Comparison of Single- and Dual-Pass Multiply-Add Fused Floating-Point Units

Romesh M. Jessani and Michael Putrino, *Senior Member*, *IEEE*

Abstract—Low power, low cost, and high performance factors dictate the design of many microprocessors targeted to the low-power computing market. The floating-point unit occupies a significant percentage of the silicon area in a microprocessor due its wide data bandwidth (for double-precision computations) and the area occupied by the multiply array. For microprocessors designed for portable products, the design-size of the floating-point unit plays an important role in the low cost factor driven by reduced chip area. Some microprocessors have multiply-add fused floating-point units with a reduced multiply array, requiring two passes through the array for operations involving double-precision multiplies. This paper discusses the design complexities around the dual-pass multiply array and its effect on area and performance. Floating-point unit areas and their associated multiply array areas are compared for a single- and dual-pass implementation in a given technology (PowerPC 604eTM and PowerPC 603eTM microprocessors, respectively).

Index Terms—Floating-point unit, multiply-add fused, multiply array, alignment shifter, sign encoding, Booth encoding.

1 Introduction

Microprocessor floating-point units (FPUs) supporting the IEEE-754 standard for single- and double-precision binary floating-point arithmetic [1] may be designed with low cost, low power, and high performance objectives in mind. Some microprocessors targeted to the low-power computing market implement on-chip FPUs that execute multiply and add operations as single instructions [3], [4], [5] as:

$$T = (A \times C) + B.$$

In such cases, the floating-point hardware is designed to accept up to three operands for executing multiply-add instructions, while other floating-point instructions requiring fewer than three operands may utilize the same hardware by forcing constants into the unused operands. For example, to execute the floating-point add instruction, T = A + B, the C operand is forced to the constant 1.0.

In general, FPUs with multiply-add implementations sometimes use a multiply array to compute the AC product, an alignment shifter to align the B operand mantissa with respect to the AC product based on the exponents of the A, B, and C operands, followed by an adder to compute AC + B. The intermediate result produced is normalized and rounded to compute the final result. The double-precision multiplication of the A and C operand mantissas (53-bit x 53-bit) leads to a 106-bit value. A 161-bit right-shifter can be used to align the B-operand mantissa with the 106-bit intermediate AC product (described in Section 3. and depicted in Fig. 4). This leads to a 161-bit adder to compute the sum of AC plus B.

The cost, or area, associated with such designs may be prohibitive and can be reduced by decreasing the size of the multiply array to implement a subset of multiplication operations in a single pass through the array, while requiring multiple passes for full-precision computation. This can be done as long as the associated reduction in performance is acceptable. The multiplier in the Cyrix83D87 [2] uses a 17 × 69 bit array. It requires four passes through the array to compute extended-precision multiplication operations.

This paper illustrates the complexity associated with a dual-pass multiply-array implementation in a multiply-add fused (MAF) FPU, along with its area reduction and performance degradation, by comparing two implementations: one having a single-pass MAF FPU design (PowerPC 604e microprocessor) and one having a dual-pass MAF FPU design (PowerPC 603e microprocessor).

The convention used in this paper to describe multibit data is x(0..n), where x(0) is the most-significant bit and x(n) is the least-significant bit.

2 Multiply-Add Fused FPUs

Microprocessors implementing a MAF FPU [3] to execute floating-point instructions may do so with three independent pipelined dataflow stages:

- Multiply stage: includes a multiply array for the multiplication of the 53-bit A operand mantissa and the 53-bit C operand mantissa; and the alignment shifting of the B operand mantissa with the intermediate AC product.
- Add stage: involves the addition of the AC product and aligned B operand mantissa; and leading zero detection (LZD) for computation of the normalize shift count.
- Normalize/Round stage: involves normalizing and rounding the intermediate result to produce the final result.

- *R. Jessani is with Ross Technology, Inc., 5316 Hwy 290 W., Suite 500, Austin, TX 78735. E-mail: romesh@ross.com.*
- *M. Putrino is with IBM Corp., 11400 Burnet Rd., International Mail Drop 6301, Austin, TX 78758. E-mail: putrino@us.ibm.com.*

Manuscript received 11 Oct. 1996; revised 19 May 1997.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 102129.

