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

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

Abstract-- The speed of arithmetic calculations in configurable hardware is limited by carry propagation even with the dedicated carry propagation 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 66MHz 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. We also present a 32-bit floating-point accumulator design with compiler-managed overflow avoidance that achieves a 66MHz clock rate. Finally, we present an application of delayed addition techniques to solve a system of linear equations using conjugate gradient method. These designs and applications demonstrate the utility of delayed addition for accelerating FPGA calculations in both the integer and floating-point domains.

Index Terms-- FPGA, delayed addition, Wallace tree, multiply-accumulate (MAC)

I. 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 -2 speed grade XC4000xl 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 inner 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 most recent microprocessors [15, 16]), Wallace trees are used to "accumulate" the partial products. 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 inner products or other repeated calculations. The technique is *effective* for configurable hardware because it removes addition's carry

The authors are with the Electrical Engineering Department of Princeton University, Princeton, NJ 08544, USA.

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 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 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. We demonstrate this promise on an example application: solving linear equations using conjugate gradient method.

The remainder of this paper is structured as follows. Section II introduces the basic idea of Delayed Addition calculation, and presents a design for a pipelined integer multiply-accumulate unit based on this approach. Section III 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 IV then presents the 66MHz floating-point accumulator with compiler-managed overflow avoidance, which is used to form the MAC unit in our application to solve linear equations using conjugate gradient method in section V. Section VI discusses issues of rounding and error theory related to these designs, Section VII presents related work, and Section VIII provides our conclusions.

II. Delayed Addition in a Pipelined Integer Multiply-Accumulator

A. 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 Xilinx 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,

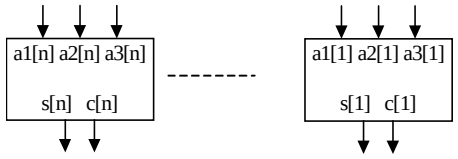

Fig 1. (a) An array of n 3-2 adders.

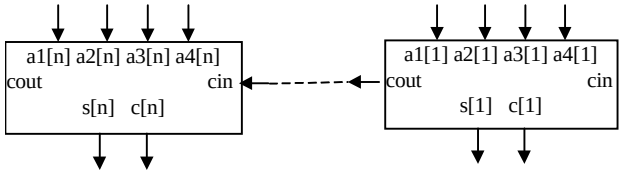

Fig 1. (b) An array of n 4-2 adders.

it is easy to map on a 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, they 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 for accumulation. Thus we would need a 64 × 64 CLB matrix to implement it [3]. However, CLB matrix of this size can barely fit into the largest Xilinx part available (XC40125XV) now (as of Sept. 1998) [20], which would involve a huge cost.

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

B. 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 adders*** (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, $S+2C = A1 + A1 + A3$

S and C are partial results that we refer to in this paper as the ***pseudo-sum***. They can be combined during a final

