Accelerating Pipelined Integer and Floating-Point Accumulations in Configurable Hardware with Delayed Addition Techniques

Zhen Luo and Margaret Martonosi, *Senior Member*, *IEEE*

Abstract—The speed of arithmetic calculations in configurable hardware is limited by carry propagation, even with the dedicated hardware found in recent FPGAs. This paper proposes and evaluates an approach called *delayed addition* that reduces the carry-propagation bottleneck and improves the performance of arithmetic calculations. Our approach employs the idea used in Wallace trees to store the results in an intermediate form and delay addition until the end of a repeated calculation such as accumulation or dot-product; this effectively removes carry propagation overhead from the calculation's critical path. We present both integer and floating-point designs that use our technique. Our pipelined integer multiply-accumulate (MAC) design is based on a fairly traditional multiplier design, but with delayed addition as well. This design achieves a 72MHz clock rate on an XC4036xla-9 FPGA and 170MHz clock rate on an XV300epq240-8 FPGA. Next, we present a 32-bit floating-point accumulator based on delayed addition. Here, delayed addition requires a novel alignment technique that decouples the incoming operands from the accumulated result. A conservative version of this design achieves a 40 MHz clock rate on an XC4036xla-9 FPGA and 97MHz clock rate on an XV100epq240-8 FPGA. We also present a 32-bit floating-point accumulator design with compiler-managed overflow avoidance that achieves a 80MHz clock rate on an XC4036xla-9 FPGA and 150MHz clock rate on an XCV100epq240-8 FPGA.

Index Terms—Delayed addition, accumulation, multiply-accumulate, MAC, FPGA.

1 Introduction

When an arithmetic calculation is carried out in a RISC microprocessor, each instruction typically has two source operands and one result. In many computations, however, the result of one arithmetic instruction is just an intermediate result in a long series of calculations. For example, dot product and other long summations use a long series of integer or floating-point operations to compute a final result. While FPGA designs often suffer from much slower clock rates than custom VLSI, configurable hardware allows us to make specialized hardware for these cases; with this, we can optimize the pipelining characteristics for the particular computation.

A typical multiplier in a full-custom integrated circuit has three stages. First, it uses Booth encoding to generate the partial products. Second, it uses one or more levels of Wallace tree compression to reduce the number of partial products to two. Third, it uses a final adder to add these two numbers and get the result. For such a multiplier, the third stage, performing the final add, generally takes about one-third of the total multiplication time [8], [9]. If implemented using FPGAs, stage 3 could become an even greater bottleneck because of the carry propagation problem. It is hard to apply fast adder techniques to speed up carry propagation within the constraints of current FPGAs. In Xilinx 4000-series chips, for example, the fastest 16-bit adder possible is the hardwired ripple-carry adder [19]. The minimum delay of such an adder (in a -9 speed grade XC4000xla part) is more than four times the delay of an SRAM-based, 4-input look-up table that forms the core of the configurable logic blocks. Since this carry propagation is such a bottleneck, it impedes pipelining long series of additions or multiplies in configurable hardware; the carry-propagation lies along the critical path, it determines the pipelined clock rate for the whole computation. Our work removes this bottleneck from the critical path so that stages 1 and 2 can run at full speed. This improves the performance of dot-products and other series calculations.

As an example, consider the summation C of a vector A: $C = \sum_{i=0}^{99} A[i]$. Our goal is to accumulate the elements of A without paying the price of 99 serialized additions. We observe that, in traditional multiplier designs (e.g., the multiply units of virtually all recent microprocessors [15], [16]), Wallace trees are used to "accumulate" the result in an intermediate format. Our work proposes and evaluates ways in which similar techniques can be used to replace time-consuming additions in series calculations with Wallace tree compression. The technique is *applicable* to configurable hardware because, in a dynamically configurable system, it is practical to consider building specific hardware for dot-products or other repeated calculations. The technique is *effective* for configurable hardware because it removes addition's carry propagation logic from the critical path of these calculations, thus allowing them to be pipelined at much faster clock rates.

By using Wallace trees to accumulate results without carry propagation overhead, we can greatly accelerate both integer and floating-point calculations. We demonstrate our

• *The authors are with the Department of Electrical Engineering, Princeton University, Princeton, NJ 08544-5263.*
E-mail: {zhenluo, mrm}@ee.princeton.edu.

Manuscript received 15 Oct. 1998; accepted 1 Mar. 2000.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 108042.

0018-9340/00/$10.00 © 2000 IEEE

ideas on three designs. The first design is an integer unit that performs pipelined sequences of MAC (multiply-accumulate) operations; this pipelined design operates at a 72 MHz clock rate on an XC4036xla-9 FPGA and 170MHz clock rate on an XV300epq240-8 FPGA. The second and third designs perform floating-point accumulations (i.e., repeated additions) on 32-bit IEEE single-precision format numbers. One of them uses a conservative stall technique to respond to possible overflows; it operates at 40MHz on an XC4036xla-9 FPGA and 97MHz on an XV100epq240-8 FPGA. The other sign relies on compiler assistance to avoid overflows by breaking calculations into chunks of no more than 512 summation elements at a time. This approach yields an 80MHz clock rate on an XC4036xla-9 FPGA and 150MHz clock rate on an XCV100epq240-8 FPGA for 32-bit IEEE single-precision summations. These clock rates indicate the significant promise of this approach in implementing high-speed pipelined computations on FPGA-based systems.

The remainder of this paper is structured as follows: Section 2 introduces the basic idea of Delayed Addition calculation and presents a design for a pipelined integer multiply-accumulate unit based on this approach. Section 3 moves into the floating-point domain, presenting a design of a pipelined 32-bit floating-point accumulator with delayed addition. Building on this basic design, Section 4 then presents the floating-point accumulator with compiler-managed overflow avoidance. Section 5 discusses issues of rounding and error theory related to these designs, Section 6 presents related work, and Section 7 provides our conclusions.

2 Delayed Addition in a Pipelined Integer Multiply-Accumulator

2.1 Overview

A multiply-accumulator unit consists of a multiplier and an adder. For adders of 16 bits or less implemented in Xilinx FPGAs, the hardwired ripple-carry adder is the fastest. For adders more than 16 bits long, a carry-select adder is a good choice for fast addition in FPGA. It uses ripple-carry adders as basic elements and a few multiplexers to choose the result. Thus, it can still utilize the hardwired ripple-carry logic on FPGA to achieve relatively high speed.

Most of the multipliers that have been implemented so far in FPGAs are based on bit-serial multipliers [2], [14]. This is because bit-serial multipliers take much less area than any other kind of multipliers. Since they have a regular layout, it is easy to map on an FPGA to achieve very high clock rate. However, bit-serial multiplier requires a very long latency to produce a result. For two multiplicands of M and N bits long, it takes M + N clock cycles to get the product [7]. Although some implementations have tried to relieve this problem by multiplying more than one bit per cycle [2], we know of no such implementations with an overall throughput of more than 10MHz.

Bit-array multipliers also have a regular layout, which makes it easy to map on FPGA and to achieve high clock rates [3]. Unlike bit-serial multipliers, these produce one product every cycle. Thus, they can achieve a very high throughput at the price of large area cost. In the case of a 32-bit integer MAC with a 64-bit final result, we would expect to have a bit-array multiplier of 63 pipeline stages for multiplication and one more pipeline stage for accumulation. Thus, we would need a 64×64 CLB matrix to implement it [3]; this is too big an area cost.

