Floating-Point Multiply-Add-Fused with Reduced Latency

Tomás Lang, *Member*, *IEEE Computer Society*, and Javier D. Bruguera, *Member*, *IEEE*

Abstract—We propose an architecture for the computation of the double-precision floating-point multiply-add-fused (MAF) operation $A + (B \times C)$. This architecture is based on the combined addition and rounding (using a dual adder) and in the anticipation of the normalization step before the addition. Because the normalization is performed before the addition, it is not possible to overlap the leading-zero-anticipator with the adder. Consequently, to avoid the increase in delay, we modify the design of the LZA so that the leading bits of its output are produced first and can be used to begin the normalization. Moreover, parts of the addition are also anticipated. We have estimated the delay of the resulting architecture considering the load introduced by long connections, and we estimate a delay reduction of between 15 percent and 20 percent, with respect to previous implementations.

Index Terms—Computer arithmetic, floating-point functional units, multiply-add-fused (MAF) operation, VLSI design.

1 Introduction

The multiply-add operation is fundamental in many scientific and engineering applications. Fixed-point implementations have been included in the instruction set of DSP processors. In this case, the delay is reduced with respect to separate multiplication and addition because the addition is merged into the adder array of the multiplication.

The concurrent processing of the steps involved in the floating-point multiply-add is much more complex than its fixed-point counterpart. However, recently, the floating-point unit of several commercial processors includes as a key feature a unified floating-point multiply-add-fused (MAF) unit [6], [7], [11], [15], which executes the double-precision multiply-add, $A + (B \times C)$, as a single instruction, with no intermediate rounding. The standard operations floating-point add and floating-point multiply can be performed using this unit by making $B = 1$ (or $C = 1$) for addition and $A = 0$ for multiplication.

The implementation of multiply-add-fused has two advantages over the case of separate floating-point adder and multiplier: 1) the operation $A + (B \times C)$ is performed with only one rounding instead of two, which permits reducing the overall delay and error of the MAF operation, and 2) there is reduction in the hardware required since several components are shared for addition and multiplication [3], [7]. On the other hand, if only one MAF unit is provided, it is not possible to concurrently perform additions and multiplications, which is possible for the case in which there is one floating-point adder and one floating-point multiplier.

The implementations of [3], [7] do not combine addition and rounding, as is done in modern implementations of floating-point addition[1] and multiplication[2] [10], [12]. In this paper, we propose a new MAF architecture based on the combination of the addition and rounding. As a consequence, the delay of the operation is reduced with respect to that of the traditional MAF implementations.

To make the combination of addition and rounding possible, we anticipate the normalization before the addition. This is different than what is done when using the combined addition/rounding approach for addition, where the normalization of the result is performed after the add/round; this is possible because no rounding is required when massive normalization occurs. On the other hand, this is not possible for the MAF operation because the rounding position is not known until the normalization has been performed. Then, we propose performing the normalization before the addition. This requires overlapping the operation of the leading-zeros-anticipator (LZA) with the operation of the normalization shifter. However, despite the overlap, the operation of the normalization shifter cannot begin at the same time as the LZA; then, there is a *time gap* where the normalization shifter is idle, waiting for the first digit of the normalization amount. We propose to fill this gap by anticipating part of the addition before the normalization. To show the practicality of these modifications, we present, in detail, the design of the resulting unit.

We estimate the delay of the proposed architecture using a family of standard cells. In the estimation, we incorporate the effect of the load and of the delay of the connections. We compare this delay with an estimate for previous implementations and we estimate that, for a double-precision format, the proposed architecture results in a delay

- *T. Lang is with the Department of Electrical Engineering and Computer Science, University of California at Irvine, Irvine, CA 92697. E-mail: tlang@uci.edu.*
- *J.D. Bruguera is with the Department of Electronic and Computer Engineering, University of Santiago de Compostela, 15782 Santiago de Compostela, Spain. E-mail: bruguera@dec.usc.es.*

Manuscript received 10 Feb. 2003; revised 30 Oct. 2003; accepted 8 Mar. 2004.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 118297.

1. The rounding step is merged with the addition using a dual adder that computes, simultaneously, the *sum* and *sum+1*. Depending on the rounding bits and the most-significant bits of *sum*, the output *sum* or *sum+1* is selected as the rounded result and then it is normalized.

2. In this case, the addition corresponds to the conversion from carry-save to nonredundant representation.

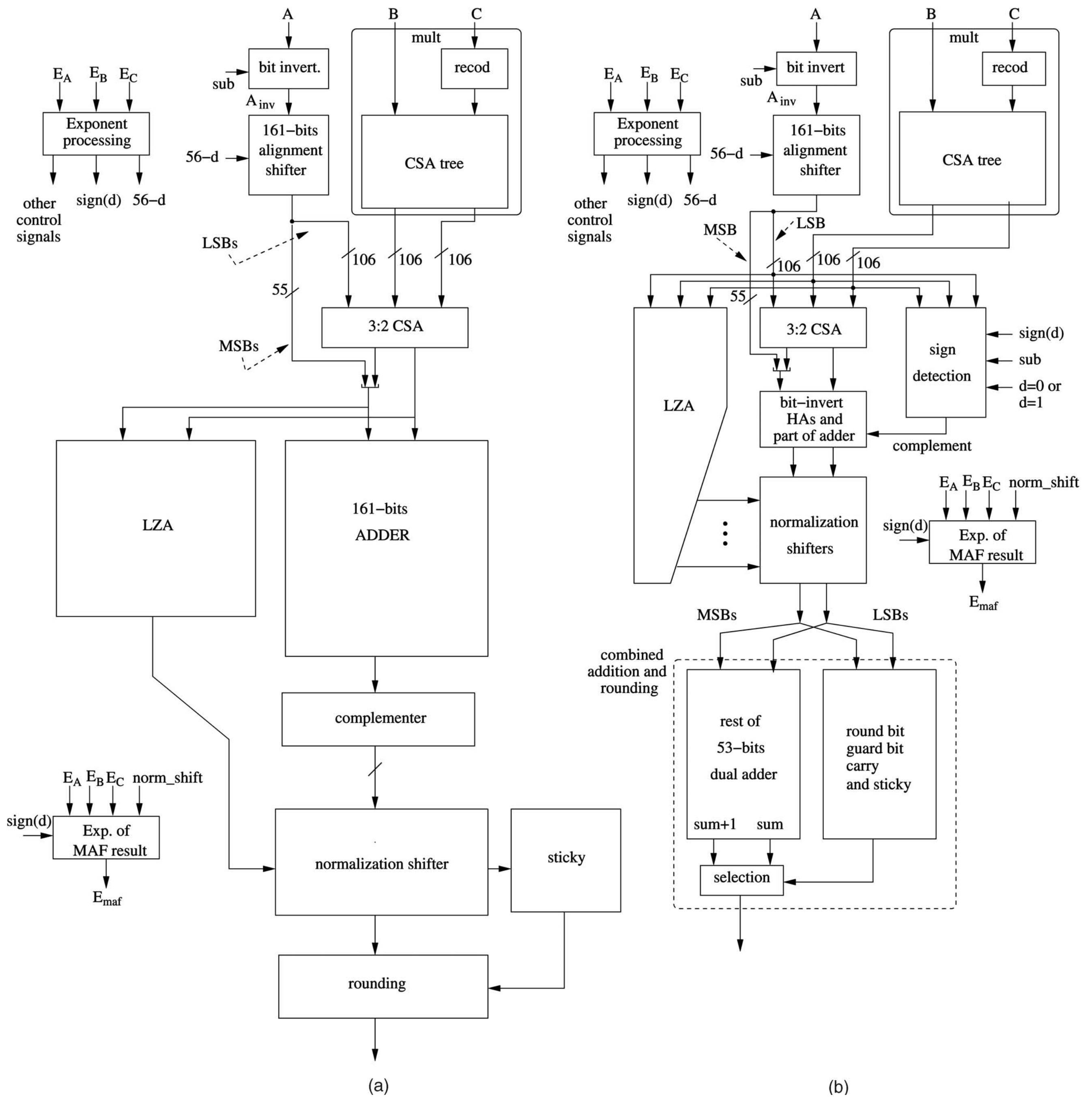

Fig. 1. Schemes for the MAF operation. (a) Basic. (b) Proposed.

reduction of between 15 percent and 20 percent with respect to basic MAF implementations.

2 Basic Scheme of the MAF Unit

The implementations of the MAF operation are for the IEEE double-precision floating-point format. In this format, a floating-point number M is represented using a sign bit s_m, an 11-bit biased exponent e_m, and a 53-bit significand X. If M is a normalized number, it represents the value $M = (-1)^{s_m} \times (1 + f_m) \times 2^{e_m - 1023}$, where $X = (1 + f_m)$, $1 \leq X < 2$ and f_m is the fractional part of the normalized significand (the 52 stored bits). The MAF architecture proposed before [8], implemented in several floating-point units of general-purpose processors [7], [8], [11], is shown in Fig. 1a. The steps in this implementation are:

1. Multiplication and alignment shift:
   - Multiplication of B and C to produce an intermediate carry-save product.
   - To reduce the latency, the bit inversion and alignment of the significand of A is done in parallel with the multiplication. The bit inversion provides the one's complement of A for an effective subtraction.[3]

3. We call A_{inv} the output of the bit inverter, regardless of whether the bit inversion is carried out ($sub = 1$) or not ($sub = 0$).

