18 Floating-Point Operations

"Spurious moral grandeur is generally attached to any formulation computed to large number of decimal places."
DAVID BERLINSKI, ON SYSTEMS ANALYSIS, 1976

In this chapter, we examine hardware implementation issues for the four basic floating-point arithmetic operations of addition, subtraction, multiplication, and division, as well as fused multiply-add. Consideration of square-rooting is postponed to Section 21.6. The bulk of our discussions concern the handling of exponents, alignment of significands, and normalization and rounding of the results. Arithmetic operations on significands, which are fixed-point numbers, have already been covered. Chapter topics include:

18.1 FLOATING-POINT ADDERS/SUBTRACTORS

A floating-point adder/subtractor consists of a fixed-point adder for the aligned significands, plus support circuitry to deal with the signs, exponents, alignment preshift, normalization postshift, and special values (± 0, $\pm\infty$, NaNs, and subnormals). Figure 18.1 is the block diagram of a floating-point adder. The major components of this adder are described in Sections 18.1–18.3. Floating-point multipliers and dividers, which are relatively simpler, are covered in Section 18.4. Implementation of fused-multiply-add units, which perform a multiplication operation followed by an addition as a single elementary operation, is discussed in Section 18.5.

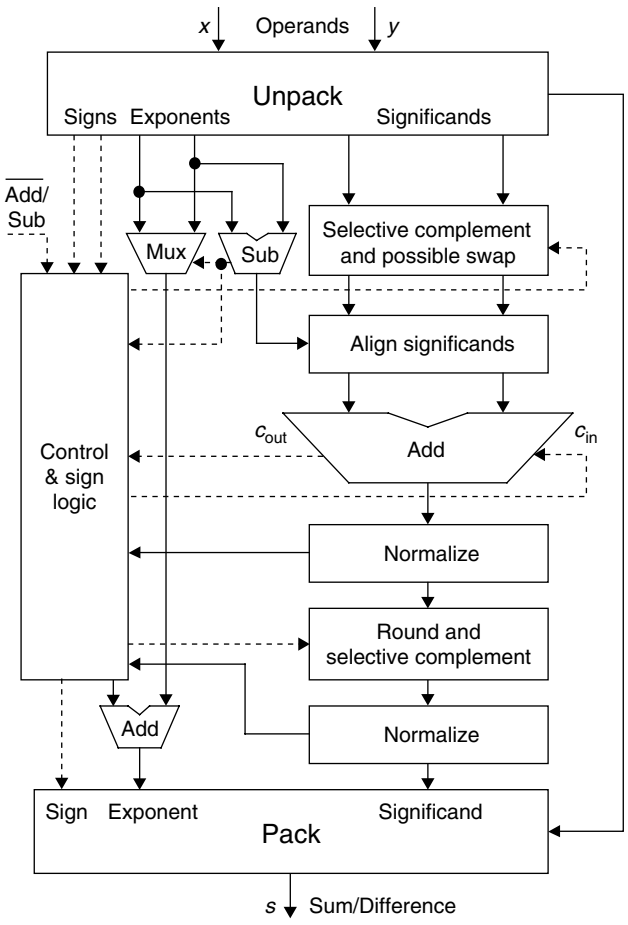

Figure 18.1 Block diagram of a floating-point adder/subtractor.

As shown in Fig. 18.1, the two operands entering the floating-point adder are first unpacked. Unpacking involves:

Separating the sign, exponent, and significand for each operand and reinstating the hidden 1.

Converting the operands to the internal format, if different (e.g., single-extended or double-extended).

Testing for special operands and exceptions (e.g., recognizing not-a-number inputs and bypassing the adder).

We will ignore subnormals throughout our discussion of floating-point arithmetic operations. In fact the difficulty of dealing with subnormals in hardware has led to reliance on software solutions, with their attendant performance penalties. A discussion of floating-point arithmetic with subnormals can be found elsewhere [Schw03].

The difference of the two exponents is used to determine the amount of alignment right shift and the operand to which it should be applied. To economize on hardware,

preshifting capability is often provided for only one of the two operands, with the operands swapped if the other one needs to be shifted. Since the computed sum or difference may have to be shifted to the left in the post normalization step, several bits of the right-shifted operand, which normally would be discarded as they moved off the right end, may be kept for the addition. Thus, the significand adder is typically wider than the significands of the input numbers. More on this in Section 18.3.

Similarly, complementation logic may be provided for only one of the two operands (typically the one that is not preshifted, to shorten the critical path). If both operands have the same sign, the common sign can be ignored in the addition process and later attached to the result. If $-x$ is the negative operand and complementation logic is provided only for y, which is positive, y is complemented and the negative sign of $-x$ ignored, leading to the result $x - y$ instead of $-x + y$. This negation is taken into account by the sign logic in determining the correct sign of the result.

Selective complementation, and the determination of the sign of the result, are also affected by the $\overline{\text{Add}}$/Sub control input of the floating-point adder/subtractor, which specifies the operation to be performed.

With the Institute of Electrical and Electronics Engineers' IEEE 754-2008 standard floating-point format, the sum/difference of the aligned significands has a magnitude in the range [0, 4). If the result is in [2, 4), then it is too large and must be normalized by shifting it 1 bit to the right and incrementing the tentative exponent to compensate for the shift. If the result is in [0, 1), it is too small. In this case, a multibit left shift may be required, along with a compensatory reduction of the exponent.

Note that a positive (negative) 2's-complement number $(x_1 x_0 . x_{-1} x_{-2} \cdots)_{\text{2's-compl}}$ whose magnitude is less than 1 will begin with two or more 0s (1s). Hence, the amount of left shift needed is determined by a special circuit known as *leading zeros/ones counter*. It is also possible, with a somewhat more complex circuit, to *predict* the number of leading zeros/ones in parallel with the addition process rather than detecting them after the addition result becomes known. This removes the leading zeros/ones detector from the critical path and improves the overall speed. Details are given in Section 18.2.

Rounding the result may necessitate another normalizing shift and exponent adjustment. To improve the speed, adjusted exponent values can be precomputed and the proper value selected once the normalization results become known. To obtain a properly rounded floating-point sum or difference, a binary floating-point adder must maintain at least three extra bits beyond the *ulp*; these are called *guard bit*, *round bit*, and *sticky bit*. The roles of these bits, along with the hardware implementation of rounding, are discussed in Sections 18.3.

The significand adder is almost always a fast logarithmic time 1's- or 2's-complement adder, usually with carry-lookahead design. When the resulting significand is negative, it must be complemented to form the signed-magnitude output. As usual, 2's-complementation is done by 1's-complementation and addition of *ulp*. The latter addition can be merged with the addition of *ulp*, which may be needed for rounding. Thus, 0, *ulp*, or 2*ulp* will be added to the true or complemented output of the significand adder during the rounding process.