Our design, as we will see next, has comparable performance to bit-array multiplier for vector MAC and is much more area efficient.

2.2 Background on Wallace Trees

Before continuing on detailed designs, we will first give a brief review on some basics of Wallace tree [10] and its derivatives [11]. One level of Wallace tree is composed of arrays of *3-2 adder*s (or *compressors*). The logic of a 3-2 adder is the same as a full adder except the carry-out from the previous bit has now become an external input. For each bit of a 3-2 adder, the logic is:

$S[i] = A1[i] \oplus A2[i] \oplus A3[i];$
$C[i] = A1[i]A2[i] + A2[i]A3[i] + A3[i]A1[i];$
For the whole array, S + 2C = A1 + A2 + A3

S and C are partial results that we refer to in this paper as the *pseudo-sum*. They can be combined during a final addition phase to compute a true sum. The total number of inputs across an entire level of a 3-2 adder array is the same as the bit-width of the inputs. Fig. 1a shows the layout of such an array example.

In some Wallace tree designs, *4-2 adder* arrays have also been used because they reduce the number of compressor levels required [11]. Each bit of such an array is composed of a 4-2 adder. The typical logic is:

$C_{out}[i] = A1[i]A2[i] + A2[i]A3[i] + A3[i]A1[i]$;
$S[i] = A1[i] \oplus A2[i] \oplus A3[i] \oplus A4[i] \oplus Cin[i];$
$C[i] = (A1[i] \oplus A2[i] \oplus A3[i] \oplus A4[i])C_in[i]$
$+ (\overline{A1[i] \oplus A2[i] \oplus A3[i] \oplus A4[i]})A4[i];$
For the whole array, S + 2C = A1 + A2 + A3 + A4

Fig. 1b shows the layout of an array example using 4-2 adders. At first glance, one might initially think that C_{in} and C_{out} are similar to the carry-in and carry-out in the ripple-carry adders. The key difference, however, is that C_{in} does not propagate to C_{out}. The critical path of an array of 3-2 or 4-2 adders is in the vertical, not horizontal direction. Furthermore, the logic shown maps well to coarse-grained FPGAs. With Xilinx 4000-series parts, we can fit each S or C, for either a 3-2 or 4-2 adder, into a single CLB using the F, G, and H function generators.

2.3 Design of Integer MAC with Delayed Addition

For an integer MAC unit, the implementation is straightforward because integers are fixed-point and are therefore aligned. Our design looks exactly like a traditional multiplier design with Booth encoding and Wallace tree except that a 4-2 adder array is inserted into the pipeline before the final addition. This MAC unit takes in two 32-bit integers as input and produces a 64-bit accumulated result. It is quite common for a MAC unit to have a wider bit-width accumulated result than its input operands in DSP processor designs. For example, in [26], the accumulated

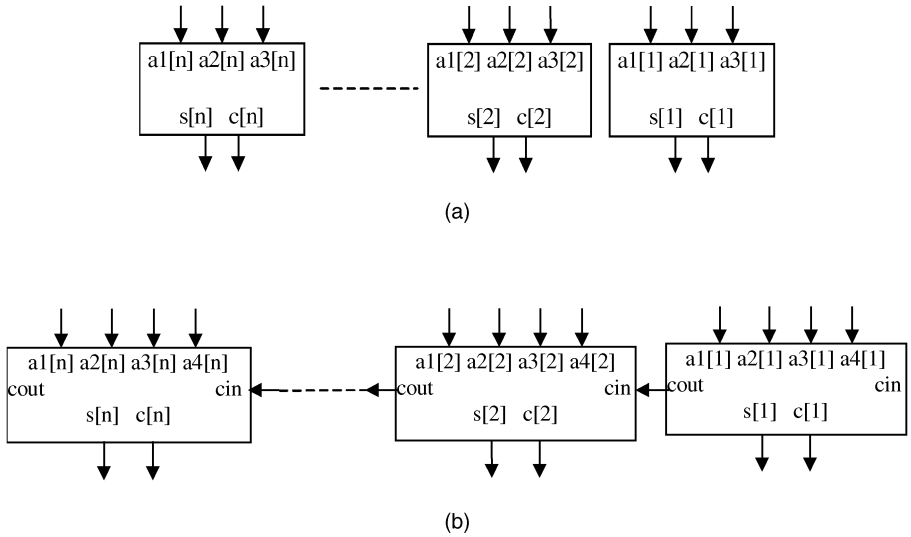


Fig. 1. (a) An array of n 3-2 adders. (b) An array of n 4-2 adders.

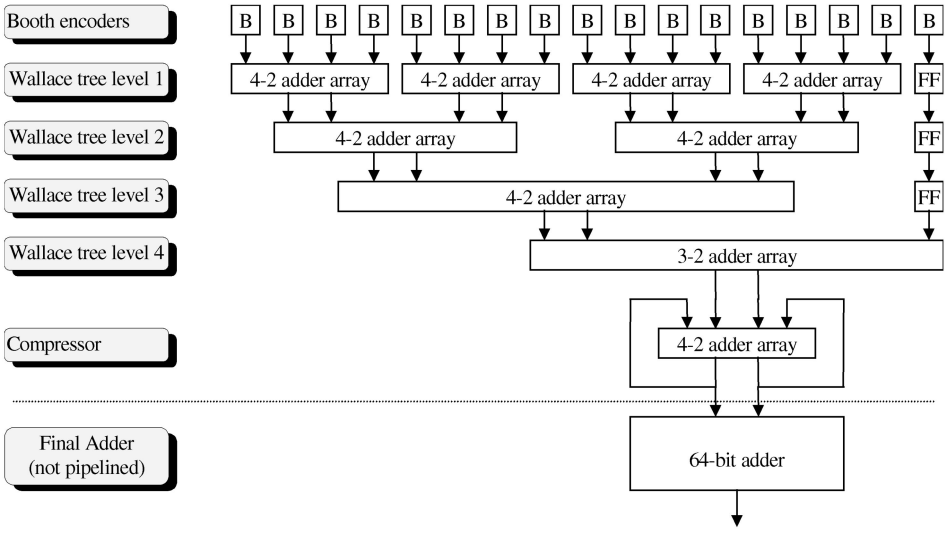


Fig. 2. Integer MAC with delayed addition.

result is 24 bits while the two input operands are all 16 bits. To achieve accumulation, we repeatedly execute:

$$\text{Pseudosum} = \text{Pseudosum} + (\text{the final two partial products of each multiplication}).$$

Recall that *pseudosum* refers to the S and C values currently being computed by a 3-2 or 4-2 adder array, awaiting the final addition that will calculate the true result. Fig. 2 shows a block diagram of our implementation.

Each level of a Wallace tree has a similar delay and this delay is also similar to that of a Booth encoder. Thus, considering in Fig. 2, a natural way to pipeline this design is to let each level of logic (above the dotted line) be one of the pipeline stages. The well-matched delays make for a very efficient pipelined implementation. The final compressor, just above the dotted line, stores and updates the pseudosum every cycle. When the repeated summation is complete, a final add (not part of the pipeline) converts this intermediate form to a true sum result.