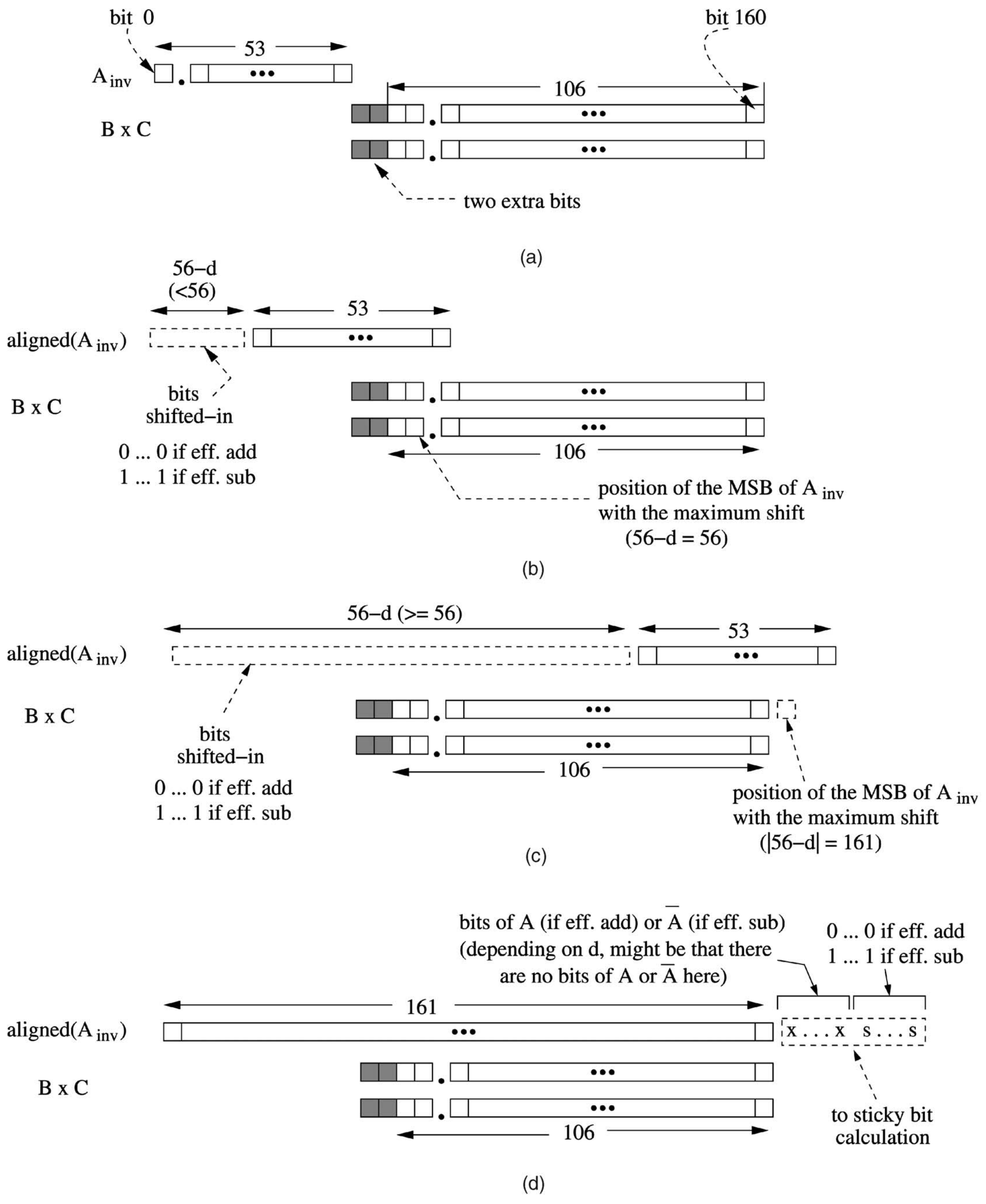

Fig. 2. Alignment of A_{inv}. (a) Before alignment. (b) Alignment with $\mathrm{d} \geq 0$. (c) Alignment with $\mathrm{d} < 0$. (d) After alignment.

The alignment is implemented as a right shift by placing A_{inv} to the left of the most significant bit of $B \times C$. This is shown in Fig. 2a. Two extra bits are placed between A_{inv} and $B \times C$ to allow correct rounding when A_{inv} is not shifted. Then, the right shift amount for the alignment depends on the value of $d = exp(A) - (exp(B) + exp(C))$, where $exp(A)$, $exp(B)$, and $exp(C)$ are the exponents of the A, B, and C operands, respectively. Two situations occur:

a. $d \geq 0$. In a conventional alignment, $B \times C$ would have to be aligned with a right shift of d bits. The maximum alignment shift would be 56 bits since, for $d \geq 56$, $B \times C$ is placed to the right of the least-significant bit of A_{inv}; in this case, $B \times C$ affects only the calculation of the sticky bit. Instead, with this implementation, A_{inv} is right shifted $56 - d$ bits; then, the shift amount is $shift\ amount = max\{0, 56 - d\}$ (see Fig. 2b).

b. $d < 0$. In a conventional alignment A_{inv} would have to be aligned with a right shift of d bits. In this case, the maximum alignment shift would be 105 bits since, for shift amounts larger than 105, $d < -105$, the operand A_{inv} is placed to the right of the least-significant bit of $B \times C$, affecting only the calculation of the sticky bit. Instead, A_{inv} is right shifted $56 - d$ bits (see Fig. 2c), then $shift\ amount = min\{161, 56 - d\}$.

Combining both cases, the shift amount is in the range $[0, 161]$, requiring a 161-bit right shifter. Moreover, the shift amount is computed as $shift\ amount = 56 - d$.

Moreover, as shown in Fig. 2d, the bits that are shifted out at right are used to compute $st1$, part of the sticky bit. The *partial* sticky bit $st1$ is used later to compute the final sticky bit. This $st1$ has value 0 when all the bits shifted out of the 161 bits are 0. However, in subtraction, we invert the bits of A before shifting. Consequently, in that case, $st1 = 0$ if all the bits shifted out are 1. Since we also shift out bits that are at the right of the least-significant bit of A, before shifting we extend A_{inv} with 0s for addition and with 1s for subtraction.[4]

2. The aligned A_{inv} is added to the carry and sum words of the product. Only the 106 least-significant bits of the aligned A_{inv} are needed as input to the 3:2 CSA, because the product has only 106 bits. The 55 most-significant bits of the aligned A_{inv} are concatenated at the output of the CSA to obtain the 161-bit sum word.
3. The next step is the addition of the carry and sum words and the determination of the shift amount for normalization.
   - The carry and sum words, obtained at the output of the CSA, are added in a 161-bit one's complement adder (with end around carry adjustment for effective subtraction). As the result can be negative, a complementer is required at the output of the adder.
   - In parallel with the significands addition, the normalization shift is determined. The LZA (Leading Zero Anticipator) produces the amount of the shift directly from the operands.
4. Once the result of the addition is obtained, the normalization shift can be performed since the shift amount has been already determined. A normalization shift is required to place the most-significant bit of the result at bit 0; as a consequence, normalization is performed to compensate for the cancellation produced in subtraction as well as to compensate for the way the alignment is performed.

   Note that, because of the two zeros included in bits 53 and 54, there is no overflow to the left of bit 0. Consequently, no right shift for normalization is required.
5. The last step is the rounding of the result.

With this scheme, the delay of the MAF operation is determined by the sum of the delays of the following components: multiplier, 3-2 CSA, 161-bit adder plus complementer, normalization, and rounding. On the other hand, the main hardware components are: multiplier, alignment shifter, 3:2 CSA, LZA, 161-bit adder, normalization shifter, and rounder.

3 Proposed MAF: General Structure

We now describe the proposed MAF architecture. Since the unit is quite complex, we present this description in two steps: In this section, we give an overview of the scheme, with just enough detail to make it understandable and believable, and leave to Section 4 the details of some of the modules. In this section, we utilize Fig. 1b to illustrate the description, whereas, in Section 4, we use the more detailed Fig. 3. The reader can postpone the understanding of the details until reading the next section or can go to the corresponding subsections while reading this section.

The objective of the proposed MAF architecture is to reduce the overall delay and, potentially, to reduce the number of cycles in a pipelined implementation. Since, in floating-point addition and multiplication, one of the approaches to reduce latency has been to combine addition with rounding [5], [10], [12], [13], we follow the same approach. For this approach, in floating-point addition and multiplication, the order of normalization and rounding is interchanged. This seems impractical to do for MAF because, before the normalization, the rounding position is not known.[5] The solution we explore is to perform the normalization before the addition. As in the basic implementation of Section 2, an LZA is used to determine the normalization amount. However, in this case, the LZA is not overlapped with the adder, so we explore the possibility of masking its delay by 1) overlapping part of the LZA with the normalization shifter and 2) anticipating part of the adder.

The resulting scheme is shown in Fig. 1b. This implementation shares several characteristics with the basic implementation described in Section 2 (Fig. 1a):

- The alignment is performed by shifting the addend A. This is done in parallel with the multiplication. Moreover, also in parallel with the multiplication, a bit-inversion of A is performed for subtraction and the partial sticky bit $st1$ is calculated.
- The result of multiplication is in carry-save representation. The three vectors are added in a 106-bit 3:2 CSA to obtain a redundant representation of the unnormalized and unrounded result of the MAF operation.

On the other hand, the proposal is different in the following aspects:

- Two's complement of A (details in Section 4.1). We use a two's complement representation for A to avoid the end around carry adjustment needed when the subtraction is carried out in a one's complement adder. This allows us to simplify the design of the combined addition and rounding. The

4. The difference between addition and subtraction appears because the bits are inverted instead of performing here the complete 2's complement operation. The calculation of the partial sticky bit is not included in the description of the basic MAF architecture in [8]; we have developed it to give a complete description of the modules also present in our proposal.

5. Although this is also true for addition, in that case, the bits introduced by the left shift normalization are all zeros, so no roundup is required when massive normalization is performed.

additional 1, needed to complete the two's complement, is added using empty slots in the 3:2 CSA.

- Normalization before the add/round (details in Section 4.3). Moreover, the LZA overlaps with the normalization shifter in such a way that the shift amount is obtained starting from the most-significant-bit and, once the first bit (MSB) is obtained, the normalization shift can start.
- Add/round (details in Section 4.4). To obtain the rounded result, the sum and $sum+1$ of the two resulting vectors of the normalization are required. This is a problem similar to the rounding of floating-point multipliers, where carry-save representation of the exact product computed in the addition tree has to be rounded [5], [12], [16], but taking into account that, since the normalization has been performed before the addition, the result is always normalized.