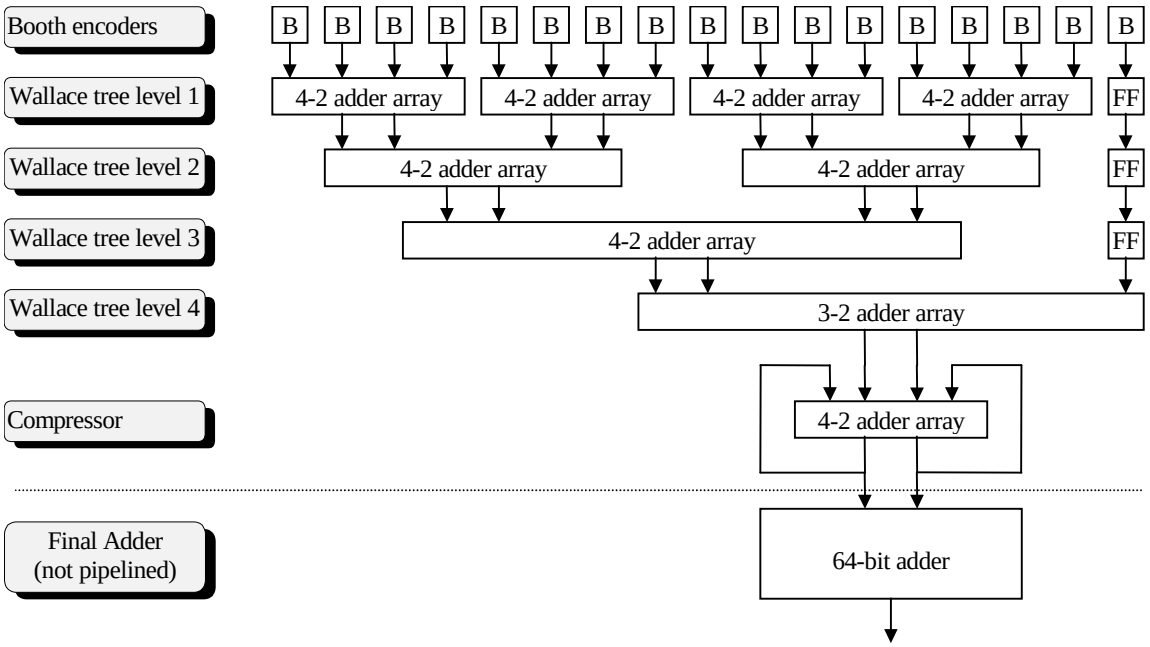

Fig. 2 Integer MAC with delayed addition.

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 1), W2 (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 Figure 2. The final addition is performed only once per summation and does not impact the pipelined clock rate.

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

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

Fig. 1 (b) shows the layout of an array example using 4-2 adders. 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.

C. 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. To achieve accumulation, we repeatedly execute:

Pseudo-sum = Pseudo-sum + (the final two partial products for 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 a block diagram of our implementation.

Designs	Xilinx part number	CLB matrix size	CLBs used	Flip-flops used	Pipeline stages	Speed (MHz)	Final Addition Delay (ns)
IMAC (delayed addition)	XC4036xlhq208-2	36×36	1287	1866	7	66.7	38.27
Traditional Integer multiplier	XC4036xlhq208-2	36×36	1243	1836	8	54.5	N/A

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

Each level of a Wallace tree has a similar delay, and this delay is also similar to that of a Booth encoder. Thus, as shown 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 pseudo-sum 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 pseudo-sum 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 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 only once per entire summation rather than once per element, will be negligible even compared to the faster vector MAC calculations of this design.

D. 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. 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. After several initial attempts, we targeted our design at the speed of 66.7 MHz and specified this information in the timing constraint file (.pcf file), where we listed this 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. Most notably, our design fits in a Xilinx 4036 part and achieves the targetted clock rate of 66.7 MHz. The final addition delay, done once per vector as a post-processing step, takes roughly 40ns or nearly 3 cycles.

To demonstrate the advantage of delayed addition, we also tried to implement an integer MAC composed of a traditional integer multiplier and an adder. However, the design was too big to fit on one XC4036 chip, so we built an integer multiplier on the chip instead. To trade off between pipeline speed and area cost, we used the carry-select adder for final addition and divided it into two pipeline stages. In the first stage, three 32-bit additions are carried out in parallel, one for the lower 32 bits and two for the upper 32 bits. In the second stage, we select the upper 32 bits between the two results by the carryout from the lower 32-bit addition. 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 faster pipeline speed than the traditional multiplier and accumulator design. According to the data above, an IMAC with the delayed addition would require

$$7 + (N-1) + 3 = N + 9$$

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 $15\text{ns} \times (N + 9) = (15N + 135)\text{ns}$. Since we could only implement the multiplier on one XC4036 chip in the traditional design, we have to add another two pipeline stages for the accumulator in our calculation. Thus the overall latency for an inner product of size N using traditional IMAC would be

$$10 + (N-1) = N + 9$$

cycles as well. Since the cycle time in the delayed addition design is 20% shorter than the traditional design, the delayed addition design has a performance speedup of 120%.

III. 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, 5 loops (loop 3, 4, 6, 9, 21) are basically long vector inner-product-like computation [17]. In certain applications, such as the conjugate gradient example in Section V, multiply and accumulation dominates the whole computation process. Thus it would be ideal if 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-

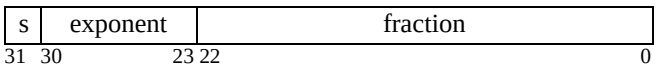

Fig. 4 IEEE single precision format.
S is the sign; exponent is biased by 127.
If exponent is not 0 (normalized), mantissa = 1.fraction
If exponent is 0 (denormalized), mantissa = 0.fraction

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.