The pseudosum is updated each cycle, but the final adder is only used when the full accumulation is done. Therefore, it is not one of the pipeline stages, but, rather, constitutes a postprocessing step, as shown in Fig. 3. With this structure, the carry propagation time for the final addition is no longer on the critical path that determines the clock rate of the pipelined MAC design. For sufficiently long vectors, this final addition time, done only once per entire summation rather than once per element, will be negligible even compared to the faster vector MAC calculations of this design.

2.4 Design Synthesis Results

For all the designs in this paper, we used Xilinx Foundation tools (V2.1) with embedded Synopsys fpga_express (V3.3) for the synthesis. In order to remove the bottleneck at the pad inputs, we added an extra pipeline stage before the booth encoder to buffer the chip inputs. We used both the popular XC4000 parts and the latest Virtex-E parts to obtain the synthesis results. Timing constraints are specified in a

cycle #	1	2	3	4	5	6	7	8	9	10	11	Combinational
Input 1	BTH	W1	W2	W3	W4	CPR						
Input 2		BTH	W1	W2	W3	W4	CPR					
Input 3			BTH	W1	W2	W3	W4	CPR				
Input 4				BTH	W1	W2	W3	W4	CPR			
Input 5					BTH	W1	W2	W3	W4	CPR		
Input 6						BTH	W1	W2	W3	W4	CPR	Final Addition

Fig. 3. Pipeline diagram of Integer MAC: $\sum_{i=0}^{5} A[i]B[i]$. The stages marked: BTH (Booth encoders), W1 (Wallace tree level 2), W3 (Wallace tree level 3), W4 (Wallace tree level 4), and CPR (Compressor) refer to the six pipeline stages shown in Fig. 2. The final addition is performed only once per summation and does not impact the pipelined clock rate.

user constraint file (.ucf file), where we listed the timing requirement for all the critical paths. The PAR (placement and routing) worked through successfully and Timing Analyzer gives all the timing information after our design is completely placed and routed. The synthesis results we get for the above design are listed in Table 1.

To demonstrate the advantage of delayed addition, we compared our design to an integer MAC composed of a traditional integer multiplier and an adder in which the adder is used for accumulating the result of the multiplier. To trade-off between pipeline speed and area cost, we divided the final adder into two pipeline stages. In the first stage, the lower 32-bit addition is carried out. In the second stage, we perform the addition of the upper 32. The synthesis result of this design is also listed in Table 1. Timing analysis shows it is exactly the two pipeline stages of the adder that are the bottleneck of the whole design. However, further pipelining the adder will involve a much larger area cost and is not likely to give any performance gain due to the long wiring delays in FPGA.

From Table 1, we can see that, by using the delayed addition algorithm, we have achieved a higher pipeline speed than the traditional multiplier and accumulator design. According to the data above, if we use XC4000 series, an IMAC with the delayed addition would require

$$7 + (\mathrm{N} - 1) + 3 = \mathrm{N} + 9$$

cycles for an integer inner product of length N to complete, where 7 stands for the number of pipeline stages, 3 stands for the cycle time for the final addition. The overall latency for this design would thus be $15\mathrm{ns} \times (\mathrm{N} + 9) = (15\mathrm{N} + 135)\mathrm{ns}$. For an IMAC with traditional multiplier and adder, the overall latency for an inner product of size N using traditional IMAC would be

$$10 + (\mathrm{N} - 1) = \mathrm{N} + 9$$

cycles as well. Thus, the delayed addition design has a performance speedup of 120 percent according to the maximum frequency listed in Table 1.

Using similar analysis, if we use Virtex-E series, an IMAC with delayed addition would require

$$7 + (\mathrm{N} - 1) + 2 = \mathrm{N} + 8$$

cycles to complete since the final addition would only require two cycles to complete in this case. However, an IMAC with traditional multiplier and accumulator design would still require

$$10 + (\mathrm{N} - 1) = \mathrm{N} + 9$$

cycles to complete. Thus, the delayed addition design has a performance speedup of 110 percent. The performance speedup is diminished in Virtex-E series because their carry logic is even faster. In Virtex-E series, a 16-bit adder takes 4.3ns, while a 64-bit adder only takes 6.3ns [25].

TABLE 1
Synthesis Results for Pipelined Integer MAC with Delayed Addition and Pipelined Integer Multiplier (with Adder)

Designs	CLBs used	Flip-flops used	Pipeline stages	Speed (MHz)	Final Adder Delay (ns)
IMAC (delayed addition)	1296	1834	7	72	34.04
IMAC (multiplier + adder)	1296	1995	10	60	N/A

(a)

Designs	CLBs used	Flip-flops used	Pipeline stages	Speed (MHz)	Final Adder Delay (ns)
IMAC (delayed addition)	1573	1834	7	170	10.97
IMAC (multiplier + adder)	1547	1995	10	155	N/A

(b)

(a) Synthesis results fo XC4000 seried (XC403xlahq208-9). (b) Synthesis results for Virtex-E series (XCV300epq240-8).

3 Using Delayed Addition in a Floating-Point Accumulator

Multiply and accumulation also appears frequently in floating-point applications. For example, of the 24 Livermore Loops, five loops (loop 3, 4, 6, 9, 21) are basically long vector inner-product-like computation [17]. In certain applications, such as using the conjugate gradient method in Space-Time Adaptive Processing [27], [28], multiply and accumulation dominates the whole computation process. Thus, it would be ideal if

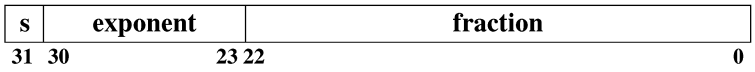


Fig. 4. IEEE single precision format. S is the sign, exponent is biased by 127. If exponent is not 0 (normalized form), mantissa = 1.fraction. If exponent is 0 (denormalized forms), mantissa = 0.fraction.

we could also use our delayed addition techniques to build a floating-point multiply and accumulator to speed up this kind of computations, like what we did in the integer case.

However, a floating-point MAC unit uses too much area to fit on a single FPGA chip. The major reason is that floating-point accumulation is a much more complex process than the integer case, as explained below. Rather than a MAC unit, we instead focus here on a floating-point accumulator using delayed addition. We first give a brief review of traditional approaches, then describe how we have used delayed addition techniques to optimize performance.

3.1 Traditional Single-Precision Addition Algorithm

As shown in Fig. 4, a traditional floating-point adder would first extract the 1-bit sign, 8-bit exponent, and 23-bit fraction of each incoming number from the IEEE 754 single precision format. By checking the exponent, the adder determines if each incoming number is denormalized. If the exponent bits are all "0," which means the number is denormalized, the mantissa is 0.fraction, otherwise, mantissa is 1.fraction. Next, the adder compares the exponents of the two numbers and shifts the mantissa of the smaller number to get them aligned. Sign-adjustments also occur at this point if either of the incoming numbers is negative. Next, it adds the two mantissas; the result needs another sign-adjustment if it is negative. Finally, the adder renormalizes the sum, adjusts the exponent accordingly, and truncates the resulting mantissa into 24 bits by the appropriate rounding scheme [2].

The above algorithm is designed for a single addition, rather than a series of additions. Even more so than in the integer case, this straightforward approach is difficult to pipeline. One problem lies in the fact that the incoming next-element-to-be-summed must be aligned with the current accumulated result. This adds a challenge to our delayed addition technique since we do not keep the accumulated result in its final form and, thus, cannot align incoming addends to it. Likewise, at the end of the computation, renormalization also impedes a delayed addition approach.

