

INTEGRATION
theVLSI journal

www.elsevier.com/locate/vlsi

A dual precision IEEE floating-point multiplier☆

Guy Even[a,1], Silvia M. Mueller[b,*,2], Peter-Michael Seidel[c,3]

[a]*Department of Electrical Engineering-Systems, Tel-Aviv University, Tel-Aviv 69978, Israel*
[b]*IBM Development Germany, Lab Boeblingen – Dept. 3173, D-71032 Boeblingen, Germany*
[c]*Computer Science and Engineering Department, Southern Methodist University, Dallas, TX 75275, USA*

Received 12 February 1999

Abstract

A new algorithm for computing IEEE-compliant rounding is presented, called *injection-based rounding*. Injection-based rounding is simple and facilitates using the same rounding circuitry for different precisions. We demonstrate the usefulness of injection-based rounding in a design of an IEEE floating-point multiplier capable of performing either a double-precision multiplication or a single-precision multiplication. The multiplier is designed to minimize hardware cost by using only a half-sized multiplication array and by sharing the rounding circuitry for both precisions. The latency of the multiplier is in single-precision two clock cycles and in double precision the latency is three clock cycles, where each pipeline stage contains roughly 15 logic levels.

Keywords: IEEE floating point arithmetic; Rounding; Floating point multiplier

1. Introduction

Fast, low precision floating-point operations have been acknowledged recently as very useful for real-time 3D graphic applications [2,5,6,9]. The new 3D graphic applications reverse the previous trend of focusing on higher and higher precisions and set a new performance goal. The new design

☆A preliminary version of this paper titled "A Dual Mode IEEE Multiplier" appeared in the Proceedings of the 2nd IEEE International Conference on Innovative Systems in Silicon (ISIS'97), pp. 282–289, 1997.

[1] Supported in part by the North Atlantic Treaty Organization under a grant awarded in 1996.

* Corresponding author. Fax: + 49-7031-16-2829.

[2] Supported by the German Science Foundation (DFG). Research done while affiliated with CS Department, University of Saarland, Germany

E-mail addresses: guy@eng.tau.ac.il (G. Even), smm@de.ibm-com (S.M. Mueller), seidel@seas.smu.edu (P.-M. Seidel).

[3] Supported by the DFG Graduiertenkolleg "Effizienz und Komplexitaet von Algorithmen und Rechenanlagen", Universitaet des Saarlandes.

0167-9260/00/$ - see front matter
PII: S0167-9260(00)00006-7

goal is to design cheap IEEE floating-point units (FPUs) capable of both, double- and single-precision operation with a higher throughput for single-precision operations.

IEEE FPUs support both, single- and double-precision operation. However, the throughput of low-precision floating-point operations is often smaller than high-precision operations. This is caused by the necessity to translate single-precision operands to double-precision format, and then translate the double-precision result back to single-precision format. In the x86 architecture (e.g. Pentium) and in the Motorola 68000 family, computations are performed in extended double precision and translation to low precision requires storing the result to memory. Our goal is to present designs of cheap and versatile IEEE FPUs that deliver good performance for both precisions.

Our multiplier design supports all four IEEE rounding modes. In single precision the latency is two clock cycles and in double precision the latency is three clock cycles, where each pipeline stage contains roughly 15 logic levels. The multiplier uses a half-sized (i.e. 27×53) multiplication array, and thus, the cost of the multiplier is reduced (see for example [1,3]). In double-precision multiplication, the multiplication array is used during the first two cycles. Therefore, new multiplications can be issued only after two clock cycles. In single-precision multiplication, a new multiplication can be issued in the next clock cycle. The hardware overhead and delay caused by supporting both precisions compared to a similar design that supports only double-precision multiplications is surprisingly small.

We present a new rounding algorithm which simplifies supporting different precisions and enables using the same rounding circuitry for both precisions. In addition, our rounding algorithm advances some of the rounding computation to the addition tree without adding a significant delay to the addition tree. Our rounding algorithm differs from previous suggestions [4,7,10,11] by injecting a value that depends only on the rounding mode.

The paper is organized as follows: In Section 2, we briefly review previous rounding algorithms and the computation of a sticky-bit from a carry-save representation. In Section 3, we define injection-based rounding. In Section 4, we describe the structure and functionality of the dual precision multiplier. In Section 5, we describe the non-standard blocks of the multiplier design. In Section 6, we prove the correctness of the implementation of the rounding algorithm. A conclusion is given in Section 7.

2. Previous work

Our rounding algorithm is designed to support multiple precisions and enables using the same rounding circuitry while still obtaining a fast rounding algorithm.

We briefly describe previous rounding algorithms that we are aware of:

1. Santoro et al. [10] and Quach et al. [7,10] reduced the four IEEE rounding modes to three modes called *round-to-zero* (RZ), *round-to-infinity* (RI), and *round-to-nearest-even* (RNE). These modes correspond to a "floor" rounding, a "ceiling" rounding, and the well-known round-to-nearest mode.