  Therefore, the add/round module consists of a dual adder of 53 bits; this reduced adder width is possible because the normalization is performed before. Then, the inputs to the add/round module are split into two parts: The 53 most-significant bits are input to the dual adder; the remaining least-significant bits are inputs to the logic for the calculation of the carry into the most-significant part and for the calculation of the rounding and sticky bits. This information is used to select the correct rounded output in the dual adder. As will be explained later, an additional row of HA is required before the dual adder to perform the rounding.
- Sign detection. A more complicated selection is required if the adder output can be negative. One way to avoid this (as shown in Fig. 1b) is to detect the sign of the adder output and complement the outputs of the CSA when the result is negative. The result can be negative only for effective subtraction. Moreover, since, in effective subtraction, we always complement the significand of A, the result can be negative only for $d \geq 0$. In this case, two situations have to be considered: 1) The result is always negative for $d \geq 2$. 2) For $d = 0$ or $d = 1$, a full comparison between the significand of A and of $B \times C$ has to be performed. Although this comparison is slow (a tree comparator), the implementation discussed in more detail in the next section shows how this delay is overlapped with parts of the dual adder.

  The two's complement of the CSA output is performed by inverting the sum and carry words and adding two in the least-significant position. This addition of two is explained in Section 4.2.
- Advance part of the adder before the inversion/normalization (details in Section 4.2). This advance is done to reduce the critical path due to the part of the LZA that cannot be overlapped with the normalization and to the sign detection.

After the alignment and multiplication, several actions can be carried out in parallel: calculation of the MSB of the shift amount in the LZA, sign detection of the adder output, and 3:2 CSA plus HAs. Among these three parallel paths, it seems reasonable that the larger delay corresponds to the LZA and that we have enough time to perform some parts of the dual addition (generation of p_i and g_i signals, at least) in parallel with the operation of the LZA.

As discussed in Section 5, the delay of the MAF unit is determined by the sum of the delay of the following components: multiplier, LZA, and the part of the 53-bit dual adder that has not been placed in parallel with the LZA. Comparing this delay with the delay of the basic architecture (see Section 2), we can expect that the total delay is reduced and, therefore, the latency of a pipelined implementation should be also reduced.

With respect to the hardware complexity, the main building blocks of the proposed architecture are: multiplier, alignment shifter, 3:2 CSA, sign detection, LZA, two normalization shifters, 53-bit dual adder, and carry calculation logic. Comparing with the basic architecture (see Section 2), we can conclude that the hardware requirements are quite similar in both architectures or slightly bigger in the proposed one.

4 Detailed Description of Some Modules of the Architecture

Fig. 3 shows the block diagram of the MAF unit. This figure is similar to Fig. 1b, but with a larger level of detail. In this section, each of the modules marked with dotted lines in the figure are explained with a larger level of detail.

4.1 Two's Complement of A

In case of effective subtraction, an additional 1 has to be added to the least-significant bit of $\overline{A}$, to complete the 2's complement of A. However, after the alignment, this bit can be anywhere between bit positions 52 and 213 (see Fig. 2). We want to add a 1 to a fixed position and let this propagate in a correct manner. Since both the truncated A_{inv} and $B \times C$ span up to bit 160, we add 2 at position 161. To see how this addition propagates correctly, consider the following two situations:

1. The least-significant bit of A_{inv} is to the left of bit 161 (see Fig. 4a). In that case, the 2 added in bit 161 produces a carry that propagates through the chain of 1's[6] to the least-significant bit of $\overline{A}$. Note that, in this case, $st1 = 0$.
2. The least-significant bit of A_{inv} is to the right of bit 160 (see Fig. 4b). In that case, the 1 added to the least-significant bit of $\overline{A}$ propagates to bit 160 only if all the bits of $\overline{A}$ shifted out are 1. This corresponds to the case $st1 = 0$. So, we add 2 in position 161 if $st1 = 0$.

Combining both cases, we add 2 in position 161 if $sub = 1$ and $st1 = 0$.

Then, the aligned A_{inv} (plus these two bits in position 161) is added to the carry and sum words of the product. As shown in Fig. 4c, since the product spans only up to bit 160, the sub and $\overline{st1}$ bits can be included before the 3:2 CSA.

6. This chain of 1s was included to correctly compute the partial sticky bit $st1$ (see Section 2).

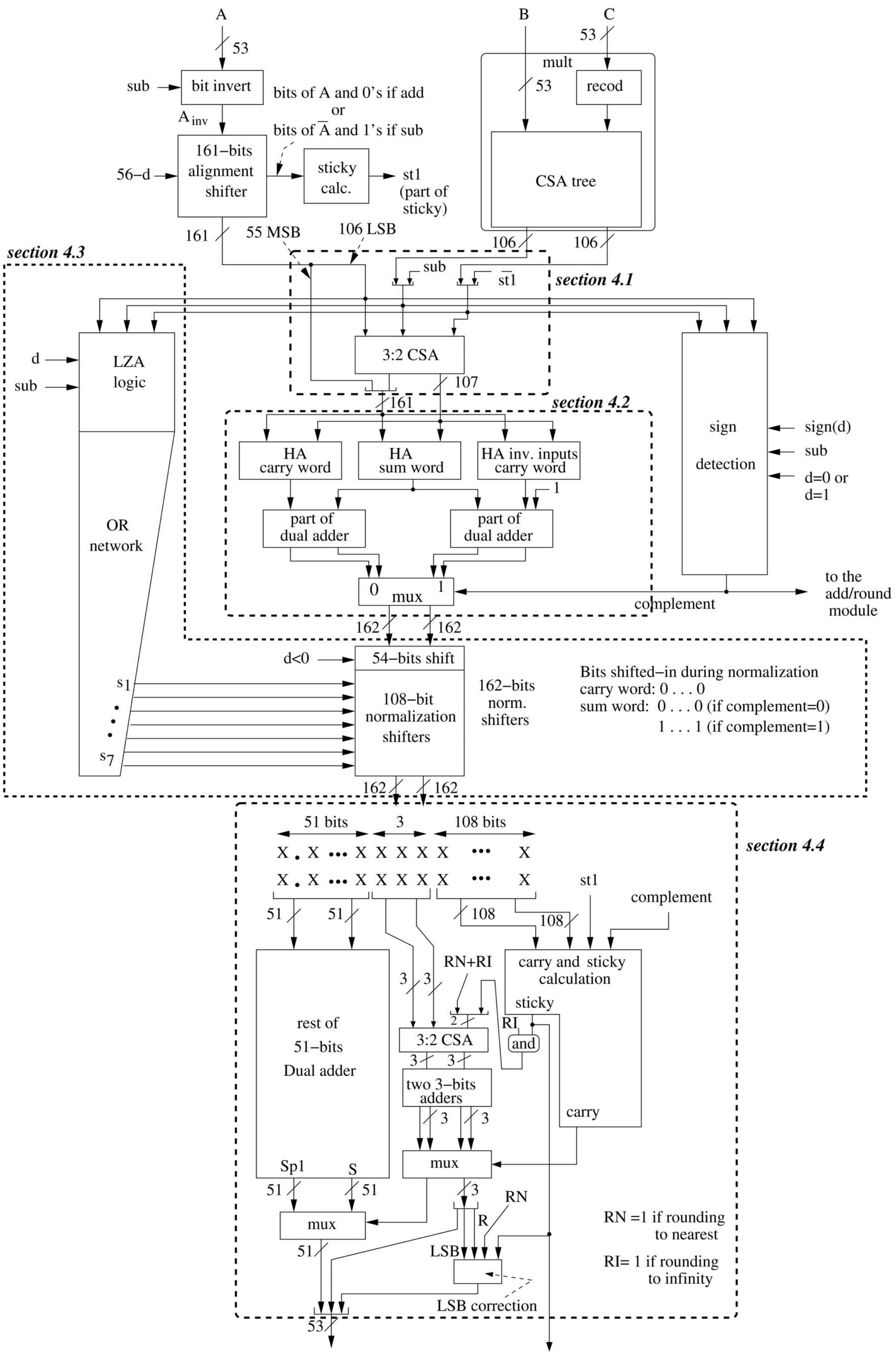

Fig. 3. Detailed block diagram of the proposed MAF.