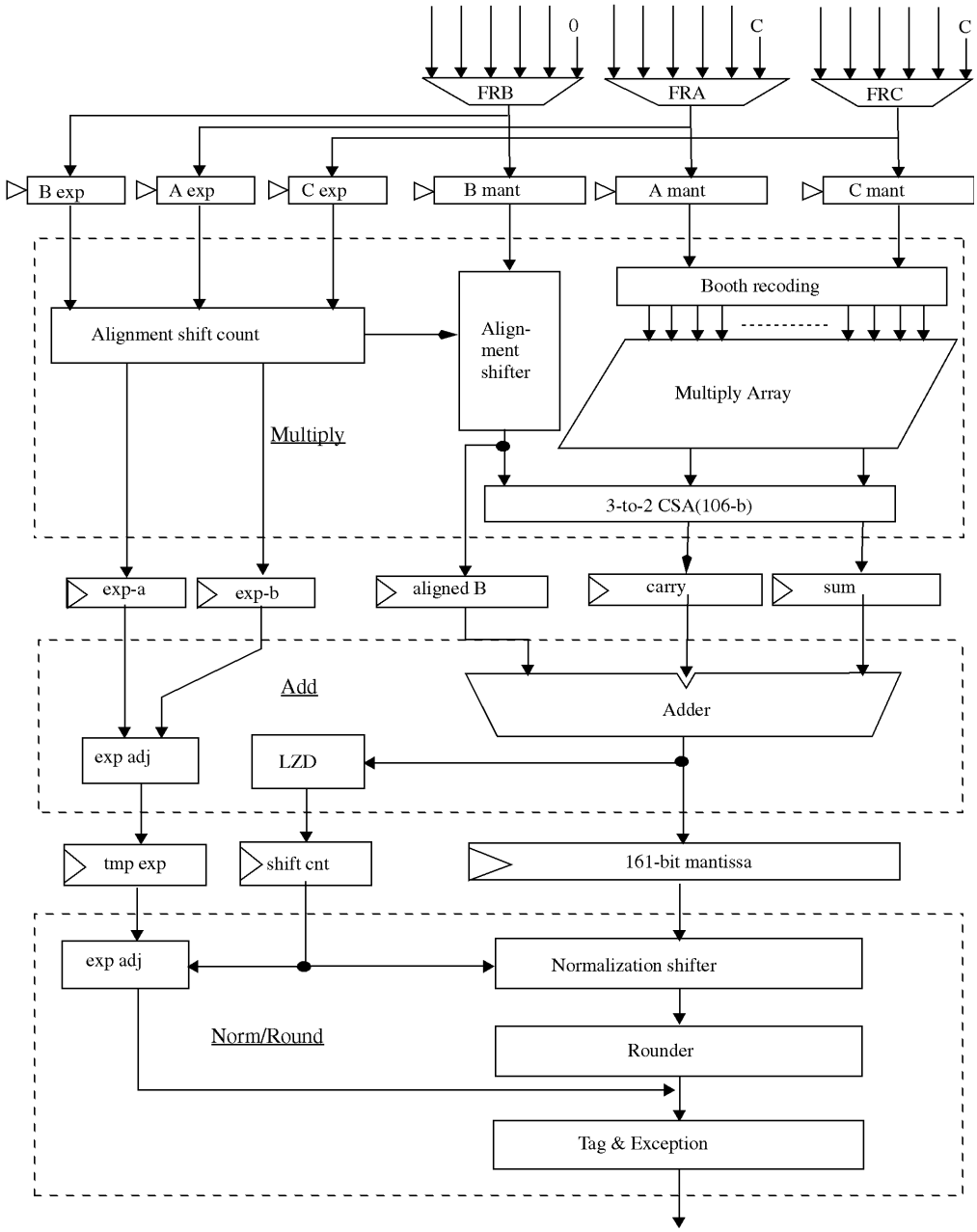


Fig. 1. Typical MAF microprocessor FPU.

Fig. 1 shows the major building blocks of a microprocessor FPU with three pipelined stages. As with FPUs of many of today's microprocessors, the multiply stage incorporates the multi-bit scanning technique (modified Booth algorithm) [6][7] and a Wallace tree of carry save adders (CSAs) [8] for the summation of partial-products to perform the multiplication of the two 53-bit mantissas.

3 Mathematics of Floating-Point Multiply and Multiply-Add Operations

The 53-bit × 53-bit multiplication for double-precision computations can be carried out using radix-4 Booth recoding of the multiplier (C operand mantissa) shown in Table 1, followed by the selection of the appropriate factor of the multiplicand (A operand mantissa) to produce the 27 partial-products which are to be summed to compute the AC product.

The radix-4 Booth recoding of groups of bits of the multiplier produces the multiplication factor to be applied to the multiplicand to obtain the partial-products. The multiplication factors are 0, +1, +2, –1, and –2. The +2 multiplication is obtained by left-shifting the A operand mantissa by one bit. The –1 and –2 factors require left-shifting by one bit (for –2), inversion of the A operand mantissa, and adding one (for 2's-complement negation) as shown:

$$-1 \times A = \overline{A} + 1$$

$$-2 \times A = \overline{2 \times A} + 1.$$

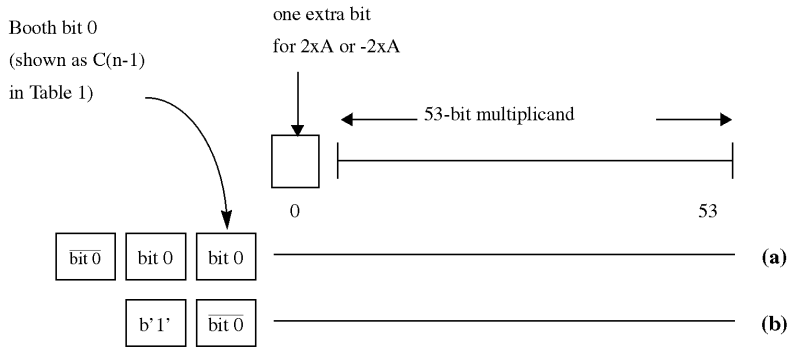


Fig. 2. Reduced sign extension for partial-products. (a) Sign extension for pp0. (b) Sign extension for pp1..pp26.

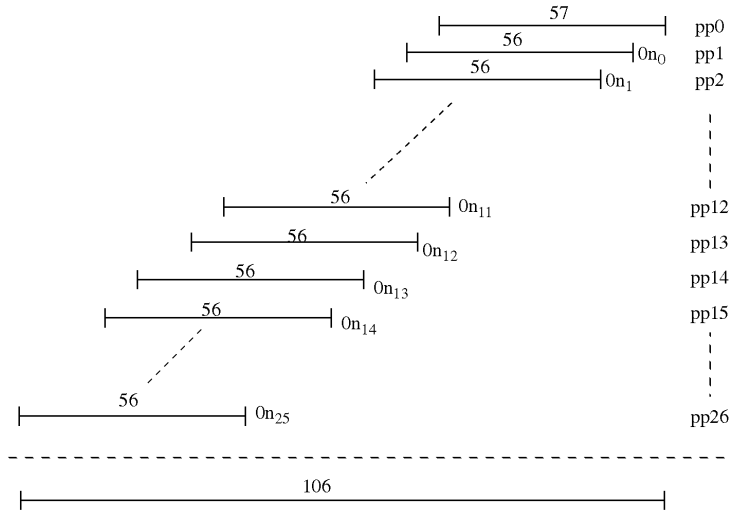


Fig. 3. Array of partial-products.

Because adding a one (termed "hot-one") requires an adder (which may compromise timing), the hot-ones of the negative partial-products are incorporated into the Wallace tree by concatenating a b'01' to the low end of the next partial-product just below the least-significant bits of the negative partial-product (because the next partial-product is displaced two bit position to the left, the alignment can easily be performed). The last partial-product is always positive in both single- and double-precision (due to a b'0' concatenated to the most-significant bit of the multiplier—the bits forming the last partial-product), thereby eliminating the requirement of adding a hot-one below the last partial-product. This configuration allows the Booth multiplexors (5:1 multiplexors for selecting 0, +1xA, +2xA, –1xA, or –2xA) to select 0, A, A shifted left one bit, A inverted, or A shifted left one bit and inverted, when forming the partial-products that feed the CSA tree.

TABLE 1
RADIX-4 BOOTH ENCODING

C(n-1)	C(n)	C(n+1)	partial-product
0	0	0	+0 x A
0	0	1	+1 x A
0	1	0	+1 x A
0	1	1	+2 x A
1	0	0	-2 x A
1	0	1	-1 x A
1	1	0	-1 x A
1	1	1	-0 x A

Because the partial-products generated can be positive or negative, the summation of the partial-products involves sign extensions up to the width of the final product (106-bits for partial-product 0). However, high-performance, hard-wired, and reduced size multipliers may incorporate the use of modified partial-products in a reduced left edge banded matrix [9], [10]. The reduced sign extension format results in a 57-bit partial-product 0 (pp0) and 56-bit pp1 through pp26, as shown in Fig. 2.

The partial-products can be added using a Wallace tree of CSAs to produce the result in sum and carry format. This results in a 106-bit product, as shown in Fig. 3, where n_0 through n_{25} represent the hot-ones for pp0 through pp25, respectively. No hot-one is required at position n_{26} because a b'0' is concatenated to the most-significant bit of the multiplier forcing pp26 to be positive.