For these two problems, we have come up with two solutions:

1. Minimize the interaction between the incoming number and the accumulated result. To achieve this, we **self-align** the incoming number on each cycle, rather than aligning it to the Pseudosum. Section 3.2.1 will describe self-alignment in more detail.
2. Use the delayed addition for accumulation only. Postpone rounding and normalization until the end of the entire accumulation. This approach is also used when implementing MAC in some full-custom IC floating-point units [12].

3.2 Our Delayed Addition Floating-Point Accumulation Algorithm

This section describes our approach for delayed addition accumulation in floating-point numbers. Similar to what we did in Integer MAC, we repeatedly execute pseudosum = pseudosum + incoming operand. Each incoming operand is an IEEE single-precision floating-point number, with 1-bit sign, 8-bit exponent (EXP[7-0]) and 23-bit fraction. For simplicity of discussion, we consider the exponent bits as three subfields: high-order exponent, a decision bit, and low-order exponent. High-order exponent refers to the EXP[7-6], the decision bit is EXP[5], and low-order exponent refers to EXP[4-0]. We take different actions according to the value of these three fields.

Like the traditional adder, our design first extends the 23-bit fraction into 24-bit mantissa. However, unlike the traditional adder, we choose not to align the incoming operand and the current pseudosum directly because that way the alignment process could easily become the bottleneck of the whole pipeline. In a traditional adder, the incoming operand interacts with the accumulated pseudosum throughout the alignment process, which makes further pipelining impossible. Instead, we keep summary information about the high-order exponent of the accumulated result and align its mantissa to a fixed boundary according the its low-order exponent. We refer to this technique as "self-alignment" and describe it below.

3.2.1 Self-Aligning Incoming Operands

There are two ways to align two floating-point numbers. The common way is to shift the mantissa of one number by d bits, where d stands for the difference of the exponents of the two numbers. Another way is to instead shift both mantissas to some common boundaries. Traditional floating-point adders employ the first method. In our case, however, the second way is used since we would like to minimize the interactions of the incoming number and the accumulated pseudosum.

We could have fully "unrolled" the incoming operand and the accumulated pseudosum by left-shifting their mantissas the number of bits denoted by their exponents except for the huge area cost involved. In that case, the shifted mantissa would be as long as 255 (the largest 8-bit exponent possible) + 24 (the width of single precision mantissa) = 279 bits. Because of this, we only left-shift the mantissa the number of bits denoted by the low-order exponent (EXP[4-0]) in our design. Since low-order exponent is a 5-bit quantity, the largest decimal it can express is 31. Thus, by left-shifting to account for low-order bits, we have extended the width of our mantissa to 55 bits. Although this is still wide, our design can fit into a single piece of Xilinx 4036, as we will see later, and this gives us the ability to garner truly high-performance single-precision floating-point from an FPGA-based design.

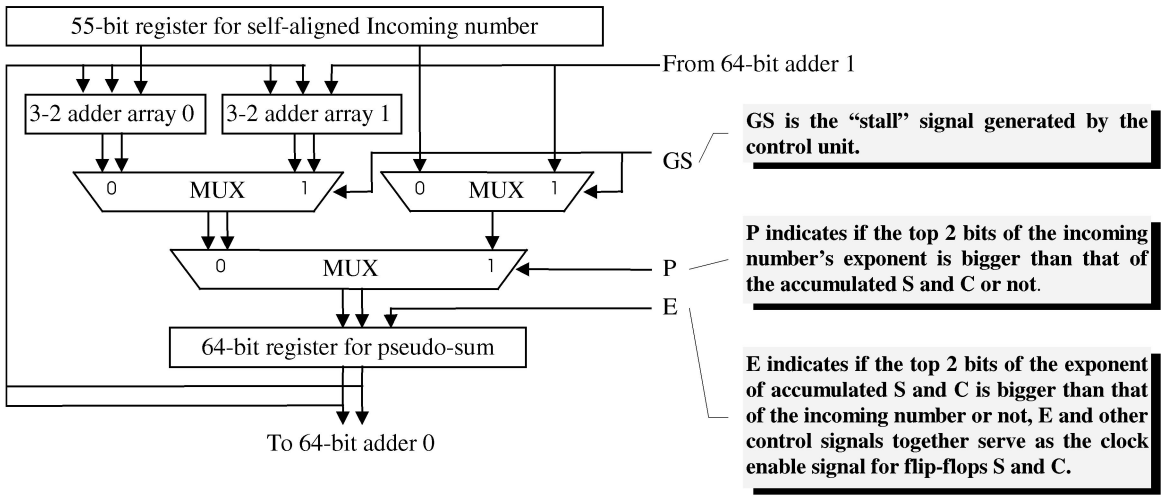


Fig. 5. Compressor design (compressor-0).

In the above self-aligning process, we did not take into consideration of the high-order exponent (EXP[7-6]) and decision bit (EXP[5]). Thus, the shifted mantissa of the incoming operand is still not perfectly aligned with that of the current pseudosum. We used the fact below to solve this remaining problem.

In single-precision IEEE floating-point, the mantissa is only 24 bits wide. Thus, if we try to add two originally normalized numbers that differ by more than 2^{24} times, alignment will cause the smaller of the two numbers to be "right-shifted" out of the expressible range for this format. For example, $2^{26} + 2^2 = 2^{26}$ in single-precision calculations. Our algorithm efficiently uses this fact to identify the similar cases and handles them appropriately.

Once self-aligned, the incoming number can be thought of as

$$mi'(\text{the 55-bit mantissa}) \times 2^{64 \cdot \text{EXP}[7\text{-}6]} \times 2^{32 \cdot \text{EXP}[5].}$$

Meanwhile, our pseudosum is stored as

$$mp'(\text{the 64-bit mantissa}) \times 2^{64 \cdot \text{EXP}[7\text{-}5]} \times 2^{32 \cdot \text{EXP}[5].}$$

If the current pseudosum and the incoming operand are **identical** in decision bit (EXP[5]), then if the high-order exponent (EXP[7-6]) of the incoming number is **bigger than** that of the pseudosum, the mantissa of the pseudosum will be shifted out of the expressible range as long as it is no more than 64 bits wide. In this case, we simply replace the current pseudosum by the incoming operand. On the other hand, if the high-order exponent of the incoming number is **smaller than** that of the pseudosum, the incoming number will be shifted out of the expressible range since mi' is less than 64 bits wide. Thus, we simply ignore the incoming operand. The compression will only take place when high-order exponent of the pseudosum is **equal to** that of the incoming number.

Note that if the current pseudosum and the incoming operand are **not identical** in EXP[5], then determining the appropriate response would actually require subtracting the two full exponents to determine by how much they differ. This would pose a bottleneck in the pipeline; thus, we hope to avoid this scenario entirely. This leads to our design described in Section 3.2.2 below.

3.2.2 Compressor Implementation Details

In order to avoid the undesirable scenario of unequal decision bits, we actually keep two running pseudosums. One compressor, referred to as compressor-0, takes care of incoming operands whose decision bit is "0" and the other compressor (compressor-1) handles those which has a decision bit of "1." We simply shunt each incoming operand to the appropriate compressor, as shown in Fig. 6. In this way, we can always take operations corresponding to the high-order exponent as described above. The two pseudosums from compressor-0 and compressor-1 are both added together during the final add stage as a postprocessing step following the pipelined computation.

