Using Delayed Addition Techniques to Accelerate Integer and Floating-Point Calculations in Configurable Hardware

Zhen Luo and Margaret Martonosi
Dept. of Electrical Engineering
Princeton University
{zhenluo,mrm}@ee.princeton.edu

Abstract

This paper proposes and evaluates an approach for improving the performance of arithmetic calculations via delayed addition. Our approach employs the idea used in Wallace trees to ***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 integer and floating-point designs that use this 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 37MHz clock rate on an XC4036XL-2 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 33 MHz clock rate. Finally, we also present a more aggressive 32-bit floating-point accumulator design that achieves a 66MHz clock rate. These designs demonstrate the utility of delayed addition for accelerating FPGA calculations in both the integer and floating-point domains.*

1 Introduction

When an arithmetic calculation is carried out in a microprocessor, each instruction typically has two source operands and one result. In many computations, however, the result of one arithmetic operation is just an intermediate result in a 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. Configurable hardware allows us to make specialized hardware for these cases, that optimizes the pipelining characteristics for the entire computation.

A typical multiplier on a full-custom integrated circuit (IC) 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 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 easily become an even greater bottleneck though; this is because it is hard to apply truly fast adder techniques within the constraints of current FPGAs. For example, in Xilinx 4000-series chips, the fastest 16-bit adder possible is a hardwired ripple-carry adder. The minimum delay for such an adder (in a -2 speed grade XC4000xl part) is more than four times the delay of an individual F or G function generator. 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 determining pipelined clock rate. Our work removes this bottleneck from the critical path, thereby improving the performance of dot-products and other series calculations.

As an example, consider the summation C of a vector A: $C = \Sigma_{i=0}^{99} A[i]$. Our goal is to accumulate the elements of A without paying the price of 99 complete additions. We observe that in traditional multiplier designs (e.g., the multiply units of virtually all recent microprocessors) 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 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 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 37MHz clock rate. 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 33MHz. The other uses a more aggressive design style which relies on compiler assistance to limit overflows by breaking calculations into chunks of no more than 512 summation elements at a time. This more aggressive approach yields a 66MHz clock rate 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 33MHz 32-bit floating-point accumulator with delayed addition. Building on this basic design, Section 4 then presents our more aggressive floating-point accumulator that can operate at 66MHz. 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 Background

Most of the multipliers that have been implemented so far in FPGAs are bit-array or bit-serial multipliers. There are two major reasons for this: (1) they can be very finely pipelined. (2) They are regular, thus easy to be mapped into a FPGA.

However, these designs have a high area cost. In the case of 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 will need a 64×64 CLB matrix to implement it [3]. This means it can't even fit into the largest Xilinx part available at this point.

Our design, as we will see next, has comparable performance 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 the traditional Wallace tree [10] and later derivatives [11]. One level of traditional 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:

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

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 width of a 3-2 adder array is the same as the bit-width of the inputs. Fig. 1 (a) 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

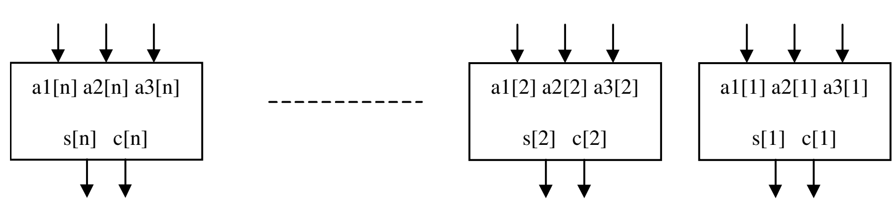

Fig 1. (a) An array of n 3-2 adders.