The multiply array may also include the addition of the aligned B operand mantissa to the sum and carry partial-products representing the AC product with 3:2 CSAs. The B

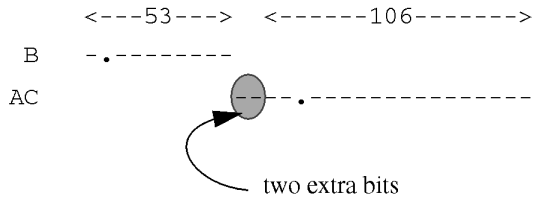


Fig. 4. Orientation of B and AC mantissas prior to alignment.

and AC mantissas can be aligned (exponents adjusted appropriately according to the shift count) by shifting the B operand mantissa with respect to the AC product (because the B operand mantissa is typically available at the start of the cycle, whereas the AC product is available only after several levels of logic). The B alignment shifter can be implemented as a right-shifter by positioning the B operand mantissa 56 bits left of the binary point of the AC product before the shift count is calculated. The shift count for the B operand mantissa is then calculated as:

$$\text{shift count} = \exp(AC) - \exp(B) + 56,$$

where exp(AC) is the exponent of the A operand added to the exponent of the C operand, and exp(B) is the exponent of the B operand.

The 53-bit B operand mantissa must be positioned at least 53 bits left of the AC product (for a right-only alignment shifter). The AC product (53-bit × 53-bit) results in a 106-bit intermediate result with two bits to the left of the binary point. The two extra bits, positioned between the B operand mantissa and the AC product, as shown in Fig. 4, allow the guard and round bits to be zero and the AC product to fall into the sticky bit calculation when the B operand is not shifted (for zero or negative shift counts). This enables correct round function calculations as dictated by the IEEE standard.

The full multiplier implementation results in the need for a right-shifter for the B operand mantissa, which can shift up to 161 bits, and an adder in the add stage that can perform a 161-bit addition.

4 Reducing the Multiply Array

Fig. 5 shows a full 53-bit × 53-bit multiplier implementation using Booth recoding and a Wallace tree of 4:2 CSAs (one group actually being 3:2 CSAs). Other commonly used CSAs are the 3:2 CSA [8], [11], the 7:3 CSA, and the 5:5:4 CSA [11], [12], [13]. The use of 4:2 CSAs was chosen for the PowerPC 604e and PowerPC 603e microprocessors FPU implementations due to availability of circuit resources and the dependance of design-time on schedule while still meeting the frequency goals of the design. It should be noted that the use of 4:2 CSAs in partial-product reduction trees has been evaluated against other techniques which have been shown to have faster partial-product reduction times [14], [15], [16].

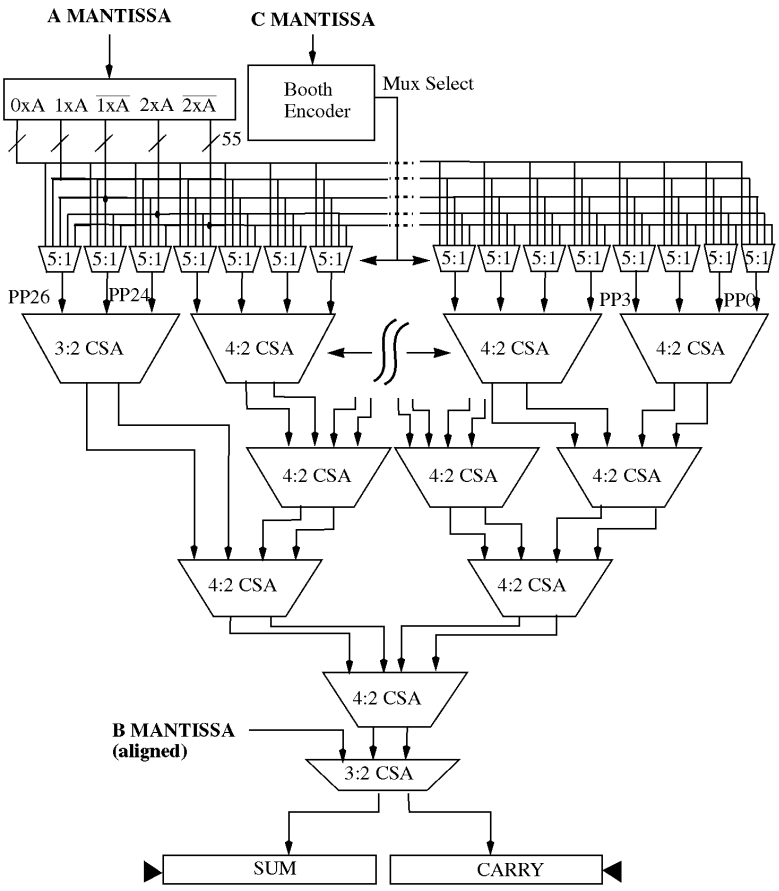


Fig. 5. 53-bit × 53-bit FPU multiplier implementation.

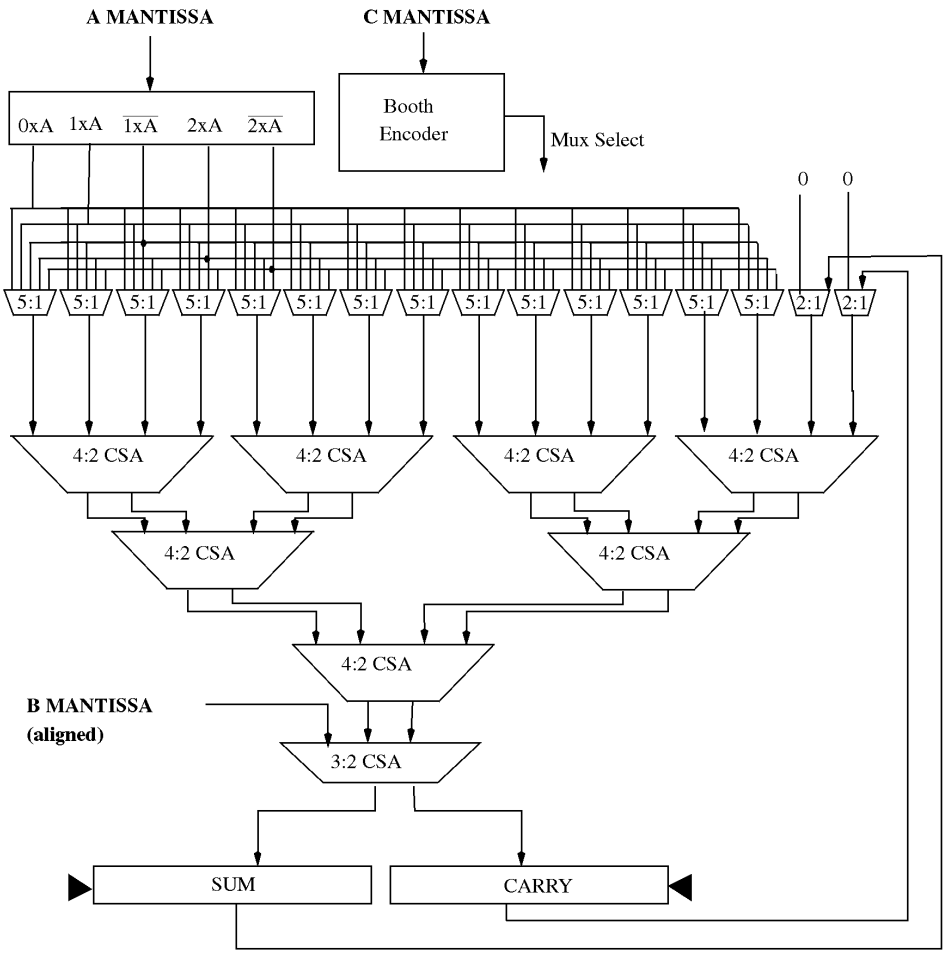


Fig. 6. Half floating-point multiplier implementation.