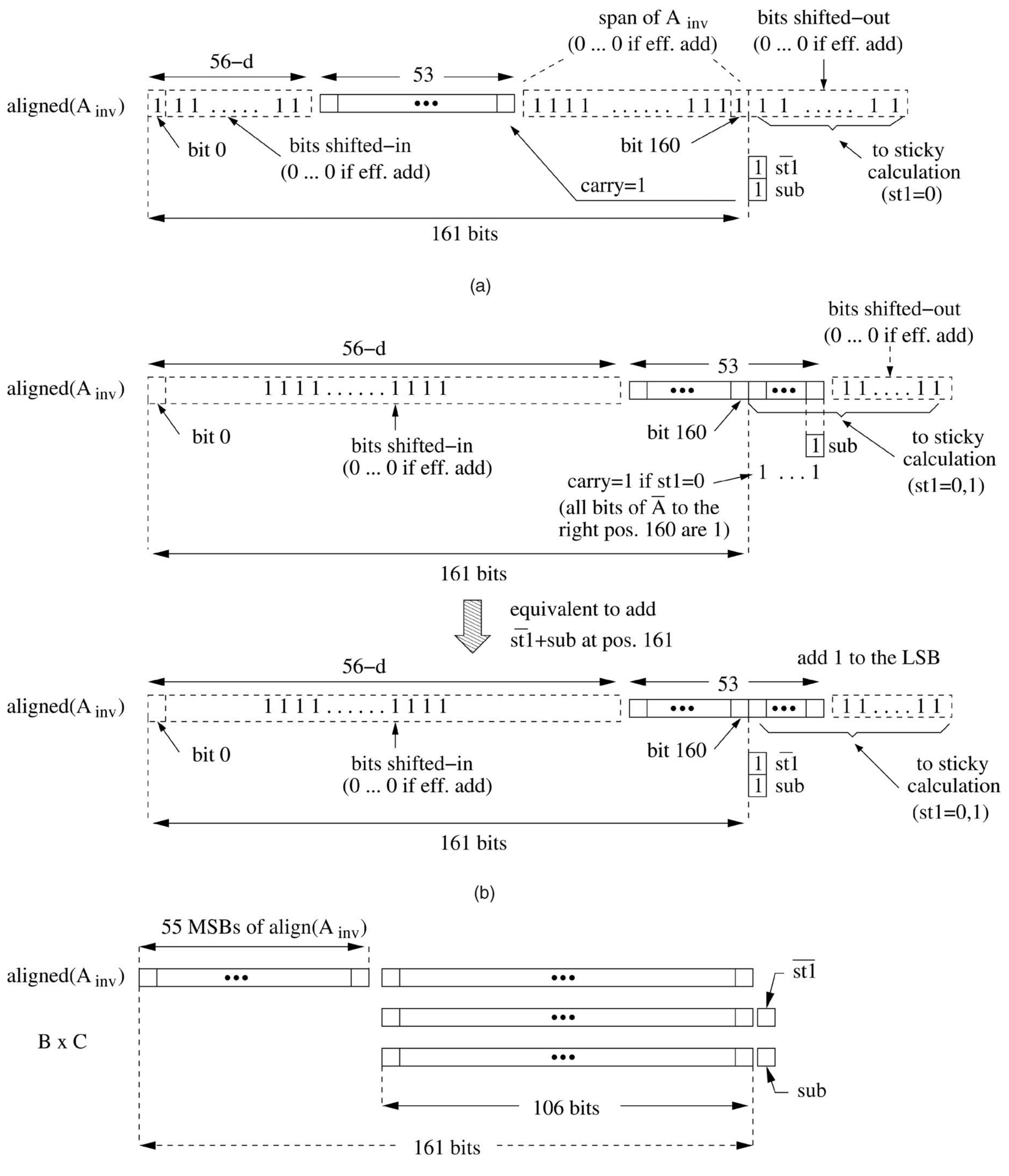

Fig. 4. Completing the 2's complement of A in effective subtraction. (a) Least-significant bit of $\mathrm{A_{inv}}$ to the left of position 161. (b) Least-significant bit of $\mathrm{A_{inv}}$ to the right of position 160. (c) Before the 3:2 CSA.

4.2 Anticipated Conditional HAs and Part of Dual Adder

As explained in Section 4.4, the HAs are required to assure a correct rounding at the output of the dual adder. In this section, we give some details about its implementation and outline the splitting of the dual adder.

- To avoid additional delays in the bit-inversion of the carry and sum words at the output of the CSA (the inversion is controlled by the the sign of adder output), it is carried out conditionally (see Fig. 3). The HAs are duplicated, with and without inverted inputs, and the correct output is selected according to the sign of the adder output ($complement = 1$ if negative). However, the sum word is the same both without inverted inputs and with inverted inputs; then, it is only necessary to replicate the calculation of the carry word.

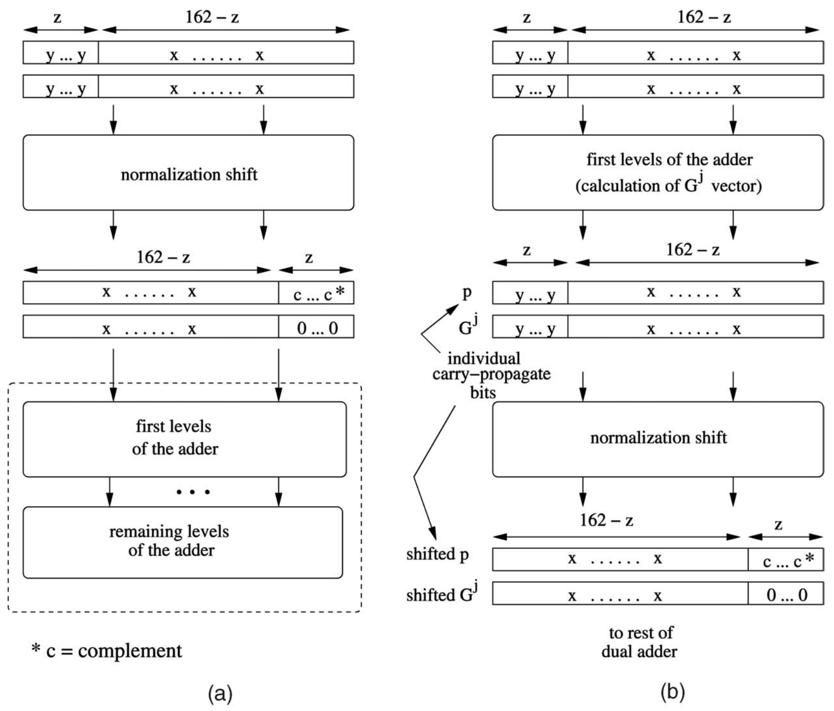
z
162 – z
y ... y
x x
normalization shift
c ... c*
0 ... 0
first levels of the adder
remaining levels of the adder
* c = complement
(a)
first levels of the adder (calculation of G^j vector)
p
G^j
individual carry–propagate bits
shifted p
shifted G^j
to rest of dual adder
(b)

Fig. 5. Anticipation of part of the adder. (a) Standard adder after normalization. (b) Generation and shifting of G^j and p.

Because of the anticipation of the HAs before the normalization, the position of the most-significant bit of the result is not known when the HA operate; therefore, it is necessary to include HAs for the 162 bits, instead of for the 54 most-significant bits, as done in [5], [12], [16] for multiplication.

- The two 1s needed to complete the two's complement of the inputs of the Half-Adders have to be added in the least-significant position of these inputs. The first 1 is incorporated as the least-significant bit of the carry word (see Fig. 3) and the second 1 is added in the add/round module ($complement = 1$), that is, after the normalization shift. However, to add this second 1 in the right position, ones have to be shifted in during the normalization (see Section 4.3).
- As said before (Section 3), the delay of the calculation of the *complement* signal is larger than the delay of 3:2 CSA and HAs. Then, to avoid additional delays, some part of the dual adder can be performed before the selection of the inverted or noninverted HAs outputs; moreover, the *anticipated* part of the dual adder also has to be replicated, considering as input the inverted and noninverted HA outputs.
- What part of the dual adder can be placed before the selection depends on the difference between the delay of the sign detection (*complement* signal) and that of the 3:2 CSA plus HA. It seems reasonable that, at least, the calculation of the bit carry-generate g_i, carry-propagate p_i and carry-kill $\bar{k}_i$ signals can be anticipated. However, some more logic could be placed before the selection; for example, the calculation of carry generated and carry kill signals of groups of 2 or 4 bits.

  On the other hand, if the delay of the computation of $s1$ is larger than the sign detection plus the multiplexer, part of the anticipated adder can be placed after the multiplexer; however, we do not consider this partitioning.
- The anticipation of the first levels of the dual adder implies some changes in its design due to the effect of the shift over the already calculated carry-generate and carry-propagate (or carry-kill) signals, G and P (or $\bar{K}$).

  Fig. 5 shows a closer look into the operands before and after the shift for a standard implementation, when the adder is placed after the normalization (Fig. 5a), and for the proposed implementation, with anticipation (Fig. 5b). In both cases, we consider that there are z most-significant zeros and, therefore, the shift amount is z. In the implementation without anticipation, after the normalization, the inputs to the dual adder are in the most-significant part of the shifter outputs. On the other hand, when some levels are moved before the shifter, the bits that input the remaining levels of the dual adder are not known until the shift has been carried out. As a consequence, the following vectors should be shifted: $P^j = (P_0^j, \ldots, P_{161}^j)$ (carry-propagate) and $G^j = (G_0^j, \ldots, G_{161}^j)$ (carry-generate), P_i^j and G_i^j being the carry propagate and carry generate of group i after level $j, 1 \leq j \leq log_2 n$ of the prefix tree[7] (group of bits from i until $i + 2^j - 1$ of the prefix tree). That is,

7. We consider that level 0 is the calculation of p_i and g_i.

$$P_i^j = p_i p_{i+1} \cdots p_{i+(2^j-1)}$$
$$G_i^j = g_i + g_{i+1} p_i + \ldots + g_{i+(2^j-1)} p_i \cdots p_{i+(2^j-2)}.$$

On the other hand, the $p_i = a_i \oplus b_i$ are used in the calculation of the result of the dual adder; therefore, vector $p = (p_0, \ldots, p_{161})$ should be also shifted. To implement the shift, there are two alternatives:

1. To have three shifters, one for each of the vectors P_i^j, G_i^j, and p. This solution, although it does not introduce any additional delay, is expensive in terms of hardware.
2. Instead of passing vectors P^j, G^j, and p, we propose to pass through the shifter only the carry-generate bits G^j and the individual carry-propagate bits p. Then, we do not pass the P^j vector. As this vector is needed to compute the G^j, it has to be recalculated after the shifter using the p_i. Note that the anticipated vectors are also used for the carry and sticky calculation (see Section 4.4.1).

4.3 Normalization Step

The main issue in the normalization step is to overlap the calculation of the shift amount (operation of the LZA) with the normalization shift itself (operation of the normalization shifter). The 162-bit normalization shifter can be implemented as a 54-bits shift followed by a 108 bit-shifter [3] in such a way that 1) the most-significant bit of the shift, the control of the 54-bit shift, can be obtained from the exponent difference and 2) the LZA computes the control for the 108-bit shift. This separation permits an additional overlap between the operation of the LZA and the normalization shifter. The separation is possible because, when the exponent difference is negative ($d < 0$), operand A_{inv} has to be shifted right past the boundary with $B \times C$ so that the shift amount is greater than 57. Under these conditions, the 54 most-significant bits of the result are zero, both in effective addition and effective subtraction. This means that, in this case, the normalization shift is, at least, of 54 bits.