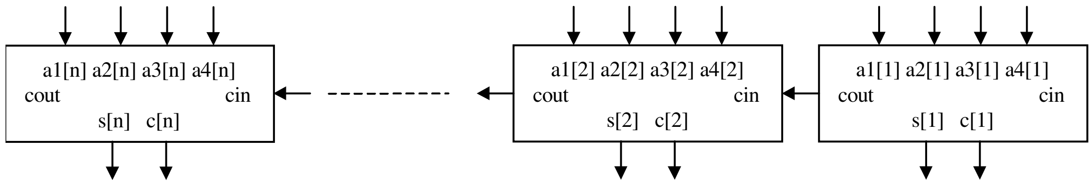

Fig 1. (b) An array of n 4-2 adders.

required [11]. Each bit of such an array is composed of a 4-2 adder. The typical logic is:

$$C_{out}[i] = A1[i]A2[i] + A2[i]A3[i] + A3[i]A1[i];$$
$$S[i] = A1[i] \oplus A2[i] \oplus A3[i] \oplus A4[i] \oplus C_{in}[i] \text{ and}$$
$$C[i] = (A1[i] \oplus A2[i] \oplus A3[i] \oplus A4[i])C_{in}[i] + \overline{(A1[i] \oplus A2[i] \oplus A3[i] \oplus A4[i])}A4[i]$$

For the whole array, $S + 2C = A1 + A2 + A3 + A4$

Fig. 1 (b) shows the layout of an array example using 4-2 adders. In both adder arrays, 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 not in the lateral direction. 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 except that a 4-2 adder array is inserted into the pipeline before the final addition. To achieve accumulation, we will repeatedly be executing

Pseudo-sum = Pseudo-sum + (the final two partial products of each multiplication)

Recall that *pseudo-sum* 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 the hardware architecture of our implementation.

Since each level of a Wallace tree has a similar delay, and since this delay is also similar to that of a booth encoder, a natural way to pipeline this design is to let each level of logic in Fig. 2 be one of the pipeline stages except for the final adder. The

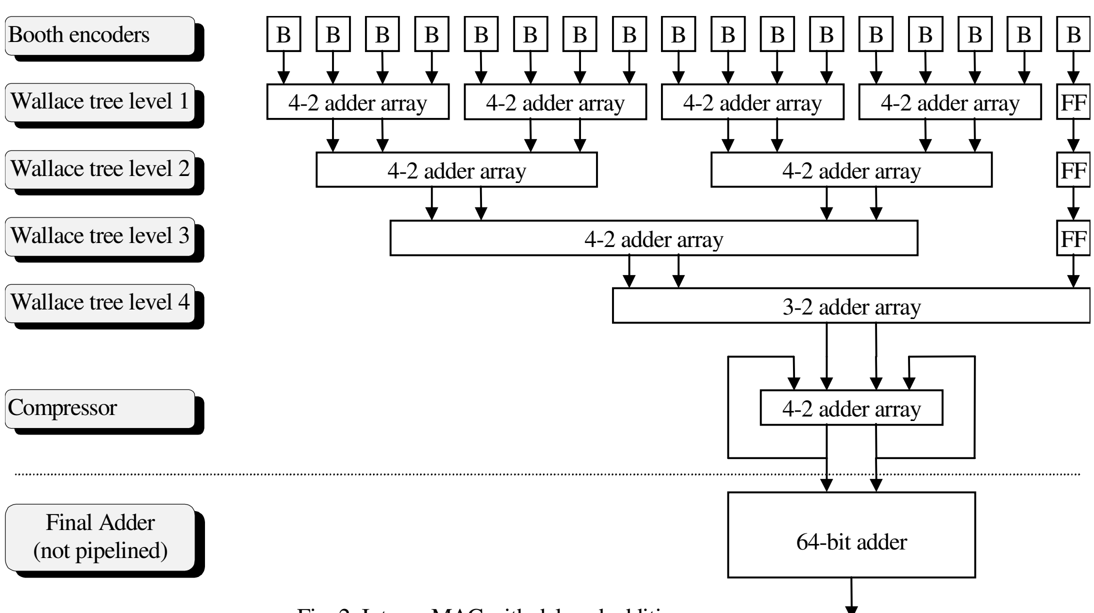

Fig. 2 Integer MAC with delayed addition.

pseudo-sum is updated each cycle, but the final adder will only be used when the full accumulation is done. Therefore, it is not one of the pipeline stages, but rather constitutes a post-processing step as shown in Figure 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 once per vector operation, will be negligible even compared to the faster vector MAC calculations of this design.