A. 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 re-normalizes 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, re-normalization 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 Pseudo-sum. Section 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].

B. 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 pseudo-sum = pseudo-sum + 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. We consider the exponent bits as three subfields: high-order exponent, a decision bit and low-order exponent for simplicity of discussion. 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, we choose not to align the incoming operand and the current pseudo-sum directly because that way the incoming operand interacts with the accumulated pseudo-sum throughout the alignment process, which makes the further pipelining impossible. Thus the alignment process could easily become the bottleneck of the whole pipeline if we still adopt the traditional alignment method. 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.

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 use 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 pseudo-sum.

We could have fully "unrolled" the incoming operand and the accumulated pseudo-sum 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 910 CLBs on a Xilinx 4036 as we will see later, and this gives us the ability to garner truly high-

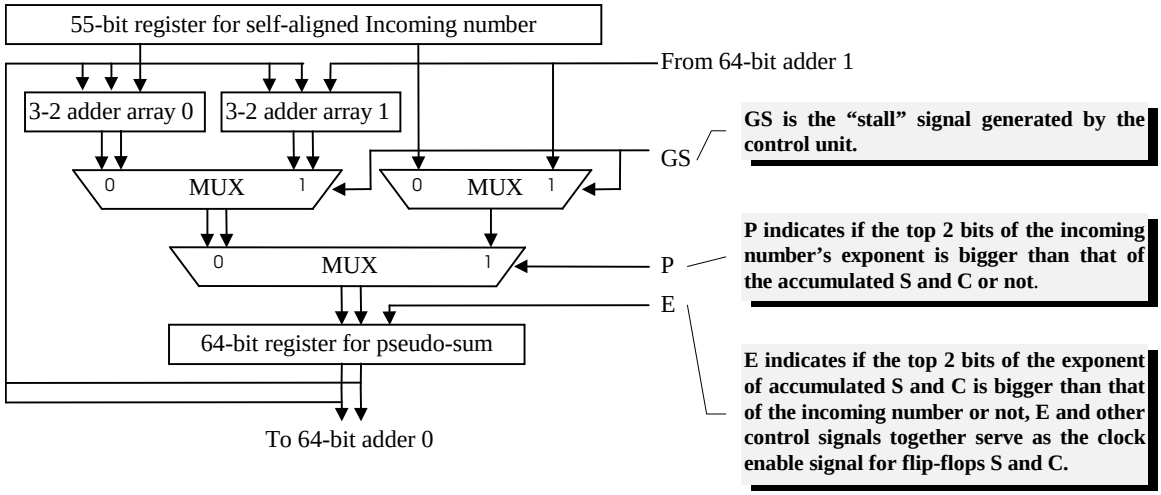

Fig. 5 Compressor design (compressor

performance single-precision floating-point from an FPGA-based design.

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 pseudo-sum. 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'*(the 55-bit mantissa) $\times 2^{64 \cdot EXP[7-5]}$. Meanwhile, our pseudo-sum is stored as *mp'*(the 64-bit mantissa) $\times 2^{64 \cdot EXP[7-5]}$. If the current pseudo-sum 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 pseudo-sum, the mantissa of the pseudo-sum will be shifted out of the expressible range as long as it is no more than 64 bit wide. In this case, we simply replace the current pseudo-sum by the incoming operand. On the other hand, if the high-order exponent of the incoming number is **smaller than** that of the pseudo-sum, the incoming number will be shifted out of the expressible range since *mi'* is less than 64 bit wide. Thus we simply ignore the incoming operand. The compression will only take place when high-order exponent of the pseudo-sum is **equal to** that of the incoming number.

Note that if the current pseudo-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. This would pose a bottleneck in the pipeline; thus we hope to avoid this scenario entirely. This leads to our design described in Section 2) below.

2) Compressor Implementation Details

In order to avoid the undesirable scenario of unequal decision bits, we actually keep two running pseudo-sums. 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 Figure 6. In this way, we can always take operations corresponding to the high-order exponent 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.