The LZA, to obtain the shift amount of the 108-bit normalization shift, is split into two parts [14]: calculation of a string of bits having the same number of leading zeros as the sum (LZA logic) and encoding the number of leading zeros (LZD). In a conventional LZA, the shift amount is known only after the encoding. In our case, we need to know the bits of the encoding of the shift as soon as possible. For this end, we replace the LZD by a set of wide OR gates (OR network in Fig. 3) and muxes.

To explain how the encoding of the shift amount is carried out, in Fig. 6, we show the implementation of the OR network for a normalization shift of 32 bits. The basic idea is to explore different groups of bits of the output of the LZA logic to determine the bits of the shift amount encoding [1]. Fig. 6a indicates which groups are explored to compute each bit. Thus, to determine the most-significant bit s_1 of the shift amount, the 16 most-significant bits of the output of the LZA logic are checked. In this way, if all the bits are zero, $s_1 = 1$, then the normalization shift includes a 16-bit shift; otherwise, $s_1 = 0$, this 16-bit shift is not necessary. To determine the remaining bits of the shift amount, as shown in the figure, different groups of bits have to be checked depending of the previous bits already calculated.

Fig. 6b shows an implementation of this algorithm. NOR gates are used to determine if there is some 1 in different groups of the string. The groups are selected by the set of multiplexers with an increasing number of inputs. Note that muxes of more than two inputs are organized as a tree, in such a way that bit s_i is used as the control signal of a 2:1 mux to obtain s_{i+1}. In this way, the shift is calculated starting from the most-significant bit. After an initial delay due to the calculation of bit s_1, each bit is obtained with the delay corresponding to a 2-input mux. The generalization to any other number of bits is straightforward, as shown in Fig. 6c for a 108-bit shift encoding.

As said before, the second 1 to complete the 2's complement of the output of the 3:2 CSA is added in the add/round module. But, as this addition is performed after the normalization, the position where the 1 should be added, the least-significant position before normalization, is unknown now. Then, in effective subtraction, bits shifted-in during normalization are $1 \ldots 1$ in the sum word and $0 \ldots 0$ in the carry word (Fig. 3); this way, the 1 is added in the least-significant position in the add/round module and it is propagated until the least-significant position before normalization.

4.4 Add/Round Module

The add/round module implements the round to nearest/even[8] (RN) and round to ∞ (RI) round modes. The round to 0 mode is not considered since this is simply a truncation of the result. Moreover, the correct round to the nearest/even result is obtained from the round to nearest/up changing the LSB [12]: Round to nearest/up produces the same result as round to nearest/even except when a tie occurs; then, the sticky bit identifies the tie case and the correct rounded to nearest/even result is obtained by forcing the LSB to 0.

The add/round module for the proposed MAF differs from the corresponding modules for floating-point addition [10], [13] and for floating-point multiplication [5], [12], [16]. The difference with respect to floating-point addition is that now there can be a carry propagating from the 108 least-significant bits to the 53 more-significant bits (see Fig. 3). This carry has to be added with the rounding bits to obtain the 53-bit rounded result. On the other hand, in floating-point addition, as both operands are 53-bit wide and one of them is aligned, there is no carry from the least-significant part, which correspond to the positions that are right shifted out in the alignment. Moreover, in the MAF case, the result is normalized, whereas, in floating-point addition, there can be an overflow or the result can be unnormalized by one position and still need rounding.

With respect to the carry, the MAF module is similar to that for floating-point multiplication. As a consequence, we adapt one of the schemes proposed for this operation.

From the three schemes that were proposed, we have determined that the one described in [16] can be easily

8. In the rest of the paper, we refer to the round to nearest/even as round to nearest.

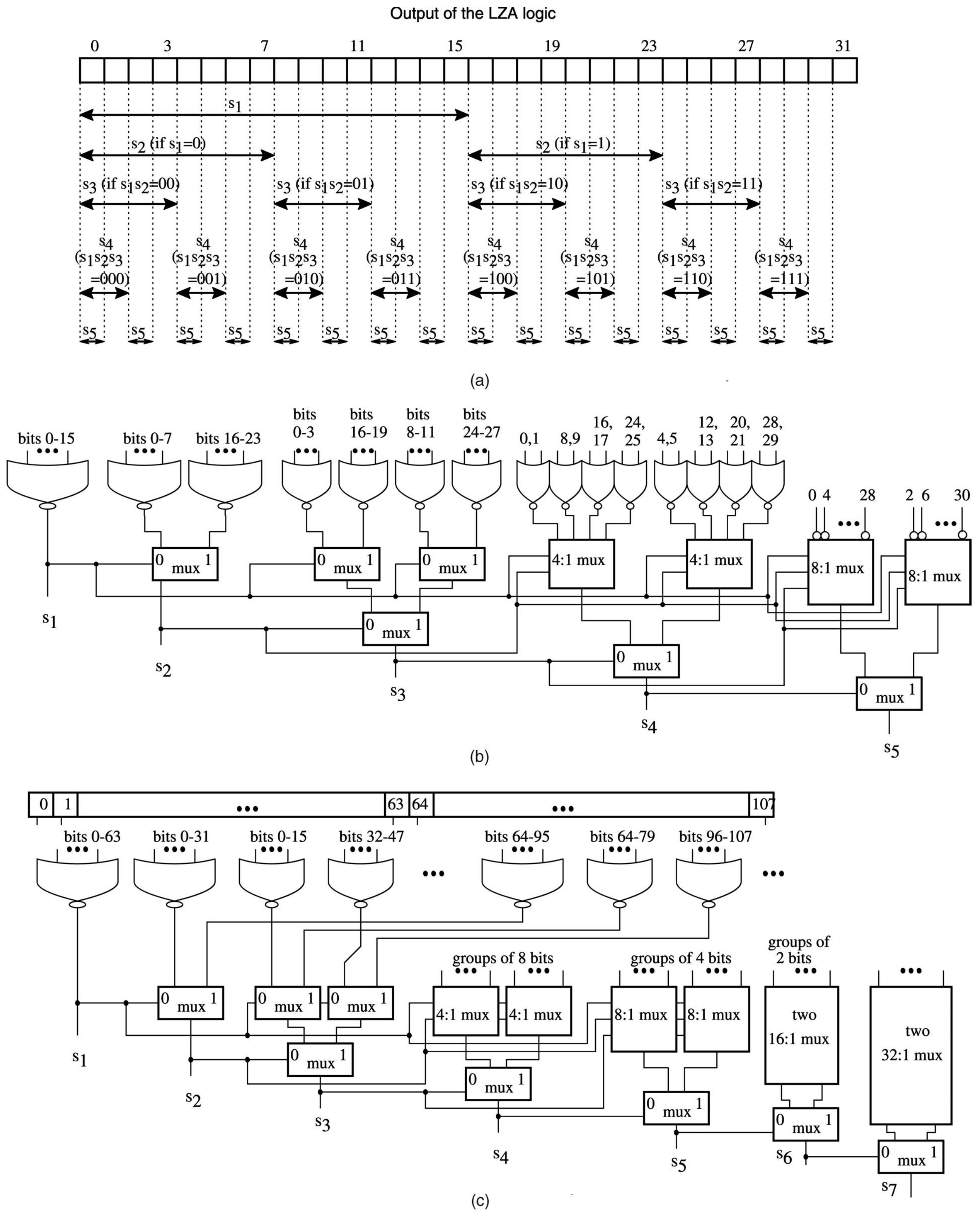

Fig. 6. Encoding of the normalization shift amount. (a) Bits checked to determine the shift amount, 32-bit example. (b) Encoding of the shift amount, 32-bit example. (c) OR network for the encoding of the normalization shift.

adapted to the MAF. On the other hand, the scheme described in [12] is hard to adapt because it requires that the half adders reduce the number of 1s to be added in position 53, but this cannot achieved in the proposed MAF because the half adders are included before the normalization. Finally, the rounding scheme proposed in [5] uses an injection that depends only on the rounding mode in order to reduce rounding to infinity and rounding to nearest to rounding to zero. Since this injection is added at a fixed position of the carry-save representation of the

