2009 19th IEEE International Symposium on Computer Arithmetic

Multi-operand Floating-point Addition

Alexandre F. Tenca
Synopsys, Inc.
tenca@synopsys.com

Abstract

The design of a component to perform parallel addition of multiple floating-point (FP) operands is explored in this work. In particular, a 3-input FP adder is discussed in more detail, but the main concepts and ideas presented in this work are valid for FP adders with more inputs. The proposed design is more accurate than conventional FP addition using a network of 2-operand FP adders and it may have competitive area and delay depending on the number of input operands. Implementation results of a 3-operand FP adder are presented to compare its performance to a network of 2-input FP adders.

1 Introduction

Floating-point (FP) addition of two operands is extensively discussed in the literature [2, 6, 3, 7]. The limited precision in the FP representation requires rounding and basically makes the FP addition sensitive to the operand order [5]. When adding multiple FP operands using a network of 2-input FP adders (FPADD2) the error in the final result can be significant (loss of accuracy). Besides, the use of several FPADD2s on a circuit may result in long delays that could be avoided with an integrated solution.

Most of the operations in algorithms that use FP consist in FP addition and accumulation (around 80%). Fused multiplication and addition operations were investigated in the past (MAC) and are usually used in graphics and digital signal processing [1]. Several processors include this type of operation. Based on this observation, since 3 operands are available to FP units (FPU) that include MACs, it would be feasible to include 3-operand FP adders as part of the FPU. The use of multi-operand adders would be most beneficial to reduce the round-off error that happens during accumulation of FP values with conventional 2-input FP adders, however, the performance of these adders must be comparable to the solution using 2-input FP adders in order to be advantageous.

This work presents the issues involved in the design of multi-operand floating-point adders, and without loss of generality, it focus on the simultaneous addition of 3 operands. We analyze the issues related to multi-operand addition and evaluate the proposed solutions in comparison to the usual approach using 2-operand FP adders. The author is not aware of any previous publication dedicated to the discussion of this topic, although there are machines with inner-products and dot-products.

A floating-point number f_i is formed by a triple (s_i, e_i, m_i), where $s_i \in \{0, 1\}$ represents the sign (0 for positive and 1 for negative), $e_i \in [0, 2^{n_e} - 1]$ represents an n_e-bit exponent that carries a signed integer in a biased representation (with bias $B = 2^{n_e - 1} - 1$), and $m_i \in [0, 2^{n_m} - 1]$ representing the n_m-bit fractional portion of the significand. The value of f_i is $(-1)^{s_i} \times 1.m_i \times 2^{e_i - B}$. For special values (zero, NaN and infinity) the exponent field and significand field assume specific values. The n-operand FP adder (FPADDn) is a component defined for values of $n \geq 2$ as:

$$f_r = FPADDn(f_1, f_2, ..., f_n) = f_1 + f_2 + ... + f_n$$

The requirements for the design of FPADDn operators with $n > 2$ include: (1) the output accuracy of the operator must be better than the accuracy of an equivalent network of FPADD2s, (2) the operator should support a commutative n-input FP addition, and (3) the delay for the solution should be better to or competitive with the network of FPADD2s without excessive extra hardware.

The general concepts involved in the addition of multiple FP operands are presented in Section 2. Section 3 provides a high-level algorithm for the operation and identifies major building blocks required in the implementation of a component to perform the operation. Other sections provide some detail on significant design issues and alternative solutions. A more detailed block diagram of a 3-input FP adder is shown in Section 4. Results obtained with experiments and conclusions are provided in Sections 5 and 6.

1063-6889/09 $25.00 © 2009 IEEE
DOI 10.1109/ARITH.2009.27

2 General issues in multi-operand addition

Let us initially discuss the difficulties related to the implementation of FPADDn and the alternatives to make it more accurate than using a network of FPADD2s. The term ”precision” is a reference to the number of bits used in the computation of the proposed function. Accuracy is used to make reference to how close the result generated by the component is to the actual result (as if infinite precision were used).

First, similarly to multiply-and-add operators [1] the internal precision used to execute multi-operand addition must be larger than the internal precision used in each FPADD2 in a network. Such a requirement is expected because the alignment, normalization and rounding steps performed by each FPADD2 in a network keep the intermediate FP values in a representation with a small number of bits. When the series of steps performed by multiple FPADD2 modules are concentrated into a single component the internal structures will require more precision to handle the wider range of relative positions between mantissas and the interference between them. So, it is not straightforward to say that there is area or delay gain in this type of solution.