   Furthermore, they described how to reduce RNE to a rounding mode called round-to-nearest-up (RNU). The difference between RNE and RNU is that in a case of a tie, the number is

rounded up in RNU, whereas in RNE it is rounded to the even significand. Given a RNU rounding unit, one can easily obtain the correct RNE result by pulling down the LSB in case of a tie that should have been rounded down rather than rounded up.

Given a precision p (where $p = 24$ in single precision and $p = 53$ in double precision), the rounding algorithms of Santoro et al. [10] and Quach et al. [7] are based on the above reduction and on truncating the sum and carry vectors in the bit-position $-(p-1)$ and adding a prediction of size $2^{-(p-1)}$ based on the rounding mode and the values of the sum and carry vectors in bit-position $[-p]$. These rounding algorithms require a 3-way compound adder that computes the sum, the incremented sum, and the sum plus 2.

2. Yu and Zyner [11,12] presented a fast rounding algorithm in which the product given as a carry-save number is divided into 3 parts: the lower part produces a carry-bit and a sticky-bit, the upper part is compressed to obtain the sum and the incremented sum, and the middle part is used to compute the least-significant bit (LSB), the guard-bit, and the round-bit. Two parallel paths are used for the rounding decision: one for the case that the product is in the range [1,2) and one for the case that the product is in the range [2,4). The final selection is based on the most significant bit (MSB) of the product and the carry generated by the rounding.
3. Gamez et al. [4] presented a rounding algorithm that avoids having to shift the operands and the product while partially sharing the same rounding circuitry for all precisions. Their algorithm is input a non-redundant representation of the number to be rounded. The algorithm is based on having two paths: In one path, the number is padded with zeros starting from the rounding position. In the second path, the number is padded by ones starting at the rounding position, and then it is incremented (increment occurs at a fixed position at the far right end of the number to allow sharing of the rounding circuitry in different precisions). The rounding decision based on the true LSB, round-bit, and sticky-bit selects one of these numbers as the rounded result. Note that this rounding algorithm requires normalizing the product before rounding so that it is in the range [1,2).
4. Saishi et al. [8] presented a rounding algorithm for fixed-point two's complement multiplication that supports two rounding modes (round-to-zero and RNU). Their rounding algorithm is based on injecting a value determined by the sign of the product and the rounding mode so that rounding is reduced to truncation. Our rounding algorithm is an extension of this idea to accommodate for different precisions in the context of floating-point multiplication.

2.1. Sticky- and carry-bit computation

The sticky- and carry-bit computation is performed on a carry-save encoded digit string having 52 digits in double precision and 23 digits in single precision. In the case of double precision, the computation is spread over two clock cycles; in each cycle 26 digits are processed.

The naive way of computing the sticky-bit of a carry-save encoded string is to compress the string to a non-redundant binary encoded string and then OR the bits. A faster method was suggested by Yu et al. [11] in which the carry-save encoded string is processed by a constant depth circuit before being input to a tree of OR-gates. The carry- and the sticky-bit computation can share some of the circuitry to reduce the cost.

3. Injection-based rounding

In this section we present a new method for implementing IEEE rounding called injection-based rounding.

The IEEE Standard defines four rounding modes: round toward 0, round toward $+\infty$, round toward $-\infty$, and round to nearest. We deal with numbers that are represented in a sign-magnitude representation. The rounding modes round toward $+\infty$ and round toward $-\infty$ can be reduced, based on the sign of the number, either to round toward zero (i.e. truncate) or round to infinity (i.e. round up) [7]. We assume henceforth that the number to be rounded is non-negative. We focus on three rounding modes: RZ – round toward zero, RI – round toward infinity, and RNE – round to nearest (even).

Following Quach et al., we implement RNE by RNU (round to nearest up) followed by an adjustment of the LSB. RNU mode differs from RNE mode only when the exact result is in the midpoint between two successive representable values; in this case RNE rounds to the representable value with the zero LSB and RNU rounds to the larger representable value. A discrepancy between RNE and RNU can occur only if the LSB in RNU is 1, in which case pulling down the LSB to zero results with the RNE result.

The role of the injection in injection-based rounding is to reduce the three rounding modes RZ, RNU, RI to RZ. Namely,

$$round_{\text{mode}}(a \cdot b) = round_{RZ}(a \cdot b + injection).$$

If $a \cdot b \in [1,2)$, then the value of the injection depends only on the rounding mode. Let p denote the length of the significand, that is: $p = 24$ in single-precision format, and $p = 53$ in double-precision format. The value of the injection is defined as follows:

$$\text{INJECTION} = \begin{cases} 0 & \text{in RZ mode,} \\ 2^{-p} & \text{in RNU mode,} \\ 2^{-(p-1)} - 2^{-2(p-1)} & \text{in RI mode.} \end{cases}$$

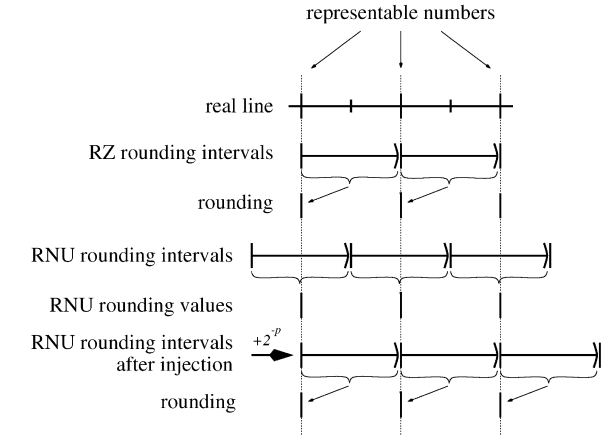
The injection reduces all three rounding modes (RZ, RNU, and RI) to truncation provided that the product is in the range [1,2). Fig. 1 depicts this reduction.

