Appearing in *IEEE Symposium on Computer Arithmetic* (ARITH 2013), April 7–10, 2013

Accurate Parallel Floating-Point Accumulation

Edin Kadric, Paul Gurniak, and André DeHon
Dept. of Electrical and Systems Engineering
University of Pennsylvania
Philadelphia, PA, USA
Email: ekadric@seas.upenn.edu

***Abstract*—Using parallel associative reduction, iterative refinement, and conservative termination detection, we show how to use tree reduce parallelism to compute correctly rounded floating-point sums in $O(\log N)$ depth at arbitrary throughput. Our parallel solution shows how we can continue to exploit Moore's Law scaling in transistor count to accelerate floating-point performance even when clock rates remain flat. Empirical evidence suggests our iterative algorithm only requires two tree reduce passes to converge to the accurate sum in virtually all cases. Furthermore, we develop the hardware implementation of a 250 MHz pipelined, native, residue-preserving IEEE-754 double-precision, floating-point adder on a Virtex 6 FPGA that requires only 48% more area than a standard adder without residue. Finally, we show how this module can be used as the base of a streaming accurate floating-point accumulation unit that can be tuned to consume m summands every cycle.**

***Keywords*-Floating-Point Arithmetic, IEEE-754, Parallel, Accumulation, Accurate, Rounding, FPGA**

I. Introduction

Microprocessor clock speeds have grown exponentially throughout the VLSI era until the mid 2000's. Limits to pipelining [1] and power density ended processor clock scaling [2], [3]. Nonetheless, Moore's Law feature size scaling continues to deliver an exponential growth in transistors per Integrated Chip (IC) or dollar. The focus has now shifted to increasing parallelism in computations rather than clock speed, leaving us with the question of how we can turn increased transistors into increased performance without increasing clock speeds.

The end of frequency scaling presents a particular challenge for floating-point (FP) arithmetic. While we can typically perform algebraic transforms that change the order of operations to increase instruction-level parallelism (ILP) for integer operations, changing the evaluation order of FP operations can lead to different results. In this paper, we focus on the problem of summing N double-precision FP values. With integer arithmetic, we can exploit the associativity of addition to reorder the sum into a tree of depth $O(\log N)$. For large N, this provides ample ILP and even opportunities for core-level parallelism without changing the result. However, since FP addition is not associative, performing a similar reordering for the FP sum would produce a *different* result.

As a result, FP addition must normally be performed sequentially as originally specified in the source code. This is a choice to sacrifice parallelism in order to maintain the determinism of the answer, an approach that not only fails to address the current need for increased parallelization, but also fails to provide guarantees on the accuracy of the answer. Although the sequential sum operation will always give the same result, it may still be an inaccurate one.

In this paper, we develop an algorithm based on [4] for parallelizing FP addition while still guaranteeing a perfectly accumulated and correctly rounded deterministic result. That is, we simultaneously address parallelism, accuracy and determinism, by formulating the basic summation as one that computes a correctly rounded result. We use an iterative, convergent summation that avoids discarding any components of the original sum until we can prove that they are not needed to correctly round the final summation (Sec. IV). Our convergence test is significantly simpler than the one suggested in [4], which requires 2 extra FLOPs (FLoating-point OPerations) per summand. We prove that the algorithm always terminates (Sec. V-C) and provide evidence that it is extremely fast in virtually all cases—usually completing in only two iterations (Sec. VI-A). Our algorithm has $O(\log N)$ depth, providing a significant speed improvement over the commonly used sequential summation technique, with only twice the delay of the non-deterministic parallel one. Finally, we show how our algorithm can be efficiently pipelined and tuned to consume m FP summands every cycle (Sec. VII-C).

II. Background

A. Correct Rounding

Given N floating-point numbers x_i, our goal is to compute the correct rounding S_c of the exact sum $S_r = \sum_{i=1}^{N} x_i$. By correct rounding, we mean that if S_r is a floating-point number, then S_c will be its exact value, whereas if it is not, S_c will be the nearest of the immediate FP neighbors of S_r. That is, S_c is the value we would get if we computed S_r with infinite precision, then rounded it to the appropriate nearest FP number. If S_r falls halfway between two FP numbers, the appropriate one is still chosen deterministically, for example using the "tie to even" approach in the IEEE754 standard. Note that correct rounding defines correctness of the result

not in terms of the order of operations but in terms of the error introduced into the final result. This gives us freedom to transform the computation for additional parallelism as long as we can guarantee to achieve the correctly rounded result. A faithful rounding of S_r is similar to the correct rounding, except that when S_r falls between two neighboring FP numbers, either one could be chosen. The literature shows sequential software algorithms that compute both faithfully and correctly rounded sums using standard IEEE-754 double-precision FPUs (*e.g.*, [5]–[7]).

B. Residue Preserving Addition

Most of the accurate accumulation software algorithms such as [5]–[7] rely on the same basic building block that is studied in detail by Kornerup et al. [8], a floating-point adder with residue (FPAR), which computes:

$$s = \text{IEEE754Round}(a+b), \quad (1)$$
$$r = (a+b) - s. \quad (2)$$

This returns the sum of two FP numbers (s)—the same sum one would get from a normal FP add—as well as the error (r) resulting from the operation, which is known to also be representable as a floating-point value [9].

The FPAR building block can be implemented on a standard IEEE-754 FP unit. Kornerup et al. [8] show that the algorithm introduced by Knuth [10] is minimal, both in terms of number of operations (six) and depth of the dependency graph (five). Furthermore, they argue that algorithms with fewer floating-point operations that also require branching are inferior (*e.g.*, [9]), due to possible drastic performance losses after a wrong branch prediction causing the instruction pipeline to drain.

Shown below is Knuth's implementation that requires no branching and does not require a priori knowledge of the ordering of the two inputs a and b.

$(s, r) = Knuth_{fpar}(a, b)$:
$s = a + b$
$b' = s - a$
$a' = s - b'$; $\delta_b = b - b'$
$\delta_a = a - a'$
$r = \delta_a + \delta_b$

Previous work suggested that a hardware implementation of an FPAR module could be significantly faster with only a minor area overhead [11], [12], as it avoids some inefficiencies in Knuth's algorithm, where residual bits computed early on are discarded and then recomputed during later stages.

C. Exploiting Hardware

There have been previous attempts at accelerating floating-point accumulation using specialized, pipelined hardware. For example, Luo and Martonosi do so by implementing delayed addition techniques [13]. They perform an associative transform of the addition operations but do not address the fact that the results would be different from the original summation (as well as inaccurate).