Finally, packing the result involves:

Combining the sign, exponent, and significand for the result and removing the hidden 1.

Testing for special outcomes and exceptions (e.g., zero result, overflow, or underflow).

Note that unlike the unpacking step, conversion between the internal and external formats is not included in the packing process. This is because converting a wider significand to a narrower one requires rounding and is best accomplished in the rounding stage, which produces the result with the desired output precision.

Floating-point adders found in various processors may differ in details from the generic design depicted in Fig. 18.1. However, the basic principles are the same, and the differences in implementation relate to clever schemes for speeding up the various subcomputations through overlapping or merging, or for economizing on hardware cost. Some of these techniques are covered in Sections 18.2 and 18.3.

18.2 PRE- AND POSTSHIFTING

The preshifter always shifts to the right by an amount equal to the difference of the two exponents. Note that with the IEEE 754-2008 short format, the difference of the two exponents can be as large as $127 - (-126) = 253$. However, even with extra bits of precision maintained during addition, the operands and results are much narrower than 253 bits. This allows us to simplify and speed up the exponent subtractor and preshift logic in Fig. 18.1.

For example, if the adder is 32 bits wide, then any preshift of 32 bits or more will result in the preshifted input becoming 0. Thus, only the least significant 5 bits of the exponent difference need to be computed, with the preshifted input forced to 0 when the difference is 32 or more.

Let us continue with the assumption that right shifts of 0 to 31 bits must be implemented. In principle, this can be done by a set of 32-to-1 multiplexers (muxes), as shown in Fig. 18.2. The multiplexer producing the bit y_i of the shifted operand selects one of the bits x_i through x_{i+31} of the (sign-extended) 32-bit input that is being aligned based on the 5-bit shift amount. Such a design, however, would lead to fan-in and fan-out problems, especially for the sign bit, which will have to feed multiple inputs of several multiplexers.

As usual, a multistage design can be used to mitigate the fan-in and fan-out problems. Figure 18.3 shows a portion of a combinational shifter that can preshift an input operand x by any amount from 0 to 15 bits. Each circular node is a 2-to-1 multiplexer, with its output fanned out to two nodes in the level below. The four levels, from top to bottom, correspond to shifting by 1, 2, 4, and 8 bits, respectively.

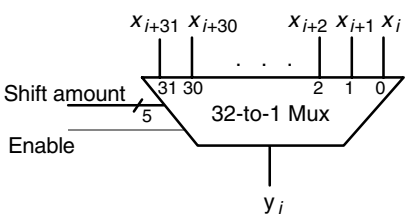

Figure 18.2 A 1-bit slice of a single-stage preshifter.

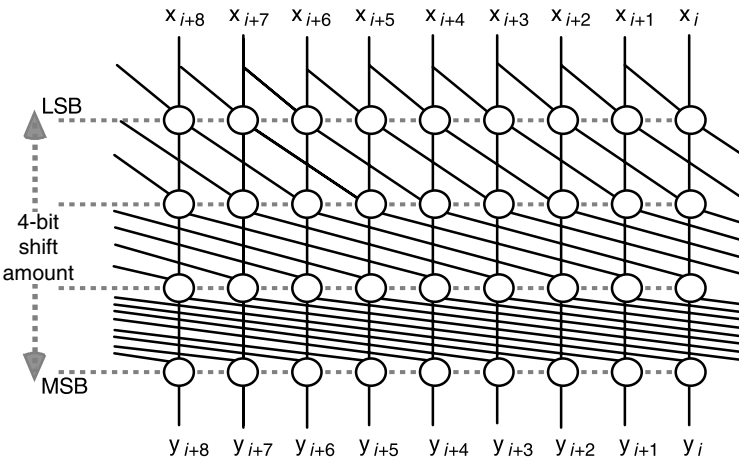

Figure 18.3 Four-stage combinational shifter for preshifting an operand by 0 to 15 bits.

In practice, designs that fall between the two extremes shown in Figs. 18.2 and 18.3 are used. For example, preshifts of up to 31 bits might be implemented in two stages, one performing any shift from 0 to 7 bits and the other performing shifts of 0, 8, 16, and 24 bits. The first stage is then controlled by the three least-significant bits (LSBs), and the second stage by the two most-significant bits (MSBs), of the binary shift amount.

Note that the difference $e1 - e2$ of the two (biased) exponents may be negative. The sign of the difference indicates which operand is to be preshifted, while the magnitude provides the shift amount. One way to obtain the shift amount in case of a negative difference is to complement it. However, this introduces additional delay due to carry propagation. A second way is to use a ROM table or programmable logic array that receives the signed difference as input and produces the shift amount as output. A third way is to compute both $e1 - e2$ and $e2 - e1$, choosing the positive value as the shift amount. Given that only a few bits of the difference need to be computed, duplicating the exponent subtractor does not have significant cost implications.

The postshifter is similar to the preshifter with one difference: it should be able to perform either a right shift of 0–1 bit or a left shift of 0–31 bits, say. One hardware implementation option is to use two separate shifters for right- and left-shifting. Another option is to combine the two functions into one multistage combinational shifter. Supplying the details in the latter case is left as an exercise.

For IEEE 754-2008 operands, the need for right-shifting by 1 bit during normalization is indicated by the magnitude of the adder output equaling or exceeding 2. Suppose the adder output is a 2's-complement number in the range $(-4, 4)$, represented as $z = (c_{\text{out}} z_1 z_0 . z_{-1} z_{-2} \cdots)_{\text{2's-compl}}$. The condition for right-shifting in this case is easily determined as $c_{\text{out}} \neq z_1$. Assuming that right-shifting is not needed for normalization, we must have $c_{\text{out}} = z_1$, with the left-shift amount then determined by the number of consecutive bits in z that are identical to z_1. So, if $z_1 = 0$ (1), we need to detect the number of consecutive 0s (1s) in z, beginning with z_0. As mentioned in Section 18.1, this is done either by applying a leading zeros/ones counter to the adder output or

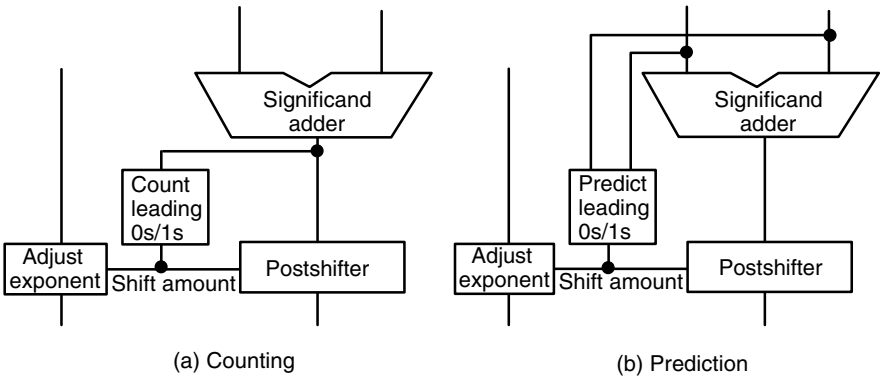

Figure 18.4 Leading zeros/ones counting versus prediction.

by predicting the number of leading zeros/ones concurrently with the addition process (to shorten the critical path). The two schemes are depicted in Fig. 18.4.