It is also well known that the result of addition of FP values using 2-input FP adders depend on the sequence of operations (non-associative operation). Consider the addition of 3 FP operands: a, b, and c, with $a = -b$ and $c << a$. The following results will occur when we execute different FP addition sequences on these operands:

- $(a +_{FP} (-b)) +_{FP} c = 0 +_{FP} c = c$
- $(a +_{FP} c) +_{FP} (-b) = a' +_{FP} (-b) = \epsilon$

where $+_{FP}$ represents FP addition, a' is a value close to a, or more specifically $a' \in \{a, a + 1ulp, a - 1ulp\}$, and ϵ is a value that can be as large as 1ulp of a. The accurate output is c, and since a is much larger than c, the error in the second result ($|\epsilon - c|$) can be large. An operator designed to satisfy the commutativity property on its inputs will always provide the most accurate result. Notice that an operator that handles this type of special cases must take catastrophic cancellations into account. Such operator will be more accurate than a network of FPADD2s.

It was initially considered as a design alternative to take advantage of the input sequence to generate a solution for the FPADDn that would be smaller and as accurate as the network of FPADD2s, but it turned out that enforcing the sequence of operations in the network and keeping the component with better or equal accuracy is complicated, and the design to cover all the special cases was not competitive with the network of FPADD2s in terms of area and delay.

Two-input FP adders swap the input operands to avoid the absolute value calculation after the addition of significands. The same is not possible in the addition of multiple operands because the sum of the smaller operands can be larger than the largest operand, and the addition of significands may be negative. There is no simple way to avoid the absolute value computation after the internal addition of aligned significands.

Another issue is related to the support of internal values larger than infinity. An initial experiment was done to detect internal overflows and set internal flags for infinities. However, exponents in FP adders overflow by only 1 unit, and internally keeping this slightly larger exponent value is beneficial in two ways: there is no need to perform overflow tests and it provides a wider range for internal values. Consider the following example: $c = -\infty$, $a = b = maxNorm$, where $maxNorm$ corresponds to the maximum normalized value in the FP system. The network of FPADD2 modules outputs NaN while the more accurate result is $-\infty$.

Finally, sticky bit calculation is more complex in the FPADDn than in FPADD2s. The determination of the correct value of the sticky bit used in the internal addition is critical to reach the desired level of accuracy. We discuss more about it in one of the following sections.

2.1 Special inputs

Extending the guidelines presented in IEEE-Std754 [4] for 2-operand addition, the addition of multiple FP operands should behave as follows for special values:

- when one of the inputs is NaN, the output is NaN;
- when inputs have one or more infinity values with the same sign, the output is infinity (matching the sign);
- when input have infinities with different signs, the output is NaN;
- when all the inputs are zeros the output is zero. However, care must be taken to keep the sign consistent with a network of FPADD2s that obey the standard. When all the zeros have the same sign, the output has this sign, independently of rounding mode. When signs are different, the output is -0 when rounding to negative infinity and +0 for all the other rounding modes. We considered (round to nearest, round to positive infinity, round to zero, round away from zero).

3 Adding multiple FP operands

A high level algorithm to perform multi-operand addition (without considering special input values) is given as follows:

1. unpack each of the FP input values f_i to obtain s_i, e_i, and m_i;

2. determine the maximum exponent $e_{max} = max(e_1, ..., e_n)$ and compute the exponent differences $\delta_i = e_{max} - e_i$;

3. align the values of $1.m_i$ based on δ_i, and determine the values of stk_i (sticky bits). The number of output bits in the alignment unit depends on the internal precision of the adder (p bits). Thus, $aligned_i[p-1:0] = (1.m_i) >> \delta_i$, where ">>" represents the bit shift operation.

4. determine the value stk to be used during addition of aligned significands based on stk_i and f_i;

5. perform addition of aligned mantissas with stk;

6. detect catastrophic cancellation of the largest f_i's, adjust the addition result and e_{max} value accordingly (discussed in Section 3.4);

7. perform normalization and rounding;

8. detect special cases, fix the FP result if necessary and pack the final FP value.