When the product is in the range [2,4), the injection should be $2 \cdot 2^{-p}$ in RNU mode, and $2^{-(p-2)} - 2^{-2(p-1)}$ in RI mode. Since the injection is added early based on the assumption that the product is in the range [1,2), we need to correct the injection when the product is in the range [2,4).

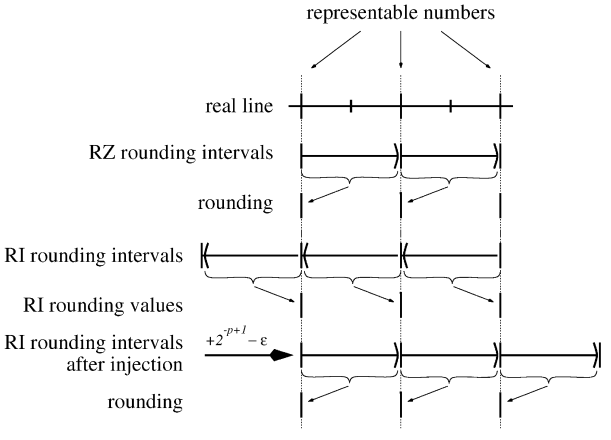
The correction of the injection depends on the rounding mode. The injection correction is defined as follows:

$$\text{COR\_INJ} = \begin{cases} 0 & \text{in RZ mode,} \\ 2^{-p} & \text{in RNU mode,} \\ 2^{-(p-1)} & \text{in RI mode.} \end{cases}$$

The correction of the injection is implemented by the *correct injection* box in Fig. 5.



(A) reduction of RNU to RZ



(B) reduction of RI to RZ

Fig. 1. Reduction of the rounding modes by injection: (A) reduction of RNU to RZ, (B) reduction of RI to RZ.

4. Description of the multiplier

In this section we describe the multiplier. A block diagram of the significand's path in the multiplier is depicted in Fig. 5. We describe the operation of the multiplier in each precision separately.

4.1. Double precision

We assume that the two significands, A and B, are normalized (namely, in the range [1,2)), and held in registers. Fig. 2 depicts the timing diagram describing the operation of the multiplier. In the

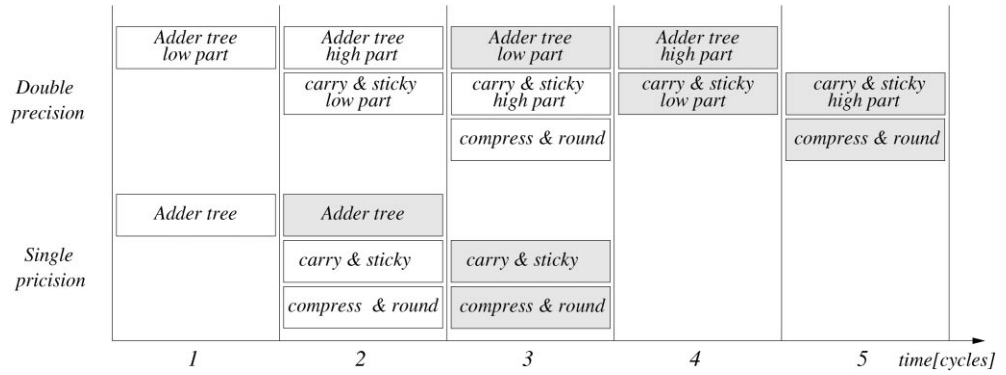


Fig. 2. Timing diagram of dual-precision multiplier. The shaded boxes depict a successive multiplication operation.

first clock cycle, A is multiplied by the 26 least-significant bits of B, namely $B[-27:-52]$. The injection is added to the product in the first clock cycle. The product, encoded as a carry-save digit string, is latched in two registers, SUM and CARRY, and has digits in positions $[-26:-104]$. In the second clock cycle, two stages of the pipeline are active: In the first stage of the pipeline, A is multiplied by the 27 most-significant bits of B, namely $B[0:-26]$. The 53 most significant carry-save digits (corresponding to positions $[-26:-78]$) of the product computed in the first cycle are fed back to the addition tree and added with the product computed in the second cycle. The higher part of the product, corresponding to positions $[1:-78]$, is latched at the end of the second clock cycle by the registers SUM and CARRY. In the second stage of the pipeline, the 26 least significant carry-save digits (corresponding to positions $[-79:-104]$) of the product computed in the first clock cycle are fed to the *carry & sticky computation* box. Note that the product computed in the first clock cycle has only 79 digits, whereas in the second cycle the computed product has 80 digits. The *stk-mux* before the *carry & sticky computation* box selects during the second cycle the 26 digits corresponding to positions $[-79:-104]$, and during the third cycle, the 26 digits corresponding to positions $[-53:-78]$ are selected. The *carry & sticky computation* box outputs the carry-bit $C[-78]$ and the sticky-bit *sticky_low* which are latched for the third clock cycle.