Leading zeros/ones counting is quite simple and is thus left as an exercise. Predicting the number of leading zeros/ones can be accomplished as follows. Note that when the inputs to a floating-point adder are normalized, normalization left shift is needed only when the operands, and thus the inputs to the significand adder, have unlike signs. Leading zeros/ones prediction for unnormalized inputs is somewhat more involved, but not more difficult conceptually.

Let the inputs to the significand adder be 2's-complement positive and negative values $(0x_0.x_{-1}x_{-2}\cdots)_{\text{2's-compl}}$ and $(1y_0.y_{-1}y_{-2}\cdots)_{\text{2's-compl}}$. Let there be exactly i consecutive positions, beginning with position 0, that propagate the carry during addition. Borrowing the carry "generate," "propagate," and "annihilate" notation from our discussions of adders in Section 5.6, we have the following:

$$p_0 = p_{-1} = p_{-2} = \cdots = p_{-i+1} = 1$$
$$p_{-i} = 0 \qquad (\text{i.e., } g_{-i} = 1 \text{ or } a_{-i} = 1)$$

In case $g_{-i} = 1$, let j be the smallest index such that:

$$g_{-i} = a_{-i-1} = a_{-i-2} = \cdots = a_{-j+1} = 1$$
$$a_{-j} = 0 \qquad (\text{i.e., } g_{-j} = 1 \text{ or } p_{-j} = 1)$$

Then, we will have j or $j - 1$ leading 0s depending on whether the carry emerging from position j is 0 or 1, respectively.

In case $a_{-i} = 1$, let j be the smallest index such that

$$a_{-i} = g_{-i-1} = g_{-i-2} = \cdots = g_{-j+1} = 1$$
$$g_{-j} = 0 \qquad (\text{i.e., } p_{-j} = 1 \text{ or } a_{-j} = 1)$$

Then, we will have $j - 1$ or j leading 1s, depending on whether the carry-out of position j is 0 or 1, respectively.

Note that the g, p, a, and carry signals needed for leading zeros/ones prediction can be extracted from the significand adder to save on hardware. Based on the preceding discussion, given the required signals, the circuit needed to predict the number of leading zeros/ones can be designed with two stages. The first stage, which is similar to a carry-lookahead circuit, produces a 1 in the jth position and 0s in all positions to its left (this can be formulated as a parallel prefix computation, since we are essentially interested in detecting one of the four patterns $pp \cdots ppgaa \cdots aag$, $pp \cdots ppgaa \cdots aap$, $pp \cdots ppagg \cdots gga$, or $pp \cdots ppagg \cdots ggp$). The second stage is an encoder or priority encoder (depending on the design of the first stage) that yields the index of the leading 1.

Finally, in the preceding discussion, we assumed separate hardware for pre- and postshifting. This is a desirable choice for higher-speed or pipelined operation. If the two shifters are to be combined for economy, the unit must be capable of shifting both to the right and to the left by an arbitrary amount. Modifying the design of Fig. 18.3 to derive a bidirectional shifter is straightforward.

18.3 ROUNDING AND EXCEPTIONS

If an alignment preshift is performed, the bits that are shifted out should not all be discarded, since they can potentially affect the rounding of the result. Recall that proper floating-point addition/subtraction requires that the result matches what would be obtained if the computation were performed with infinite precision and the result rounded. It may thus appear that we have to keep all bits that are shifted out in case left-shifting is later needed for normalization. Keeping all the bits that are shifted out effectively doubles the width of the significand adder.

We know from earlier discussions that the significand adder must be widened by 1-bit at the left to accommodate the sign bit of its 2's-complement inputs. It turns out that widening the adder by 3 bits at the right is adequate for obtaining properly rounded results. Calling the three extra bits at the right G, R, and S, for reasons to become apparent shortly, the output of the significand adder can be represented as follows:

$$\text{Adder output} = (c_{\text{out}} z_1 z_0 . z_{-1} z_{-2} \cdots z_{-l} GRS)_{\text{2's-compl}}$$

In the preceding equation, z_1 is the sign indicator, c_{out} represents significand overflow, and the extra bits at the right are

G : Guard bit

R : Round bit

S : Sticky bit

We next explain the roles of the G, R, and S bits and why they are adequate for proper rounding. The explanation is in terms of the IEEE 754-2008 binary floating-point format, but it is valid in general.

When an alignment right-shift of 1 bit is performed, G will hold the bit that is shifted out and no precision is lost (so, G "guards" against loss of precision). For alignment right shifts of 2 bits or more, the shifted significand will have a magnitude in [0, 1/2). Since the magnitude of the unshifted significand is in [1, 2), the difference of the aligned significands will have a magnitude in [1/2, 2). Thus, in this latter case, the normalization left shift will be by at most 1 bit, and G is still adequate to protect us against loss of precision.

In case a normalization left shift actually takes place, the "round bit" is needed for determining whether to round the resulting significand down ($R = 0$, discarded part $< ulp/2$) or up ($R = 1$, discarded part $\geq ulp/2$). All that remains is to establish whether the discarded part is exactly equal to $ulp/2$. This information is needed in some rounding schemes, and providing it is the role of the "sticky bit," which is set to the logical OR of all the bits that are shifted through it. Thus, following an alignment right shift of 7 bits, say, the sticky bit will be set to the logical OR of the 5 bits that move past G and R. This logical ORing operation can be accommodated in the design of the preshifter (how?).