In contrast, Kapre and DeHon demonstrate that an iterative approach can be used to exploit parallelism and still generate the same result as a sequential sum [14]. Although this introduces determinism in the parallel approach, it spends more time to reach a solution that is still inaccurate.

Our work addresses the accuracy and parallelism issues at the same time: We speed up computations by exploiting parallelism while guaranteeing that the provided result is accurately accumulated and correctly rounded. We use a tree-like structure in the spirit of Leuprecht and Oberaigner [4] but with a much more efficient convergence test.

III. Add with Residue Unit

This section describes the hardware design of a 7-stage, pipelined, IEEE-754 compliant, double-precision Floating-Point Adder (FPA) and its extension into an FPAR. We implement both designs on a Virtex 6 FPGA and find that the FPAR only occupies 48% more area than the FPA, while still running at the same speed.

A. Floating-Point Adder

A 64-bit IEEE-754 double-precision number contains 1 sign bit, 11 exponent bits and 52 mantissa bits plus an implicit leading 1 and handles denormalized numbers, zero, infinity and Not A Number (NaN). The adder computes a rounded IEEE-754 sum, s, according to Eq. 1.

Our FPA is divided into 7 pipeline stages. The yellow part of Fig. 1 shows the general organization, which is similar to the improved single-path floating-point adder in [15], except that we use a Leading One Detector (LOD) instead of a Leading One Predictor (LOP) in order to save area. The datapath is as follows: First, the mantissas are compared and the exponents subtracted (stage 1), before being used to right shift the smaller one's mantissa (stage 2). During the right shift, the discarded bits are used to compute rounding requirements in stage 4. Before that, the mantissas are added together (stage 3) and the LOD determines the resulting change in exponent (stage 4), which is then used to determine rounding requirements (also stage 4), as well as in stage 5 to normalize the mantissa. Stage 5 rounds and shifts the number, which can be parallelized to improve delay since the costly dynamic left shift is only needed when a and b are completely overlapping, and thus no rounding is needed. Otherwise the position of the mantissa's MSB can only change by at most one in either direction, so that in the case where rounding is added, normalization is performed with a much cheaper 1-bit right or left shift as suggested in [15]. A multiplexer is then used to select the proper case. Stage 6 checks if the result is zero (zero mantissa), denormalized (negative exponent), and if it has overflowed to infinity (maximum exponent), at the same time as computing a denormalized version assuming the exponent is negative.

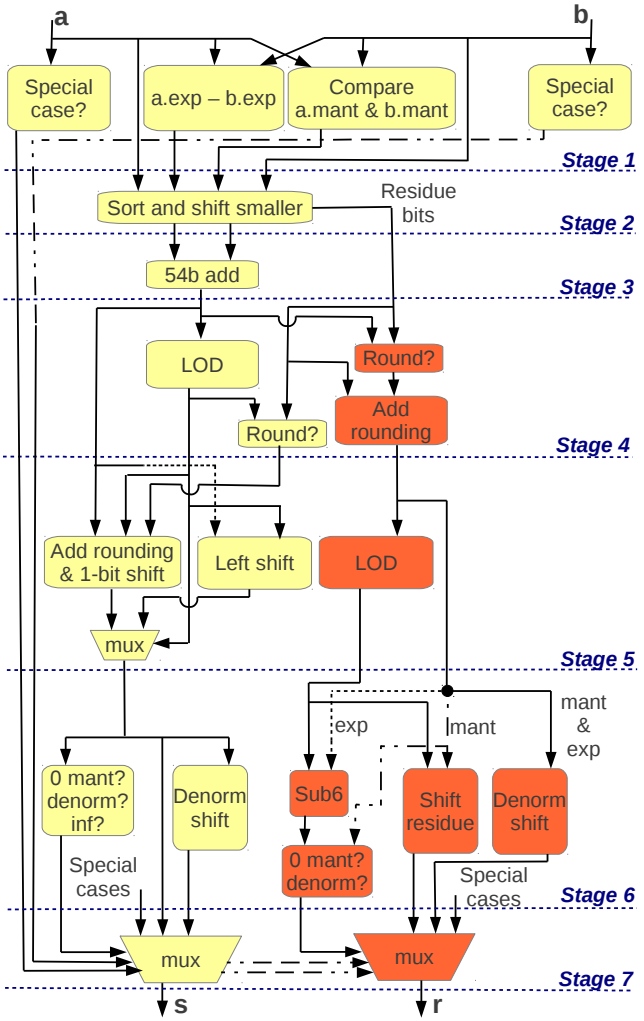

Figure 1. Structure of the FPAR unit.

Finally, stage 7 selects between the rounded normalized number, its denormalized version, a 0 output, and a special case number determined back in stage 1.

This hardware design was described in Bluespec [16] and implemented on a Virtex 6 FPGA. It occupies an area of 1517 Lookup Tables (LUTs) and runs at 250 MHz (the critical path delay is 3.996 ns after place and route). The slowest and most area consuming operations are the dynamic shifts, the additions, and the 54-bit comparison in stage 1.

B. Floating-Point Adder with Residue

The FPAR can be built by extending the FPA. Fig. 1 shows its organization, with the additional hardware shown in orange. Stages 1, 2 and 3 are mostly the same, except for the additional registers to communicate the residue bits to the subsequent stages instead of discarding them. Unfortunately, we cannot perform a leading 1 search on r at the same time as s (stage 4) since we first need to know where the s mantissa will end in order to know where to start the search for r. Therefore, this step is moved to stage 5, and is performed after residue rounding, moved to stage 4, which is controlled by a rounding unit similar to the one for the sum: adding 1 to the LSB of s is coupled with subtracting 1 one position above the MSB of r and vice versa. However, we only need to know whether the MSB of the sum has moved by one bit at most in order to determine the position at which the residue should be rounded, since moving by more than one position would mean that a and b are cancelling each other and that the residue is zero. This three-case check can be performed at low cost during stage 4 even without knowing the sum's leading 1 information, and is followed by an adder to compute the rounded residue. Stage 5 then performs a leading 1 search on the residue. Stage 6 normalizes and computes a denormalized version, both of which require dynamic shifts, together with checking whether the residue is 0 (0 mantissa) or denormalized (this time checking for a negative normalized exponent $exp_{norm} = exp - LO_{index}$, where LO_{index} is the index of the leading 1). Finally, during stage 7, a second multiplexer is added to choose between the different possible residue outputs.