The full 53-bit × 53-bit multiplication requires 27 Booth recoders in radix-4 Booth recoding, which leads to 27 partial-products to be added with the 4:2 CSAs chosen for the Wallace tree implementation.

The 27 partial-products are added with 13 groups of CSAs (seven in the first level for adding the 27 partial-products; three in the second level to add the six sum and carry outputs from the first level; two in the third level to add the three sum and carry outputs form the second level and the remaining sum and carry outputs from the first level; and one in the fourth level to add the two sum and carry outputs from the third level) to reduce the addition to two partial-products (sum and carry outputs form the last 4:2 CSA level). The aligned B operand mantissa is then added to the sum and carry partial-products of the Wallace tree with a 3-to-2 CSA level.

The large number of Booth recoders and CSAs used leads to high area occupancy, which may not be desirable for a microprocessor targeted to the portable computer market. However, the product can be computed by implementing the multiply array in half the size by carrying out half the multiplication (with approximately the lower half of the multiplier operand) in one clock cycle, carrying over the result of the addition of the partial-products of the first half of the multiplication to the next clock cycle to be added to the partial-products of the second half of the multiplication (carried out with the remaining upper bits of the multiplier operand).

The multiplier operand is divided into two parts, the lower part consisting of 28 bits (generating 14 partial-products that are added with four groups of 4:2 CSAs in the first level leaving two inputs to these CSAs free for multiplexing the sum and carry result of the first pass in the next cycle) and the upper part consisting of 25 bits (generating 13 partial-products that are added by the first level of 4:2 CSAs along with the sum and carry results of the first pass). The multiplier implementation is shown in Fig. 6.

An additional partial-product is created by concatenating a b'000' to the most-significant bit of the multiplier to equalize the number of partial-products in both cycles, thereby eliminating the need to multiplex between pp13 and a zero value. The division of 25 and 28 bits of the C operand mantissa (with b'000' concatenated to the most-significant bit) allows double-precision multiplication computations to be carried out in two passes, while single-precision multiplication computations, requiring 26 bits of the C operand mantissa (24-bit single-precision mantissa with b'00' concatenated at the least-significant bit) can easily be carried out in one pass by ignoring the lower 27 bits of the 53-bit C operand mantissa and without additional multiplexing in the Wallace tree. The division of the C operand mantissa for dual-pass multiplication is shown in Fig. 7.

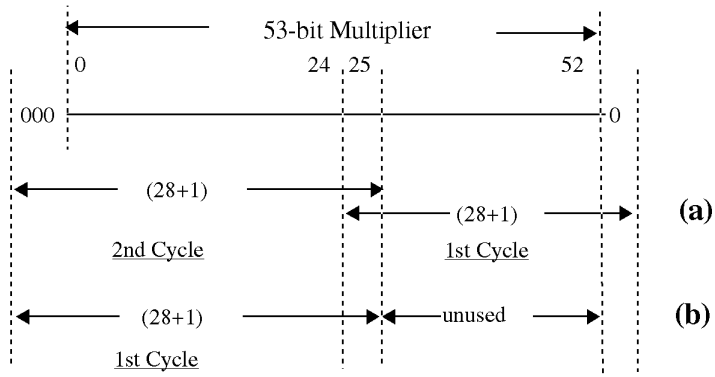


Fig. 7. Multiplier operand subdivision for dual-pass multiply arrays. (a) Double precision. (b) Single precision.

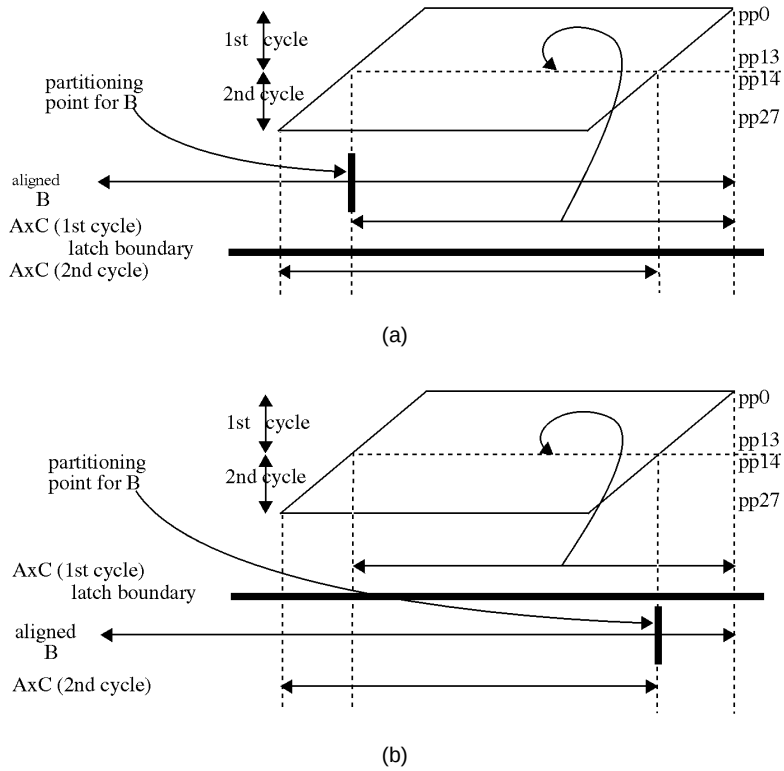


Fig. 8. Partitioning the B operand mantissa. (a) 3:2 CSA in multiply stage. (b) 3:2 CSA in add stage.

It can be seen that an implementation having a dual-pass multiply array reduces the number of Booth recoders (27 to 14), the number of Booth multiplexors (27 to 14), and the number of CSA groups (13 to seven). Moreover, it can also be seen that there is a reduction in one level of 4:2 CSAs in the Wallace tree possibly allowing for higher attainable frequencies of operation.

5 Effect on the B Alignment Shifter

For single-pass multiply arrays, the B operand mantissa can be aligned in a 161-bit range around the 106-bit AC product. The dual-pass multiplier requires partitioning the 161-bit aligned B mantissa into two parts, one for each pass of the multiply. The aligned B mantissa is added with a 3:2 CSA group to the sum and carry outputs from the 4:2 CSA Wallace tree. The partitioning of the aligned B mantissa is dependent on whether the 3:2 CSA group resides in the multiply stage or in the add stage.