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 Pipeling diagram of Integer MAC of n (n = 6) input numbers. BTH (Booth encoders), W1 (Wallace tree level 1), W2 (Wallace tree level 2), W3 (Wallace tree level 3), W4 (Wallace tree level 4), CPR (Compressor) refer to the six pipeline stages shown in Figure 2.

2.4 Design Synthesis Results

For all the designs in this paper, we used the Synopsys fpga_analyzer tool (1997.08) to generate a .sxnf file from our VHDL input and we used Xilinx Foundation tools (V1.3) for the rest of the synthesis. All the timing information is given by Timing Analyzer after our design is completely placed and routed. The synthesis results we get from the above design are listed in Table 1. Most notably, our design easily fits in a Xilinx 4036 part and achieves a clock rate of 37MHz. The final addition delay, done once per vector as a post-processing step, takes roughly 36ns, or about 1.5 of the pipelined clock cycles. In traditional MAC designs, this additional 50% overhead for the add would be on the critical path, and thus the design would be forced to run at no better than 28MHz.

Xilinx part number	CLB matrix size	CLBs used	Flip-flops	Pipeline stages	Speed (MHz)	Final Addition Delay (ns)
XC4036xlhq208-2	36×36	1296	1699	6	37	35.7

Table 1 Synthesis results for pipelined integer MAC with delayed addition.

3 Using Delayed Addition in a Floating-Point Accumulator

A floating-point MAC unit uses too much area to fit on a single chip, so 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

In a traditional floating-point addition, the 1-bit sign, 8-bit exponent and 23-bit fraction (the portion of the mantissa to the right of its implied binary point) of each incoming number are first extracted from the IEEE 754 single precision format. In the first stage, the exponents are compared and the mantissas are shifted so that the two numbers are aligned. Sign-adjustments also occur at this point. Next, the two mantissas are added, and the sum is re-normalized [2].

The above algorithm is designed for a single addition rather than a series of accumulations. 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, re-normalization also impedes a delayed addition approach.

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

a. Minimize the interaction between the incoming number and the accumulated result. "Self-align" the incoming number on each cycle, rather than aligning it to the Pseudo-sum. Section 3.2 will describe self-alignment in more detail.

b. Use the delayed addition for accumulation and 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 will repeatedly be executing pseudo-sum = pseudo-sum + incoming operand. Each incoming operand is an IEEE single-precision floating-point number, with 8-bit exponent (EXP[7-0]) and 23-bit fraction. During the computation the pseudo-sum is stored as a 64-bit quantity, but it is normalized and truncated to a single-precision number during the final addition post-processing stage after the pipelined computation has completed.

The most notable characteristic of our approach is its ability to minimize the interactions between the incoming operand and the accumulated pseudo-sum. Most importantly, we cannot directly align the incoming operand and the current accumulated result, because we are delaying the final addition that calculates the accumulated result. Instead, we keep minimal information about the high-order exponent bits of the accumulated result, and perform the rest of the calculations using a fixed alignment. We refer to this technique as "self-alignment".

3.2.1 Self-Aligning Incoming Operands

A basic aspect of our technique notes that in single-precision IEEE floating-point, the mantissa is only 23 bits wide. Thus, if we try to add two originally-normalized numbers that differ by more than 2^{23}, alignment will cause the smaller of the two numbers to be "right-shifted" out of the expressible range for this format. (For example, $2^{56} + 2^{2} = 2^{56}$ in single-precision calculations.) Our algorithm efficiently identifies these cases and handles them appropriately.