The multiplier array and the addition tree need to support adding the injection during the first cycle and adding the feedback during the second cycle. The main task of the multiplier array is to multiply 53 bits by 27 bits. This means that the array has 27 rows if Booth recoding is not used, and 14 rows if radix 4 Booth recoding is used. In either case, one can add two additional rows to the addition tree without incurring a significant delay to the addition tree. These extra two rows can be used to feed the injection during the first cycle and the feedback during the second cycle.

In the third clock cycle, the 26 least-significant bits of SUM and CARRY (bit positions $[-53:-78]$) are input to the carry computation box. The sticky computation box is fed by the digits in positions $[-54:-78]$. The *carry & sticky computation* box gets also the inputs of the carry-bit $C[-78]$ and the sticky-bit *sticky_low* computed during the second clock cycle (for positions $[-79:-105]$). The results of this computation are the carry-bit $C[-52]$ (to be input to position $[-52]$), the round-bit R (that corresponds to position $[-53]$), and the sticky-bit corresponding to positions $[-54:-105]$.

During the third cycle, the 54 most-significant bits of S and C (bit positions $[1:-52]$) are added by a 3-way adder that computes the sum, the incremented sum, and the sum plus 2. (To be precise, the values computed are: the sum, the sum plus 2^{-52}, and the sum plus 2^{-51}.) After the 3-way addition, three selections take place to determine the rounded product:

1. The carry-bit $C[-52]$ controls the selection of the pair $sum, sum+1$ or the pair $sum+1, sum+2$. We denote the selected pair by $x, x+1$.
2. The MSB (bit position [1]) of the sum together with the rounding-mode and the carry-bit R are fed to the *correct injection* box to determine which value is selected: x or $x+1$. This selection implements the correction of the injection in case the product is in the range [2,4).
3. The MSB determines whether the product is in the range [1,2) or in the range [2,4). If the MSB equals 1, then the product is shifted to the right so that it will be in the range [1,2]. Note, that the MSB includes the injection but ignores the carry-bit $C[-52]$. Nevertheless, we prove in Section 5 that the rounding is correct.

The order of the selections is chosen to obtain a reasonable delay and cost. If a further reduction in timing is desired, one could perform the right-shifting that is controlled by the MSB before the selection which implements the injection correction. However, such a change would require having two right-shifters rather than one.

Finally, two corrections take place: (a) the LSB of the rounded product is fixed (in the case of a discrepancy between RNE and RNU, as explained later); and (b) the MSB of the rounded product is fixed since significand overflow might occur. Therefore, the product has to be brought to the range [1,2). This is done by OR-ing the two most-significant bits of the product (the two bits left of the radix point) to produce the final most-significant bit of the product.

Note, that the first stage of the pipeline is engaged in double-precision multiplication during the first and second clock cycles. Therefore, a new multiplication can be issued only after two clock cycles, yielding a throughput of one double-precision multiplication per two clock cycles as depicted in Fig. 2.

4.2. Single precision

We assume that Registers A and B hold 24-bit normalized significands. Note, that one could consider packing two single-precision significands in each register, and perform the corresponding multiplications one after the other. We omit this packing possibility to simplify the description.

In single-precision multiplication, multiplexer *Amux* shifts A by 29 positions to the right. This "right-justification" causes the least-significant bit of the single-precision product (i.e. bit $[-23]$) to be placed in the same position as the least-significant bit of the double-precision product (i.e. bit $[-52]$). The advantage of this shifting is that the bit positions required for rounding in both precisions become identical and hence the same circuitry can be used. Multiplexer *Bmux* selects B but keeps it "left-justified" (i.e. the radix point is aligned with the radix point in double precision).

In the first clock cycle the multiplication array computes $A \cdot B + injection$. The computed products are latched in the SUM and CARRY registers, and no feedback is used.

During the second clock cycle, the product, stored in registers SUM and CARRY, is compressed and rounded. The lower part is input to the carry & sticky computation box, and the upper part is

added using the 3-way adder. The only modification compared to the double-precision rounding is the position of the MSB. Since the single-precision product is aligned with the double-precision product with respect to the L-bit position, the MSB of the single-precision number appears in position $[-28]$. The selection of the correct MSB is done by the *msb-mux*. At the end of the computation, the product needs to be shifted back 29 positions to the left so that it is "left-justified".

Note, that the first stage of the pipeline is engaged in single-precision multiplication only during the first clock cycle. Therefore, a multiplication can be issued immediately after a single-precision multiplication, yielding a throughput of one single-precision multiplication per clock cycle as depicted in Fig. 2.