Figure 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 Figure 6. The running pseudo-sum 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 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

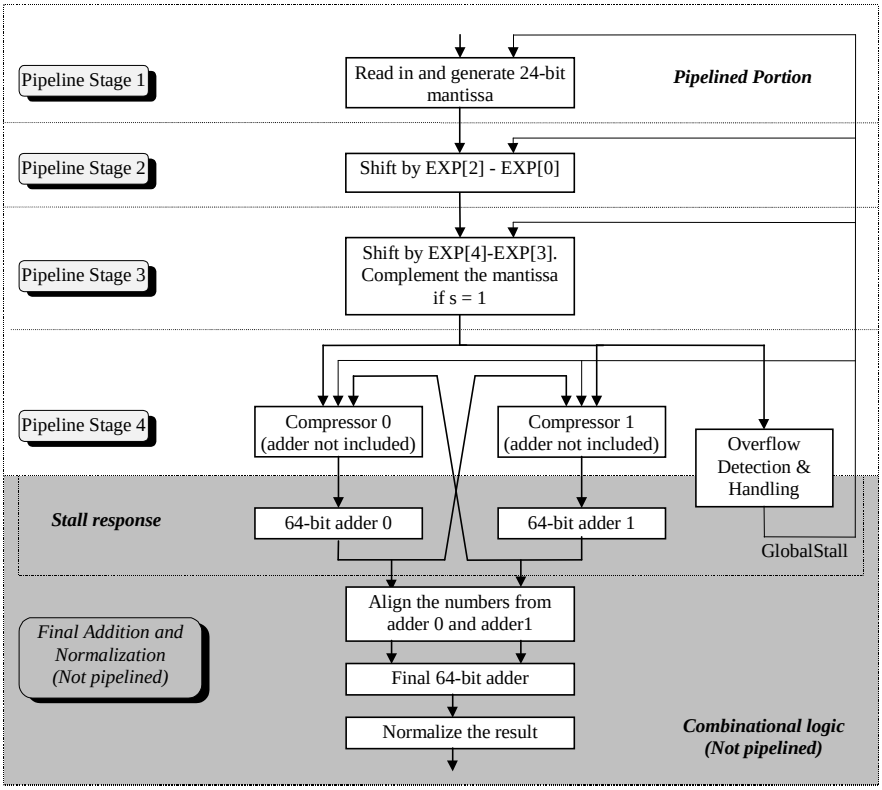

Fig. 6 Floating-point accumulator pipelining scheme

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 floating-point accumulation with delayed addition

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 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 pseudo-sums in parallel during the stall.

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 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 pseudo-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 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 pseudo-sum in its current form.

C. Experimental Results

Figure 6 shows the block diagram of this design and Table 2 summarizes the synthesis results. Because this is a