We are thus able to exploit parallelism and switch the order of operations to produce the residue without affecting the clock frequency and number of pipeline stages. The FPAR occupies an area of 2252 LUTs, only 48% more than the FPA. It is also comprised of 7 pipeline stages running at 250 MHz: The critical path delay is 3.999 ns. While we quote specific results from our FPGA implementation, we expect a custom implementation would achieve similar results—achieving the same latency as the base FPA and requiring only fractionally more area. Manoukian and Constantinides have implemented a single-precision FPAR on a Virtex 6, also with no delay overhead, and 47% area overhead compared to a base FPA unit [17].

C. Impact

Software accurate accumulators that use an FPAR unit in their algorithm [5]–[7] usually implement it using multiple FPA units as a building block (Sec. II-B). Compared to the 5 sequential instructions in Knuth's algorithm [10], our FPAR offers a 5x latency improvement over current solutions, while occupying only 48% more area than a standard FPA unit, a 75% total work reduction (6x area of an FPA for previous work versus 1.5x for ours).

IV. Parallel Accumulation Strategy

Pipelining the floating-point units does not actually help a sequential summation, since we must wait for the result of the previous addition to complete in order to add the next value. We must extract parallelism in order to exploit the throughput benefits of a pipelined floating-point adder.

Our basic strategy for performing an accurate parallel summation, similarly to [4], is as follows:

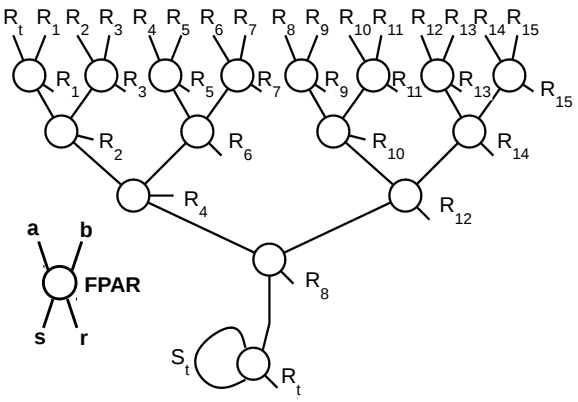

Figure 2. Basic binary reduce tree structure for $N = 16$ summands.

1) Perform a parallel tree reduce sum on the summation output of the FPAR with $\lceil \log N \rceil$ stages and $N - 1$ nodes; keep the residue output of each FPAR for potential refinement and error estimation (Fig. 2).
2) Perform the same parallel tree reduce on the FPAR residues and compute a conservative upper bound on the magnitude of the sum of residues, and, hence, potential error in our calculation (Sec. V).
3) Iterate reducing the residues and adding their sum into the overall sum, S_t, until our conservative upper bound indicates the residues can be ignored.

In Fig. 2, R_t is the residue resulting from adding the sum of the residues to the accumulating total sum S_t, and it is updated after each iteration.

V. Detailed Algorithm

In this section, we provide more details on the functioning of our algorithm. We start by developing an expression for efficiently computing convergence detection. We then describe the detailed algorithm, and we finish by proving two theorems that ensure its termination. Without loss of generality, we assume the IEEE-754 standard "round to nearest, ties to even" rounding mode throughout the section, which requires the most care to achieve proper rounding and is the default for binary floating-point.

A. Termination Condition

To determine when we have enough information to return the correctly rounded result, we calculate a conservative upper bound to the magnitude of the sum of the $N - 1$ remaining residues, rsb, the Residue Sum Bound. We can then check for convergence by testing the condition $conv(S_t, R_t, rsb)$ (Tab. I). If the condition succeeds, we conclude that even an upper bound on what is left in the tree is not sufficient to influence the total accumulating sum, S_t, and it can be returned as the final sum of the algorithm.

Before deriving an expression for rsb and $conv$ more precisely, we first define $nzcnt$ as the number of non-zero residues left in the tree and $maxexp$ as the maximum exponent of these residues, both of which can be trivially computed during the summation tree reduction (Sec. VII-B). We then write an upper bound on the sum of the residues:

$$|error| \leq nzcnt \times 2^{maxexp+1}. \tag{3}$$

By definition, rsb is an upper bound on $|error|$, hence:

$$rsb = nzcnt \times 2^{maxexp+1} \geq |error|. \tag{4}$$

We can now approximate rsb as an exponent only:

$$rsb.exp = \lceil \log(nzcnt) \rceil + maxexp + 1. \tag{5}$$

Eq. 5 gives us an upper bound on the error.

Finally, we need to define $conv(S_t, R_t, rsb)$. The function takes two non-overlapping inputs S_t and R_t, with $|S_t| \geq |R_t|$, as well as rsb (since S_t and R_t are outputs from an FPAR, this relation always holds). Tab. I shows a case analysis that helps us define and implement the $conv$ function. Based on the values of S_t, R_t and rsb, we are able to decide whether S_t is the final converged sum (shown with a green check mark), or whether other iterations of the algorithm are required (red X). The $conv$ function considers convergence to be successful if and only if the following condition is satisfied:

$$round(S_t + R_t + rsb) = round(S_t + R_t - rsb)$$

In the table, R_t patterns are shown assuming their MSB starts right after the LSB of S_t. Also, we define a function lco (Lowest in Chain of Ones), which gives the index of the least significant 1 in the first chain of ones of a string—the most significant of these chains when there are more than one (*e.g.*, Fig. 3). We define $rsb.i$ (rsb index) to compare how rsb aligns with R_t as

$$rsb.i = S_t.exp - 53 - rsb.exp. \tag{6}$$

As shown in Tab. I, if $rsb.i < 0$, then it overlaps with S_t and convergence fails (a different result is obtained whether rsb is added or subtracted). Otherwise, we continue by looking at the R_t pattern and identify three cases:

A) There are two zeros after the LSB of S_t and before any R_t bit is set (at indices 0 and 1). If $rsb.i$ is not immediately after the LSB of S_t (if $rsb.i > 0$), then neither $R_t + rsb$ nor $R_t - rsb$ have large enough magnitudes to set the LSB of S_t after rounding: convergence succeeded. Otherwise, if $rsb.i = 0$, it acts as a tying bit and we must look at R_t: if it is non-zero, we would round differently whether rsb is added or subtracted: convergence failed. Otherwise, if R_t is zero, this is a tie case: For the tie to even rounding mode, we must look at the LSB of S_t, if it is 0, convergence has been achieved; if it is 1 it has failed, since a positive or negative rsb would make the number round to a different even value.