The effect of 1-bit normalization shifts on the rightmost few bits of the significand adder output is as follows

Before postshifting (z)	$\cdots$	z_{-l+1}	z_{-l}	G	R	S
1-bit normalizing right-shift	$\cdots$	z_{-l+2}	z_{-l+1}	z_{-l}	G	$R \vee S$
1-bit normalizing left-shift	$\cdots$	z_{-l}	G	R	S	0
After normalization (Z)	$\cdots$	Z_{-l+1}	Z_{-l}	Z_{-l-1}	Z_{-l-2}	Z_{-l-3}

where the Z_h are the final digit values in the various positions, after any normalizing shift has been applied. Note that during a normalization right shift, the new value of the sticky bit is set to the logical OR of its old value and the value of R. Given a positive normalized result Z, we can round it to the nearest even by simply dropping the extra 3 bits and

Doing nothing	if $Z_{-l-1} = 0$ or $Z_{-l} = Z_{-l-2} = Z_{-l-3} = 0$
Adding $ulp = 2^{-l}$	otherwise

Note than no rounding is necessary in the case of a multibit normalizing left shift, since full precision is preserved in this case (the sticky bit must be zero). Other rounding modes can be implemented similarly.

Overflow and underflow exceptions are easily detected by the exponent adjustment adder near the bottom of Fig. 18.1. Overflow can occur only when we have a normalizing right shift, while underflow is possible only with normalizing left shifts. Exceptions involving not-a-numbers and invalid operations are handled by the unpacking and packing blocks in Fig. 18.1. One remaining issue is the detection of a zero result and encoding it as the all-zeros word. Note that detection of a zero result is essentially a by-product of the leading zeros/ones detection discussed earlier. Determining when the "inexact" exception must be signaled is left as an exercise.

As discussed earlier in this section, a preshift of 2 bits or more rules out the possibility of requiring a multiple-bit postshift to remove leading 0s or 1s. Very fast floating-point adder designs take advantage of this property in a dual-path arrangement. When the exponents differ by no more than 1 (the inputs are close to each other in order of

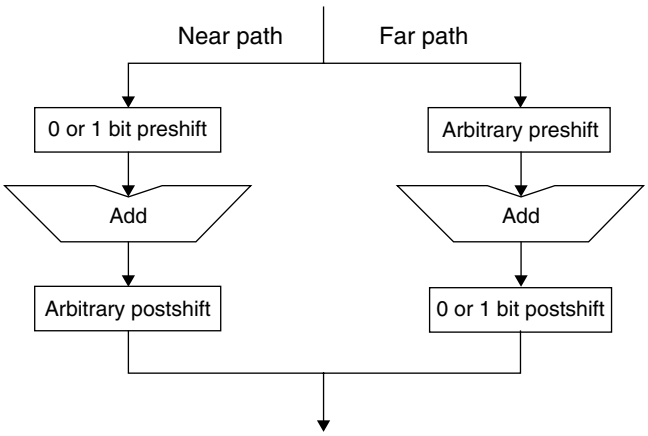

Figure 18.5 Conceptual view of significand handling in a dual-path floating-point adder.

magnitude), the significands are routed to the "near" data path that contains a leading 0s/1s predictor and a full postshifter; in fact, one can even avoid rounding in this path. Otherwise, that is, with a preshift of 2 bits or more, the "far" datapath with a full preshifter and simple postshifter is employed. Note that the 1-bit normalizing right shift that may be required can be combined with rounding. A block diagram of such a dual-path floating-point adder is shown in Fig. 18.5. Description of a modern processor that uses this approach can be found elsewhere [Nain01].

18.4 FLOATING-POINT MULTIPLIERS AND DIVIDERS

A floating-point multiplier consists of a fixed-point multiplier for the significands, plus peripheral and support circuitry to deal with the exponents and special values (± 0, $\pm \infty$, NaNs and subnormals). Figure 18.6 depicts a generic block diagram for a floating-point multiplier. The role of unpacking is exactly as discussed for floating-point adders at the beginning of Section 18.1. Similarly, the final packing of the result is done as for floating-point adders. The sign of the product is obtained by XORing the signs of the two operands.

A tentative exponent is computed by adding the two biased exponents and subtracting the bias from the sum. With the IEEE 754-2008 short format, subtracting the bias of 127 can be easily accomplished by providing a carry-in of 1 into the exponent adder and subtracting 128 from the sum. This latter subtraction amounts to simply flipping the MSB of the result.

The significand multiplier is the slowest and most complex part of the unit shown in Fig. 18.6. With the IEEE 754-2008 binary format, the product of the two unsigned significands, each in the range $[1, 2)$, will be in the range $[1, 4)$. Thus, the result may have to be normalized by shifting it one position to the right and incrementing the tentative exponent. Rounding the result may necessitate another normalizing shift and exponent

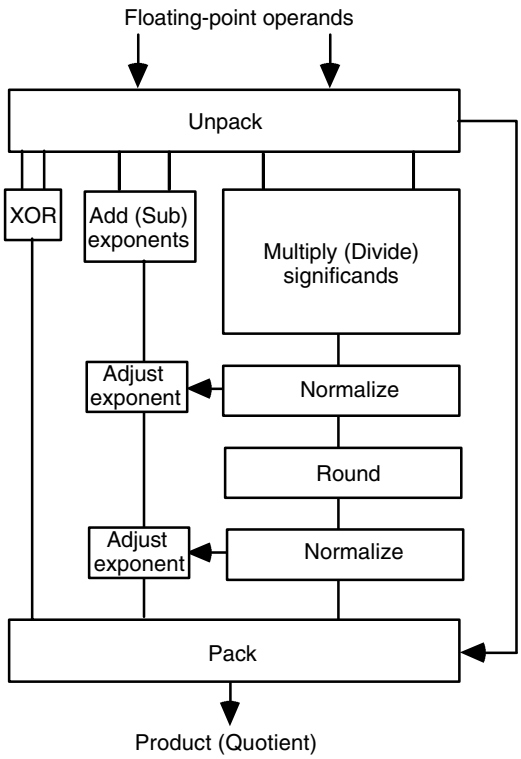

Figure 18.6 Block diagram of a floating-point multiplier (divider).

adjustment. When each significand has a hidden 1 and l fractional bits, the significand multiplier is an unsigned $(l+1) \times (l+1)$ multiplier that would normally yield a $(2l+2)$-bit product. Since this full product must be rounded to $l+1$ bits at the output, it may be possible to discard the extra bits gradually as they are produced, rather than in a single step at the end. All that is needed is to keep an extra round bit and a sticky bit to be able to round the final result properly. Keeping a guard bit is not needed here (why?).

To improve the speed, the incremented exponent can be precomputed and the proper value selected once it is known whether a normalization postshift is required. Since multiplying the significands is the most complex part of floating-point multiplication, there is ample time for such computations. Also, rounding need not be a separate step at the end. With proper design, it may be possible to incorporate the bulk of the rounding process in the multiplication hardware.

To see how, note that most multipliers produce the least-significant half of the product earlier than the rest of the bits. So, the bits that will be used for rounding are produced early in the multiplication cycle. However, the need for normalization right shift becomes known at or near the end. Since there are only two possibilities (no postshift or a right shift of 1 bit), we can devise a stepwise rounding scheme by developing two versions of the rounded product and selecting the correct version in the final step. Alternatively, rounding can be converted to truncation through the injection of corrective terms during multiplication [Even00].