A general block diagram of an adder for multiple FP operands is shown in Figure 1. In the following sections we provide details for each block in the diagram and discuss the design decisions made for each of them. The approach looks rather straightforward but the multi-path approach used in 2-input adders is not attractive for FPADDns given the number of combinations to be considered.

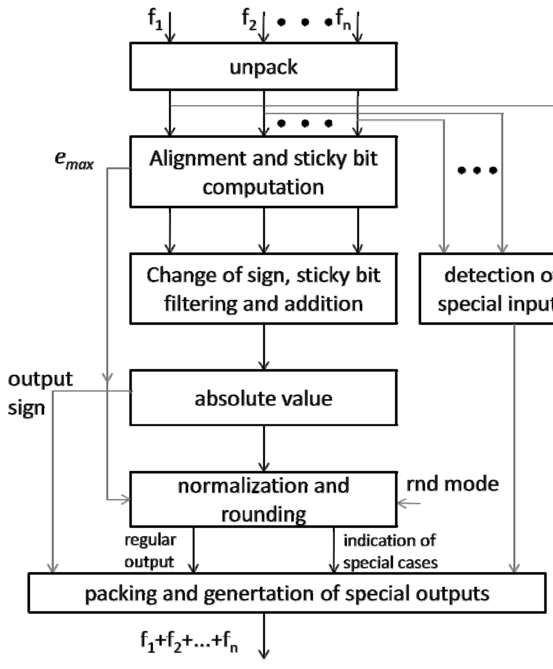
f1
f2
fn
unpack
emax
Alignment and sticky bit computation
Change of sign, sticky bit filtering and addition
detection of special inputs
output sign
absolute value
normalization and rounding
rnd mode
regular output
indication of special cases
packing and genertation of special outputs
f1+f2+...+fn

Figure 1. General block diagram

3.1 Alignment of significands

The alignment of significands involves also the computation of the sticky bit used for addition. The alignment of operands requires the detection of the maximum exponent and the computation of the relative difference between exponents. Once the differences between exponents is computed, the alignment is done using variable shifters. We discuss the use of two alternatives to perform this task.

3.1.1 Alignment with detection of maximum exponent

The maximum exponent may be detected with a network of comparators and multiplexers. Let us define a component (MAX) that is formed by a comparator and a multiplexer as shown in Figure 2.

A tree structure formed with this component provides a solution to identify the maximum value among several inputs. The tree is based on the associative and commutative properties of the $MAX(A, B)$ function.

Once the maximum exponent of n FP numbers being added is obtained, the shifting distance for proper alignment is computed by several small subtractors (exponents usually use a small number of bits). The output of each subtractor is the shifting distance control applied to a variable right shifter that aligns the corresponding significand.

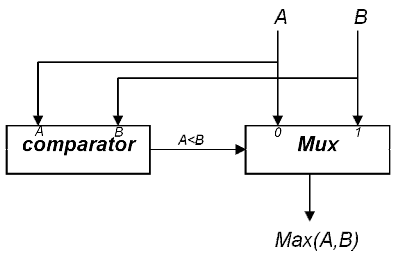

Figure 2. *$MAX(A, B)$ block for detection of maximum exponent*

The alignment unit using this method for a 3-operand FP adder is shown in Figure 3. For this example, the alignment unit requires 2 MAX blocks (3 comparators and 3 muxes) to compute the maximum exponent, 3 subtractors to obtain the shifting distances and 3 right shifters. The critical path passes by two MAX blocks, one subtractor, and one shifter.

An important parameter for this block is the bit width of the variable shifter outputs (k). They depend on the width of the internal adder for aligned significands, which is discussed in section 3.3.