B) There is a 1 at index 0. What follows the 1 *must* be all zeros, by definition of the FPAR unit (another case would

Table I
THE $conv(S_t, R_t, rsb)$ FUNCTION.

rsb.i<0					
Yes	X				
No	**Rt pattern**				
	00xxx..	**rsb index**			
		rsb.i > 0	√		
		rsb.i = 0	**Rt = 0?**		
			No	X	
			Yes	**St.lsb=0?**	
				Yes	√
				No	X
	1xxx..	**Xxx.. = 000..**			
		No	Impossible		
		Yes	**rsb = 0?**		
			No	X	
			Yes	√	
	01xxx..	**rsb index**			
		rsb.i > Rt.lco	√		
		0≤rsb.i<Rt.lco	X		
		rsb.i = Rt.lco	**Rt pattern**		
			01..10xxx..	**xxx.. = 000..**	
				No	X
				Yes	**St.lsb=0?**
					Yes √
					No X

have caused rounding that would have been absorbed into the sum and would have returned a residue with a 0 in place of the 1 at index 0). If rsb is 0, we know how to round the final sum depending on $S_t.LSB$. Otherwise, if rsb is non-zero, adding or subtracting it will make the tie deviate in a different direction, thus failing the convergence test.

C) Index 0 is zero and index 1 is set:

1) If $rsb.i > R_t.lco$, then neither $R_t + rsb$ nor $R_t - rsb$ can cause a carry that will set index 0, and we have converged.
2) If $0 \leq rsb.i < R_t.lco$, then $R_t + rsb$ and $R_t - rsb$ have a different effect on index 0: In one of the cases it would be zero, and in the other it would be one. Either way, at least one bit would be set after index 0, so that convergence fails.
3) If $rsb.i = R_t.lco$, we are in a similar situation as in (B), but now we also need to check whether there is a bit set after index 0. If there is, then convergence fails just as in (B). If no other bit is set, then one of $R_t + rsb$ or $R_t - rsb$ (the one that sets index 0) will become a tie, which is dealt with similarly to case 1): If $S_t.LSB$ is even, then both the R_t+rsb and R_t-rsb cases suggest no rounding: convergence is achieved; otherwise one of them suggests no rounding while the other is a tie suggesting a rounding away from the odd value: convergence fails.

B. Algorithm

The complete accurate parallel accumulation algorithm is shown in Fig. 4. The loop calls a tree reduction on every

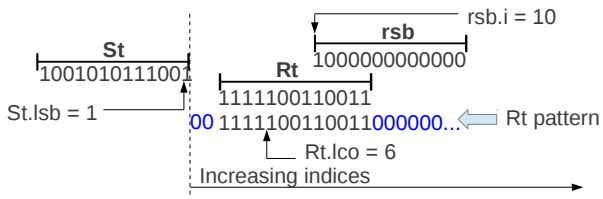

Figure 3. Example of an input configuration in the $conv$ function.

iteration, before checking for convergence and returning the result. If convergence fails, we proceed with another iteration until it is successful. However, we also need to check for specific rare cases such as the one shown next for a reduced mantissa length of 4 and assuming rsb is set by the only residue remaining in the tree:

$$S_t = 1.1010e27, R_t = 1.0000e22, rsb = 1.0000e10$$
$$S_t + R_t + rsb = 1.1011e27, S_t + R_t - rsb = 1.1010e27$$

In this example, while convergence is not achieved, R_t is non-overlapping with the remaining residue and no reduction can be performed. This is why our algorithm shown in Fig. 4 performs another check when convergence fails:

$$(rsb.exp < R_t.exp)$$

If the latter condition succeeds after convergence fails we say that we have reached a "Big Tie". A Big Tie means that only the LSB of S_t can now be affected by what is left in the tree (plus potential carry), and that we can thus simplify our convergence condition in order to resolve cases such as the one shown in the previous example. Theorem 1 in Sec. V-C proves that either convergence or a Big Tie must be reached by the algorithm.

In order to resolve a Big Tie, we start by subtracting a 1 right after the LSB of S_t (index 0 in Fig. 3). This way, we have removed the tying bit, and we then only check for the following condition:

$$sign(R_t + rsb) = sign(R_t - rsb)$$

Indeed, we now only need to know whether what is left in the tree is positive or negative to determine whether it will shift the tie up or down, and hence whether it will affect the LSB of S_t or not. Doing this prevents clogging the tree and allows smaller values to interact and resolve. Theorem 2 in Sec. V-C proves that if a Big Tie is reached, it will be resolved, and thus that our algorithm is complete and always terminates.

In fact, this handling of a Big Tie deals with the case that Kornerup et al. use to prove the impossibility of correctly rounding the sum of three or more FP numbers using a "Round to nearest tie to even" approach with depth less than 1939 double-precision IEEE-754 operations [8]. We are able to cover such a depth with limited resources because repeated iterations allow us to dynamically increase the DAG depth until termination, exceeding 1939 if needed.

Init: $S_t = 0$; $BigTie = False$;
while True **do**
  Compute Reduce Tree (Fig. 2) on initial values for the first pass, on residues for subsequent passes;
  Obtain $nzcnt$, $maxexp$, rsb, S_t, R_t;
  if $!BigTie$ **then**
    if $conv(S_t, R_t, rsb)$ **then**
      return S_t;
    else
      if $rsb.exp \geq R_t.exp$ **then**
        Continue; //will resolve to another case (Th. 1)
      else
        //Residues can only affect the LSB of S_t
        $BigTie = True$;
        $Rts = R_t.sign$;
        Subtract 1 right after the LSB of S_t;
      end if
    end if
  else
    //Resolving a Big Tie
    if $sign(R_t + rsb) = sign(R_t - rsb)$ **then**
      if $sign(R_t \pm rsb) > 0$ **then**
        return $(Rts > 0)?rnd\_up(S_t) : S_t$
      else if $sign(R_t \pm rsb) < 0$ **then**
        return $(Rts > 0)?S_t : rnd\_down(S_t)$
      else if $sign(R_t \pm rsb) = 0$ **then**
        return $round(S_t$ with tying bit);
      end if
    else
      Continue; //will resolve to previous case (Th. 2)
    end if
  end if
end while

Figure 4. Correctly Rounded Sum Algorithm.

C. Convergence proofs

In this section, we prove two theorems. Theorem 1 ensures that the algorithm in Fig. 4 will either converge to the result, or to a Big Tie case where only the LSB of the sum is unknown. Theorem 2 ensures that if a Big Tie is encountered it will always be resolved. First, let us establish two lemmas.

Lemma 1: $\log(nzcnt)$ eventually reduces to 40.