First, note that if the current pseudo-sum and the incoming operand are **identical** in EXP[5], then the bits EXP[7-6] will determine which action to take. If the incoming operand's EXP[7-6] is larger than EXP[7-6] of the current accumulated result, then the incoming operand is much larger than the current accumulated result. Thus it will completely replace the current accumulated result, because of the right-shifting phenomenon discussed in the previous paragraph. On the other hand, if the incoming EXP[7-6] is smaller than that of the current accumulated value, we can discard the incoming operand because it is too small to make a difference in the accumulated pseudo-sum. When the incoming operand's EXP[7-6] is equal to that of the accumulated result (and we've already said that their EXP[5] bits are equal) then the two numbers differ by a small-enough amount that we actually need to do some calculations! In this case, we compress the incoming operand value with the current pseudo-sum using a Wallace tree approach. This is discussed in Section 3.2.2.

Next, note that if the current sum 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. Since we are using Wallace tree compression and are delaying true accumulations, we do not know the full exponent bits for the accumulated sum, and thus we want to avoid this scenario entirely.

To avoid the case of unequal EXP[5] bits, we actually keep two running pseudo-sums. One compressor, referred to as compressor-0, takes care of incoming operands whose EXP[5] bit is 0 and the other compressor (compressor-1) handles those where the EXP[5] bit is 1. In this way, we can always operate using the actions based on EXP[7-6] as described above. The two pseudo-sums from compressor-0 and compressor-1 are both added together during the final add stage as a post-processing step following the pipelined computation. Section 3.2.2 below discusses the implementation details of these compressors. This section also discusses the stall and swap operations that must happen when one of these running pseudo-sums gets large enough that its EXP[5] bit changes values. Figure 5 shows the overall structure of our design.

So far, we have not discussed EXP[4-0]. Since our scheme handles EXP[7-5] using the special cases outlined above, only EXP[4-0] need to be handled using shifting. Since EXP[4-0] is a 5-bit quantity, the largest exponent that it can express is 31. Thus, we can fully “unroll” this floating-point number by left-shifting the mantissa the number of times denoted by EXP[4-0]. In single-precision IEEE floating point, the mantissa is 23 bits wide and (in normalized form) is preceded by an implicit 1 bit. That is, a mantissa “X” would represent the binary number 1.X. During calculations, we make the implicit leading 1 into an explicit bit, which gives us a 24 bit mantissa. Then, by left-shifting to account for EXP[4-0], we need 31 additional bits,

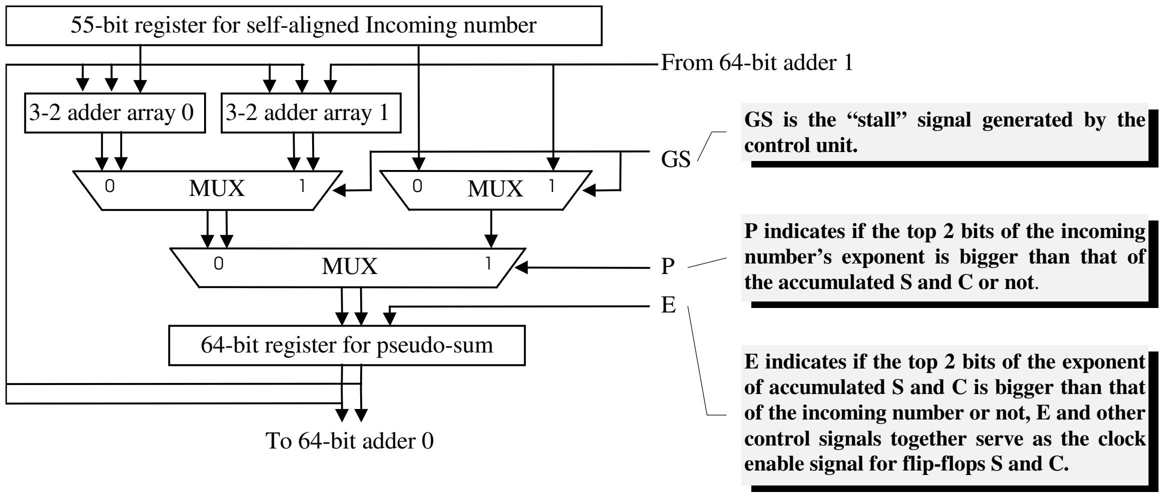

Fig. 4 Compressor design (compressor 0)

resulting in 55-bit wide calculations. Although this is quite wide, our design still fits into 910 CLBs on a Xilinx 4036, and this gives us the ability to garner truly high-performance single-precision floating-point from an FPGA-based design.