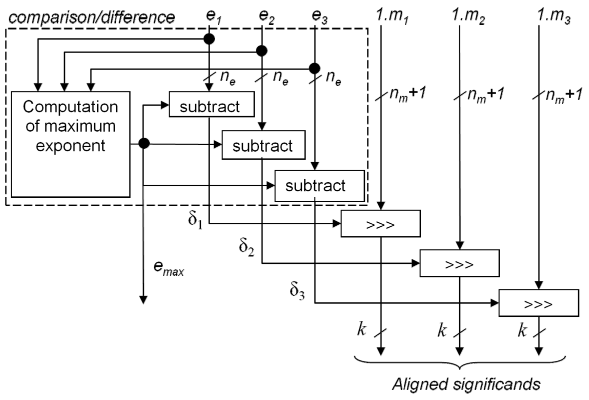

Figure 3. Alignment unit for a 3-input FP adder

Inputs				Mux control			sign/zero ctr		
c_{12}	c_{23}	c_{31}	e_{max}	M_1	M_2	M_3	α	β	γ
0	0	0	N/A	-	-	-	-	-	-
0	0	1	e_3	1	0	-	0	1	2
0	1	0	e_2	0	-	1	1	2	0
0	1	1	e_2	0	-	1	1	2	0
1	0	0	e_1	-	1	0	2	0	1
1	0	1	e_3	1	0	-	0	1	2
1	1	0	e_1	-	1	0	2	0	1
1	1	1	d.c.	-	-	-	-	-	-

Table 1. Comparison logic behavior

3.1.2 Alignment using results of relative differences

The previous circuit takes a significant portion of the FPADDn. An alternative approach comes from the realization that comparators basically use subtraction to find out which operand is the largest or smaller than the other, so, the result of the subtractions between exponents used for comparison could be also used for the computation of differences (used in the alignment shifters). Conversely, the computation of the relative differences between exponents can be used to determine which one is the largest. This alternative is attractive for FPADDns with small number of inputs. A diagram that illustrates this alternative is shown in Figure 4. The comparison logic uses the carry-out bits from the subtractions between exponents to come to a conclusion about the largest exponent, as shown in Table 1. In the table we have $e_{xy} = e_x - e_y$, for $x, y \in \{1, 2, 3\}$, and thus $e_{xy} = -e_{yx}$. Mux_x feeds a change-of-sign unit that is also able to zero its output. The control signals for the change of sign units are α, β, and γ, which can have the values 0 (do not change sign), 1 (change the sign), or 2 (zero output).

The carry-out bit c_{ij} generated by the operation $e_i - e_j$ is 1 when the operands have the same value or the minuend is larger than the subtrahend. The absolute value of the

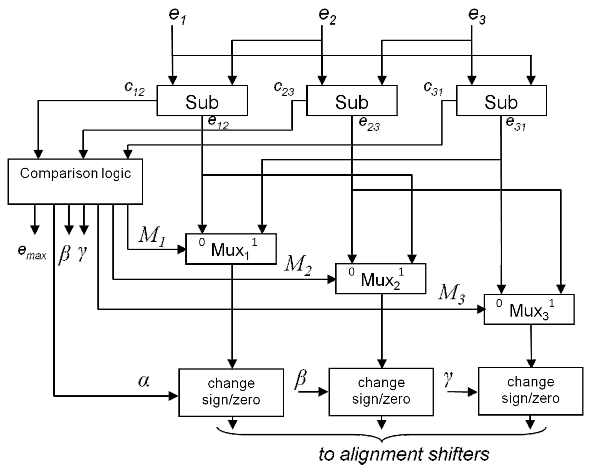

Figure 4. Another alternative for comparison/difference computation block

differences requires the change of sign for some of the subtractor's results. The N/A case for e_{max} indicates a case that does not happen. The output "d.c." means that any of the exponents can be used. Once e_{max} is obtained, change of sign blocks and muxes are used to pick up the correct absolute difference between exponents. For example, when e_1 is the largest exponent, the change of sign/zero logic connected to δ_1 is commanded to send a zero to the output, while e_{12} is selected in Mux_2 with the change of sign logic passing the input straight through (making $\delta_2 = e_{12}$), and e_{31} is selected in Mux_3 with change of sign to become $\delta_3 = e_{13}$. The alternatives are shown in the Table.

The critical path includes a subtractor, a change of sign block, a multiplexer and an alignment shifter. The result of the performance comparison between this alternative and the one presented in the previous subsection is given in section 5.

3.2 Sign of aligned significands

Once each significand is aligned, the sign of the FP number associated to it must be incorporated. This operation is required for the cases when the FP inputs have different signs. To avoid having change of sign logic for every aligned significand, it is better to anchor one of them and change the sign of the others relative to it. If the anchored significand is in fact negative, the result of the addition has the opposite sign. For an n-operand addition, we have the sign of the final FP number (s_r) computed from the sign of the adder output (s_{add}) and the sign of the input operand to be anchored (s_a) as $s_r = s_{add} \oplus s_a$. Each aligned significand is changed to a negative value based on the difference between its sign (s_i) and the anchor, or simply $change\_sign = s_i \oplus s_a$.