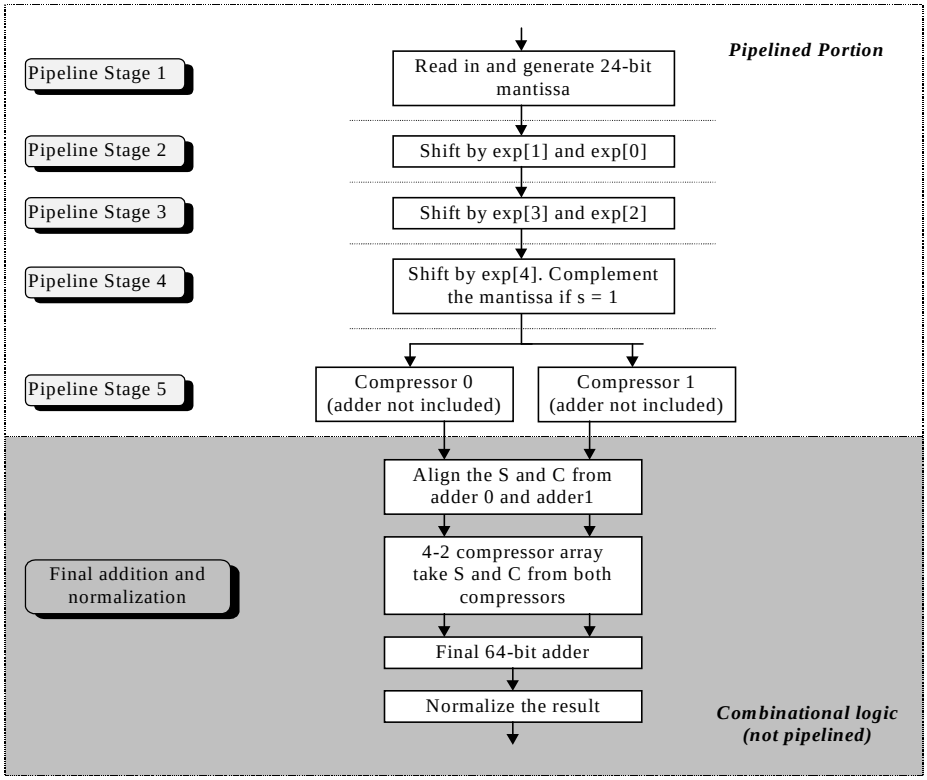

Fig. 7 Floating-point accumulator with compiler-managed overflow avoidance pipelining scheme

floating-point accumulator rather than a MAC unit, it is actually smaller than the integer MAC unit discussed in the previous section. Using 4 pipeline stages, 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. According to the data above, using the same analysis as in section III and assuming there is no stall during the computation, we will have to wait

$$4 + (N-1) + 3 = N + 6$$

cycles for an accumulation of length N to complete.

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 IV, we can prove that for vectors shorter than 512 elements, there is no chance of stalling at all.

IV. 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 because of the possible overflow of the pseudo-sum 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…1100…00 (the first 24 bits are '1's and

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 results of floating-point accumulation with delayed addition and compiler-managed overflow avoidance.

the rest 31 bits are '0's). This is less than 2^{55}. Thus if the pseudo-sum is stored in an n-bit (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 pseudo-sum width to be 64 as in the design from Section III. 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. However, since 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 Figure 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. Figure 7 shows a resulting 5-pipeline-stage design.

Synthesis results summarized in table 3 show that we have doubled the speed of the conservative design and have achieved a high clock rate of 66MHz. For an accumulation of size N, we will need 5 + (N-1) + 5 = N + 9 cycles to complete the whole computation.

V. Applications of Delayed Addition Techniques

In previous sections, we have examined MACs and accumulations in isolation. We now evaluate the utility of our approach within a larger application. Our design is especially useful for those applications in which inner-product computations dominate the overall execution time. One such example is to solve a system of linear equations using conjugate gradient method. The system of linear equations takes the form:

$$A\,x = b, \qquad (5.1)$$

where A is a N×N matrix and x, b are 1×N vectors. Such computations arise frequently in scientific matrix-oriented code, such as in Space-Time Adaptive Processing (STAP) [18, 21]. STAP is a problem for which configurable accelerators are often explored, so a high-performance, FPGA-based inner product unit would form the core of such an accelerator.

A. Conjugate Gradient Method: Background

The conjugate gradient method is based on the idea of minimizing the function

$$f(x) = \tfrac{1}{2} x^T A x - bx \qquad (5.2)$$

The function is minimized when its gradient $\nabla f = Ax - b$ is zero, which is exactly the solution of (5.1).

	N-way ip	a+b×c	Div	Sub
Eq. 5.4: α_i	N + 2	-	1	-
Eq. 5.5: x_{i+1}	-	N	-	-
Eq. 5.6: g_{i+1}	N	-	-	N
Eq. 5.7: d_{i+1}	N + 1	N	1	-

Table 4: Number of different operations in Equation (5.4 – 5.7) ip stands for inner product, Div for division, Sub for subtraction.

We can use regression to find the minimum x in (5.2). Set the initial conditions to be

$$x_0 = 0 \quad \textit{(initial solution)}$$
$$d_0 = b \quad \textit{(initial direction)}$$
$$g_0 = -d_0 \quad \textit{(initial gradient)} \qquad (5.3)$$

and iteratively calculate the following steps:

$$\alpha_i = \frac{g_i^T d_i}{d_i^T A\, d_i} \qquad (5.4)$$

$$x_{i+1} = x_i + \alpha_i d_i \qquad (5.5)$$

$$g_{i+1} = A\, x_{i+1} - b \qquad (5.6)$$

$$d_{i+1} = -g_{i+1} + \frac{g_{i+1}^T A\, d_i}{d_i^T A\, d_i} d_i \qquad (5.7)$$

until

$$| x_{i+1} - x_i | < err \qquad (5.8)$$

where *err* is a pre-specified error limit.