3.2.2 Compressor Implementation Details

Figure 4 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 Figure 5. The running pseudo-sum is stored as the Wallace tree's S and C partial results in the 64-bit registers shown.

Once self-aligned, the incoming number can be thought of as *m'*(the 55-bit mantissa) $\times 2^{64 \cdot EXP[7-6]}$. If EXP[7-6] of the incoming number is bigger than that of the pseudo-sum, the pseudo-sum will be shifted out of the expressible range as long as the compressor is no more than 64 bit wide. On the other hand, if EXP[7-6] of the incoming number is smaller than that of the pseudo-sum, the incoming number will be shifted out of the expressible range since *m'* is less than 64 bit wide. The compression thus will only take place when EXP[7-6] of the pseudo-sum is equal to that of the incoming number.

Were it not for the possibility of the pseudo-sum 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 pseudo-sum 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 pseudo-sum 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 pseudo-sum and the sign bit from the 55-bit incoming operand. We have used *espresso* to form a minimized truth table generating the ***GlobalStall*** signal (***GS*** in Fig. 4) as a Boolean function of these 7 bits. As shown in Fig. 5, the GlobalStall signal is used as the clock enable signal on the first three pipestages; 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. Our design 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.

When a stall occurs, our response is to sum the S and C portions of each compressors' pseudo-sum using the 64-bit adders shown in the Stall Response box in Figure 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-