Because floating-point multiplication consists of several sequential stages or subcomputations, it is quite simple and natural to pipeline it for increased throughput. Pipeline latches can be inserted across the natural block boundaries in Fig. 18.6 as well as within the significand multiplier if the latter is of the full-tree or array variety. Chapter 25 presents a detailed discussion of pipelining considerations and design methods.

A floating-point divider has the same overall structure as a floating-point multiplier (Fig 18.6). The two operands of floating-point division are unpacked, the resulting components pass through several computation steps, and the final result is packed into the appropriate format for output. Unpacking and packing have the same roles here as those discussed for floating-point adders in Section 18.1 (the divide-by-0 exception is detected during unpacking). The sign of the quotient is obtained by XORing the operand signs.

A tentative exponent is computed by subtracting the divisor's biased exponent from the dividend's biased exponent and adding the bias to the difference. With the IEEE 754-2008 short format, the bias of 127 must be added to the difference of the two exponents. Since adding 128 is simpler than adding 127, we can compute the difference less one by holding c_{in} to 0 in a 2's-complement subtraction (normally, in 2's-complement subtraction, $c_{in} = 1$) and then flipping the MSB of the result.

The significand divider is the slowest and the most complex part of the unit shown in Fig. 18.6. With the IEEE 754-2008 format, the ratio of two significands in [1, 2) is in the range (1/2, 2). Thus, the result may have to be normalized by shifting it one position to the left and decrementing the tentative exponent. Rounding the result may necessitate another normalizing shift and exponent adjustment.

As in the case of multiplication, speed can be gained by precomputing the adjusted exponent and selecting the proper value when the need for normalization becomes known. Since dividing the significands is the most complex part of floating-point division, there is ample time for such computations. Considerations for pipelining of the computations are also quite similar to those of floating-point multiplication.

One main difference between floating-point division and multiplication is in rounding. Since the significand divider's output may have to be left-shifted by 1 bit for normalization, the quotient must be developed with an extra 2 bits that serve as the guard and round bits (see the discussion of rounding for floating-point addition in Section 18.3). In division schemes that produce a remainder, the final remainder is used to derive the value of the sticky bit (how?). Then, the rounding process discussed at the end of Section 18.3 is applied. Convergence division creates some difficulty for rounding in view of the absence of a remainder.

As was the case for fixed-point multipliers and dividers, floating-point multipliers and dividers can share much hardware. In particular, when the significand division is performed by one of the convergence methods discussed in Chapter 16, little additional hardware is required to convert a floating-point multiplier into a floating-point multiply/divide unit.

18.5 FUSED-MULTIPLY-ADD UNITS

Fused-multiply-add (FMA) operation, that is, computing $p = ax+b$ is a natural operation for direct hardware implementation, once we move beyond the four basic arithmetic

operations. Fused multiply-add is useful in two very common computation sequences, namely, polynomial evaluation and vector dot product. Polynomial evaluation, based on Horner's rule, uses the iteration

$$s := sz + c(j) \qquad \text{for } j \text{ from } n-1 \text{ down to } 0$$

where $c^{(j)}$ is a coefficient of the polynomial $f(z) = c^{(n-1)}z^{n-1} + c^{(n-2)}z^{n-2} + \cdots + c^{(1)}z + c^{(0)}$ and the running total s is initialized to 0. Similarly, the dot product of the n-vectors u and v, with their elements indexed from 0 to $n-1$, can be evaluated via

$$s := s + u^{(j)}v^{(j)} \qquad \text{for } j \text{ from 0 upto } n-1$$

beginning with $s = 0$.

The simplest way to implement an FMA unit to compute $ax + b$ is to cascade a floating-point multiplier, that keeps its entire double-width product of the significands of a and x, with a double-width floating-point adder. However, we can do substantially better in terms of latency if we opt for an optimized merged implementation. One simple optimization is to build the multiplier to keep its product in carry-save form, thus avoiding the final carry-propagate portion of the multiplication algorithm. This makes the addition process only slightly slower, because it now involves a three-operand addition (a carry-save adder level, followed by a conventional fast adder).

Figure 18.7 shows the block diagram of an FMA unit with the aforementioned optimization and two other enhancements. The first of these enhancement concerns the

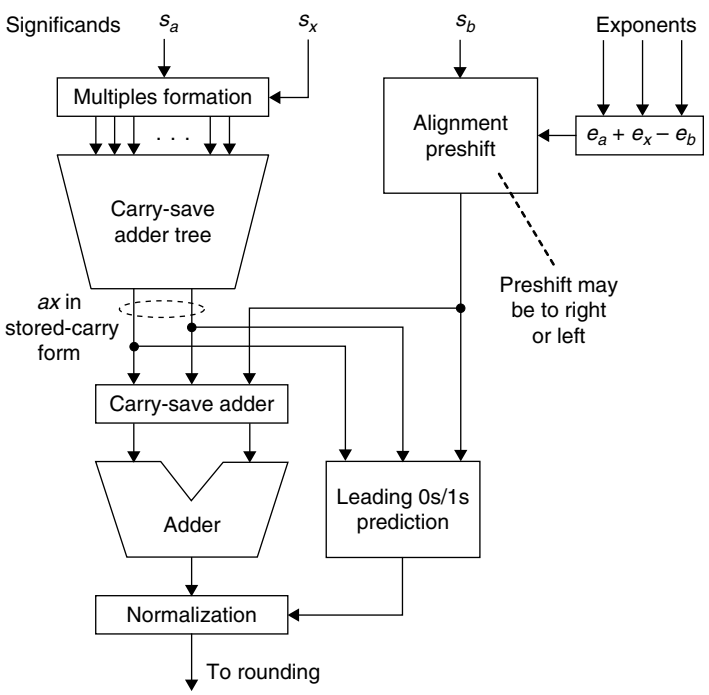

Figure 18.7 Block diagram of a fast FMA unit.

alignment preshift. In floating-point addition, we normally shift the significand of the operand with the smaller exponent. In the design shown in Fig. 18.7, we always shift the significand s_b of b. When b has a larger exponent than that of ax, it must be preshifted to the left for proper alignment. The preshifted version of s_b is taken to be a triple-width number, to accommodate the extra positions created by the preshift in either direction. To understand why the preshifted version of s_b need not be more than $3k$ bits wide, where k is the precision of the input significands, consider the two cases of right and left preshifts. If the right-shift amount for s_b is more than k bits, all the bits shifted past the kth extra bit to the right can be summarized into a sticky bit for use in rounding (see Section 18.3). Similarly, if the left-shift amount for s_b is more than k bits, the product $s_a s_x$ is very small and will only affect the final result via the rounding of s_b. So, in either case, no more than k extra bits are created as a result of right/left preshift.