As we can see from (5.4) to (5.7), each iteration is divided into four steps and these four steps must be serialized since the operation of each step relies on the full knowledge of previous step's result. Each step has significant parallelism, however. The number of different kinds of calculations required for each step is summarized in table 4. Wherever N appears in a table entry, the corresponding operation has N-fold parallelism.

The above analysis shows that using conjugate gradient method to solve a system of linear equations is potentially a good candidate for highlighting the utility of FPGA computation with our specially designed accumulation units. Next, we present a configurable computing system that implements this application.

B. Proposed Architecture

The current configurable computing systems can be roughly categorized into closely-coupled architectures and loosely-coupled architectures. For closely-coupled architectures, the configurable hardware is part of a microprocessor and is generally treated as a special functional unit (SFU). The SFU behaves similarly to other functional units on chip; it also takes two operands and produces a result for each

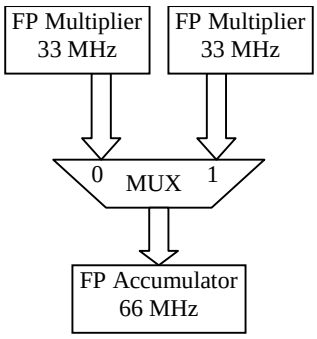

Fig. 8 A Floating-Point MAC: The accumulator takes in operands alternatively from the two multipliers. The whole MAC runs at 66MHz.

operation. This architecture, however, is not suitable for our application for several reasons.

First of all, all state-of-the-art microprocessors have floating-point multipliers and adders. Generally these functional units are pipelined and can run at a very high speed. Some microprocessors can even use them to form two MAC units, like HP PA-RISC. Our SFU, on the other hand, has a much slower speed. What's more, as we can see from previous discussion, it takes a lot of hardware resource to implement a MAC unit in FPGA, which is beyond the capability of most current closely-coupled configurable computing system. For those that have this capability, the number of SFUs that can be implemented is very limited, too. Thus the parallelism in the algorithms can hardly be exploited.

Loosely-coupled configurable computing system, on the other hand, can provide us with plenty of hardware resource for taking advantage of the mass parallelism in applications. In this architecture, we normally have one host machine and multiple FPGA boards connected to the I/O bus, with 10-60 FPGAs and associated memory on each board. The host machine configures and initializes each FPGA board and is responsible for communication with other machines while the FPGA boards takes care of major calculation. The host machine starts the operation on FPGA boards after initialization and FPGA board talks to the host machine when the job is done by interrupt.

For our design, each FPGA board will be used to implement several MAC units, and will also contain some local memory. We use floating-point accumulators and multipliers to form MAC units. Since the size of the matrix A is generally known before computation, we choose a floating-point accumulator with compiler-managed overflow avoidance as described in Section IV. On a separate FPGA, we will have an 8-stage pipelined floating-point multiplier; our design can run at 33MHz on –2 Xilinx parts. Thus we can form a MAC unit with two floating-point multipliers and a floating-point accumulator, with the floating-point accumulator taking in the products alternatively from each one of the two floating-point multipliers each clock cycle (Fig 8). In this way, the MAC unit can run at the full speed of the accumulator, 66MHz.

Local memory on the FPGA board will help manage the I/O data flow from the host machine to the FPGA board. It holds the data of matrix A and the results for each iteration including α_i, x_{i+1}, g_{i+1}, d_{i+1}, and also provides space for intermediate results. When solving more than one system of linear equations simultaneously, it would be more efficient to have two memory chips. This would allow us to have one chip doing computation while the other one is downloading a new system of linear equations from the host machine. The memories would alternate their functions once the computation is done. The communication latency between host machine and each FPGA board will thus be totally hidden except for the first time.

C. Performance of FPGA-based Conjugate Gradient Accelerator

Suppose we have L (L>2) MAC units on a FPGA board. This FPGA board is thus capable of processing all the multiplication and accumulation in (5.4) – (5.8) except for the division, which is handled by CPU via interrupt. To simplify the calculation, we assume the compiler guarantees N is less than 512. Thus we don't have to worry about breaking long inner products into small pieces. We also assume each MAC unit is fed from local memory at its optimum bandwidth of two 32-bit operands at 66MHz. Suppose we have ***L*** MAC units on the FPGA board, the required local memory bandwidth would thus be 528×L Meg bytes per second.