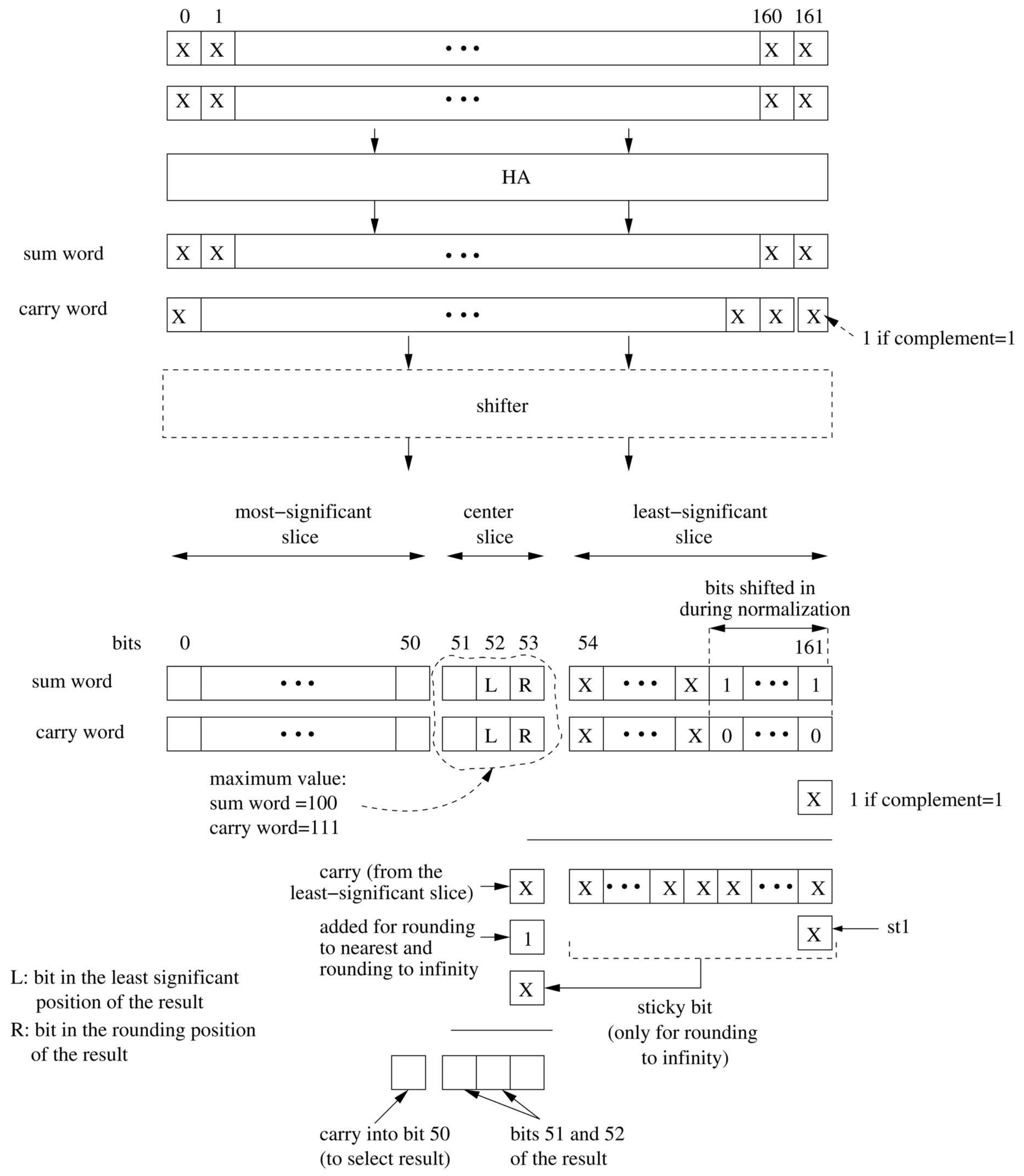
0
1
160
161
HA
sum word
carry word
1 if complement=1
shifter
most−significant slice
center slice
least−significant slice
bits shifted in during normalization
bits
0
50
51 52 53
54
161
L
R
maximum value: sum word =100 carry word=111
1 if complement=1
carry (from the least−significant slice)
added for rounding to nearest and rounding to infinity
st1
sticky bit (only for rounding to infinity)
L: bit in the least significant position of the result
R: bit in the rounding position of the result
carry into bit 50 (to select result)
bits 51 and 52 of the result

Fig. 7. Carry propagation from the lower part and dual adder output selection (for clarity, this figure illustrates the splitting of the dual adder inputs when no part of the adder has been anticipated before the shifter).

product, in the case of the MAF module it would have to be added after the normalization; this would be contrary to the advantage of this scheme which, in floating-point multiplication, adds the injection as part of the addition tree of the multiplier.

We now describe the adaptation of the scheme in [16]. In contrast to the case of floating-point multiplication, in the proposed MAF, the result has been normalized before the add/round module so that no overflow is possible and the rounding operation is simplified. As shown in Fig. 7, the module has three slices: a least-significant slice (bits 54-161), a center slice (bits 51-53), and a most-significant slice (bits 0-50). These have the following characteristics:

1. Least-significant slice (bits 54-161) is input to the logic that computes the carry into the upper part and the sticky bit. In an effective subtraction, a 1 has to be added into the least-significant bit ($complement = 1 \Rightarrow add + 1$). Moreover, the *partial sticky bit* $st1$, calculated with the bits shifted out during the alignment, has to be taken into account in the calculation of the sticky bit.

2. Center slice (bits 51-53). Following [16], these 3 bits are added, together with the carry from the least-significant slice, the sticky bit (only in the rounding to infinity mode), and the 1 that always gets added in position 53 for rounding to nearest and rounding to infinity, to compute the two least-significant bits of the result.[9]

   We take into account that the delay of the calculation of the carry from the least-significant slice is larger than the delay of the calculation of the sticky bit plus the delay of the 3-bit addition; then, the addition is implemented conditionally, computing the 3-bit sum and $sum + 1$ and, once the carry is known, the correct value is selected. Note that the addition of the 1 in position 53 plus the sticky bit plus the 3-bit slices of the sum and carry words requires a 3:2 CSA and then a 3-bit adder, as shown in Fig. 3.

   As in [16], a row of HAs spanning from bit 0 to bit 53 is used to limit the maximum value in bits 51 to 53 and assure that only one carry is produced to bit 50 during the rounding and the assimilation of the lower part. Specifically, the maximum value is produced for the following situation (bits 1 to 50 are marked as $x \dots x$):

bits#	1-54 of the HA inputs	≡	x...xx 1111
			x...xx 0111
bits#	1-53 of the sum word	≡	x...xx 100
bits#	1-53 of the carry word	≡	x...x0 111.

   Any other HA's inputs produce a smaller amount in bits 51-53, for example:

bits#	1-54 of the HA inputs	≡	x...xx 1111
			x...xx 1111
bits#	1-53 of the sum word	≡	x...xx 000
bits#	1-53 of the carry word	≡	x...x1 111.

   Consequently, the addition of a maximum of three in bit 53 produces only one carry to position 50.

   As stated in Section 4.2, because of the anticipation of the HAs before the normalization, it is necessary to include HAs for the 162 bits.

3. Most-significant slice (bits 0-50). The 51 most-significant bits are input to the dual adder to obtain the sum (S) and the sum plus 1 ($Sp1$). The carry of the 3-bit addition in the center slice decides whether $S = X + Y$ or $Sp1 = X + Y + 1$ should be selected at the output of the dual adder.

The correct LSB of the result in the round to nearest rounding mode is determined from the bit in the rounding position and the sticky bit [12],

$$LSB_{corrected} = 0 \text{ if } (R \cdot \overline{sticky}) \cdot RN = 1.$$

9. This scheme presents some variations with respect to the implementation in [16]. In [16], as the result may have 1 bit of overflow, an additional 1 has to be added in position 52.

4.4.1 Correction Due to Anticipation

In the design of the dual adder, we have to take into consideration that some levels of the prefix tree have been anticipated before the normalization and that only the carry-generate vector G^j and the the individual carry-propagate bits p pass through the normalization shifter (see Section 4.2). For the rest of the prefix tree, we require vectors P and G, which correspond to level j. The calculation of these vectors is done as follows:

1. The vector P is calculated directly from the p_i, passed through the shifter. This requires a set of j-input AND gates.
2. The vector G uses the vector G^j passed through the shifter. However, since G^j was calculated before the shifter, it includes bits that are now outside of the addition range, as illustrated in the following example. Consequently, some bits have to be corrected. Fig. 8a shows an example with a reduced number of bits: 16-bit operands before the shifter, with $z = 6$, and an 8-bit dual adder.[10] We consider that two levels of the prefix tree are anticipated. Note that vector G^2 at the output of the shifter is not as needed for carries generation in the dual adder since it includes information from the bits that are out of the addition range (to simplify notation, we call G^j this shifted G^j); therefore, some of the least-significant bits have to be corrected before beginning the operation of the next prefix tree level.

   The number of bits needing correction ($correct$) depends on the number of levels anticipated before the shifter. It is easy to show that $correct = 2^j - 1$, j being the number of levels anticipated (three in the example). The relation between the corrected vector G and G^j is given by the following expression:

$$G_i^j = G_i + G_i^{rest} P_i,$$

   where G_i^{rest} corresponds to the carry generate (jth level) produced by the bits not inside the range of the adder. Consequently,

   - If $G_i^j = 0$, then $G_i = 0$.
   - If $G_i^j = 1$, then we consider two cases:
     - If $P_i = 1$, then $G_i = 0$ (mutually exclusive),
     - If $P_i = 0$, then $G_i = 1$.

Then, as a summary of the previous considerations:

$$G_i = \begin{cases} G_i^j \cdot \overline{P}_i & \text{if correction needed} \\ G_i^j & \text{otherwise.} \end{cases}$$

The remaining levels correspond to a standard implementation [2].