3.2.1 Detection and processing of sticky bits

Consider that each aligned significand may have a sticky bit stk_i, and we need to define the bit stk to be used in the addition of significands.

When performing multi-operand addition there is a possibility of having more than one aligned significand out of range (completely shifted out of range during alignment) or partially out of range. In any case, the sticky bit set for each significand after alignment phase must be used to define the sticky bit to be incorporated to the addition process. The decision about which sticky bit should be kept is decisive to generate an accurate result. The solution is trivial when the operands getting out of range (completely or partially) have the same sign, we just pick any one of them. The critical situation happens when the sticky bits area associated to operands with different signs.

There are a couple of cases that must be considered when multiple stk bits have non-zero value and the significands have different signs:

- there is at least one partially out-of-range (POR) significand: one simple approach would consist in discarding all the stk bits. It would work like a truncation of the significands. When there is enough precision in the internal adder (as discussed later), the error introduced by truncation does not substantially affect the result.

- all the aligned significands are completely out-of-range (COR): in this case, we make $stk = 1$ and the sign to be used for this bit is given by the following function:

$$sign_{stk} = sign(\sum_{i=1}^{n} stk_i * f_i)$$

  which means that for a large number of operands, an addition of aligned significands coming out of range is required for accurate computation of the stk bit. This requirement is a big burden for multi-operand addition when $n > 4$.

When $n = 3$ the solution to the problem is to *always take the stk bit of the largest FP value f_i that has $stk_i = 1$*. Full comparison of the FP values applied to the component is required to make a decision.

For $n > 3$ something more elaborate must be used since we may have the case when 3 COR significands occur, and addition of two of the smaller FP values that correspond to two of these COR significands may result in a larger value than the FP value of the third COR significand, and has opposite sign. We could still adopt the same approach applied to $n = 3$, but we would have issues with rounding in some cases. This simple approach would be however, equivalent to some results obtained with a network of FPADD2s, but the FPADD3 wouldn't be more accurate than the network in these cases.

Other alternatives would involve sorting of operands and comparisons in order to properly identify the value of the sticky bit to be used.

3.3 Internal adder precision

Most of the time, the bits of one aligned significand may have a weight that is completely different than the weight of bits in another aligned significand. It is practically impossible (for usual FP formats) to have enough internal precision to keep all the aligned significand bits of all operands. Therefore, a decision must be made regarding the precision of the internal adder for significands, and the degree of truncation that must happen as a consequence of this decision.

A regular two-operand adder uses only 3 extra LS bits (besides the bits in the width of a significand) to account for all cases in the FP addition/subtraction. In the multi-operand sum, the number of extra bits used to bound the error in the computation will depend on the number of operands.

Consider the addition of the following FP values: x, $-(x - \epsilon_1)$, $-(\epsilon_1 - \epsilon_2)$, $-(\epsilon_2 - \epsilon_3)$. Each value is such that ϵ_1 is close to 1 ulp of x, ϵ_2 is close to 1 ulp of ϵ_1, and ϵ_3 is close to 1 ulp of ϵ_2. When these operands are added in the FPADDn the result must be equal to ϵ_3. Notice that when the operands are added in a network of FPADD2s, depending on the order of operations, the result can be zero or close to zero, which may be significantly different from ϵ_3. Based on this scenario, the internal adder should have a precision of approximately $n - 1$ times the precision of a significand. This situation illustrates the *partial cancellation* of significant bits.

In another scenario, the truncation that happens when significands are partially placed out of range introduces an error that can affect the round bit position. Therefore, the adder must be large enough to keep carry chains that in a long-precision calculation would modify the round bit.

Instead of trying to solve the general problem we look at these two situations when applied to 3-operand addition. Let us consider that the significand has f bits. The two cases described above are considered as follows:

1. partial cancellation of operands: as shown in Figure 5, the cancellation of MS bits between the largest aligned significands result in a value near the LS bit position of $1.m_2 \times 2^{\delta_2}$, which when combined with the smallest aligned significand should keep enough significant

bits in the adder to generate the output. The total number of bits in this case is 1 bit for the sign, 2 bits for overflow (for the case of effective additions of 3 significands), $f + 1$ bits for the overlapped area between the two largest significands, f bits for the second significand and 1 bit position for the sticky bit, coming to a total of $p = 2f + 5$ bits.