Before going down to calculation details, we first define ***K-ip time*** as the number of clock cycles needed for our MAC unit to compute the inner product of two vectors of length K. In our case, we need

$$8 + 5 + (k-1) + 5 = k + 17$$

cycles to complete such a computation, where 8 comes from the number of pipeline stages for multiplier, 5 comes from the number of pipeline stages for the accumulator, k-1 is the vector length minus 1, and 5 counts for the combinational delay for the final addition.

We can then calculate the latency of one iteration of computation as follows:

1) Latency for step 1: Computing $\boldsymbol{\alpha_i = \dfrac{\mathbf{g_i^T d_i}}{\mathbf{d_i^T A d_i}}}$

In this step (5.4), we first calculate $d_i^T A$. The calculation consists of N inner products, which requires ⌈N/L⌉ N-ip time. Since $d_i^T A$ is an N×1 vector, $(d_i^T A)\, d_i$ is simply one inner product of length N, and so is $g_i^T d_i$. Thus we can compute $(d_i^T A)\, d_i$ and $g_i^T d_i$ simultaneously within one N-ip time. In all, we need ⌈N/L⌉ +1 N-ip time for inner

products as well as one interrupt for division. Since $d_i^T A\, d_i$ will be used later as a divisor in step 4, we can save one interrupt by sending back both α_i and $1/\, d_i^T A\, d_i$.

2) Latency for step 2: Computing $\boldsymbol{x_{i+1} = x_i + \alpha_i d_i}$

In this step (5.5), we have to compute each element of x_{i+1} and we have N such computations. For element j of x_{i+1}, we have $x_{i+1,j} = x_{i,j} + \alpha_i\, d_{i,j}$. This can be computed as an inner product of vector length two. Although inner product of size two is not economical at all using our MAC unit, it is far better than to send these numbers back to host machine for computation and send the results back, which would incur a high penalty in communication time. Thus we need $\lceil N/L \rceil$ 2-ip time for this step.

3) *Latency for step 3: Computing* $\boldsymbol{g_{i+1} = A\, x_{i+1} - b}$

In this step (5.6), we also have to compute each element of g_{i+1} and we have N such computations. Since b is fixed through iterations, we can store the $-b$ in the memory beforehand. Thus for element j of g_{i+1}, we have $g_{i+1,j} = (-\, b_j) + \Sigma\ \ A_{j,k}\, x_{i,k}$. This can be computed as an inner product of vector length N+1, so we need $\lceil N/L \rceil$ (N+1)-ip time for this step.

4) Latency for step 4:

$$\textit{Computing } \mathbf{d_{i+1} = -g_{i+1} + \frac{g_{i+1}^T A\, d_i}{d_i^T A\, d_i}\, d_i}$$

In this step (5.7), we need $\lceil N/L \rceil$ +1 N-ip time to calculate $g_{i+1}^T A\, d_i$ for the same reason stated in (5.3.1). Since $1/\, d_i^T A\, d_i$ is available from step 1, we need one multiplication time for $g_{i+1}^T A\, d_i \times (1/\, d_i^T A\, d_i)$. And we need another $\lceil N/L \rceil$ 2-ip time to get the result for the same reason stated in 5.3.2. In all, we need $\lceil N/L \rceil$ +1 N-ip time + $\lceil N/L \rceil$ 2-ip time + 1 multiplication time for this step.

5) Overall Performance

By summing up the time needed in each step, we get the time for one iteration of the computation. In all, it takes 2 ($\lceil N/L \rceil$ + 1) N-ip time + 1 $\lceil N/L \rceil$ (N+1)-ip time + 2$\lceil N/L \rceil$ 2-ip time +1 multiplication time + 1 interrupt time for each iteration. If we plug in the N-ip time and 2-ip time, and we take 8 cycles to be one multiplication time, and we assume that 1 interrupt takes 500 cycles, then we need

$$(3N + 90)\lceil N/L \rceil + 2N + 542$$

cycles for each iteration.

Fig. 9 shows the cycle count chart for different problem sizes and number of MAC units. From this table we can see, with the increase of problem size, the number of cycles needed becomes inversely proportional to the number of MAC units L. This is because the cost of interrupt has become trivial in overall latency when N gets large.