10. The adder shown is a Kogge-Stone prefix adder, but any other prefix adder could be used.

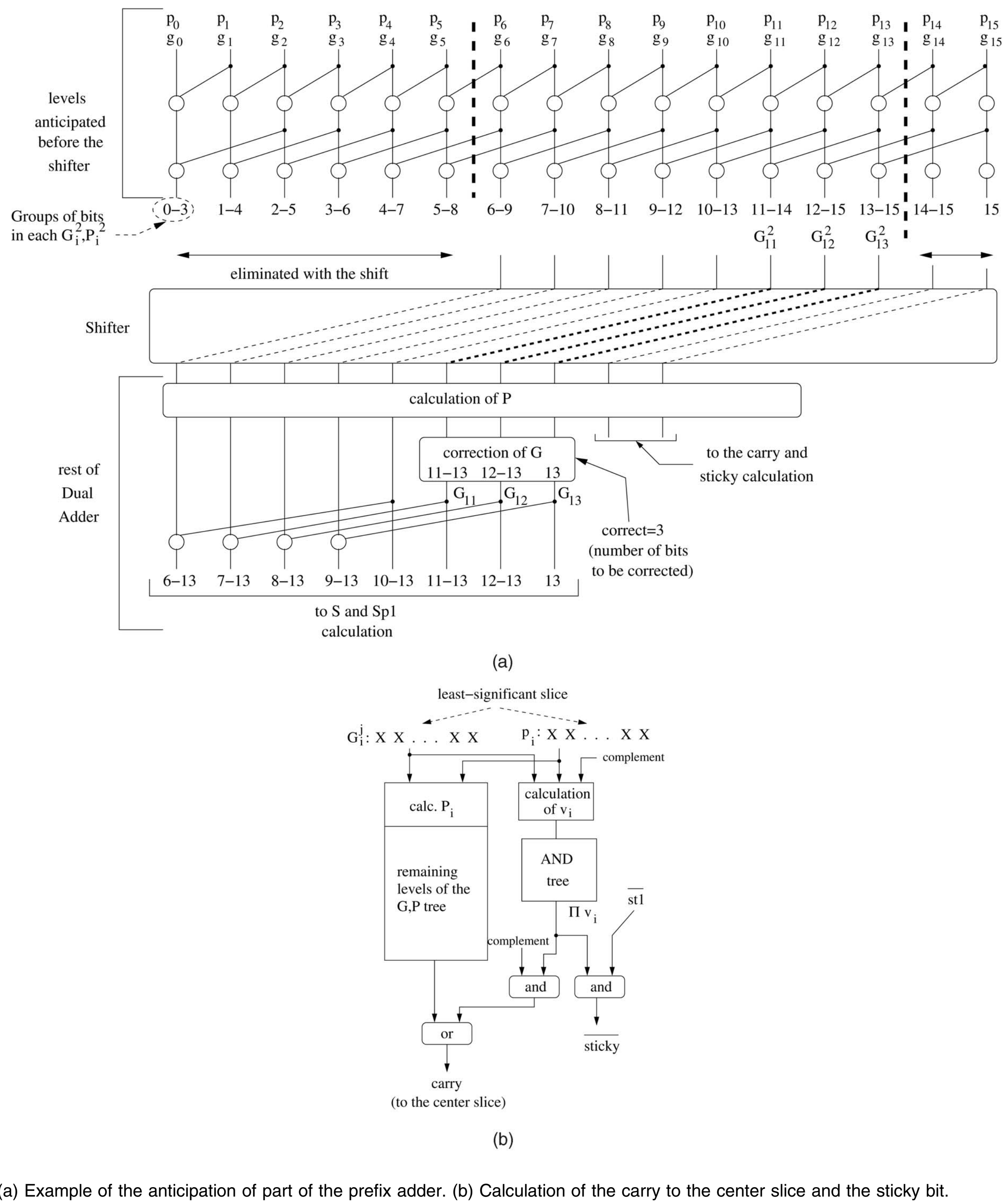
levels anticipated before the shifter
Groups of bits in each G_i^2, P_i^2
eliminated with the shift
Shifter
calculation of P
correction of G
to the carry and sticky calculation
correct=3 (number of bits to be corrected)
rest of Dual Adder
to S and Sp1 calculation
(a)
least−significant slice
calc. P_i
remaining levels of the G,P tree
calculation of v_i
AND tree
complement
st1
and
or
carry (to the center slice)
sticky
(b)

Fig. 8. (a) Example of the anticipation of part of the prefix adder. (b) Calculation of the carry to the center slice and the sticky bit.

4.4.2 Carry from the Least-Significant Slice and Sticky Bit

As shown in Fig. 7, the addition in the least-significant slice can propagate a carry to the center slice. Moreover, the sticky bit is computed in the least-significant slice. But, since part of the addition has been anticipated before the normalization, the carry and the sticky bits have to be computed from vectors G^j and p; moreover, we have to consider that a 1 is added in the least-significant position when $complement = 1$. Note that this 1 cannot be included in the G,P tree (*carry in = complement*) because of the anticipation of several levels before the normalization. In this slice, it is not necessary to correct the bits of G^j. Fig. 8b shows the block diagram for the calculation of the carry and the sticky bit. Two parts can be distinguished:

- Carry calculation. When $complement = 0$, the carry is calculated with a standard prefix tree, implementing the nonanticipated levels. When $complement = 1$ (add +1), additionally to the carry generated in the G,P tree, a carry propagates to the center slice when[11] $\Pi_{i \in least.sign.slice} p_i = 1$.

11. That means that the addition of the two operands is $1 \ldots 1$; if a $+1$ is added, a carry is propagated to the center slice.

- Sticky bit calculation. The sticky bit is composed of two components: $st1$, obtained from the bits shifted out during the alignment of A_{inv}, and $st2$, obtained from the least-significant slice after the normalization shifter. The final sticky bit is $st = st1 + st2$. To obtain the partial sticky bit $st2$, we have to determine if the addition in the least-significant slice is zero,

$$st2 = 0 \quad if \quad \begin{cases} Sum.word + Carry.word \\ = 0 \ldots 0 & \text{and } complement = 0 \\ Sum.word + Carry.word \\ = 1 \ldots 1 & \text{and } complement = 1. \end{cases}$$

First, we consider the case of $complement = 0$. We adapt to our MAF the method described in [4] for a fast evaluation of condition $A + B = 0$ without carry propagation. In summary, the method proposed in [4] is as follows: Given two operands $A = a_0 \ldots a_{n-1}$ and $B = b_0 \ldots b_{n-1}$, to evaluate condition $A + B = 0$ vector $E = (e_0, \ldots, e_{n-1})$ is computed, $e_i = k_{i+1} \oplus p_i$ being $p_i = a_i \oplus b_i$ and $k_i = \overline{a_i + b_i}$. Then,

$$A + B = 0 \quad \text{if} \quad Z_{compl=0} = \prod_{i=0}^{n-1} e_i = 1.$$

In order to utilize this method, we need to compute bits k_i from vector G^j and bits p_i. This computation is based on $G_i^j = g_i + p_i \cdot G_{i+1}^j$ and two cases can be considered:

- $p_i = 0$. Then, $g_i = G_i^j$ and, therefore, $k_i = \overline{G_i^j}$.
- $p_i = 1$. Then, $k_i = 0$.

Consequently, $k_i = \overline{p_i} \cdot \overline{G_i^j}$.

On the other hand, when $complement = 1$, we have

$$A + B = 0 \quad \text{if} \quad Z_{compl=1} = \prod_{i=0}^{n-1} p_i = 1.$$

Combining both situations, *complement=0* and *complement=1*, we get

$$A + B = 0 \quad \text{if} \quad Z = \prod_{i=0}^{n-1} v_i = 1$$

with

$$\begin{aligned} v_i &= k_{i+1} \cdot \overline{complement} \oplus p_i \\ &= \overline{(p_{i+1} + G_{i+1}^j + complement)} \oplus p_i. \end{aligned}$$

Consequently, $\overline{st2} = \prod v_i$ and, finally,

$$\overline{sticky} = \overline{st1} \cdot \overline{st2} = \overline{st1} \cdot \prod v_i.$$

Therefore, as shown in Fig. 8b, the calculation of the carry and the sticky bits requires a G,P tree, the logic for computing v_i, and an AND tree for computing $\prod v_i$.

5 Delay Estimation and Comparison

We estimate the delay of the proposed MAF and compare it with the basic MAF. The estimates are performed at a *qualitative* level. A more accurate evaluation depends on a large number of technological factors and on the floating-point unit requirements.

5.1 Critical Path Delay in the Proposed MAF Architecture

As shown in Fig. 3, the critical path of the proposed MAF is given by the multiplier (t_{mult}), parallel paths ($t_{upper\,par\,paths}$), normalization ($t_{norm\,shift}$), and the part of the add/round module that has not been anticipated ($t_{add/round}$) (this delay includes the calculation of the G and P vectors). Then,

$$t_{maf} = t_{mult} + t_{upper\,par\,paths} + t_{norm\,shift} + t_{add/round} \qquad (1)$$

where $t_{upper\,par\,paths}$ is the largest delay of the three parallel paths 1) sign detection and 54-bits shift ($t_{sign} + 2t_{mux}$), 2) calculation of the MSB of the shift amount ($t_{lza\,string} + t_{s1}$), and 3) 3:2 CSA plus HA plus part of the adder ($t_{csa} + t_{ha} + t_{part\,of\,adder}$), that is,

$$\begin{aligned} t_{upper\,par\,paths} = \max\{&(t_{sign} + 2t_{mux}), (t_{csa} + t_{ha} + t_{part\,of\,adder} \\ &+ 2t_{mux}), (t_{lza\,string} + t_{s1})\}. \end{aligned} \qquad (2)$$

Note that $t_{part\,of\,adder}$ depends on the number of levels anticipated before the shifter. More specifically, the part of the adder that is anticipated is such that the delay of this path matches the delay of the longest of the other two paths.

In the add/round module, there are also three parallel paths (*lower parallel paths*) and a final result selection mux. The paths are: the path going through the dual adder ($t_{rest\,of\,adder}$), the path involving the calculation of the carry and the selection of the carry to bit 51 ($t_{carry\,calc} + t_{mux}$) and the path involving the calculation of the sticky bit ($t_{sticky} + t_{csa} + t_{2b\,add} + t_{mux}$). Therefore, the delay of the add/round module is given by

$$\begin{aligned} t_{add/round} = \max\{&t_{rest\,of\,adder}, (t_{carry\,calc} + t_{mux}), \\ &(t_{sticky} + t_{csa} + t_{2b\,add} + t_{mux})\} + t_{mux}. \end{aligned} \qquad (3)$$

5.2 Critical Path Delay in the Basic MAF Architecture and Comparison

The Basic MAF architecture is shown in Fig. 1a. To reduce the delay, the 161-bit one's complement adder is implemented as a 106-bit adder and a 53-bit incrementer for the most-significant part [7], [8], [11]. The carry generated in the 106-bit adder is used to select the incremented or nonincremented most-significant part. This way, the adder delay is the delay of a 106-bits one's complement adder ($t_{106-bit\,adder}$) plus the delay of the output bit inversion ($t_{1'comp}$). Then,

$$\begin{aligned} t_{maf\,basic} = t_{mult} + t_{csa} + t_{106-bit\,adder} + t_{1'comp} + t_{norm\,shift} \\ + t_{round}, \end{aligned} \qquad (4)$$

t_{round} being the delay of the rounding that, basically, consists of another 54-bit adder.