Fig. 5 shows the design layout for one of the two compressor units in the design, namely compressor-0. Compressor-1 has essentially identical structure except that it cross-connects with adder-0, as shown in Fig. 6. The running pseudosum is stored as the Wallace tree's S and C partial results in the 64-bit registers shown.

Were it not for the possibility of either pseudosum overflowing, the design would now be complete. Since the accumulated result may exceed the register capacity, we have also devised a technique for recognizing and responding to potential pseudosum overflows. Since we are not doing the full carry-propagation of a traditional adder, we cannot use the traditional overflow-detection technique of comparing carry-in and carry-out at the highest bit. In fact, without performing the final add to convert the pseudosum to the true sum, it is impossible to **precisely** know a priori when overflows will occur.

Our approach instead relies on conservatively determining whenever an overflow **might** occur and, then, stalling the pipeline to respond. We can conservatively detect possible overflow situations by examining the top three bits of the S and C portions of the pseudosum and the sign bit from the 55-bit incoming operand. We have used *espresso* to form a minimized truth table generating the *GlobalStall*

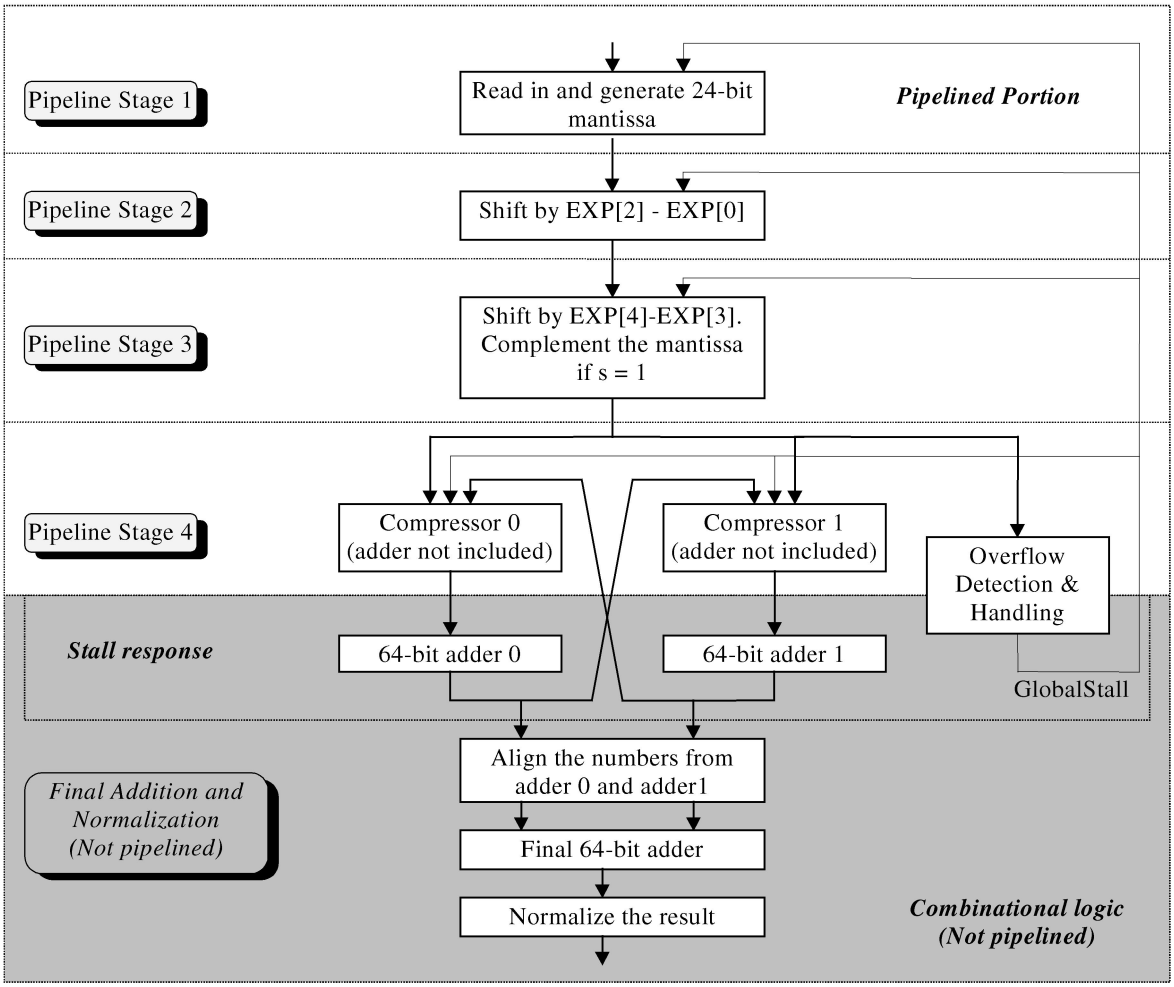


Fig. 6. Floating-point accumulator pipelining scheme.

signal (*GS* in Fig. 4) as a Boolean function of these seven bits. As shown in Fig. 5, the GlobalStall signal is used as the clock enable signal on the first three pipeline stages; when it is asserted, the pipeline stalls and no new operands are processed until we respond to the possible overflow.

Since the design's two compressors are summing different numbers, they will, of course, approach overflow at different times since only one number is added a time. Our design, however, does overflow processing in both compressors whenever either compressor's GlobalStall signal is asserted. This coordinated effort avoids cases where overflow handling in one compressor is immediately followed by an overflow in the other compressor and it potentially reduces the number of stalls needed, too, since we process these two pseudosums in parallel during the stall.

When a stall occurs, our response is to sum the S and C portions of each compressors' pseudosum using the 64-bit adders shown in the Stall Response box in Fig. 5. This is a traditional 64-bit addition incurring a significant carry-propagation delay, but, since it occurs during the stall-time, it does not lie on the critical path that determines the design's pipelined clock rate. (As long as stalls are infrequent, it does not noticeably impact performance.) The decision of what to do with the newly formed sum depends on its value, i.e., it depends on whether 1) an overflow truly occurred or 2) we were overly conservative in our stall detection. In cases where an overflow does occur, the value of EXP[5] in the pseudosum will change. Recall that compressor-0 is to handle the accumulation of incoming operands whose EXP[5] bit is 0, with a pseudosum whose EXP[5] bit is also 0. If the pseudosum overflow causes EXP[5] to change value, then we need to pass the newly computed full sum over to the other compressor. This is why the design in Fig. 6 includes the cross-coupled connections of adder-1 to compressor-0 and vice versa. When we are overly conservative in predicting a stall, EXP[5] will not change values. In this case, we retain the pseudosum in its current form.

3.3 Experimental Results

Fig. 6 shows the block diagram of this design and Table 2 summarizes the synthesis results. Because this is a floating-point accumulator rather than a MAC unit, it is actually smaller than the integer MAC unit discussed in the previous section. Using four pipeline stages, our design attains a clock rate of 40MHz with an XC4000 part and 97MHz with a Virtex-E part. Because of extra bookkeeping required to renormalize the final result, the postprocessing

TABLE 2
Synthesis Results for Pipelined Floating-Point Accumulation with Delayed Addition

Xilinx part number	CLB matrix size	CLB used	Flip-flops	Pipeline stages	Speed (MHz)	Final Add and Norm. Delay (ns.)
XC4036xlahq208-9	36×36	939	378	4	40	96.5
XCV100epq240-8	$20 \times 30 \times 2$	903	378	4	97	48.9

delay in this design is larger. For the XC4000 part, this delay corresponds to roughly four pipelined clock cycles, while, for the Virtex-E part, this delay corresponds to roughly five clock cycles. As in the integer case, this difference between the final add time and the pipelined clock cycle time highlights the utility of delayed addition. By pulling this delay off the vector computation's critical path, we pay for it only once per vector, not on each clock cycle.