The second optimization is to perform leading 0s/1s prediction based on the stored-carry representation of $s_a s_x$ and the preshifted version of s_b. Three-input leading 0s/1s prediction is slightly slower than the two-input version depicted in Fig. 18.4b. However, this circuit is likely to be faster than the carry-propagate adder anyway. Besides, we also have a little more leeway here, given the extra carry-save adder level before the carry-propagate adder.

The resulting design in Fig. 18.7 has a latency comparable to that of a floating-point multiplier, in either the single-stage version shown or a two-stage pipelined version with latches inserted after the carry-save adder tree and the alignment preshifter. Thus, it is quite feasible to forego the inclusion of separate adder and multiplier circuits in a floating-point arithmetic unit, using instead the FMA unit for these operations (by setting $x = 1$ for addition and $b = 0$ for multiplication).

18.6 LOGARITHMIC ARITHMETIC UNIT

As discussed in Section 17.6, representing numbers by their signs and base-b logarithms offers the advantage of simple multiplication and division, in as much as these operations are converted to addition and subtraction of the logarithms, respectively. In this section, we demonstrate the algorithms and hardware needed for adding and subtracting logarithmic numbers and present the design of a complete logarithmic arithmetic unit.

We noted, in Section 17.6, that addition and subtraction of logarithmic numbers can, in principle, be performed by table lookup. One method of reducing the size of the required table is via converting the two-operand (binary) operation of interest to a single-operand (unary) operation that needs a smaller table. Consider the add/subtract operation

$$(Sx, Lx) \pm (Sy, Ly) = (Sz, Lz)$$

for logarithmic operands and assume $x > y > 0$ (other cases are similar). Then

$$\begin{aligned} Lz = \log z &= \log(x \pm y) = \log(x(1 \pm y/x)) \\ &= \log x + \log(1 \pm y/x) \end{aligned}$$

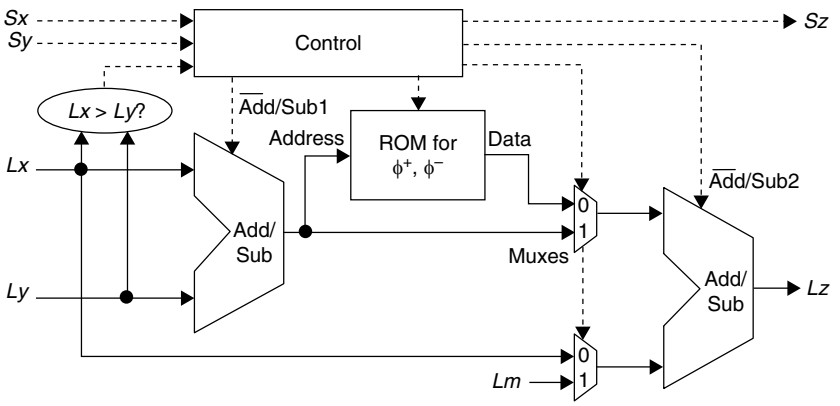

Figure 18.8 Arithmetic unit for a logarithmic number system.

Note that $\log x$ is known and $\log(y/x)$ is easily computed as $\Delta = -(\log x - \log y)$. Given Δ, the term

$$\log(1 \pm y/x) = \log(1 \pm \log^{-1} \Delta)$$

is easily obtained by table lookup (two tables, ϕ^+ and ϕ^-, are needed). Hence, addition and subtraction of logarithmic numbers can be based on the following computations:

$$\log(x + y) = \log x + \phi^+(\Delta)$$
$$\log(x - y) = \log x + \phi^-(\Delta)$$

Figure 18.8 depicts a complete arithmetic unit for logarithmic numbers. For addition and subtraction, Lx and Ly are compared to determine which one is larger. This information is used by the control box for properly interpreting the result of the subtraction $Lx - Ly$. The reader should be able to supply the details.

The design of Fig. 18.8 assumes the use of scaling of all values by a multiplicative factor m so that numbers between 0 and 1 are also represented with unsigned logarithms. Because of this scaling, the logarithm of the scale factor m (or the constant bias Lm) must be subtracted in multiplication and added in division. Thus for addition/subtraction, the first adder/subtractor performs the subtraction $Lx - Ly$ and the second one adds Lx to the value read out from the ROM table. In multiplication (division), the first adder computes $Lx + Ly(Lx - Ly)$ and the second one subtracts (adds) the bias Lm.

PROBLEMS

18.1 Exponent arithmetic in floating-point adder

a. Design the "Subtract exponents" block of the floating-point adder in Fig. 18.1 for the IEEE 754-2008 64-bit floating-point format. Assume that a 6-bit difference, plus a "force to 0" output, is to be provided.

b. Repeat part a, this time assuming that the output difference is to be forced to 63 if the real difference exceeds 63.

c. Compare the designs of parts a and b and discuss.

18.2 Sign logic in floating-point adder

Consider the "Sign logic" block in the floating-point adder of Fig. 18.1.

a. Explain the role of the output from this block that is fed to the "Normalize" and "Adjust exponent" blocks.
b. Supply a complete logic design for this block, assuming the use of a 2's-complement significand adder.

18.3 Alignment preshifter

Design an alignment preshifter for IEEE 754-2008 short format that produces a shifted output with guard, round, and sticky bits.

18.4 Precision in floating-point adders

Referring to the discussion at the beginning of Section 18.3, why would the width of the significand adder double if we were to keep all the bits that are shifted out during the alignment preshift? In other words, doesn't the presence of 0s in those extra positions of the unshifted operand mean that the addition width will not change? Of course, the same question applies when we keep only an extra 3 bits of precision. Do we really have to extend the adder width by 3 bits? *Hint:* The answer depends on which operand is complemented.

18.5 Leading zeros/ones counter

a. Design a ripple-type leading zeros/ones counter for the normalization stage of floating-point addition and derive its worst-case delay. Is this a viable design?
b. Show that the problem of leading zeros/ones detection can be converted to parallel prefix logical AND.
c. Using the result of part b, design a logarithmic time, leading zeros/ones counter.

18.6 Leading zeros/ones counter

a. Use a programmable logic array to design an 8-input leading zeros/ones counter with the following specifications: eight data inputs, two control inputs, three address (index) outputs, and one "all-zeros/ones" output. One of the control inputs specifies whether leading 0s or leading 1s should be counted. The other control input turns the tristate drivers of the address outputs on or off, thus allowing the address outputs of several modules to be tied together. The tristate drivers are also turned off when the "all-zeros/ones" output is asserted.
b. Show how two leading zeros/ones counters of the type described in part a can be cascaded to form a 16-bit leading zeros/ones counter.
c. Can the cascading scheme of part b be extended to wider inputs (say 24 or 32 bits)?