Fig. 8 shows the two described possibilities for positioning the 3:2 CSA group used to incorporate the aligned B mantissa into the intermediate result.

In Fig. 8a, the 3:2 CSA resides in the multiply stage. The feedback of the upper bits of the sums and carries of AC (first pass) includes the middle bits of the aligned B mantissa. The lower bits of AC (first pass), that include the lower bits of the aligned B mantissa, are latched for transfer to the add stage.

In Fig. 8b, the 3:2 CSA resides in the add stage. The feedback of the upper bits of the sums and carries of the AC product (first cycle) does not include the middle bits of the aligned B mantissa. The lower bits of the AC product (first cycle), which include the lower bits of the aligned B mantissa, are latched for the add stage.

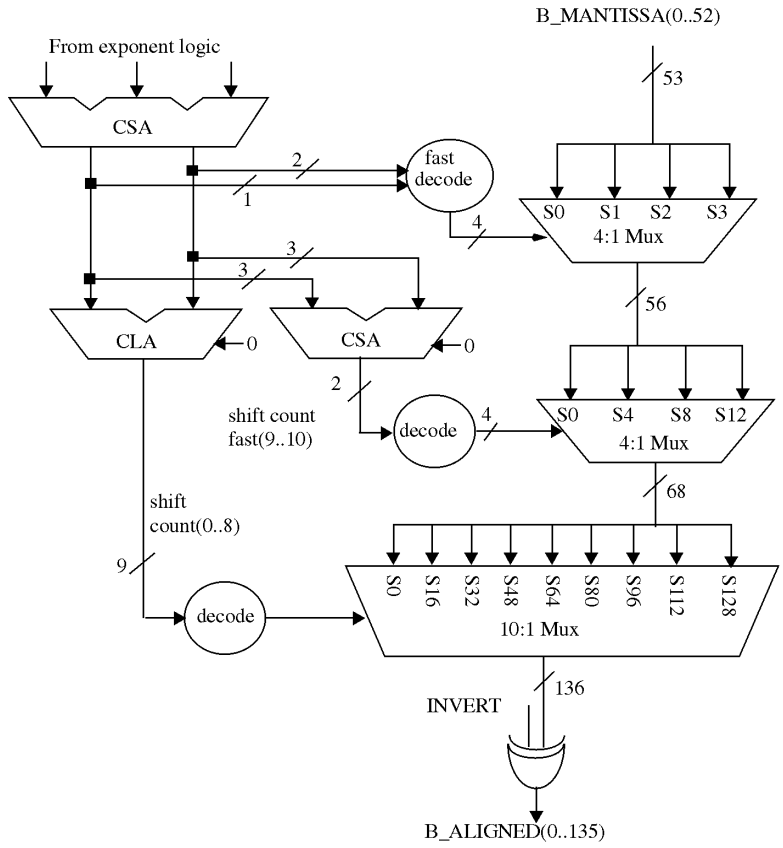


Fig. 9. B alignment shifter.

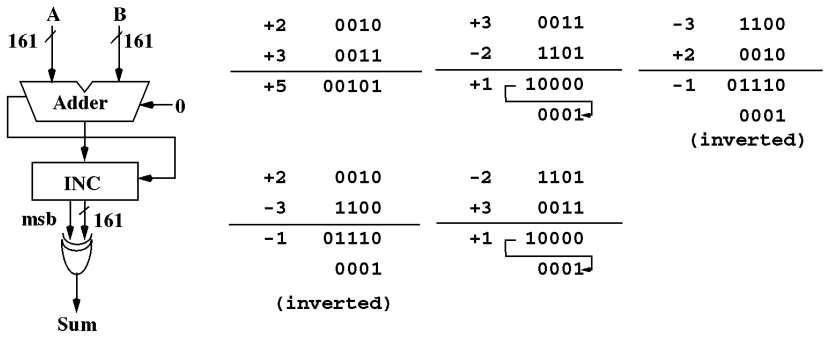


Fig. 10. One's-complement adder examples.

The choice of positioning the logic incorporating the aligned B mantissa is made depending on logic complexity and timing requirements.

Fig. 9 shows a 53-bit input, 136-bit output B alignment shifter that can be used with the dual-pass multiply array as it would apply to the case of Fig. 8b. The exponent logic calculates the shift count by examining the exponents of the A, B, and C operands as discussed in Section 3. The constant 56 is added to the shift count to allow for the initial offset in alignment between the B operand and AC product. In the dual-pass multiplier implementation, the constant is different for the two passes through the array. In the first-pass, the constant 30 is used (56-26) to shift the appropriate low-order 26 bits of the B operand mantissa to the least-significant bit of the 136-bit output of the alignment shifter (these 26 bits would normally have shifted to the lower 26 bits of the 161-bit B aligned mantissa in a single-pass FPU); in the second pass, the standard constant 56 is used.

6 Effect on the Adder

The equivalent of a 161-bit carry propagate adder (CPA) is required to add the sums and carries of the 3:2 CSA level that reduces the B operand mantissa and the sums and carries of the Wallace tree representing the AC product to two values. The adder can be implemented to carry out a 161-bit addition in one's-complement form (with end around carry adjustment for effective-subtraction) [17], as shown in Fig. 10.

Because of the relative positions of the AC product and the 161-bit aligned B operand, the upper 55-bits of the 161-bit aligned B operand mantissa is mathematically being

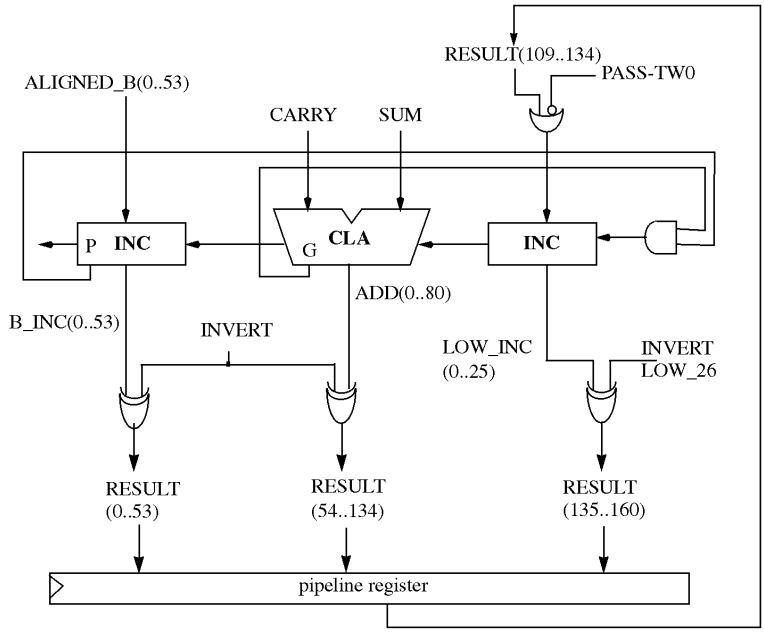


Fig. 11. The 161-bit adder.

added to zero, with a carry-in representing the carry-out of the addition of the lower 106-bits of the aligned B operand mantissa to the AC product. This allows the upper 55-bits of the CPA to be replaced with a 55-bit incrementer whose carry-in is fed by the carry-out of the lower-order 106-bit CPA. In practice, a 54-bit incrementer and a 107-bit CPA are used because the carry partial-product from the Wallace tree is shifted one bit left of the sum partial-product, and the aligned B operand mantissa is added up to the width of the combined overlapped sum and carry partial-products of the Wallace tree. In addition, the incrementer does not wait for the carry-out of the 107-bit CPA; instead, the carry-out of the CPA chooses between the incremented or nonincremented versions of the aligned B operand's upper bits via a carry-select multiplexor.