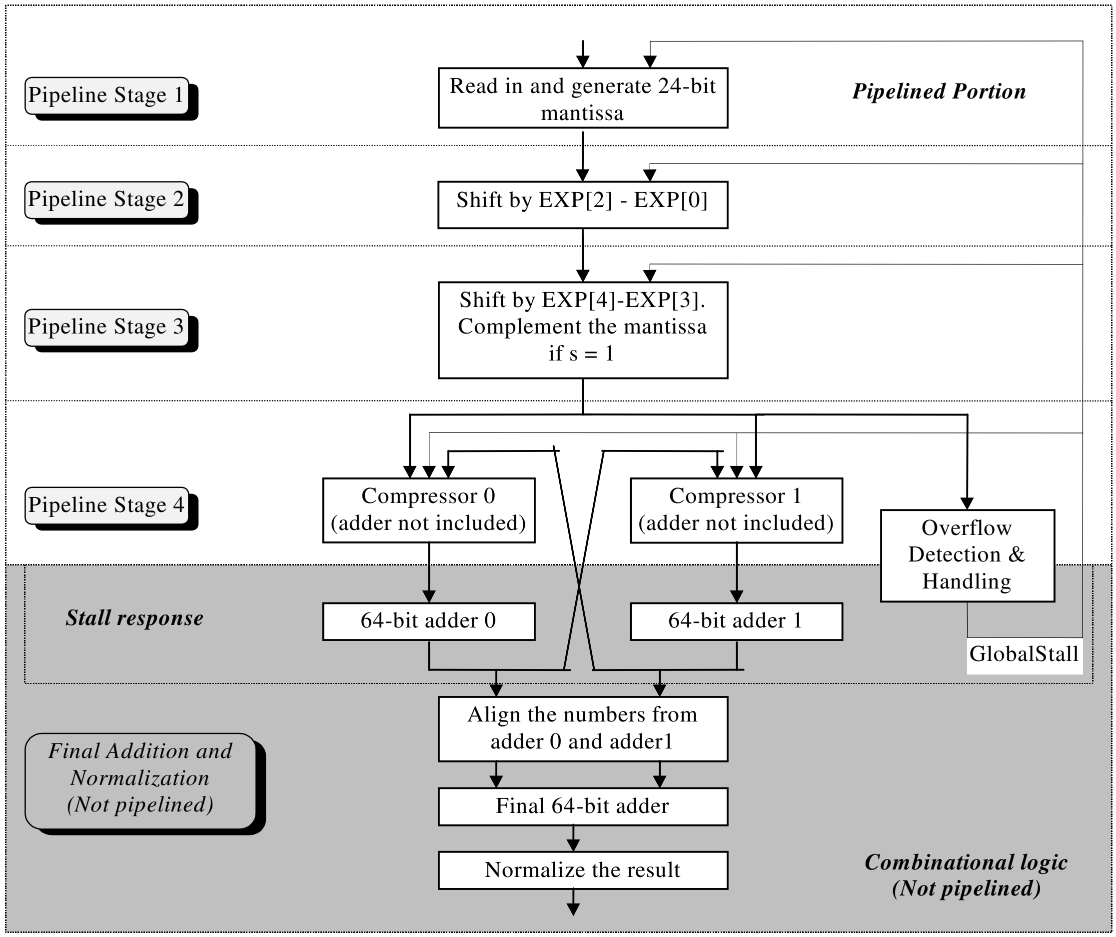

Fig. 5 Floating-point Delayed accumulator pipelining scheme

formed sum depends on its value, i.e., it depends on whether (i) an overflow truly occurred or (ii) we were overly conservative in our stall detection. In cases where an overflow does occur, the value of EXP[5] in the sum will change. Recall that compressor-0 is to handle the accumulation of incoming operands whose EXP[5] bit is 0, with a pseudo-sum whose EXP[5] bit is also 0. If the pseudo-sum 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 Fig. 5 indicates 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 pseudo-sum in its current form.

3.3 Experimental Results

Synthesis results for this design are shown in Table 2. 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 4 pipestages, our design attains a clock rate of 33MHz. Because of extra bookkeeping required to renormalize the final result, the post-processing delay in this design is larger. At 72.4ns, this delay corresponds to roughly 2.4 of the pipelined 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.

Xilinx part number	CLB matrix size	CLB used	Flip-flops	Pipeline stages	Speed (MHz)	Final Add and Norm. Delay (ns.)
XC4036xlhq208-2	36×36	910	378	4	33	72.4

Table 2 Synthesis results for pipelined floating-point accumulation with delayed addition.

4 A More Aggressive Floating-Point Accumulator Design

The main reason why we have the overflow detection and handling logic in the previous design is because of the possible overflow of the Pseudo-sum over a number of accumulations. 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 would like to resort to the compiler to break a large accumulation into small pieces so that overflow will be guaranteed ***not*** to occur when each of these small pieces is executed.

Our design, like many others in FPGA computing, relies on taking advantage of large chunk of data to have an overall increase in performance. We will not have much gain here if we have to break an accumulation into very small pieces. How often will the overflow happen then? The largest incoming mantissa that can be fed into one compressor is 11…1100…00 (the first 24 bits are '1's and the rest 31 bits are '0's). It is less than 2^{55}. Thus if the pseudo-sum is stored in a n-bit ($n > 55$) register, we will have an overflow at least after 2^{n-55} accumulations. We can thus use this formula to choose suitable n for specific applications.

Based on the above analysis, a more aggressive design is proposed here. In this design, we choose n = 64 like in the conservative design. Thus a big accumulation will have to be divided into smaller accumulations of less than 512 numbers by compilers. However, since we no longer need the overflow checking and handling in this aggressive design, all the related components in our previous design have been removed and the two 64-bit adders right below the compressors are replaced by an array of 4-2 adders. This greatly simplified the control logic on the critical path, which enables us to further pipeline our