Proof: Assuming $N \leq 2^{40}$, it is trivially true that $\log(nzcnt) \leq \log(N-1) \leq 40$. For larger sums we can guarantee that $nzcnt$ will reduce with a suitable residue feedback structure (carefully ordering the residues back into the first stage of the reduce tree so that cancellations occur). However, as we shall see in Sec. VIII, it is likely this sum would follow an operation that reduced the number of inputs well below 2^{40}.

Lemma 2: If $maxexp > R_t.exp - 53$, then the residue that set $maxexp$ will be reduced, thus reducing $maxexp$.

Proof: Consider two cases:

- $maxexp$ is set by a uniquely largest exponent. Since R_t has 53 bits, if $maxexp$ is larger than $R_t.exp$-53, it will interact with R_t and be reduced.
- $maxexp$ is not set by a uniquely largest exponent. The two residues setting $maxexp$ will be combined into one. A process that repeats until $maxexp$ is set by a uniquely largest exponent (the previous case).

In either case, R_t is reduced.

We can now conclude the following two theorems.

Theorem 1: Eventually, $rsb.exp < R_t.exp$.

Proof:
Eq. 5: $rsb.exp = \lceil \log(nzcnt) \rceil + maxexp + 1$
Lemma 1: Eventually, $\log(nzcnt) \leq 40$
Lemma 2: Eventually, $maxexp \leq R_t.exp - 53$
Combining Lemmas 1 and 2 into Eq. 5: Eventually,

$$rsb.exp \leq R_t.exp - 12 < R_t.exp. \tag{7}$$

Theorem 1 shows that rsb will become smaller than R_t, in which case only the LSB of S_t can still be affected by the combined effect of R_t and rsb. As shown in Fig. 4, this forces the algorithm to either converge or reach a Big Tie, which Theorem 2 addresses.

Theorem 2: Eventually, $sign(R_t - rsb) = sign(R_t + rsb)$.

Proof:
By Theorem 1: Eventually, $rsb.exp < R_t.exp$

$$\Rightarrow rsb < 2^{R_t.exp} \leq |R_t|, \tag{8}$$

$$\Rightarrow |R_t| - rsb \geq 0. \tag{9}$$

Since rsb is nonnegative, Eq. 9 implies

$$sign(R_t - rsb) = sign(R_t + rsb). \tag{10}$$

Theorem 2 proves that a Big Tie will always be resolved since its convergence condition $(sign(R_t + rsb) = sign(R_t - rsb))$ must eventually be met. Together, Theorems 1 and 2 show that the algorithm always terminates.

VI. Experimental Results

A. Experimental setup and results

In order to test the speed and practicality of our algorithm, we perform software simulations using several different datasets. To measure how much the sum can change with respect to a small change in the summands, we define the condition number of the N x_i datapoints similarly to [18]:

$$\kappa = \frac{\sum_{i=1}^{N} |x_i|}{|\sum_{i=1}^{N} x_i|}$$

For low κ, the data is said to be well-conditioned, and the algorithm should easily converge, whereas for high κ, it is said to be ill-conditioned and is expected to make convergence more difficult.

We generate datasets similar to those used by Zhu and Hayes [19]. Data #1 contains randomly generated positive floating-point numbers, so that the condition number is $\kappa = 1$. Data #2 is similar except that it contains both positive and negative numbers, resulting in a low condition number. Data #3 is similar to Anderson's ill-conditioned data [5]: we first generate data #2, then compute the mean using standard floating-point arithmetic (thus introducing some error), before subtracting it from each data point. This results in ill-conditioned data with a high κ. Finally, data #4 contains half randomly generated positive floating-point numbers, together with their exact opposites in the second half, such that the exact sum is 0, and $\kappa = \infty$.

In addition to the uniformly distributed random data used in previous work (*e.g.*, [5], [19]), we also use an exponential one. We define an exponential distribution as one where the individual bits of the IEEE-754 representation are randomly picked from a uniform distribution, thus tending to produce numbers with all allowed exponent values instead of concentrating them on the highest positive and negative exponents. This prevents the numbers from being too close together and reduces mantissa overlap, resulting in data that does not reduce as efficiently as a uniform distribution.

We define the parameter δ as the maximum possible difference in exponent ranges in the original summands, which is taken into account when generating the random data. We use 2^{12} as the number of summands in the dataset (as suggested in Sec. VIII), and we repeat each experiment 1000 times. Tabs. II-a and II-b show results when the numbers are generated with an exponential and a uniform distribution respectively. Both tables report the average number of iterations the algorithm takes before the convergence condition is met, together with the associated standard deviation from that average. Then, the average number of non-zero values remaining in the tree when convergence is met is shown, followed by the average percentage error when computing the sum using a simple sequential software non-exact summation.

B. Discussion of the results

We observe that in most cases, the algorithm always takes exactly 2 iterations to converge (with a 0 standard deviation in number of iterations), no matter how ill-conditioned the data is, except for the extreme case of $\kappa = \infty$ and exponentially distributed data. This empirically validates our inexpensive convergence test since we get the same number of iterations suggested in [4], whose termination detection was much more expensive (two full FP additions per tree node). We also note that the highly unlikely Big Tie case was never encountered with any of these datasets.

Fig. 5 illustrates why exactly two iterations are needed most of the time. As shown, after two numbers a and b are fed into an FPAR unit, they come out as two new numbers with non-overlapping mantissas, where the MSB

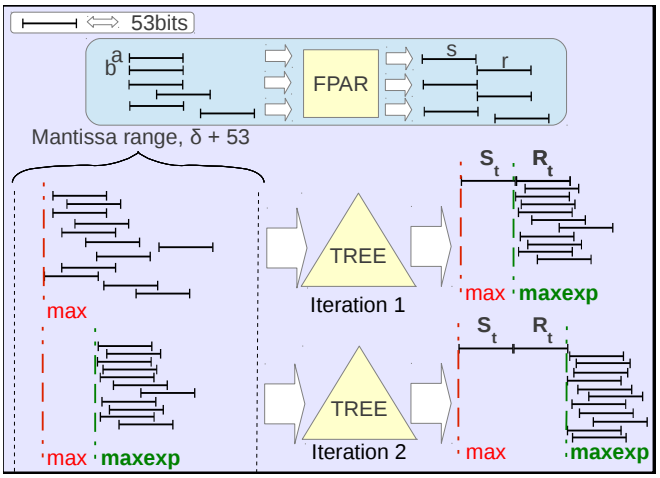

Figure 5. Effects of the algorithm on the mantissa ranges.