2. propagation of truncation errors: Figure 6 shows the minimum adder precision required to maintain the longest carry chain among the aligned significands. The Figure shows three aligned significands $x_1 = 1.m_1 \times 2^{\delta_1}$, $x_2 = 1.m_2 \times 2^{\delta_2}$ and $x_3 = 1.m_3 \times 2^{\delta_3}$. Clearly, $\delta_1 = 0$. The value x_3 indicates an aligned significand that was partially shifted out of range and has a bit that causes a carry propagation with x_2. Between x_1 and x_2 there is a 1-bit gap that corresponds to the round bit (R). In the shown configuration, the adder operates without a problem, but if the adder precision is one bit shorter, x_3 would be completely out of range, and the LS bits of x_2 would be in the sticky bit position. Based on the previous discussion about sticky bits, only one of them is preserved, and therefore, the propagation of carries will no longer happen, affecting the value of the round bit (R). Based on this observation, the internal adder precision needs to cover 1 bit for the sign, 2 bits for overflow, $f + 1$ bits for the largest significand including round bit, f bits for the second significand and 1 bit position for the sticky bit, also coming to a total of $p = 2f + 5$ bits.

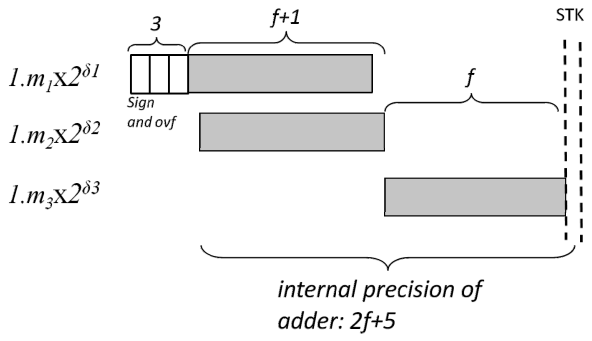

Figure 5. Minimum adder size when partial cancellation of largest operands occurs ($1.m_1 \times 2^{\delta_1} - 1.m_2 \times 2^{\delta_2} \neq 0$)

As a result of this discussion, an internal adder with $2f+5$ bits is required to implement a FPADD3 component.

A formal verification tool provided by Synopsys (Formality) was used to confirm the lower bound on the internal adder size for the 3-operand FP adder. Two models were

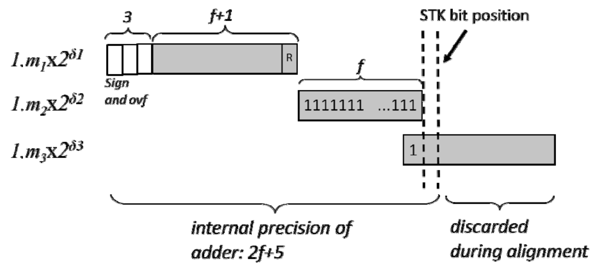

Figure 6. Worse case for carry-chain formation in a 3-input FP adder

used. A reference model A with large precision and a test module B with adjustable internal precision. The procedure consisted of decreasing the precision of model B until it reached a point where formal verification failed. The verification started with an internal adder precision in B that was slightly larger than the value $2f + 5$. Test vectors obtained when the verification failed indicate the situations that lead to inaccurate results, and they always include the cases described in this section.

For more operands, a larger precision is required for the internal adders, which must in this case handle more possibilities for the alignment of significands and cancellations.

3.4 Catastrophic cancellation

Another special situation that must be handled by the FPADDn is caused by *catastrophic cancellation* of the largest aligned significands, resulting in a complete loss of significant bits if the exponent difference between the two largest FP numbers and the next FP number is greater than the precision of the internal adder. This problem cannot be solved with a reasonable adder size. The best alternative is to use detectors of catastrophic cancellations and a correction step that takes the next group of aligned significands. The number of combinations for more than 4 FP inputs become very large, which makes the solution of this problem extremely difficult, and forbids the use of FPADDn for large $n > 4$.

Full comparison of FP values is required for both sticky bit computation and detection of catastrophic cancellation, allowing some sharing of hardware resources in this case.

4 A 3-operand FP adder

After we have covered the major building blocks required for multi-operand FP addition, we can provide a little bit more detail on the 3-input FP adder (FPADD3) design. The block diagram for a 3-operand adder is shown in Figure 7. The alignment unit was described earlier. The change of sign block adjusts the sign of significands based on the floating-point input f_3 (anchor), as described in Section 3.2. The sign of f_3, called s_3 is passed to the sign adjustment logic to compute the final result sign.