To determine the delay reduction achieved with the proposed MAF, (1) and (4) are compared. Considering that

$t_{mux} = t_{1'comp}$, the delay difference between the Basic MAF and the proposed MAF is

$$\Delta = (t_{csa} + t_{106-bit\,adder} + t_{round}) - (t_{upper\,par\,paths} + t_{lower\,par\,paths}). \quad (5)$$

We now estimate the delay of each of the terms, using as delay unit the delay of an inverter with load of 4, t_{inv4}. Although this measure is being used to be independent of the technology, there are several modules which have alternative implementations, with different delays, areas, and energy dissipation; consequently, we provide here an estimation of the improvement that is achieved with some reasonable implementation of these modules. Under these conditions, we have estimated that the delays of the three paths in the upper parallel paths are almost equal and roughly equal to the delay of a 54-bit adder, $t_{upper\,par\,paths} \approx t_{54-bit\,adder}$. Similarly, we have estimated that the delays of the three paths in the lower parallel paths are roughly similar, $t_{lower\,par\,paths} \approx t_{rest\,of\,adder}$. Then, considering $t_{round} \approx t_{54-bit\,adder}$ and replacing in (5),

$$\Delta = t_{csa} + t_{106-bit\,adder} - t_{rest\,of\,adder}. \quad (6)$$

To estimate $t_{rest\,of\,adder}$, we have to determine the number of levels of the dual adder that can be placed before the normalization shifter. Since

$$t_{csa} + t_{ha} + t_{part\,of\,adder} + 2\,t_{mux} \leq t_{par\,paths} \approx t_{54-bit\,adder},$$

we have

$$t_{part\,of\,adder} \leq t_{54-bit\,adder} - (t_{csa} + t_{ha} + 2\,t_{mux}). \quad (7)$$

The delays for each element in (7) are: $t_{csa} = 8\,t_{inv4}$, $t_{ha} = 4\,t_{inv4}$, and $t_{mux} = 4\,t_{inv4}$. We assume that the 54-bits adder is formed by a 6-levels prefix tree plus the logic for the calculation of the g_i and $\bar{k}_i$ and the XOR gates to obtain the sum vector. To obtain more realistic estimations, we consider the gate load and the additional load introduced by long wires [9]. Then, according to our estimations, $t_{54-bit\,adder} = 36\,t_{inv4}$ and it is possible to anticipate part of the adder such that $t_{part\,of\,adder} = 16\,t_{inv4}$. With this bound, and considering the delay in the calculation of the G and P vectors, $t_{corr} = 4\,t_{inv4}$, the calculation of the p_i, g_i, and $\bar{k}_i$ bits, and two levels of the prefix tree can be anticipated. Therefore,

$$t_{rest\,of\,adder} = t_{54-bit\,adder} + t_{corr} - t_{part\,of\,adder} \approx 24\,t_{inv4}. \quad (8)$$

To determine the delay reduction, the delay of a 106-bit adder needs to be estimated. Considering a 7-level prefix tree, the logic for the end-around carry adjustment, and the delay due to long wires, we have estimated $t_{106-bit\,adder} = 46\,t_{inv4}$. Then, replacing, in (6), the delay reduction obtained with the proposed architecture is $30\,t_{inv4}$.

To put in perspective this reduction, we estimate the global delay of the basic MAF architecture with the following delays for the remaining components of the unit: 1) 53×53 multiplier, we assume a radix-4 coding for the multiplier and a CSA tree with 4 levels of 4:2 CSA ($t_{mult} = 60\,t_{inv4}$); and 2) normalization shifter, the delay of the normalization shifter is about 40 percent of the delay of the 106-bits 1's complement adder [7] ($t_{norm\,shift} \approx 20\,t_{inv4}$). We can conclude that the delay of the basic MAF unit is $t_{maf\,basic} \approx 175\,t_{inv4}$ and, therefore, the proposed architecture achieves a delay reduction of between 15 percent and 20 percent.

Since the unit is pipelined, the delay reduction is effective in improving the performance if it allows the reduction of the cycle time and/or the number of pipeline stages. If this is the case, it depends on specific implementation constraints, such as the intended processor cycle time and the limitations on partitioning into stages.

6 Conclusions

An architecture for a floating-point Multiply-Add-Fused (MAF) unit that reduces the latency of the traditional MAF units has been proposed. The proposed MAF is based on the combination of the final addition and the rounding, by using a dual adder. This approach has been used previously to reduce the latency of the floating-point addition and multiplication. However, it can be used only if the normalization is performed after the rounding and this is not possible for the MAF operation because the rounding position is not known until the normalization has been performed. To overcome this difficulty, we propose that the normalization be carried out before the addition. This required a careful design of some other parts of the MAF, the leading-zeros-anticipator (LZA), and the dual adder. The LZA is implemented so that its operation overlaps with the normalization shifter. Moreover, part of the dual addition, the calculation of g, p, and $\bar{k}$ vectors and some of the first levels of the prefix tree, is anticipated before the normalization. This permits us to balance the delay of the several parallel paths in the MAF and to reduce the delay of the critical path.

The delay of the proposed architecture has been estimated and compared with the basic MAF architecture. The conclusion is that, for a double-precision floating-point representation, the proposed MAF reduces the delay by about 15-20 percent.

Acknowledgments

This work was performed in part while J.D. Bruguera was with the University of California at Irvine. This work was supported in part by the *Ministry of Science and Technology* of Spain under contract MCYT-FEDER TIC2001-3694-C02 and by US National Science Foundation grant MIP-9726244.

References

[1] E. Antelo, M. Boo, J.D. Bruguera, and E.L. Zapata, " A Novel Design for a Two Operand Normalization Circuit," *IEEE Trans. Very Large Scale of Integration (VLSI) Systems,* vol. 6, no. 1, pp. 173-176, 1998.

[2] N. Burgess, "The Flagged Prefix Adder for Dual Additions," *Proc. SPIE ASPAII7,* 1998.

[3] C. Chen, L.-A. Chen, and J.-R. Cheng, "Architectural Design of a Fast Floating-Point Multiplication-Add Fused Unit Using Signed-Digit Addition," *Proc. Euromicro Symp. Digital System Design (DSD 2001),* pp. 346-353, 2001.

[4] J. Cortadella, J.M. Llaberí, "Evaluation of $A + B = K$ Conditions without Carry Propagation," *IEEE Trans. Computers,* vol. 41, no. 11, pp. 1484-1488, Nov. 1992.

[5] G. Even and P.M. Seidel, “A Comparison of Three Rounding Algorithms for IEEE Floating-Point Multiplication,” *IEEE. Trans. Computers,* vol. 49, no. 7, pp. 638-650, July 2000.

[6] C. Heikes and G. Colon-Bonet, “A Dual Floating Point Coprocessor with an FMAC Architecture,” *Proc. IEEE Int'l Solid State Circuits Conf. (ISSCC96),* pp. 354-355, 1996.

[7] E. Hokenek, R. Montoye, and P.W. Cook, “Second-Generation RISC Floating Point with Multiply-Add Fused,” *IEEE J. Solid-State Circuits,* vol. 25, no. 5, pp. 1207-1213, 1990.

[8] R.M. Jessani and M. Putrino, “ Comparison of Single- and Dual-Pass Multiply-Add Fused Floating-Point Units,” *IEEE Trans. Computers,* vol. 47, no. 9, pp. 927-937, Sept. 1998.

[9] S. Knowles, “A Family of Adders,” *Proc. IEEE 14th Symp. Computer Arithmetic,* pp. 30-34, 1999.

[10] S.F. Oberman, H. Al-Twaijry, and M.J. Flynn, “The SNAP Project: Design of Floating-Point Arithmetic Units,” *Proc. IEEE 13th Symp. Computer Arithmetic,* pp. 156-165, 1997.

[11] F.P. O'Connell and S.W. White, “ POWER3: The Next Generation of PowerPC Processors,” *IBM J. Research and Development,* vol. 44, no. 6, pp. 873-884, 2000.

[12] M.R. Santoro, G. Bewick, and M.A. Horowitz, “Rounding Algorithms for IEEE Multipliers,” *Proc. IEEE Ninth Symp. Computer Arithmetic,* pp. 176-183, 1989.

[13] P.M. Seidel and G. Even, “How Many Logic Levels Does Floating-Point Addition Require?” *Proc. Int'l Conf. Computer Design (ICCD '98),* pp. 142-149, 1998.

[14] M.S. Schmookler and K.J. Nowka, “Leading Zero Anticipation and Detection—A Comparison of Methods,” *Proc. IEEE 15th Symp. Computer Arithmetic,* pp. 7-12, 2001.

[15] H. Sharangpani and K. Arora, “Itanium Processor Microarchitecture,” *IEEE Micro,* vol. 20, no. 5, pp. 24-43, Sept./Oct. 2000.

[16] R.K. Yu and G.B. Zyner, “167 MHz Radix-4 Floating-Point Multiplier,” *Proc. IEEE 12th Symp. Computer Arithmetic,* pp. 149-154, 1995.

Tomás Lang received the electrical engineering degree from the Universidad de Chile in 1965, the MS degree from the University of California, Berkeley, in 1966, and the PhD degree from Stanford University in 1974. He is a professor in the Department of Electrical Engineering and Computer Science at the University of California, Irvine. Previously, he was a professor in the Computer Architecture Department of the Polytechnic University of Catalonia, Spain, and a faculty member of the Computer Science Department at the University of California, Los Angeles. His primary research and teaching interests are in digital design and computer architecture with current emphasis on high-speed and low-power numerical processors and multiprocessors. He is coauthor of two textbooks on digital systems, a textbook on digital arithmetic, two research monographs, one IEEE tutorial, and the author or coauthor of research contributions to scholarly publications and technical conferences. He is a member of the IEEE Computer Society.

Javier D. Bruguera received the BS degree in physics and the PhD degree from the University of Santiago de Compostela (Spain) in 1984 and 1989, respectively. He is a professor in the Department of Electronic and Computer Engineering at the University of Santiago de Compostela. Previously, he was an assistant professor in the Department of Electrical, Electronic, and Computer Engineering at the University of Oviedo, Spain, and an assistant professor in the Department of Electronic Engineering at the University of A Corunna, Spain. He was a visiting reseacher in the Application Center of Microelectronics at Siemens in Munich, Germany and in the Department of Electrical Engineering and Computer Science at the University of California, Irvine. His primary research interests are in the area of computer arithmetic, processor design, digital design for signal and image processing, and parallel architectures. He is a member of the IEEE.