has typically only shifted by a few bits at most: a major shift of m positions is extremely unlikely for randomly chosen bits, in the order of 2^{-m}. Therefore, with extremely high probability, the final sum S_t at the end of the tree will occupy the 53 bits to the immediate left of the final residue R_t, whereas R_t will occupy the same bits as several other residues remaining in the tree, including the largest one that determines $maxexp$. Hence, since $rsb.exp = \log(nzcnt) + maxexp + 1$ (Eq. 5), rsb easily overlaps with S_t and the convergence test fails. However, after a second pass in the tree, the new R_t still occupies the bits to the immediate right of S_t, but this time the remaining residues and $maxexp$ occupy the bits to the immediate right of R_t, making it extremely unlikely that $\log(nzcnt)$ would be large enough to cause an overlap with S_t. A dataset needs to be carefully designed to get more than two iterations, as was done with the exponentially distributed dataset #4. In that case, major cancellations throughout the tree translate into a $maxexp$ that is often larger than S_t, thus failing the convergence test. Therefore, in order to see more than two iterations, we had to define extremely ill-conditioned cases that are unlikely to occur in practice (condition number $\kappa \to \infty$). We conclude that our algorithm is extremely fast and reliable.

Tab. II-a shows that, even when there are many non-zeros left in the tree (datasets #1 and #2), our algorithm is able to determine that they will not be sufficient to affect the final sum, so that it can discard them and converge faster. Tab. II also reminds us that the non-accurate summation error can become intolerably large (datasets #3 and #4).

VII. Comparisons

A. Asymptotic Performance

Our algorithm achieves the asymptotically optimal FLOP count $\Theta(N)$. This is the same as simple summations that ignore precision and other efficient correctly rounded approaches (*e.g.* [6], [20]). With sufficient parallelism, our

Table II
EXPERIMENTAL RESULTS FOR EXPONENTIAL (A) AND UNIFORM (B) DISTRIBUTIONS OF DATA.

a) Exponential distribution of data

	Data 1 κ = 1			Data 2 κ = 78.0 on average			Data 3 κ = 6.33e16 on average			Data 4 κ → ∞		
δ	# its, σ	% non-0	s/w seq % error	# its, σ	% non-0	s/w seq % error	# its, σ	% non-0	s/w seq % error	# its, σ	% non-0	s/w seq % error
10	2, 0	0.000	1.45E-13	2, 0	0.000	6.34E-13	2, 0	0.000	5.5909	2, 0	0.000	∞
100	2, 0	42.129	2.28E-13	2, 0	41.760	4.14E-13	2, 0	0.000	137.4309	3, 0	0.000	∞
1000	2, 0	93.850	3.54E-14	2, 0	93.581	9.94E-14	2, 0	0.000	4855.618	20, 0.17	0.000	∞
2000	2, 0	96.856	2.54E-14	2, 0	96.698	1.39E-13	2, 0	0.000	86870.63	39, 0.045	0.000	∞

b) Uniform distribution of data

	Data 1 κ = 1			Data 2 κ = 330 on average			Data 3 κ = 1.59e17 on average			Data 4 κ → ∞		
δ	# its, σ	% non-0	s/w seq % error	# its, σ	% non-0	s/w seq % error	# its, σ	% non-0	s/w seq % error	# its, σ	% non-0	s/w seq % error
10	2, 0	0.000	1.26E-13	2, 0	0.000	5.93E-13	2, 0	0.000	2.881778	2, 0	0.000	∞
100	2, 0	0.000	1.28E-13	2, 0	0.000	5.61E-13	2, 0	0.000	3.044484	2, 0	0.000	∞
1000	2, 0	0.000	1.24E-13	2, 0	0.000	6.49E-13	2, 0	0.000	1.993717	2, 0	0.000	∞
2000	2, 0	0.000	1.24E-13	2, 0	0.000	7.62E-13	2, 0	0.000	1.822635	2, 0	0.000	∞

Table III
FLOP COUNT COMPARISON.

Accumulation Algorithm	**FLOPs/summand**	**Depth**
iFastSum [6]	12	$O(N)$
with our FPAR	3	
Simple, inaccurate	1	$O(\log N)$
Optimistic Sequential [14]	40	$O(\log N)$
Leuprecht and Oberaigner tree [4]	14	$O(\log N)$
with our FPAR	5	
This Work	3	$O(\log N)$

design can achieve $O(\log N)$ latency, which is superior to traditional correctly rounded summations including Demmel's Radix sort [20] and Zhu's convergent summation [6], both of which are $\Theta(N)$. Only Leuprecht and Oberaigner's correct summation algorithm achieved $O(\log N)$ latency [4]. Furthermore, our $O(\log N)$ latency is the same as a precision-ignoring sum, and Kapre and DeHon's sequential-semantics-preserving sum [14].

B. Floating Point Operation Count

Tab. III compares the FLOP counts of the major, representative algorithms, including the revision of prior work using our FPAR. We count 1.5 FLOPs for our FPAR due to its area (Sec. III). All three correct rounding algorithms (iFastSum, Leuprecht and Oberaigner, and ours) take 2 iterations in almost all cases. We estimate Kapre and DeHon as 8 iterations with 5 FLOPs per iteration. Leuprecht and Oberaigner use two FLOPs per tree node for convergence detection. We do not count our termination computation as a FLOP since it takes less than 4% of the area of the FPA (see mn unit in Sec. VII-C). Our algorithm achieves a lower FLOP count than previous algorithms, tying only with [6], which has depth $O(N)$ instead of our $O(\log N)$ depth.

C. Concrete, Tunable Streaming Accumulation

In this section, we show how our algorithm can be efficiently implemented in limited area using a fixed number of FPAR modules. Our unit is a "streaming" accumulator because it deals with inputs as they are presented to it on every cycle. Furthermore, we show how it can be tuned to consume several inputs every cycle.

Fig. 6-a shows the basic building block that we use to achieve this. We call it a Tree Streaming Block (TSB). The TSB instantiates one FPAR module and functions with two phases. During phase 1, the FPAR computes the sum of floating-point values already present in the A1 and B1 FIFOs, storing the results in A2 and B2. At the same time, as new inputs are presented to the TSB unit, they are inserted into A1 and B1, which can both store $N/2$ values. For N inputs to our algorithm, phase 1 takes $N/2$ cycles, it captures half the inputs to the algorithm into A1 and B1, and uses the FPAR to compute the first tree level ($N/2$ operations).