According to the data above, using the same analysis as in Section 3.3 and assuming there is no stall during the computation, we will have to wait

$$4 + (\mathrm{N} - 1) + 4 = \mathrm{N} + 7$$

cycles for an accumulation of length N to complete for an XC4000 part and

$$4 + (\mathrm{N} - 1) + 5 = \mathrm{N} + 8$$

cycles for a Virtex-E part.

Since each stall causes a 3-cycle bubble in the pipeline and too many stalls may eventually incur expensive system interrupt, we also want to make sure how frequent stalls might be when we accumulate N numbers. We did two simulations. Simulation I used 100,000 uniformly distributed floating-point numbers with their absolute values ranging from 2^{-31} to 2^{31}. Because positives and negatives are balanced, we did not even meet one case of stalling. Simulation II uses 100,000 uniformly distributed positive floating-point numbers ranging from 0 to 2^{31} and we only found 24 cases of stalling. Summing these 100,000 numbers would need 100,006 cycles so that 72 stall cycles are negligible. From this experiment, we conclude that overflow and stalling pose little problem for most applications as long as we have a reasonably large local buffer for operands. As we will show and exploit in Section 4, we can prove that, for vectors shorter than 512 elements, there is no chance of stalling at all.

4 Floating-Point Accumulator with Compiler-Managed Overflow Avoidance

The main reason why we have the overflow detection and handling logic in the previous design is to avoid possible overflow of the pseudosum after a number of operations. However, the stall-related logic is very complicated and has a big area cost. Worst of all, it sits on the critical path of our design and slows down the pipeline speed. To avoid the area and speed overhead due to overflow detection and handling, we present a different style design here. This design omits overflow handling by relying on the compiler to break a large accumulation into smaller pieces so that overflow is guaranteed *not* to occur when each of these pieces is executed.

Avoiding area overhead for stall handling is desirable, but we will not have much gain in our design if we have to break an accumulation into very small pieces. Our goal is to determine a bound of how often the stall will occur. The largest incoming mantissa that can be fed into one compressor is $11\ldots1100\ldots00$ (the first 24 bits are "1"s and the rest 31 bits are "0"s). This is less than 2^{55}. Thus, if the pseudosum is stored in an n-bit ($\mathrm{n} > 55$) register, we may have an overflow every $\mathbf{2^{n-55}}$ accumulations. We can use this formula to choose a suitable n for specific applications.

In this design, we choose the pseudosum width to be 64, as in the design from Section 3. Since $64 - 55 = 9$, overflows may occur every $2^9 = 512$ accumulations. We will rely on the compiler to break summations into 512-element vectors. For the following loop, we can transform the source code on the top to the bottom. A procedure ACCUMULATE(A, n) is used to compute the accumulation in configurable hardware, where A is the pointer to the floating point array and n is the number of floating point numbers to be accumulated.

```
for (i=0; i< N; i++)
  S += A[i];
          ↓
for (i=0; i< (N>>9); i+=512)
  S += ACCUMULATE(A[i],512);
```

Since now we no longer need the overflow checking and handling in this design, all the related components in our previous design can be removed and the two 64-bit adders below the compressors in Fig. 4 are replaced by an array of 4-2 adders. This also greatly simplifies the control logic on the critical path and enables us to further pipeline our design. Fig. 7 shows a resulting 5-pipeline-stage design.

Synthesis results summarized in Table 3 show that we have dramatically increased the speed of the conservative design and have achieved a high clock rate of 80MHz with the XC4000 part and 150MHz with the Virtex-E part. For an accumulation of size N, we will need $5 + (\mathrm{N} - 1) + 7 = \mathrm{N} + 11$ cycles to complete the whole computation with an XC4000 part and $5 + (\mathrm{N} - 1) + 7 = \mathrm{N} + 11$ cycles, too, with a Virtex-E part.

5 Discussion

In this section, we discuss some of the issues raised by our delayed addition technique, particularly with respect to floating-point calculations. The IEEE floating-point

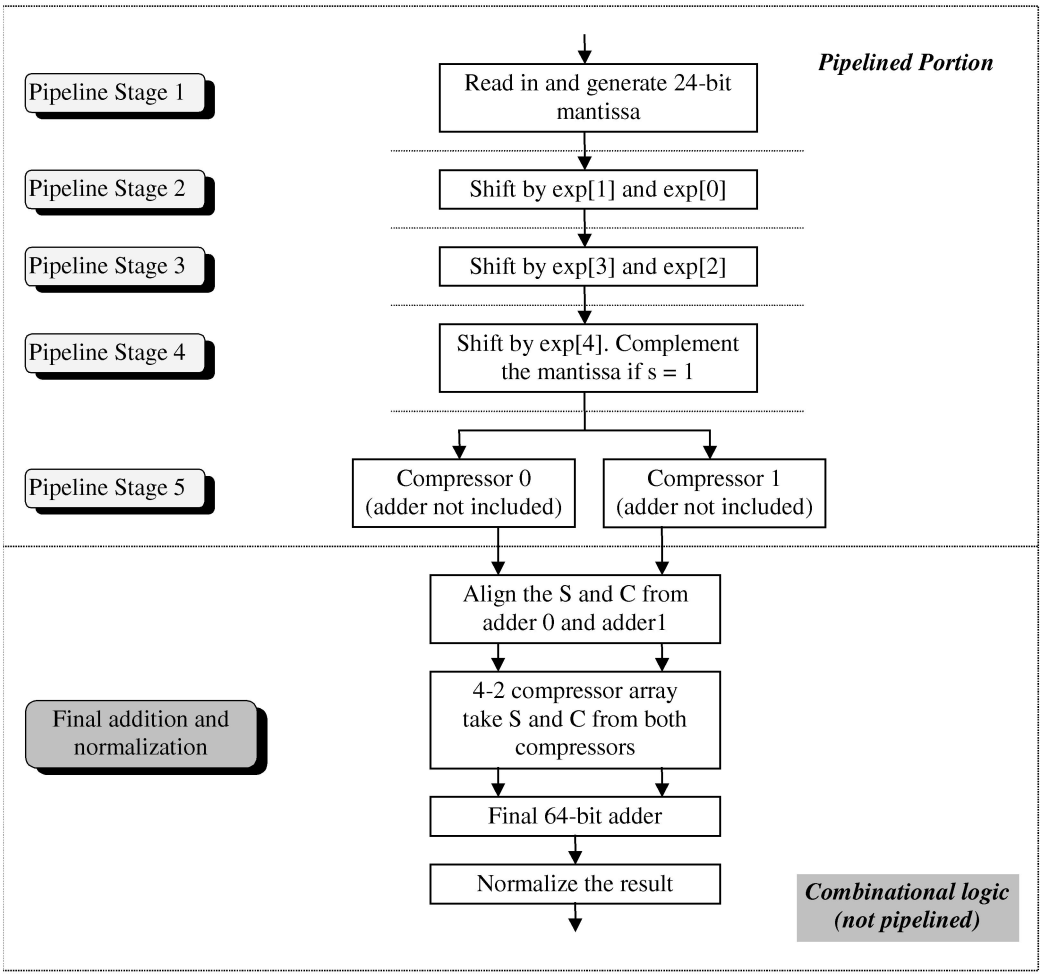


Fig. 7. Floating-point accumulator with compiler-managed overflow avoidance pipelining scheme.