18.7 Leading zeros/ones prediction

Extend the results concerning leading zeros/ones prediction, presented at the end of Section 18.2, to unnormalized inputs. *Hint:* Consider three separate cases of positive inputs, negative inputs, and inputs with unlike signs.

18.8 Rounding in floating-point operations

a. Extend the round-to-nearest-even procedure for a positive value, given near the end of Section 18.3, to a 2's-complement result Z.
b. Occasionally, when performing double-precision arithmetic, we would like to be able to specify that the result be rounded as if it were a single-precision number, with the single-rounded result then output in double-precision format. Why might such an option be useful, and how can it be implemented?
c. Show how the guard, round, and sticky bits can be used when an "inexact" exception is to be indicated following the rounding process.

18.9 Rounding in floating-point operations

Given that an intermediate 2's-complement result for a floating-point operation with guard, round, and sticky bits is at hand, describe how each of the following rounding schemes can be implemented:

a. Round to nearest away from 0.
b. Round toward 0.
c. Round toward $+\infty$.
d. Round toward $-\infty$.
e. R* rounding (see Fig. 17.9).

18.10 Floating-point multipliers

In multiplying the significands of two floating-point numbers, the lower half of the fractional part is not needed, except to properly round the upper half. Discuss whether, and if so, how, this can lead to simplified hardware for the significand multiplier. Note that the significand multiplier can have various designs (tree, array, built of additive multiply modules, etc.).

18.11 Inner-product computation unit

Having an FMA basic operation allows us to speed up an inner-product computation and to reduce its error. Sketch the design of a hardware unit that is specifically optimized for computing inner products. The unit should allow several products to be computed in sequence, while maintaining a running sum of greater precision. Rounding should be postponed to the very end of the inner-product computation.

18.12 Rounding in floating-point division

a. Explain how the sticky bit needed for properly rounding the quotient of floating-point division is derived from the final remainder.

b. Explain how a properly rounded result might be derived with convergence division.

18.13 On-the-fly rounding in division

To avoid a carry-propagate addition in rounding the quotient of floating-point division, one can combine the rounding process with the on-the-fly conversion of the quotient digits from redundant to conventional binary format [Erce92]. Outline the algorithm and hardware requirements for such an on-the-fly rounding scheme.

18.14 Floating-point operations on subnormals

Based on what you have learned about floating-point add/subtract, multiply, and divide units in this chapter, briefly discuss design complications if subnormal numbers of IEEE 754-2008 were to be accepted as inputs and produced as output.

18.15 Logarithmic arithmetic

Consider a 16-bit sign-and-logarithm number system, using $k = 6$ whole and $l = 9$ fractional bits for the logarithm. Assume that the logarithm base is 2 and that 2's-complement representation is used for negative logarithms.

a. Find the representations of $x = 2.5$ and $y = 3.7$ in this number system.
b. What is the required ROM size for the arithmetic unit of Fig. 18.8?
c. Perform the operations $x + y$ and $x - y$, supplying the needed table entries ϕ^+ and ϕ^-.

18.16 Flexible floating-point processor

Consider a 64-bit floating-point number representation format where the sign bit is followed by a 5-bit "exponent width" field. This field specifies the exponent field as being 0–31 bits wide, the remaining 27–58 bits being a fractional significand with no hidden 1. Do not worry about special values such as $\pm\infty$ or not-a-number.

a. Enumerate the advantages and possible drawbacks of this format.
b. Outline the design of a floating-point adder to add two numbers in this format.
c. Draw a block diagram of a multiplier for flexible floating-point numbers.
d. Briefly discuss any complication in the design of a divider for flexible floating-point numbers.

18.17 Double rounding

Consider the multiplication of two-digit, single-precision decimal values .34 and .78, yielding .2652. If we round this exact result to an internal three-digit, extended-precision format, we get .265, which when subsequently rounded to single precision by means of round-to-nearest-even, yields .26. However, if the exact result were directly rounded to single precision, it would yield .27.

a. Can double rounding lead to a similar problem if we always round up the halfway cases instead of applying round-to-nearest-even?
b. Prove that for floating-point operands x and y with p-bit significands, if $x + y$ is rounded to p' bits of precision ($p' \geq 2p + 2$), a second rounding to p bits of precision will yield the same result as direct rounding of the exact sum to p bits.
c. Show that the claim of part b also holds for multiplication, division, and square-rooting.
d. Discuss the implications of the preceding results for converting the results of double-precision IEEE floating-point arithmetic to single precision.

18.18 Rounding in ternary arithmetic

If we had ternary as opposed to binary computers, radix-3 arithmetic would be in common use today. Discuss the effects of this change on rounding in floating-point arithmetic.

18.19 Floating-point addition

a. Compute the sum of the two floating-point operands $x = +.9988 \times 10^{+09}$ and $y = -.1001 \times 10^{+10}$, represented in decimal format, assuming that there is no guard digit.
b. Repeat part a with a single guard digit.
c. Comment on errors in the results of parts a and b.

18.20 Logarithmic arithmetic unit

In Fig 18.8, the five control signals generated by the control unit are not independent, in the sense that some pairs of signals have identical or complementary values.

a. How many independent control signals is the control unit required to produce?
b. Design a combinational circuit to produce the control signals identified in part a.

18.21 Floating-point division via reciprocation

We noted in Chapter 16 that performing the division z/d via the multiplication $z \times (1/d)$ is highly beneficial when several numbers must be divided by the same divisor d. An optimizing compiler may detect this situation and issue the appropriate instructions to take advantage of the common divisor. Argue that optimizations are possible even when a single division by d is to be performed. *Hint:* the two parameters z and d may not become available at the same time.

18.22 Residue logarithmic number system

It has been suggested that the benefits of logarithmic and residue number representation can be combined by representing a discrete version of the logarithm of a number in residue number system [Arno05]. Study this class of number

representation systems and present your findings in a two-page report, focusing on advantages and potential implementation problems.

18.23 Lookup table in logarithmic arithmetic unit

Plot the functions ϕ^+ and ϕ^- on graph paper for values of Δ in the range $[-10, 0]$. Based on your graph, comment on simplifications and table size reductions that might be possible so as to allow the use of wider words in logarithmic arithmetic.