In the dual-pass multiplier implementation, the lower 26 bits of the final result are generated in the first cycle of the add stage, and are latched to be fed back for the second add cycle. In the second add cycle, the lower 26 bits need only be incremented if an end around carry adjustment is required. This allows a reduction of the width of the 106-bit carry look-ahead adder (CLA) by 26 bits, reducing it to 81-bits. Therefore, in the second add stage, the low-order 26 bits are fed to an incrementer, whose carry output feeds the carry-in of the 81-bit CLA. The modified CPA is shown in Fig. 11.

The reduction in the overall CPA size further reduces the silicon area occupied by the FPU implementing a dual-pass MAF design, and requires a shorter logic delay as compared to a full 106-bit CLA of a single-pass MAF design. It should be noted that there are more efficient methods of implementing the final adder in a parallel multiplier based on signal-arrival profiles [18], [19]. The CPA was chosen for the PowerPC 604e and PowerPC 603e microprocessor FPUs because all the signals feeding the CPA come from the output of latches due to the nature of the pipeline, thereby all having the same arrival times. However, implementations having fewer pipeline stages may take advantage of the methods in [18], [19].

7 Complications Associated with Implementing Dual-Pass Multiply Arrays

This section points out some areas where complexity of design is increased when the FPU of the microprocessor is architected to use a reduced dual-pass multiplier.

7.1 Complication 1

The dual-pass mechanism requires a feedback path to loop back the sum and carry partial-products of the first pass, so that these can be summed along with the addition of the partial-products of the second pass. The 4:2 CSA tree allows 16 partial-products to be added every cycle. Of these, 14 are used in the first pass (with zeroes inputted to the remaining two). In the second pass, 14 partial-products are added (including the zeroed pp27), and the remaining two inputs allow the sum and carry fed back from the first pass to be added by the Wallace tree. Control logic and additional multiplexing is thereby added.

7.2 Complication 2

In the reduced array, pp0 through pp13 are generated in the first pass of the double-precision multiply, and pp14 though pp27 are generated in the second pass. In the full multiplier implementation, the problem of placement of the hot-one of pp26 in the Wallace tree is eliminated by the fact that pp26 is positive. In the dual-pass multiply array implementation, the problem is extended to finding positions in the Wallace tree for the hot-one of pp13 in the first pass, and the hot-one of pp27 in the second pass. There is no hot-one required for pp27; however, pp13 in the first pass could be positive, negative or zero and its hot-one cannot be ignored. This problem is solved by saving the hot-one of pp13 and feeding it back to the multiply array in the second pass to be

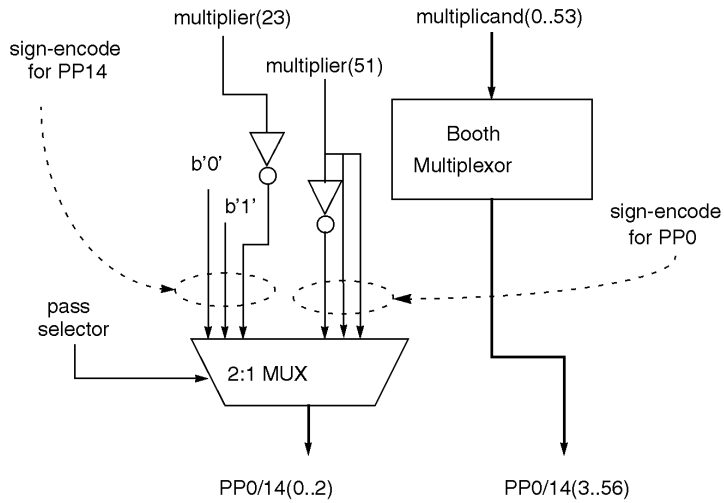


Fig. 12. Sign-encoding PP0 (PP14).

positioned with pp14 (as it normally would have been positioned in the full-multiplier implementation).

This, however, leads to a 2-bit complication in the Wallace tree of 4:2 CSAs. Because the dataflow for pp0 through pp13 is shared with pp14 through pp27 respectively for the two passes of the dual-pass multiplication, the dataflow for the corresponding partial-products of the two passes must match exactly for each individual bit position to prevent the need for additional multiplexing for the dataflow between the two cycles.

Because pp14 is applied to the matrix with the concatenation of the hot-one, pp0 must also have a similar extension to eliminate the need for multiplexing. It can be seen that this problem is solved by adding two extra CSAs in the first group of CSAs in the first level (which have pp0 and pp14 as inputs), and positioning pp0 and pp14 as:

$$pp0 \text{ ———————— } 00$$

$$pp14 \text{ ———————— } 0n_{13}$$

7.3 Complication 3

In the generation of the partial-products, the sign extension used for pp0 involves adding three bits left of bit 0, while pp1 through pp27 are sign extended by adding two bits to the left of bit 0 [9]. However, in the dual-pass multiplier, pp0 and pp14 share the dataflow from the A operand mantissa (one requires a 3-bit sign-extension while the other requires a 2-bit extension, respectively). This necessitates adding a multiplexor for the first three bits of pp0 (pp14), which allows for selecting the sign extend bits needed, depending on whether the first or second pass is being performed and whether the operation is single- or double-precision. This is shown in Fig. 12.

7.4 Complication 4

The first-pass through the multiply array (adding 14 partial-products) produces an 82-bit value. Feeding the entire 82-bits into the second-pass (to be added to the rest of the partial-products) involves larger groups of CSAs because pp14 through pp27 are positioned at least 28 bits left of pp0 through pp13, respectively. This adds 28 CSAs for at least one group of CSAs in each level. The problem is solved by feeding back the upper 56 bits of the first-pass result, while latching the lower 26 bits. These lower 26 bits of the first pass represent the low 26 bits of the 161-bit final result (AC + B) before normalization and rounding, and are not modified by the second pass. The lower 26 bits are represented as the latched feedback path feeding the low-order incrementer in Fig. 11. Again, control logic complexity is increased.