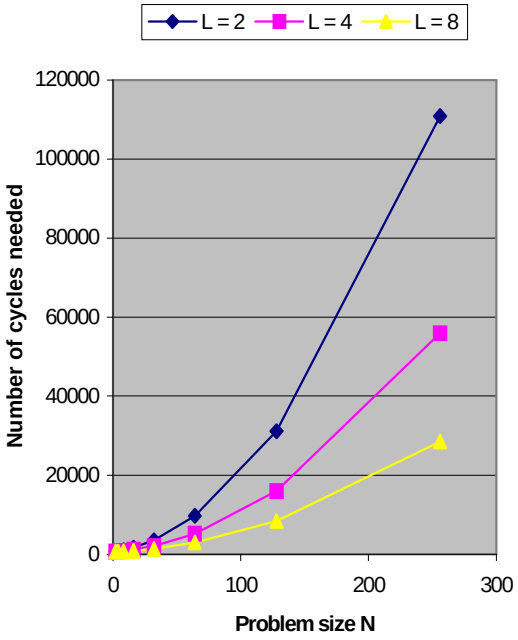

Fig. 9 Cycle count for one iteration with regard to problem size N and number of MAC units L.

It is worth mentioning here that the major computation in checking for convergence (5.8), $|\ x_{i+1} - x_i\ |$, is also an inner product operation. From (5.5) we know that $x_{i+1} - x_i = \alpha_i d_i$, each element of which is calculated by the multiplication process in step 2. Thus we just have to save all these elements of $\alpha_i\, d_i$ in the memory and calculate its norm later. A scheduling that definitely won't put extra latency on the system would be to calculate it during the last N-ip time in step 1 and check if $|\ x_{i+1} - x_i\ | < err$ during the interrupt.

VI. 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 III, IV, and V, 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. 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].

VII. 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]. 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] provides 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. C. S. 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 et al. [11].

Early in 1994, Canik et al. [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. Ligon 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.

At the application level, there are many papers about Space-Time Adaptive Processing (STAP). Some background knowledge can be found in Gupta [18]. In addition, Smith et al. [19] talks about using conjugate gradient method for STAP and Gupta [18] discussed about the possibility of using configurable hardware for STAP.

Finally, several authors have discussed rounding and error theory [4,5,12]. We can see how people deal with the error in physics from Taylor [4]. Wilkinson [5] and Heikes et al. [12] touch on rounding error in MAC and inner-product units.

VIII. 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. We have also used our designs in a real application: solving system of linear equations using conjugate gradient method. 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.

References

[1] IEEE Standards Board. “IEEE Standard for Binary Floating-Point Arithmetic”. Technical Report ANSI/IEEE Std. 754-1985, 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.

[14] Walter B. Ligion III, Scott McMillan, Greg Monn, Fred Stivers and Keith D. Underwood. "A Re-evaluation of the Practicality of Floating-Point Operations on FPGAs", IEEE Symposium on FPGAs for Custom Computing Machines, Napa Valley, CA, April 1998.

[15] Dileep P. Bhandarkar, "Alpha Implementations and Architecture, complete reference and guide", Digital Press, 1996

[16] Yu, R.K. et al. "167 MHz radix-4 floating point multiplier", Proceedings of the 12th Symposium on Computer Arithmetic, p.149-54, Bath, UK, July 1995.

[17] McMahon, F. M. "The Livermore FORTRAN kernels: A computer test of numerical performance range," Tech. Rep. UCRL-55745, Lawrence Livermore National Laboratory, Univ. of California, Livermore, December 1986.

[18] Nikhil D. Gupta, "Reconfigurable Computing for Space-Time Adaptive Processing", master thesis proposal, Department of Computer Science, Texas Tech University, Fall 1997

[19] Xilinx, "XC4000E and XC4000X Series Field Programmable Gate Arrays, product specification", V1.4, Nov. 1997

[20] Xilinx, "XC4000XV Family Field Programmable Gate Arrays, Advance Product Specification", V1.1, May 1998

[21] Smith, S.T. et al. "Linear and Nonlinear Conjugate Gradient Methods for Adaptive Processing", 1996 IEEE International Conference on Acoustics, Speech and Signal Processing, Atlanta, GA, May 1996

[22] Microelectronics group of 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