Once phase 1 completes, we switch to phase 2, which uses the FPAR in order to compute the sum of the data in A2 and B2 and feed it back to A2 and B2 again, each of which can store $N/4$ values. Phase 2 can thus compute level 2 of the tree ($N/4$ cycles), followed by level 3 ($N/8$ cycles), and so on, until it has computed all tree levels in a total of $N/2$ cycles, plus $p \times \log(q) - \sum_{i=0}^{log(q)-1} 2^i$ due to the pipelining of the FPAR, where p is the pipeline depth and $q = 2^{\lceil \log(p) \rceil}$. Hence, phase 2 also takes roughly $N/2$ cycles to complete, a time during which it can write the second half of the N inputs into A1 and B1, so that when we switch back to phase 1, the buffers are already full and the FPAR can be used right away. Throughout both stages, the residues are sent outside of the TSB module.

We have described the TSB in Bluespec and implemented it on our FPGA, with memories wide enough to support $N =$

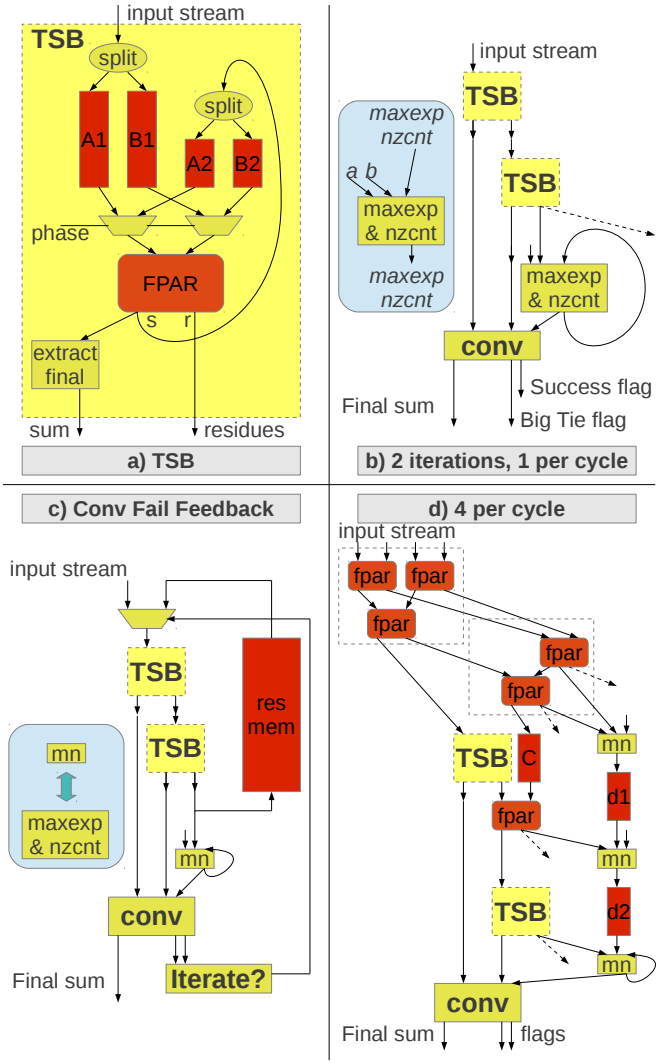

Figure 6. Streaming Tree Reduce Configuration a) TS Block b) 1 value per cycle c) Feedback for failed convergence d) 4 values per cycle.

2^{12} floating-point inputs (Sec. VIII justifies this choice). It was found to consume only 7% more area than the FPAR module alone: The block was synthesized with a total of 2410 LUTs (plus the Block RAMs), and a critical path of 3.967 ns, thus still running at 250 MHz and only adding a minor area overhead.

The TSB can in turn be used in several configurations in order to implement our summation algorithm. First, in Fig. 6-b we show how we can consume $m = 1$ floating-point value every cycle using two TSBs and an additional $conv$ module which evaluates convergence as described in Sec. V-A (as well as a "Big Tie" flag). In the simplest case, the $conv$ module can take the S information from the two TSBs and compute their sum and residue, S_t and R_t, which are used to determine convergence. In order to save area, this final addition can also be performed on the already instantiated FPAR module in the second TSB (the mechanism for this is not shown in Fig. 6), with only a minor latency overhead. Now the $conv$ module also needs to compute rsb, for which it needs the $nzcnt$ and $maxexp$ information. Both of these are updated throughout the tree after each FPAR operation: Whenever a residue is to be discarded (or put on the side for subsequent iterations, shown with dashed arrows), we update $maxexp$ and $nzcnt$ locally and propagate the result to the next node, together with the sum. The $conv$ module then receives the $maxexp$ and $nzcnt$ from the last level of the tree. We denote the module updating $maxexp$ and $nzcnt$ by mn, and it performs the following operations:

$$(maxexp, nzcnt) = mn(a, b, maxexp_{in}, nzcnt_{in}):$$
$$maxexp = max(maxexp_{in}, a.exp, b.exp)$$
$$nzcnt = nzcnt_{in} + (a \neq 0) + (b \neq 0)$$

Implementing the mn block on our FPGA, we measure a critical path of 2.378 ns and only 53 LUTs used. For the $conv$ block, we have used four pipeline stages, for which we find a critical path of 3.937 ns and 682 LUTs used. Putting all of this together into the structure shown in Fig. 6-b, we get a full design with a critical path of 3.986 ns and 5648 LUTs, only 17% more than two TSB modules alone.

We suggest implementing two TSB modules directly since we have found that the algorithm generally takes only 2 iterations to complete. However, in order to deal with convergence issues, we can also add a feedback path to store the residues and use them in the first TSB block again during subsequent iterations (Fig. 6-c). We do not show how to deal with this nor Big Tie handling, since it can even be dealt with using higher level software mechanisms.

Finally, Fig. 6-d shows how the design can be tuned to consume m values every cycle (case $m = 4$ shown). In order to achieve this, we add more FPAR units to build part of the tree directly: $m-1$ FPAR units are arranged in a tree to compute one sum and $m-1$ residues. The sum stream is used as the input stream to the first TSB, whereas the $m-1$ residues are further added together using $m-2$ more FPAR units, producing a sum of the residues saved in a "C" FIFO. The residue outputs of the TSB are then summed with the outputs from C using another FPAR module, before being fed into the second TSB. This structure effectively directly implements the first $\log(m)$ stages of the tree in hardware, thus allowing the designer to trade off between the speed-ups allowed by the tree structure and area consumption. Of course, we also need to replicate the mn units, one for each FPAR, and propagate them through additional "D" FIFOs as shown in Fig. 6-d.

This shows how to implement our accurate accumulation algorithm in limited area while maintaining a high throughput of m values per cycle, a latency of $\frac{3N}{m}$, and $O(N)$ work.