18.24 Rounding in floating-point division

Prove the following theorem about floating-point division, attributed to William Kahan, after establishing the lemmas that precede it. The width of a floating-point number is the number of bits in the significand $x_0.x_{-1}x_{-2}\ldots x_{-l}$ needed for its exact representation (one more than the index of the rightmost 1 in the significand). Assume that the radix r of the representation is a prime number throughout.

a. Lemma: The product of two floating-point numbers of widths u and v has width $u + v - 1$.
b. Lemma: The exact ratio of two floating-point numbers of width w or less cannot have a width greater than w.
c. Lemma: If the ratio of two positive integers is nonterminating and the divisor is in $[r^{j-1}, r^j - 1]$ for some $j > 0$, no more than $j - 1$ consecutive 0 (or $r - 1$) digits can appear in the result.
d. Theorem: In binary floating-point with precision p, if a quotient is approximated to $2p + 2$ bits, with an error of less than one unit in position $2p + 2$, the approximation has at least p significant bits.

18.25 Logarithmemic arithmetic

Consider an 8-bit unsigned logarithmic number system in which the base-2 logarithm is represented with $k = 3$ whole and $l = 5$ fractional bits.

a. Represent the numbers $x = 7$ and $y = 11$ as accurately as possible in this number system.
b. Compute the representation of the product $p = x \times y$ and analyze its accuracy.
c. Compute the representation of the sum $s = x + y$ and analyze its accuracy.

18.26 Leading zeros counter

Show that a leading zeros counter for a word of width $k = 2^a$ can be built recursively by using two $(k/2)$-input leading zeros counters and a two-way multiplexer. Then, generalize your construction to the case where k is not a power of 2.

18.27 Monotonicity in floating-point arithmetic

This problem is attributed to W. Kahan. Consider a computer that performs floating-point multiplication by truncating (rather than rounding) the exact $2p$-digit product of p-digit normalized fractional significands to p digits; that

is, it does not develop the lower p digits of the exact product, or simply drops them.

a. Show that, for a radix r greater than 2, this causes the monotonicity of multiplication to be violated (i.e., there exist positive floating-point numbers a, b, and c such that $a < b$ but $a \times_{\text{fp}} c > b \times_{\text{fp}} c$). *Hint:* when $x \times_{\text{fp}} y < 1/r$, postnormalization causes the least significant digit of the final product to be 0.

b. Show that multiplication remains monotonic in radix 2 (i.e., $a \leq b$ implies $a \times_{\text{fp}} c \leq b \times_{\text{fp}} c$).

REFERENCES AND FURTHER READINGS

[Ande67] Anderson, S. F., J. G. Earle, R. E. Goldschmidt, and D. M. Powers, "The IBM System/360 Model 91: Floating-Point Execution Unit," *IBM J. Research and Development*, Vol. 11, No. 1, pp. 34–53, 1967.

[Arno05] Arnold, M. G., "The Residue Logarithmic Number System: Theory and Implementation," *Proc. 17th Symp. Computer Arithmetic*, pp. 196–205, 2005.

[Bose87] Bose, B. K., L. Pei, G. S. Taylor, and D. A. Patterson, "Fast Multiply and Divide for a VLSI Floating-Point Unit," *Proc. 8th Symp. Computer Arithmetic*, pp. 87–94, 1987.

[Cole08] Coleman, J. N., et al., "The European Logarithmic Microprocessor," *IEEE Trans. Computers*, Vol. 57, No. 4, pp. 532–546, 2008.

[Coon80] Coonen, J. T., "An Implementation Guide to a Proposed Standard for Floating-Point Arithmetic," *IEEE Computer*, Vol. 13, No. 1, pp. 69–79, 1980.

[Davi74] Davis, R. L., "Uniform Shift Networks," *IEEE Computer*, Vol. 7, No. 9, pp. 60–71, 1974.

[Erce92] Ercegovac, M. D., and T. Lang, "On-the-Fly Rounding," *IEEE Trans. Computers*, Vol. 41, No. 12, pp. 1497–1503, 1992.

[Even00] Even, G., 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, 2000.

[Gok07] Gok, M., "A Novel IEEE Rounding Algorithm for High-Speed Floating-Point Multipliers," *Integration, the VLSI Journal*, Vol. 40, No. 4, pp. 549–560, 2007.

[Gosl71] Gosling, J. B., "Design of Large High-Speed Floating-Point Arithmetic Units," *Proc. IEE*, Vol. 118, pp. 493–498, 1971.

[Le07] Le, H. Q., et al., "IBM POWER6 Microarchitecture," *IBM J. Research & Development*, Vol. 51, No. 6, pp. 639–662, 2007.

[Mont90] Montoye, R. K., E. Hokonek, and S. L. Runyan, "Design of the Floating-Point Execution Unit in the IBM RISC System/6000," *IBM J. Research and Development*, Vol. 34, No. 1, pp. 59–70, 1990.

[Nain01] Naini, A., A. Dhablania, W. James, and D. Das Sarma, "1-GHz HAL SPARC64 Dual Floating Point Unit with RAS Features," *Proc. 15th Symp. Computer Arithmetic*, pp. 173–183, 2001.

[Ober97] Oberman, S. F., and M. J. Flynn, "Design Issues in Division and Other Floating-Point Operations," *IEEE Trans. Computers*, Vol. 46, No. 2, pp. 154–161, 1997.

[Omon94] Omondi, A. R., *Computer Arithmetic Systems: Algorithms, Architecture and Implementation*, Prentice-Hall, 1994.

[Schw03] Schwarz, E. M., M. Schmookler, and S. D. Trong, "Hardware Implementations of Denormalized Numbers," *Proc. 16th IEEE Symp. Computer Arithmetic*, June 2003, pp. 70–78.

[Schw06] Schwarz, E. M., "Binary Floating-Point Unit Design: The Fused Multiply-Add Dataflow," in *High-Performance Energy-Efficient Microprocessor Design*, V. G. Oklobdzija and R. K. Krishnamurthy (eds.), pp. 189–208, Springer, 2006.

[Sode96] Soderquist, P., and M. Leeser, "Area and Performance Tradeoffs in Floating-Point Divide and Square-Root Implementations," *ACM Computing Surveys*, Vol. 28, No. 3, pp. 518–564, 1996.

[Tron07] Trong, S. D., M. S. Schmookler, E. M. Schwarz, and M. Kroener, "P6 Binary Floating-Point Unit," *Proc. 18th Symp. Computer Arithmetic*, pp. 77–86, 2007.

[Wase82] Waser, S., and M. J. Flynn, *Introduction to Arithmetic for Digital Systems Designers*, Holt, Rinehart, & Winston, 1982.

[Yu06] Yu, X. Y., et al., "A 5 GHz+ 128-bit Binary Floating-Point Adder for the POWER6 Processor," *Proc. 32nd European Solid-State Circuits Conf.*, pp. 166–169, 2006.