TABLE 3
Synthesis Results of Floating-Point Accumulation with Delayed Addition and Compiler-Managed Overflow Avoidance

Xilinx part number	CLB matrix size	CLB used	Flip-flops	Pipeline stages	Speed (MHz)	Final Addition Delay (ns)
XC4036xlahq208-9	36×36	940	441	5	80	86.08
XCV100epq240-2	$20 \times 30 \times 2$	1059	441	5	150	45.91

standard [1] specifies the format of a floating-point number of either single or double precision, as well as rounding operations and exception handling. It provides us with both a representation standard and an operation standard. The representation standard is helpful for transporting code from one system to another. The operation standard, together with the representation standard, works to ensure that same result can be expected for floating point calculations on different platforms (if they all choose the same rounding scheme). Our designs, described in Sections 3 and 4, abide by the representation portion of the IEEE standard and implement single-precision IEEE floating-point including denormalized numbers.

In the accumulator design, we make the basic assumption that the additions performed are commutative. This assumption is also routinely made by most current-generation microprocessors, where out-of-order execution also assumes that floating-point operations on independent registers are commutative. Similarly, many compiler optimizations geared at scientific code also assume commutativity; optimizations such as loop interchange and loop fusion reorder computations as a matter of course.

Our operations do, in effect, reorder computations. For any accumulation sequence $A_1 + A_2 + \ldots + A_n$, assume there is no overflow in either compressor throughout the computation and assume for now that our rounding scheme is exactly like in a normal adder. We first divide array A into two arrays A0 and A1 according to their EXP[5]. All the numbers in A0 $(A0_1, A0_2, \ldots A0_m)$ have EXP[5] = 0, while all the numbers in A1 $(A1_1, A1_2, \ldots A1_{n-m})$ have EXP[5] = 1.

Let $A0_1, A0_2, \ldots A0_K$ be the numbers with the largest high order exponent bits in A0 and $A0_{K+1}, A0_{K+2}, \ldots A0_m$ be the rest numbers in A0. Similarly, let $A1_1, A1_2, \ldots A1_J$ be the numbers with the largest high order exponent bits in A1 and $A1_{J+1}, A1_{J+2}, \ldots A1_{n-m}$ be the rest numbers in A1. Our accumulator virtually did the accumulation $(A0_1 + A0_2 + \ldots + A0_K) + (A1_1 + A1_2 + \ldots + A1_J)$ while ignoring all the other numbers in A0 and A1. This result is the same as that we get by using a normal floating-point adder to do the accumulation in the following order

$$(A0_1 + A0_{K+1} + A0_{K+2} + \ldots + A0_m + A0_2 + A0_3 + \ldots + A0_K) + (A1_1 + A1_{J+1} + A1_{J+2} + \ldots + A1_{n-m} + A1_2 + A1_3 + \ldots + A1_J).$$

This is because, in a normal adder,

$$A0_1 + A0_{K+1} = A0_1, A0_1 + A0_{K+2} = A0_1, \ldots.$$

Thus, the above computation is equivalent to

$$(A0_1 + A0_2 + \ldots + A0_K) + (A1_1 + A1_2 + \ldots + A1_J).$$

Furthermore, we use a different rounding system in the accumulator design. Instead of rounding each time after the addition, we only round once for each compressor throughout the accumulation if there is no compressor overflow. This, in fact, reduces the rounding error for the whole accumulation since we do fewer roundings throughout the computation, but it will certainly produce a different result from what we get in a general-purpose microprocessor even if computation are carried out in exactly the same order. However, according to [18], even for all the systems that conform to IEEE 754, "most programs will actually produce different results on different systems for a variety of reasons." And, even for the same executable running on the same machine under the same operating system, we could have different the results due to different run time environment. For example, an extra cache miss in the second run of a program could cause the floating point instructions to be reordered in a different way from how they were reordered in the first fun, potentially producing a different result. Thus, we care more about if the computation results are reasonable than if the results are the same as in a microprocessor system.

6 Related Work

This paper touches on areas related to both computer arithmetic and configurable computing. Patterson et al. provide an overview of computer arithmetic in Appendix A of [6]. They concentrate on logic principles of various designs of basic arithmetic components. In addition, innumerable books and papers go into more detail on adder and multiplier designs at the transistor level. For example, Weste and Eshraghian [7] provide a detailed discussion on various multiplier implementations and Wallace trees. Examples of recent full-custom multiplier design can be found in the work by Ohkubo et al. [8] and Makino et al. [9]. We can also find recent multiplier designs in state-of-the-art microprocessors such as DEC Alpha 21164 [15] and SUN Ultrasparc [16]. These designs, as with most, use Booth encoders and Wallace trees

Our approach employs the idea used in Wallace trees. Wallace used 3-2 adders to build up the first Wallace tree [10]. There have been many derivatives since then. The most important change is to use 4-2 adders to replace the 3-2 adders in the original implementation. In many designs, pass transistors, rather than full CMOS logic gates, are used to build 4-2 adders to improve the circuit speed, as we can see in Heikes and Colon-bonet [12].

Early in 1994, Canik and Swartzlander [3] discussed how to map a bit-array multiplier to Xilinx FPGA. They built an 8×8 bit-array multiplier for integer multiplication with Xilinx 3000 series and their fully pipelined implementation on XC3190-3 achieved more than 100 Mhz. Now, almost all the major FPGA vendors have provided their implementations of integer multiplier or multiply-accumulator of 16 bit or shorter length [22], [24]. A comparison of the speed of these implementations can be found in [23]. Most of them are based on bit-serial or bit-array multipliers. However, bit-array multiplier has too big an area cost for long integer multiplication. We know of no implementations of 32-bit-array multiplier nor have we seen any implementations of 32-bit integer multiplier or multiply-accumulators based on bit-serial or other algorithms.

More recent work has examined implementing floating-point units in FPGAs [2], [3], [13], [14]. Louca et al. [2] present approaches for implementing IEEE-standard floating-point addition and multiplication in FPGAs. They used a modified bit-serial multiplier and prototyped their designs on Altera FLEX8000s. Ligion et al. [14] also discuss the implementation of IEEE single precision floating-point multiplication and addition and they accessed the practicability of several of their designs on XILINX 4000 series FPGA. Shirazi et al. [13] talk about the limited precision floating point arithmetic. They have adapted IEEE standard for limited precision computation like FFT in DSP.

Finally, several authors have discussed rounding and error theory [4], [5], [12], [21]. We can see how people deal with the error in physics from Taylor [4]. Wilkinson [5] and Heikes and Colon-bonet [12] touch on rounding error in MAC and inner-product units. In chapter 4 of [21], Higham discussed how rounding error depends on the order of summation. Goldberg [20] presents a detailed description on IEEE 754 standard error analysis of floating-point operations under this standard and Priest [18] talks about the impact on the error analysis of different implementations of IEEE standard.