5. Details

In this section, we describe the functionality of the non-standard boxes that appear in Fig. 5: the *injection* box, the *carry & sticky computation* box, the *correct injection* box and the *fix LSB & MSB* box.

Injection box: The injection box reduces the four IEEE rounding modes to three rounding modes based on the sign of the product. The value of the injection is then determined by the reduced rounding mode. The shifting of the multiplicand A by 29 positions to the right in the case of single precision unifies the injection value for single precision and double precision.

Carry & sticky computation box: Fig. 3 depicts the functionality of the carry & sticky computation box with the *sticky low* and *carry low* registers. The SUM and CARRY strings are given two logical names: SUM_1, CARRY_1 and SUM_2, CARRY_2, where the subscript refers to the clock cycle in which the stored value is latched. The bit positions of SUM_1 and CARRY_1 are $[-26:-105]$ and the bit positions of SUM_2 and CARRY_2 are $[1:-78]$.

The computation of the carry, round and sticky bit proceeds as follows: In a double-precision computation in the second clock cycle, the SUM_1 and CARRY_1 are input into the 26 carry-save digit

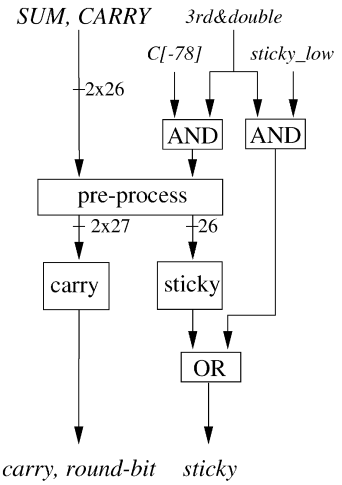


Fig. 3. Carry and Sticky computation box.

sticky and carry-bit computation. The sticky-bit output is latched by the *sticky low* register. The carry-bit output, which is the $C[-78]$ carry bit, is latched by the *carry low* register. In the third clock cycle, the SUM_2 and CARRY_2 and the *carry low* are input into the 26 carry-save digit sticky and carry bit computation. The sticky-bit output is combined with the *sticky low* to obtain the sticky bit S. The carry-bit output is the $C[-53]$ carry bit. The round bit R (corresponding to the bit in position $[-53]$ in the non-redundant binary encoded sum of SUM and CARRY) and the carry-bit $C[-52]$ are computed by adding $C[-53]$ with $\text{SUM}_2[-53]$ and $\text{CARRY}_2[-53]$.

In single-precision multiplication, only SUM_2 and CARRY_2 are valid, and the computation of the carry, round and sticky bit is completed in one cycle.

The *pre-process* box before the *carry* and *sticky* boxes computes the bitwise AND, XOR and OR of the inputs which are used by the *carry* box and the *sticky* box.

Correct injection box: The correct injection box deals with adding the correction to the injection when the product is in the range [2,4). Assuming that the *msb* bit signals this case correctly, the correction of the injection is as follows:

1. In RZ, no correction is required.
2. In RNU, the correction value is $2^{-p} = 2^{-53}$. This is one bit to the right of the LSB of the product. However, if the round-bit R equals 1, then the correction together with the round-bit amount to 2^{-52}, which implies that the sum needs to be incremented.
3. In RI, the correction value is $2^{-(p-1)} = 2^{-52}$. This means that the sum needs to be incremented.

Therefore, the COR_INJ signal which controls the *inc-mux* is defined by

$$\text{COR\_INJ} = msb \text{ AND } (RI \text{ OR } (RNU \text{ AND } R)).$$

Fix LSB & MSB box: In case of the rounding mode RNE, our rounding algorithm computed RNU instead of RNE. To fix the LSB of the rounded product, we introduce the signal *npd* (no pull down) which equals 0 when the LSB should be pulled down.

We define the *npd* signal by

$$npd = \overline{RNE} \text{ OR } R \text{ OR } S \text{ OR } (msb \text{ AND } Z[-53]).$$

The correctness proof of the signal *npd* is given in Section 6. Fig. 4 depicts the functionality of the *Fix LSB & MSB* box. The multiplexer deals with shifting back the product to the left in case of single precision. The position of the MSB in single-precision multiplication is $[-28]$.

The *msb* might fail to report overflow correctly, namely, the *sum* does not overflow, but the incremented sum does. In this case, the rounded result equals 2, as proved below, and therefore, the post-normalization shift is implemented by an OR-gate, which fixes the MSB.