7.5 Complication 5

The B alignment shifter, if implemented as a right-shifter, shifts the B operand mantissa in a 161-bit range. For a dual-pass multiply array, the lower 26 bits of the result of the first-pass is not fed back, but pipelined into the add stage. The B alignment shifter must provide the lower 26 bits of the 161-bit value in the first-pass, and the upper 135-bit value in the second-pass. This reduces the size of the B alignment shifter, but involves extra complications in providing different shift counts for each pass.

The partitioning point of the aligned-B operand mantissa is shown in Fig. 8b, which divides the operand into a lower 26 bits for the first pass and an upper 135 bits for the second pass.

7.6 Complication 6

The sticky bit (used for the round/truncate function involved in producing the final result) is the logical OR of all bits beyond the 53rd bit (for double-precision) in the intermediate result representing AC + B. During the alignment shifting of the B mantissa, all bits of B which are shifted beyond bit 161 are logically ORed into the sticky bit. In the dual-pass multiplier implementation, for a double-precision multiply, the bits of the aligned B mantissa that fall into the sticky bit for the first pass are fed back to be logically ORed into the sticky bit for the second pass.

7.7 Complication 7

The 161-bit sum and carry partial-products to be added in the add stage are provided in two passes (26 low-order bits in the first pass, with the remaining bits in the second-pass).

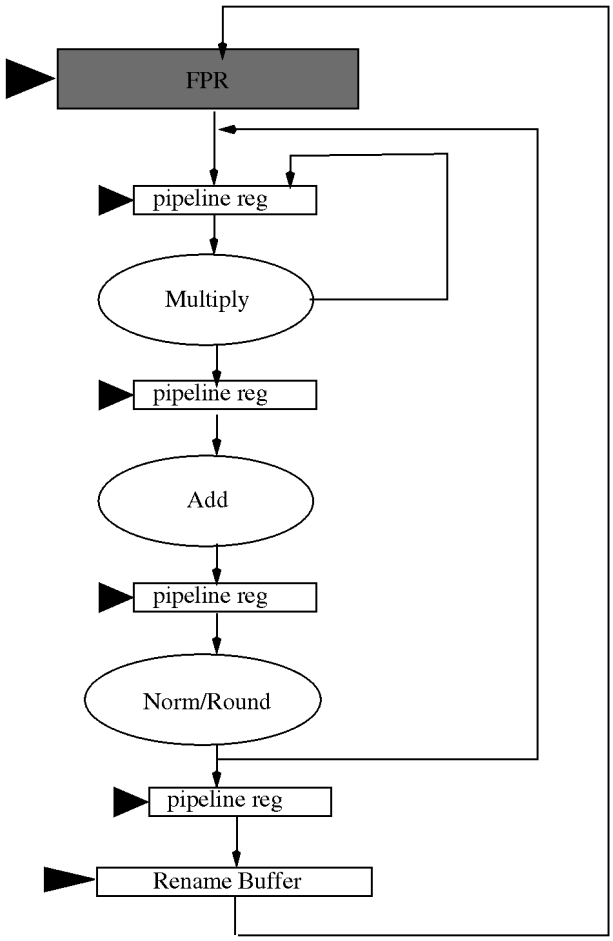


Fig. 13. Major pipeline stages of the dual-pass FPU.

The 161-bit Adder is reduced to a 136-bit adder. The 26 bits of the first pass are latched and transferred to the next pass, where they are fed to a 26-bit incrementer to form the low-order bits of the 161-bit result as shown in Fig. 11 (incrementer implements the end-around carry addition producing the proper sign-magnitude result). The CLA/incrementer combination adds complexity of design over that of a standard sign-magnitude adder.

8 Performance Impact of the Reduced Multiply Array

Fig. 13 shows the typical pipeline stages of an MAF floating-point unit implementing a dual-pass multiplier. This configuration allows single-precision floating-point multiplication to be computed at the rate of one instruction per cycle. For double-precision multiplication, the multiply stage (and the add stage) is exercised twice reducing the rate of operation to one instruction every two cycles. The effect on the throughput for floating-point instructions is as follows:

- Single-pass instructions (nonmultiply instructions and single precision multiplies) have a one cycle throughput;
- Dual-pass instructions (instructions involving double-precision multiply operations) have a two cycle throughput.

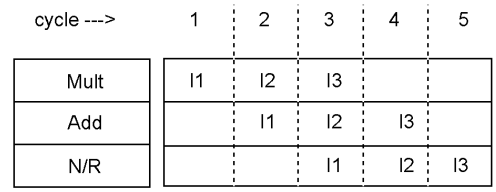


Fig. 14. Single-pass multiply instruction timing.

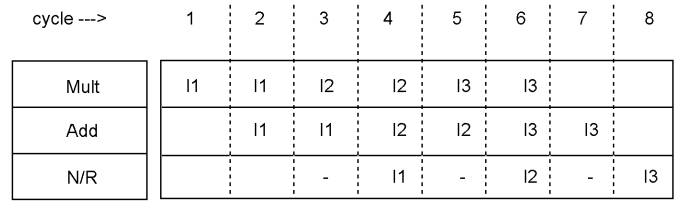


Fig. 15. Dual-pass multiply instruction timing.

Fig. 14 and Fig. 15 show the timing diagram for three consecutive instructions (I1, I2, and I3) for the case of single- and dual-pass instructions, respectively. The first multiply-stage pass of a dual-pass instruction stalls the dispatching of another floating-point instruction for one cycle due to the fact that the multiply stage will be required in the next cycle for further computation of the dual-pass instruction.

The PowerPC 604e microprocessor has a single-pass MAF FPU implementation. It has an estimated SPECfp95 of 6.6. This estimation was based on a system with a 200 MHz PowerPC 604e microprocessor, 1 MB of L2 cache running at 66 Mhz, with a memory bus frequency of 66 MHz. A similarly equipped 200 MHz PowerPC 603e system implemented with a dual-pass MAF FPU yields an estimated SPECfp95 of 3.7. It can be seen that the SPECfp95 performance of the dual-pass implementation is slightly greater than half that of the single-pass implementation.

9 Size Comparisons

The PowerPC 604e microprocessor is implemented in a 0.5mm CMOS technology with an FPU area of 19.5 mm^2 including the floating-point registers. Its multiply array has an area of 5.1 mm^2. The PowerPC 603e microprocessor is implemented in the same technology with an FPU area of 11.7 mm^2 including the floating-point registers. Its multiply array has an area of 2.5 mm^2. The multiply array of the dual-pass implementation occupies about 50 percent of the area of the single-pass array, while the total area of the dual-pass FPU occupies about 60 percent of the area of the single-pass FPU.