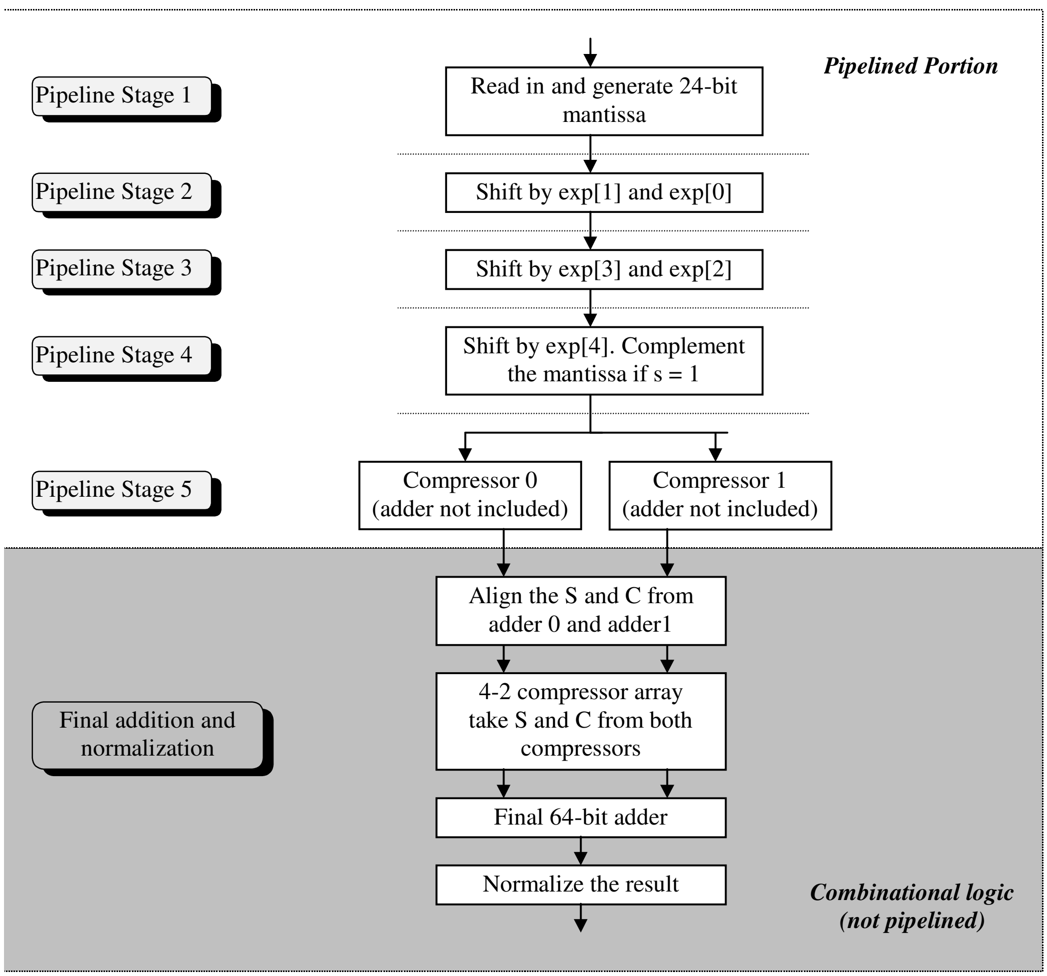

Fig. 6 Floating-point Delayed accumulator pipelining scheme(Aggressive Design)

design from 4 pipeline stages in the previous design to 5 pipeline stages (see fig. 6). Synthesis results show that we have doubled speed of the conservative design and have achieved a high clock rate of 66Mhz.

Xilinx part number	CLB matrix size	CLB used	Flip-flops	Pipeline stages	Speed (MHz)	Final Addition Delay (ns)
XC4036xlhq208-2	36×36	839	417	5	66	73.23

Table 3 Synthesis result of Floating-point accumulation with delayed addition (aggressive design).

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 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, 4, and 5, abide by the representation portion of the IEEE standard. Our approaches implement single-precision IEEE floating-point including denormalized numbers. Our operations do reorder computations, however, and make the basic assumption that the additions performed are commutative. Space limitations prevent a full discussion of rounding techniques and error theory, but we note here that 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. When users are concerned, error theory in computations can be used to determine to what extent such reordering is safe [4,5].