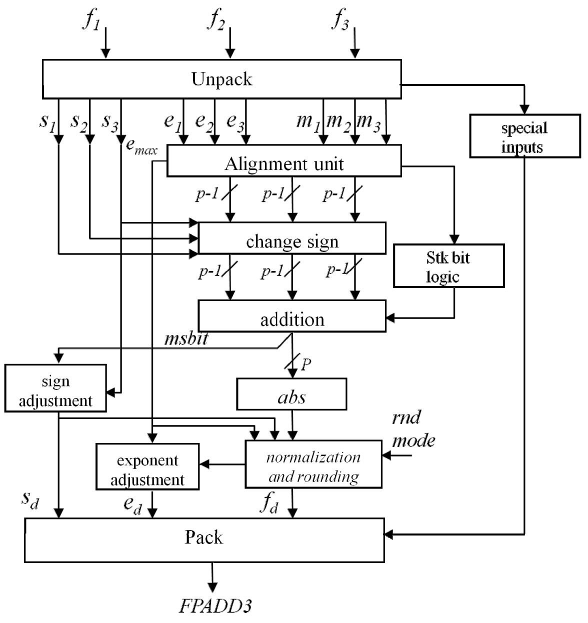

Figure 7. Block diagram for the 3-input FP Adder

The sticky bit logic process the stk_i values coming from the alignment unit and let only one of them be used during addition. The blocks that follow the internal addition of aligned significands are similar to the blocks in a two-input FP adder.

In order to have exact rounding, the internal adder used in the FPADD3 must have $2f + 5$ bits of precision, as discussed earlier.

The catastrophic cancellation detection is done in the *special inputs* block. This block is also responsible for the detection of other input conditions, such as multiple zeros and infinities, the presence of NaNs, and combinations of special values. Comparators are used to identify the catastrophic cancellation cases during addition and control a bypass circuit in the *pack* block to forward the smallest FP input to the output.

By following the rules discussed earlier, the FPADD3 generates a perfectly rounded result. This means that its output is computed as it would be done with infinite precision, and then rounded to the finite FP format. The design enforces commutativity in the operation by treating all the inputs the same way. This characteristic cannot be obtained with a network of FPADD2s.

One potential optimization to reduce area in the implementation of the FPADD3 would be to use only two alignment shifters instead of three since only 2-out-of-3 significands are aligned. However, in order to do this type of processing, the inputs should be sorted before they were applied to the alignment shifters, which would increase the critical path.

5 Implementation results and analysis

The design of the 3-operand FP adder (FPADD3) was described in Verilog and Synopsys tools were used to implement and test it. A few target CMOS cell libraries were used for synthesis with Design Compiler. Functional simulation was performed using VCS-MX.

First, the experimentation with the two options for alignment of the significand shows that the first option (Section 3.1.1) is larger and slower than the second option (Section 3.1.2) for tight time constraint. When the time is very relaxed the first option becomes smaller. The best solution for delay was used.

A comparison of the area and delay results for the FPADD3 after synthesis using a TSMC 90nm CMOS standard cell library is shown in Figure 8. The curves show the various design points for which the component can be used. As we relax the time constraint (delay) required for the design, the area is reduced. This particular plot was obtained for an FPADD3 that handles single precision operands (8 bits of exponent and 23 bits in the significand). In the figure, the quality of results of the 3-input FP adder is compared against the network of 2 FPADD2s. The FPADD2 implementation used in this experiment came from the Synopsys DesignWare Library (DW_fp_add). The 3-input FP adder is superior in terms of speed, but it is not as small as the network of 2 DW_fp_add components for more relaxed time constraint. However, the 3-input FP adder is much more accurate than the network of small adders. Designers are always looking for the best possible performance with the smallest area impact, and the use of a FPADD3 delivers extra performance and better numerical behavior, with a modest area penalty in this case (the area increase with the reduction in the critical path delay follows the same trend shown by the network of FP adders).