VIII. Extensions

An interesting extension to our work would be to include it as part of an even more general streaming algorithm able

to compute the sum of an arbitrary number of inputs (instead of N previously) with constant memory, based on the accumulator strategy presented by Demmel [20]. In particular, as shown by Zhu and Hayes [19], if we use an algorithm such as ours (or iFastSum in their case) as a building block for an accumulator-based algorithm, our implementation only needs to provide enough memory for computing the sum of 2^{12} numbers. The accumulator algorithm consists in first reducing a large number of inputs into a limited number of them using the fact that their exponent range is limited and that they are forced to overlap. Then, once all the summands have been compressed into 2^{12} new ones, we can apply our algorithm on those new summands, which can be done with constant memory.

Furthermore, we could use our streaming unit with no changes to support an accurate dot product, and therefore several important linear algebra routines. We would need a multiplier that produced the full $2\times$(mantissa width) result and encoded that as two double-precision, floating-point numbers to the sum. Since the output of the multipliers is only $2N$ double-precision values, an accumulator on $2N$ inputs will serve to perform the accurate reduction.

IX. Conclusions

We have shown how to perform parallel associative floating-point summation and obtain a correctly rounded result in $O(\log N)$ depth. We have also provided evidence that our algorithm is fast and reliable, only requiring two iterations in virtually all cases. Furthermore, thanks to our FPAR hardware unit, we are able to directly improve the running time and total work of other algorithms that were proposed in the literature. We have implemented the unit as an extension of a standard FPA, and because we were able to perform all the additional computations in parallel with already necessary operations, we were able to make the unit run as fast as the standard FPA, while requiring only 48% more area. Finally, we have shown how to implement our summation algorithm based on a streaming floating-point accumulation hardware that can be tuned to consume m summands every cycle.

Acknowledgment

Support for Paul Gurniak from the Rachleff Scholar's Program was instrumental in initiating this work. This material is based in part upon work supported by the Office of Naval Research (ONR) under Contract No. N000141010158. The views expressed are those of the authors and do not reflect the official policy or position of ONR or the U.S. Government. Valuable feedback from the anonymous reviewers improved the clarity, precision, and presentation of the paper.

References

[1] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger, "Clock rate versus IPC: The end of the road for conventional microarchitectures," in *ISCA*, 2000, pp. 248–259.

[2] M. Horowitz, E. Alon, D. Patil, S. Naffziger, R. Kumar, and K. Bernstein, "Scaling, power, and the future of CMOS," in *IEDM*, December 2005, pp. 7–15.

[3] S. H. Fuller and L. I. Millett, Eds., *The Future of Computing Performance: Game Over or Next Level?* The National Academies Press, 2011. [Online]. Available: http://www.nap.edu/catalog.php?record_id=12980

[4] H. Leuprecht and W. Oberaigner, "Parallel algorithms for the rounding exact summation of floating point numbers," *Computing*, vol. 28, pp. 89–104, 1982.

[5] I. J. Anderson, "A distillation algorithm for floating-point summation," *SIAM J. Sci. Comput*, vol. 20, pp. 1797–1806, 1999.

[6] Y.-K. Zhu and W. B. Hayes, "Correct rounding and a hybrid approach to exact floating-point summation," *SIAM J. Sci. Comput.*, vol. 31, no. 4, pp. 2981–3001, July 2009. [Online]. Available: http://dx.doi.org/10.1137/070710020

[7] S. M. Rump, "Ultimately fast accurate summation," *SIAM J. Sci. Comput.*, vol. 31, no. 5, pp. 3466–3502, September 2009. [Online]. Available: http://dx.doi.org/10.1137/080738490

[8] P. Kornerup, V. Lefèvre, N. Louvet, and J.-M. Muller, "On the computation of correctly rounded sums," *IEEE Trans. Comput.*, vol. 61, no. 3, pp. 289–298, March 2012.

[9] T. J. Dekker, "A floating-point technique for extending the available precision," *Numerische Mathematik*, vol. 18, pp. 224–242, 1971. [Online]. Available: http://dx.doi.org/10.1007/BF01397083

[10] D. E. Knuth, *The Art of Computer Programming: Seminumerical Algorithms*, 3rd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997, vol. 2.

[11] F. de Dinechin, J. Detrey, O. Cret, and R. Tudoran, "When FPGAs are better at floating-point than microprocessors," ENS Lyon, Tech. Rep. ensl-00174627, 2007. [Online]. Available: http://prunel.ccsd.cnrs.fr/ensl-00174627

[12] W. Dieter, A. Kaveti, and H. Dietz, "Low-cost microarchitectural support for improved floating-point accuracy," *Comp. Arch. Lett.*, vol. 6, no. 1, pp. 13–16, January–June 2007.

[13] Z. Luo and M. Martonosi, "Accelerating pipelined integer and floating-point accumulations in configurable hardware with delayed addition techniques," *IEEE Trans. Comput.*, vol. 49, no. 3, pp. 208–218, March 2000.

[14] N. Kapre and A. DeHon, "Optimistic Parallelization of Floating-Point Accumulation," in *Proc. IEEE Symp. on Comp. Arith.*, June 2007, pp. 205–213.

[15] M. Ercegovac and T. Lang, *Digital Arithmetic*, ser. The Morgan Kaufmann Series in Computer Architecture and Design. Morgan Kaufmann, 2003.

[16] Bluespec, Inc., "Bluespec SystemVerilog." [Online]. Available: http://www.bluespec.com

[17] M. V. Manoukian and G. A. Constantinides, "Accurate floating point arithmetic through hardware error-free transformations," in *Proc. Intl. Conf. on Reconf. Comp.* Springer-Verlag, 2011, pp. 94–101. [Online]. Available: http://dl.acm.org/citation.cfm?id=1987535.1987550

[18] N. J. Higham, "The accuracy of floating point summation," *SIAM J. Sci. Comput*, vol. 14, pp. 783–799, 1993.

[19] Y.-K. Zhu and W. B. Hayes, "Algorithm 908: Online exact summation of floating-point streams," *ACM Trans. Math. Softw.*, vol. 37, no. 3, pp. 37:1–37:13, September 2010. [Online]. Available: http://doi.acm.org/10.1145/1824801.1824815

[20] J. Demmel and Y. Hida, "Accurate and efficient floating point summation," *SIAM J. Sci. Comput.*, vol. 25, no. 4, pp. 1214–1248, April 2003. [Online]. Available: http://dx.doi.org/10.1137/S1064827502407627