6. Correctness

The main issue that we address in this section is whether the usage of the *msb* signal for deciding if an overflow occurred is correct. As long as the *msb* bit indicates whether the exact product is greater than or equal to 2, it is easy to see that the algorithm rounds correctly. But one should also consider the following cases in which the *msb* signal is "wrong": (a) $msb = 0$ and the exact product is

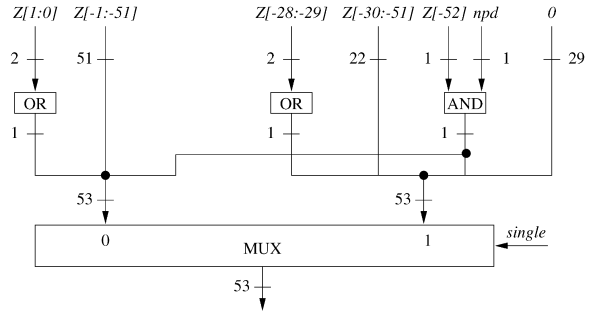


Fig. 4. Fix LSB & MSB box.

greater than or equal to 2; and (b) $msb = 1$ and the exact product (without the injection) is less than 2. The following claim shows that correct rounding can be based on the signal *msb* even though the *msb* signal might be wrong. Moreover, the proof of Claim 1 shows that, whenever the *msb* signal is wrong, the rounded product equals 2.

We prove the correctness of rounding in double-precision computations. The proof in case of single-precision computations follows the same lines.

Claim 1. *Let* EXACT *denote the exact product, namely* $\text{EXACT} = |A| \cdot |B|$. *Consider the registers* SUM$[1:-104]$ *and* CARRY$[1:-104]$ *which satisfy*

$$|\text{SUM}[1:-104]| + |\text{CARRY}[1:-104]| = \text{EXACT} + \text{INJECTION}.$$

Let $sum = |\text{SUM}[1:-52]| + |\text{CARRY}[1:-52]|$ *and define msb by*

$$msb = \begin{cases} 1 & \text{if } sum \geqslant 2, \\ 0 & \text{otherwise.} \end{cases}$$

Then correct rounding of EXACT *can be computed as follows*:

$$r_{mode}(\text{EXACT}) = \begin{cases} r_{RZ}(\text{EXACT} + \text{INJECTION}) & \text{if } msb = 0, \\ r_{RZ}(\text{EXACT} + \text{INJECTION} + \text{COR\_INJ}) & \text{if } msb = 1, \end{cases}$$

where $mode \in \{RZ, RNU, RI\}$.

Proof. We consider two main cases: (1) $msb = 0$; and (2) $msb = 1$:

1. Suppose $msb = 0$. If $\text{EXACT} < 2$ then the claim follows from the definition of injection. If $\text{EXACT} \geqslant 2$, then

$$\text{EXACT} + \text{INJECTION} \in [2, sum + 2^{-51}), \tag{1}$$

The reason for this is that the contribution of $|\text{SUM}[-53:-104]| + |\text{CARRY}[-53:-104]|$ is in the range $[0, 2^{-51})$.

Since $msb = 0$, it follows that $sum \leqslant 2 - 2^{-52}$. Substituting the upper bound on *sum* in Eq. (1) yields

$$\text{EXACT} + \text{INJECTION} \in [2, 2 + 2^{-52}). \tag{2}$$

We assume that the exact product is in the range $[2, 4)$, and hence, the injection should be corrected by adding an injection correction. The correction of the injection, COR_INJ, is in the range $[0, 2^{-52}]$, therefore

$$\text{EXACT} + \text{INJECTION} + \text{COR\_INJ} \in [2, 2 + 2^{-51}). \tag{3}$$

The definition of injection and the injection correction implies that $r_{mode}(\text{EXACT}) = r_{RZ}(\text{EXACT} + \text{INJECTION} + \text{COR\_INJ})$. Eqs. (2) and (3) imply that

$$r_{RZ}(\text{EXACT} + \text{INJECTION} + \text{COR\_INJ}) = r_{RZ}(\text{EXACT} + \text{INJECTION}) = 2$$

and part 1 follows.

2. Suppose $msb = 1$. If $\text{EXACT} \geqslant 2$ then the claim follows from the definition of injection and the injection correction. If $\text{EXACT} < 2$, then since $\text{INJECTION} \in [0, 2^{-52})$, it follows that

$$\text{EXACT} + \text{INJECTION} \in [2, 2 + 2^{-52}). \tag{4}$$

The proof now follows the proof in case $msb = 0$. □

We now prove that correct rounding is computed by the circuitry described in Fig. 5.

Claim 2. *Consider the notation used in Claim* 1:

1. *If* $msb = 0$ *then*

$$r_{RZ}(\text{EXACT} + \text{INJECTION}) = sum + C[-52] \cdot 2^{-52}.$$

2. *If* $msb = 1$ *then*

$$r_{RZ}(\text{EXACT} + \text{INJECTION} + \text{COR\_INJ}) = ((sum/2 + (C[-52] + \text{COR\_INJ}) \cdot 2^{-53}) div\, 2^{-52}) \cdot 2^{-52}.$$

Proof. The definitions of *sum* and $C[-52]$ imply that

$$\text{EXACT} + \text{INJECTION} = sum + C[-52] \cdot 2^{-52} + tail, \tag{5}$$

where $tail \in [0, 2^{-52})$. If $msb = 0$, it follows that the rounded product is in the range $[1, 2]$, and part (1) of the claim follows.