In order to have another design for comparison we synthesized the adder published in [7]. It is a very specialized adder for low-latency and double-precision FP format. The synthesis of this adder was done for the same conditions and technology as the previous experiment. It has an area of 29,323 and delay of 2.7ns when synthesized for very tight time constraint (without pipeline stages). Two of those adders in a network would add up to an area of almost 51,200 (considering an overall area reduction of 17% when two FP adders are combined) with a delay of 5.4ns. The synthesis of the double-precision FPADD3 with the same

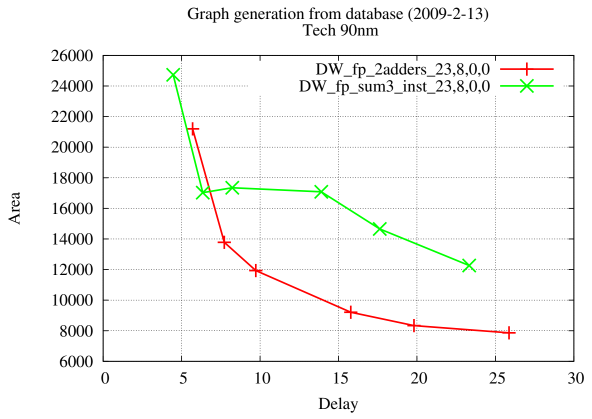

Figure 8. Performance comparison between 3-input FP adder and network of FPADD2s for a 90nm standard cell library synthesis - single precision FP format

constraints result in the fastest circuit at 5.6ns and area of 47,600. Therefore, the FPADD3 has again comparable critical path to the network of 2-input FP adders with superior accuracy and less area (7%).

Other area/delay experiments were performed for other technologies and the results reforced the same trend observed for the 90nm technology. However, the network of FPADD2s is always smaller for loose time constraint. Thus if the application requires a small solution, the accuracy is not a concern, and the time may be relaxed, the addition of multiple FP values should be done with the network of 2-input adders.

Verification of the FPADD3 component operation was done using Matlab and extensive simulation runs. Small FP sizes were used to enable an almost exhaustive test of the input range. For larger FP formats, a guided random test was applied, for which the inputs are generated on the neighborhood of special values, or inputs are such that force special conditions (total cancellation, partial cancellation, multiple stk bits, etc). The first step in the verification process consisted in applying tests vectors to the multi-operand adder and the corresponding network of FPADD2s, generating this way two sets of results. The two sets were compared, and the mismatching test vectors were submitted as inputs to a Matlab program that was written to calculate the addition of multiple FP operands using very large precision arithmetic. This way, the program was able to compute the result as it would be done by an infinitely precise operator and calculate the roundoff error in each case. Using this methodology it was possible to verify that the multi-operand adder was always more accurate than the network of FPADD2s. Any results violating this premise were used to identify corner cases where the design was not working properly and eliminate bugs.

6 Conclusion

In this paper we demonstrated the feasibility of implementing multi-operand floating-point adders to get more accurate operations than equivalent networks of FPADD2s. The work was concentrated on 3-input FP adders but the discussion about design issues and alternative solutions is also applicable to adders for more operands. The experimental results show that the 3-input FP adder design can be synthesized to reach shorter or similar delays than the network of 2-input FP adders, with comparable or better area, and more accuracy. The capability of this component to generate outputs honoring the commutative property for its inputs is very advantageous to eliminate the ordering problem (non-associative behavior) imposed at the algorithm level on networks of FPADD2s.

References

[1] J. Bruguera and T. Lang. Floating-point fused multiply-add: reduced latency for floating-point addition. In *17th IEEE Symposium on Computer Arithmetic*, pages 42–51, 2005.
[2] M. D. Ercegovac and T. Lang. *Digital Arithmetic*. Morgan Kaufmann, 2003.
[3] D. Goldberg. What Every Computer Scientist Should Know About Floating-Point Arithmetic. *ACM Computing Surveys*, March 1991.
[4] IEEE Std 754-1985. *IEEE Standard for Binary Floating-Point Arithmetic*. IEEE, 1985.
[5] N. Kapre and A. DeHon. Optimistic Parallelization of Floating-Point Accumulation. In *18th IEEE Symposium on Computer Arithmetic*, pages 205 – 216, 2007.
[6] B. Parhami. *Computer Arithmetic: Algorithms and Hardware Designs*. Oxford, 2000.
[7] P.-M. Seidel and G. Even. Delay-optimized implementation of IEEE floating-point addition. *IEEE Transactions on Computers*, 49(7):638 – 650, July 2000.