7 Conclusions

Within many current FPGAs, carry propagation represents a significant bottleneck that impedes implementing truly high-performance pipelined adders, multipliers, or Multiply-accumulate (MAC) units within configurable designs. This paper describes a delayed addition technique for improving the pipelined clock rate of designs that perform repeated pipelined calculations. Our technique draws on Wallace trees to accumulate values without performing a full carry-propagation; Wallace trees are universally used within the multiply units in high-performance processors.

The unique nature of configurable computing allows us to apply these techniques not simply within a single calculation, but, rather, across entire streams of calculations.

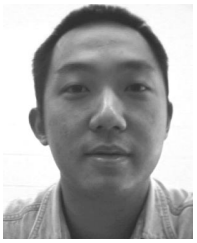
We have demonstrated the significant leverage of our approach by presenting three designs exemplifying both integer and floating-point calculations. The designs operate at pipelined clock rates from 40 to 72 MHz on Xilinx 4000 series and from 97 to 170 MHz on Xilinx Virtex-E series. These techniques and applications should help to broaden the space of integer and floating-point computations that can be customized for high-performance execution on current FPGAs.

ACKNOWLEDGMENTS

This research was supported in part by DARPA Grant DABT63-97-1-0001 and by a grant from the U.S. National Science Foundation (NSF) Experimental Systems Program. In addition, Zhen Luo was supported in part by a Princeton University Gordon Wu fellowship and Margaret Martonosi is the recipient of an NSF Career Award. This paper is a revised version of the paper "Using Delayed Addition Techniques to Accelerate integer and floating point Calculations in Configurable Hardware" presented at SPIE 98.

REFERENCES

[1] IEEE Standards Board, "IEEE Standard for Binary Floating-Point Arithmetic,"Technical Report ANSI/IEEE Std. 754-1985, IEEE, New York, 1985.
[2] L. Louca, T.A. Cook, and W.H. Johnson, "Implementation of IEEE Single Precision Floating Point Addition and Multiplication on FGPAs," *Proc. IEEE Symp. FPGAs for Custom Computing Machines,* Apr. 1996.
[3] R.W. Canik and E.E. Swartzlander, "Implementing Array Multipliers in XILINX FPGAs," *Proc. 1994 28th Asilomar Conf. Signals, Systems, and Computers,* 1994.
[4] J.R. Taylor, *An Introduction to Error Analysis.* Univ. Science Books, 1982.
[5] J.H. Wilkinson, *Rounding Errors in Algebraic Processes.* Prentice Hall, 1963.
[6] D.A. Patterson, J.L. Hennessy, and D. Goldberg, *Computer Architecture, A Quantitative Approach,* Appendix A, second ed. Morgan Kaufmann, 1996.
[7] N.H.E. Weste and K. Eshraghian, *Principles of CMOS VLSI Design,* second ed. Addison-Wesley, 1993.
[8] N. Ohkubo et al., "A 4.4 ns CMOS 54×54 Bit Multiplier Using Pass-Transistor Multiplexer," *IEEE J. Solid State Circuits,* vol 30, no. 3, pp. 251-256, Mar. 1995.
[9] H. Makino et al., "An 8.8 ns 54×54 Bit Multiplier with High Speed Redundant Binary Architecture," *IEEE J. Solid State Circuits,* vol. 31, no. 6, pp. 773-783, Mar. 1995.
[10] C.S. Wallace, "Suggestions for a Fast Multiplier," *IEEE Trans. Electronic Computers,* vol. 13, pp. 114-117, Feb. 1964.
[11] Y. Kanie et al., "4-2 Compressor with Complementary Pass-Transistor Logic," *IEICE Trans. Electron,* vol. E77-c, no. 4, pp. 789-796, Apr. 1994.
[12] C. Heikes and G. Colon-bonet, "A Dual Floating Point Coprocessor with an FMAC Architecture," *ISSCC Digest Technical Papers,* pp. 354-355, 1996.
[13] N. Shirazi, A. Walters, and P. Athanas, "Quantitative Analysis of Floating Point Arithmetic on FPGA Based Custom Computing Machines," *Proc. IEEE Symp. FPGAs for Custom Computing Machines,* pp. 155-162, Apr. 1995.
[14] W.B. Ligion III, S. McMillan, G. Monn, F. Stivers, and K.D. Underwood, "A Re-Evaluation of the Practicality of Floating-Point Operations on FPGAs," *Proc. IEEE Symp. FPGAs for Custom Computing Machines,* Apr. 1998.
[15] D.P. Bhandarkar, *Alpha Implementations and Architecture, Complete Reference and Guide.* Digital Press, 1996.
[16] R.K. Yu et al., "167 MHz Radix-4 Floating Point Multiplier," *Proc. 12th Symp. Computer Arithmetic,* pp. 149-54, July 1995.
[17] F.M. McMahon, "The Livermore FORTRAN Kernels: A Computer Test of Numerical Performance Range," Technical Report UCRL-55745, Lawrence Livermore Nat'l Laboratory, Univ. of California, Davis, Dec. 1986.
[18] D. Priest, "Differences among IEEE 754 Implementations," http://www.validgh.com/goldberg/addendum.html, 1997.
[19] Xilinx, "XC4000E and XC4000X Series Field Programmable Gate Arrays, Product Specification,"V1.4, Nov. 1997.
[20] D. Goldberg, "What Every Computer Scientist Should Know about Floating-Point Arithmetic," http://www.validgh.com/goldberg/paper.ps, 1991.
[21] N.J. Higham, *Accuracy and Stability of Numerical Algorithms.* SIAM, 1996.
[22] Microelectronics Group, Lucent Technologies, "Create Multiply-Accumulate Functions in ORCA FPGAs," Feb. 1997.
[23] Altera, "FLEX 10K v.s. FPGA performance,"Technical Brief 12, Sept. 1996.
[24] Altera, "Implementing Multipliers in Flex 10K Devices,"Application Note 53, Mar. 1996.
[25] Xilinx, "Virtex-E 1.8V Field Programmable Gate Arrays Datasheet Description v1.1," 1999.
[26] M. Nomura et al., "A 300-MHz 16-b 0.5 um BiCMOS Digital Signal Processor Core LSI," *IEEE J. Solid State Circuits,* vol. 29, no. 3, Mar. 1994.
[27] N.D. Gupta, "Reconfigurable Computing for Space-Time Adaptive Processing," master's thesis proposal, Dept. of Computer Science, Texas Tech Univ., Fall 1997.
[28] S.T. Smith et al., "Linear and Nonlinear Conjugate Gradient Methods for Adaptive Processing," *Proc. 1996 Int'l Conf. Acoustics, Speech, and Signal Processing,* May 1996.



Zhen Luo received his BS degree from the Computer Science and Technology Department of Peking University in 1996. He has been a PhD student in the Electrical Engineering Department at Princeton University since then. His research interests include configurable computing, computer arithmetic, and CAD for VLSI.

Margaret Martonosi earned her PhD from Stanford University in 1993 and also holds a master's degree from Stanford and a bachelor's degree from Cornell University, all in electrical engineering. She is currently an associate professor at Princeton University, where she has been on the faculty in the Department of Electrical Engineering since 1994. Her research interests are in computer architecture and the hardware-software interface and her group's current research focus is on hardware and software strategies for power-efficient computing. She is a senior member of the IEEE and a member of the IEEE Computer Society.