The second part of the claim follows from the shifting of the significand by one position to the right and chopping off the tail in positions $[-53{:}-105]$. □

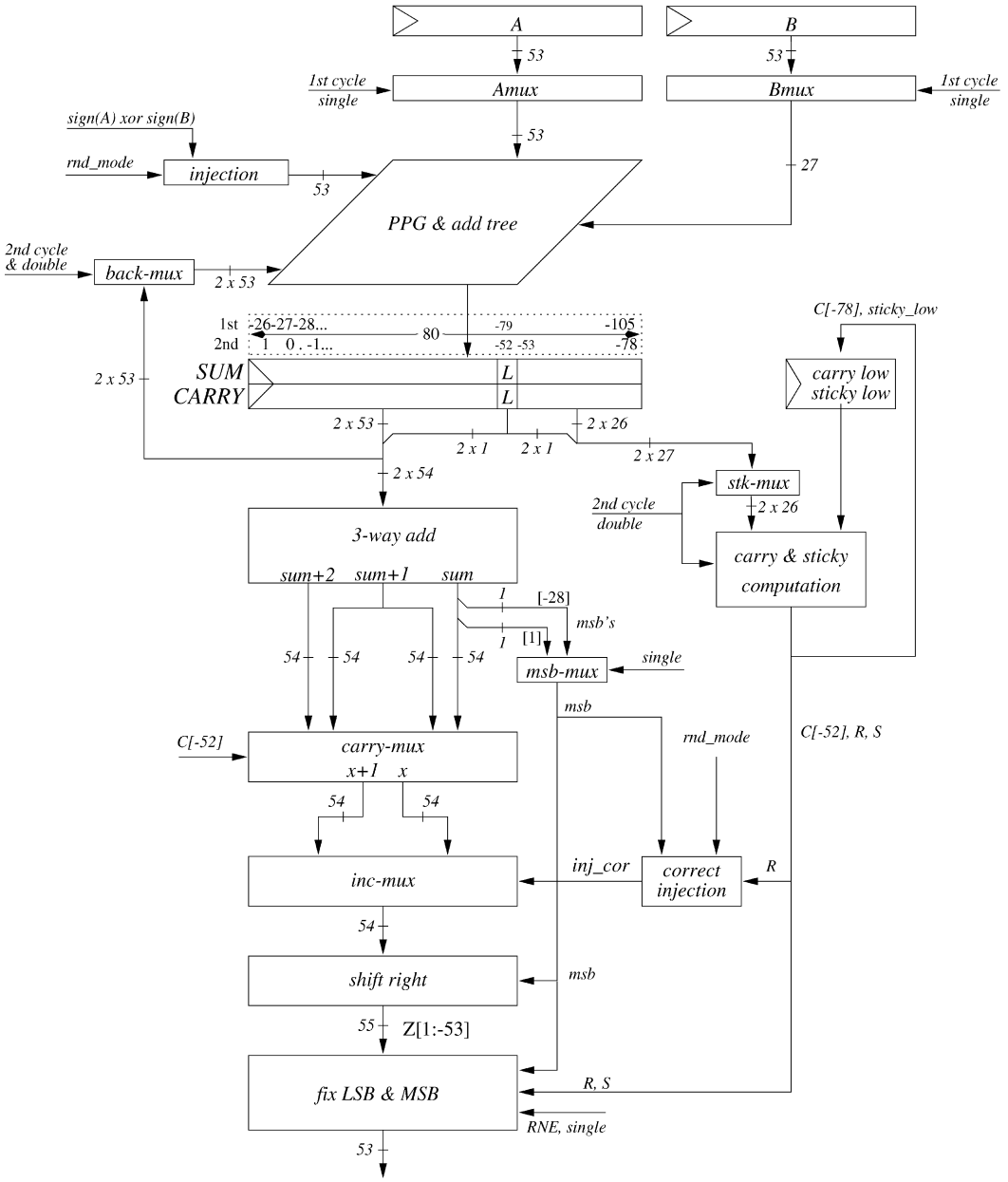


Fig. 5. Block diagram of the dual precision multiplier.

Finally, we prove that the *npd* bit for the L-bit fix in RNE-mode is computed correctly.

Claim 3. *Consider the signal npd defined by*

$$npd = \overline{RNE} \text{ OR } R \text{ OR } S \text{ OR } (msb \text{ AND } Z[-53]).$$

Then, npd equals zero iff a tie occurs in RNE.