6 Related Work

This paper touches on areas related to both computer arithmetic and configurable computing. Hennessy and Patterson provide an overview of computer arithmetic in Appendix A of [6]. In addition, innumerable books and papers go into more detail on adder and multiplier designs. For example, Weste and Eshraghian [7] provides a detailed discussion on various multiplier implementations and Wallace trees. Work by Wallace [10] and Heikes et al. [11] discuss Wallace tree implementations for high-performance multiplier designs in full-custom integrated circuits. Work by Ohkubo et al. [8] and Makino et al. [9] are also examples of recent multiplier designs in full-custom ICs. These designs, as with most, use Booth encoders and Wallace trees.

More recent work has examined implementing floating-point units in FPGAs [2,3,13]. Louca et al. [2] present approaches for implementing IEEE-standard floating-point addition and multiplication in FPGAs. They use bit-serial adders for multiplication. 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. Canik et al. [3] talks about how to map a bit-array multiplier to FPGA. Although it was published in 1994, their results were still excellent even by today's point of view.

Finally, several authors have discussed rounding and error theory [4,5,12]. Wilkinson [5] and Heikes et al. [12], in particular, touch on rounding error in MAC and inner-product units.

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 between 33 and 66MHz. These techniques should help to broaden the space of integer and floating-point computations that can be customized for high-performance execution on current FPGAs.

References

[1] IEEE Standards Board. "IEEE Standard for Binary Floating-Point Arithmetic". Technical Report ANSI/IEEE Std. 754-1985, the Institute of Electrical and Electronics Engineers, New York, 1985.

[2] Loucas Louca, Todd A. Cook, William H. Johnson, Implementation of IEEE Single Precision Floating Point Addition and Multiplication on FGPAs. IEEE Symposium on FPGAs for Custom Computing Machines, Napa Valley, California, April, 1996.

[3] Robert W. Canik, Earl E. Swartlander, Implementing Array Multipliers in XILINX FPGAs, Proceedings of 1994 28th Asilomar Conference on signals, systems and computers, Pacific Grove, CA 1994.

[4] John R. Taylor, "An Introduction to Error Analysis", University Science Books, 1982.

[5] J. H. Wilkinson, "Rounding Errors in Algebraic Processes", Prentice-hall Inc. 1963.

[6] David A. Patterson, John L. Hennessy, David Goldberg, "Computer Architecture, A quantitative approach" Appendix A, 2nd edition, Morgan Kaufmann publisher, 1996.

[7] Neil H. E. Weste, Kamran Eshraghian, "Principles of CMOS VLSI design", 2nd edition, Addison-Wesley Publishing Company, 1993.

[8] Norio Ohkubo, Makoto Suzuki, etc. "A 4.4 ns CMOS 54×54 bit multiplier using pass-transistor multiplexer", IEEE Journal of Solid State Circuits, Vol 30, No. 3, p.251-256, March 1995.

[9] Hiroshi Makino, Yasunobu Nakase, etc. "An 8.8 ns 54×54 bit multiplier with high speed redundant binary architecture", IEEE Journal of Solid State Circuits, Vol 31, No. 6, p.773-783, March 1995.

[10] C.S. Wallace, "Suggestions for a fast multiplier", IEEE Trans. Electron, computers, Vol EC-13, p.114-117, February, 1964.

[11] Y. Kanie, Y. Kubota, etc. "4-2 compressor with complementary pass-transistor logic", IEICE Trans. Electron, vol. E77-c, no. 4, p.789-796, April, 1994.

[12] Craig Heikes, Glenn Colon-bonet, "A dual floating point coprocessor with an FMAC architecture", ISSCC Digest of technical papers, p.354-355, 1996.

[13] Nabeel Shirazi, Al Walters, and Peter Athanas. "Quantitative Analysis of Floating Point Arithmetic on FPGA Based Custom Computing Machines", IEEE Symposium on FPGAs for Custom Computing Machines, p.155-162, April 1995.