10 Summary

Floating-point unit implementations conforming to the IEEE Standard for Binary Floating-point Arithmetic sometimes occupy a major portion of the area on a microprocessor chip, especially those targeted to the portable computer products marketplace. Further complicating the issue of size is the choice of implementing the MAF concept, such as was done in the PowerPC 603e microprocessor [5]. Table 2

TABLE 2
SINGLE- AND DUAL-PASS COMPARISON

	PowerPC 604e (single-pass)	PowerPC 603e (dual-pass)
FPU Area	19.5 mm^2	11.7 mm^2
Multiply Array Area	5.1mm^2	2.5mm^2
Design Tradeoffs	Straightforward design of multiply array, alignment shifter, sticky logic, and adder	Added complexity in: multiply array sign extension, dataflow, and controls; alignment shift-count computation; sticky logic; adder implementation
Estimated SPECfp95	6.6	3.7

shows a comparison of size and performance between the PowerPC 604e and PowerPC 603e microprocessor FPUs.

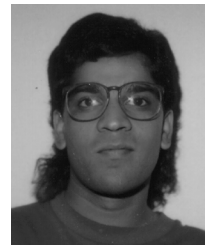
It was shown that the area of the FPU can be reduced by the implementation of a dual-pass multiplier for double-precision multiply and multiply-add operations, while maintaining the single-pass nature for single-precision and nonmultiply oriented operations. However, reduction of size comes at the cost of a degradation of performance and an increase in logic complexity associated with the multiply array, B alignment shifter, sticky logic, and adder implementation.

ACKNOWLEDGMENTS

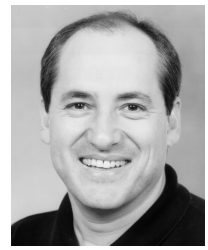
PowerPC 604e and PowerPC 603e are trademarks of the International Business Machines Corporation. Other company, product, and service names may be trademarks or service marks of others.

REFERENCES

[1] IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985.
[2] W.S. Briggs and D.W. Matula, "A 17 × 69 Bit Multiply and Add Unit with Redundant Binary Feedback and Single Cycle Latency," *Proc. 11th IEEE Symp. Computer Arithmetic*, pp. 163-170, June 1993.
[3] E. Hokenek, R.K. Montoye, and P.W. Cook, "Second-Generation RISC Floating Point with Multiply-Add Fused," *IEEE J. Solid-State Circuits*, vol. 25, no. 5, pp. 1,207-1,213, Oct. 1990.
[4] R.K. Montoye, E. Hokenek, and S.L. Runyon, "Design of the IBM RISC System/6000 Floating-Point Execution Unit," *IBM J. Research and Development*, vol. 34, no. 1, pp. 59-70, Jan. 1990.
[5] R. Jessani and C. Olson, "The Floating-Point Unit of the PowerPC 603e," *IBM J. Research and Development*, vol. 40, no. 5, pp. 559-566, Sept. 1996.
[6] A.D. Booth, "A Signed Binary Multiplication Technique," *Quarterly J. Mechanical and Applied Math.*, vol. 4, part 2, 1951.
[7] S. Vassiliadis, E.M. Schwarz, and D.J. Hanrahan, "A General Proof for Overlapped Multiple-Bit Scanning Multiplications," *IEEE Trans. Computers*, vol. 38, no. 2, pp. 172-183, Feb. 1989.
[8] C.S. Wallace, "A Suggestion for Fast Multipliers," *IEEE Trans. Electronic Computers*, vol. 13, pp. 14-17, Feb. 1964.
[9] S. Vassiliadis, E.M. Schwarz, and B.M. Sung, "Hard-Wired Multipliers with Encoded Partial Products," *IEEE Trans. Computers*, vol. 40, no. 11, pp. 1,181-1,197, Nov. 1991.
[10] G. Bewick, "High Speed Multiplication," Electronic Research Seminar, Stanford Univ., 12 Mar. 1993.
[11] L. Dadda, "Some Schemes for Parallel Multipliers," *Alta Frequenza*, vol. 34, pp. 349-356, May 1965.
[12] W.J. Stenzel, W.J. Kubitz, and G.H. Garcia, "Compact High-Speed Parallel Multiplication Scheme," *IEEE Trans. Computers*, vol. 26, no. 10, pp. 948-957, Oct. 1977.
[13] L. Dadda, "Composite Parallel Counters," *IEEE Trans. Computers*, vol. 29, no. 10, pp. 942-946, Oct. 1980.
[14] V.G. Oklobdzija and D. Villeger, "Improving Multiplier Design by Using Improved Column Compression Tree and Optimized Final Adder in CMOS Technology," *IEEE Trans. VLSI Systems*, vol. 3, no. 2, pp. 292-301, June 1995.
[15] V.G. Oklobdzija, D. Villeger, and S.S. Liu, "A Method for Speed Optimized Partial Product Reduction and Generation of Fast Partial Multipliers Using an Algorithmic Approach," *IEEE Trans. Computers*, vol. 45, no. 3, pp. 294-306, Mar. 1996.
[16] C. Martel, V.G. Oklobdzija, R. Ravi, and P.F. Stelling, "Design Strategies for Optimal Multiplier Circuits," *Proc. 12th IEEE Symp. Computer Arithmetic*, pp. 42-49, 1995.
[17] S. Vassiliadis, D.S. Lemon, and M. Putrino, "S/370 Sign-Magnitude Floating-point Adder," *IEEE J. Solid-State Circuits*, vol. 24, no. 4, pp. 1,062-1,070, Aug. 1989.
[18] V.G. Oklobdzija and P. Stelling, "Design Strategies for the Final Adder in a Parallel Multiplier," *Conf. Record 29th Ann. Asilomar Conf. Signals, Systems, and Computers*, vol. 1, pp. 591-595, Oct. 1996.
[19] P. Stelling and V.G. Oklobdzija, "Design Strategies for Optimal Hybrid Final Adders in a Parallel Multiplier," *J.VLSI Signal Processing*, vol. 14, no. 3, 1996.



Romesh M. Jessani received his BE degree in electronics and communication engineering from the Regional Engineering College, Kurukshetra, India, and his MS degree in electrical engineering from the University of Texas at Dallas. After graduation, he worked at Advanced Micro Devices on the 80C51 microcontroller. Subsequently, he joined Motorola to work with IBM on the development of the PowerPC microprocessor architecture and logic design. He worked on various aspects of several PowerPC microprocessors, including the floating-point unit, L1 and L2 cache controllers, array built-in self-test, and computer architecture. Currently, he is involved in the development of a SPARC-V9 microprocessor at Ross Technology, Inc.



Michael Putrino (M'87-SM'88) received the BS degree (cum laude) in electrical engineering from Syracuse University, Syracuse, New York, in 1977. He joined IBM in May 1977 and is currently a development engineer manager at the Somerset Design Center, an IBM, Motorola, and Apple alliance, where he manages a team of core, integer, and floating-point logic designers developing high performance PowerPC microprocessors. His main research interests lie within the scope of computer arithmetic, computer architecture, and hardware design of pipelined computers and parallel RISC microprocessors. He is the inventor of six U.S. patents, some of which have also been issued in France, Germany, the United Kingdom, and Japan, and has 12 addition U.S. patents pending. He is a senior member of the IEEE and a technical member of VHDL International.