Proof. In RNE, a tie occurs, iff the rounding operand lies at the midpoint between two representable numbers. In the non-overflow case, this is indicated by $R = 1$ and $S = 0$, in the overflow case $L = 1$, $R = 0$, and $S = 0$. However, the bits L, R, and S do not correspond to the product $a \cdot b$, but the number $a \cdot b + injection$. In the non-overflow case the injection is an increment in the R-position, and for the overflow case it is an increment in the L-position. Thus, we get the following conditions for a tie:

RNE	*ovf*	*L*	*R*	*S*
1	0	don't care	0	0
1	1	0	0	0

Consider the signal *npd′* defined by

$$npd' = \overline{RNE} \text{ OR } R \text{ OR } S \text{ OR } (ovf \text{ AND } Z[-53]).$$

The discussion above implies that $npd' = 0$ iff a tie occurs. The signal *npd* is obtained from the definition of signal *npd′* by substituting *msb* for *ovf*. As proved in Claim 1, if $msb \neq ovf$, then the RNU value equals 2. Therefore, if $msb \neq ovf$, there is no need to fix the LSB, and the claim follows. □

7. Conclusions

A new rounding algorithm, called injection-based rounding has been presented. This rounding algorithm is compared with previous rounding algorithms. Injection-based rounding is advantageous in (a) simplifying the circuitry required for rounding; and (b) supporting multiple precisions.

We demonstrate these advantages in a design of an IEEE floating-point multiplier capable of performing single- and double-precision multiplication. In single precision, the latency is two clock cycles and a new multiplication operation can be started every clock cycle. In double precision, the latency is three clock cycles and a new multiplication operation can be started only after two clock cycles. The multiplier saves hardware by using a half-sized multiplication array and by using the same rounding circuitry for both precisions.

Acknowledgements

We would like to thank Holger Leister, Stuart Oberman, and Wolfgang Paul for helpful discussions. Access to patent descriptions provided by the IBM Patent Server helped us find related patents.

References

[1] W.S. Briggs, D.W. Matula, A 17 × 69 bit multiply and add unit with redundant binary feedback and single cycle latency, Proceedings 11th Symposium on Computer Arithmetic, number 11, 1993, pp. 163–170.

[2] Chromatic's Mpact 2 boosts 3D, Microprocessor Rep. 10 (15) (1996) pp. 1, 6–10.
[3] J. Fandrianto, B.Y. Woo, VLSI floating-point processors, Proceedings Seventh Symposium on Computer Arithmetic, number 7, 1985, pp. 93–100.
[4] C. Gamez, R. Pang, Apparatus and method for rounding operands, U.S. patent 5258943, 1993.
[5] C. Hansen, MicroUnity's media processor architecture, IEEE Micro 16 (4) (1996) pp. 34–41.
[6] Hitachi SH-4 gets graphically superscalar, Microprocessor Rep. 10 (14) 1996.
[7] N. Quach, N. Takagi, M. Flynn, On fast IEEE rounding, Technical Report CSL-TR-91-459, Stanford University, January 1991.
[8] M. Saishi, T. Minemaru, Multiplication circuit having rounding function, U.S. patent 5500812, 1996.
[9] Samsung launches media processor, Microprocessor Rep. 10 (11) (1996) pp. 1, 6–9.
[10] M.R. Santoro, G. Bewick, M.A. Horowitz, Rounding algorithms for IEEE multipliers, Proceedings Ninth Symposium on Computer Arithmetic, 1989, pp. 176–183.
[11] R.K. Yu, G.B. Zyner, 167 MHz radix-4 floating point multiplier, Proceedings 12th Symposium on Computer Arithmetic, 1995, pp. 149–154.
[12] G. Zyner, Circuitry for rounding in a floating point multiplier, U.S. patent 5150319, 1992.

Guy Even received his B.Sc. degree in Mathematics and Computer Science from the Hebrew University in Jerusalem in 1988, and his M.Sc. and D.Sc. degrees in Computer Science from the Technion in Haifa in 1991 and 1994, respectively. During 1995–1997, he was a post-doctoral fellow in the Chair of Prof. Wolfgang Paul at the University of the Saarland at Saarbruecken. Since 1997, he has been a faculty member in the Electrical Engineering – Systems Department in Tel-Aviv University. His current areas of research include computer arithmetic and the design of IEEE compliant floating-point units, approximation algorithms for NP complete problems related to VLSI design, and the design of systolic arrays.

Silvia Melitta Mueller – B.Sc. in Mathematics and Computer Science (1988), M.Sc. in Mathematics (1989), D.Sc. in Computer Science (1991), and Habilitation in Computer Science (1998) from the University of Saarland at Saarbruecken. During 1992–1993, a post-doctoral fellow in the International Computer Science Institute in Berkeley. During 10/97 and 08/98, visiting scientist in the Chair of Prof. Charles Leiserson at MIT, Laboratory for Computer Science. During 1998–1999, a faculty member in the Computer Science Department in University of Saarland. Since 09/99, engineer in the S/390 processor development department in IBM Germany, Laboratory Boeblingen. Current areas of research include design of advanced multi-processor systems, microprocessors, and floating-point units, and formal hardware verification.

Peter-Michael Seidel studied Computer Science and Electrical Engineering at the University of Hagen, Germany, from which he graduated in 1996. He completed his PhD in computer science in 1999 at the chair of Prof. Wolfgang Paul at the University of the Saarland, Germany, where he was supported by a fellowship of the German National Science Foundation (DFG). Currently, he is an assistant professor in the Computer Science and Engineering Department at Southern Methodist University in Dallas, Texas. His current areas of research include: computer arithmetic, computer architecture, and the design of IEEE compliant floating-point units.