Modular Design of Fully Pipelined Accumulators

Miaoqing Huang, David Andrews
Department of Computer Science and Computer Engineering, University of Arkansas
Fayetteville, AR 72701, USA
{mqhuang,dandrews}@uark.edu

***Abstract*—Fast and efficient accumulation arithmetic circuits are critical for a broad range of scientific and embedded system applications. High throughput accumulation circuits are typically hand designed for specific vector lengths requiring the circuit to be modified when the lengths are changed. In this work we present a new design approach that can achieve low latency and near optimal throughput for input data vectors of arbitrary length. The flexibility of the design allows it to be used for both integer and floating-point operations. By providing a simple and efficient interface to the user and a modular architecture for the designer, the proposed technique has broad impact across a wide range of custom hardware designs.**

I. Introduction

Field-Programmable Gate Array (FPGA) technology has been an enabling technology for a wide range of application domains. Platform FPGAs have enjoyed particular success within the embedded systems domain, with their ability to serve as programmable multiprocessor systems on chip and application specific custom accelerator solutions. Researchers and manufacturers have also focused on bringing the benefits of FPGA technology into the high performance scientific computing community. These efforts have met with mixed success due to strong competition from cheap and economical cluster computers, as well as challenging design issues associated with the computational requirements of scientific codes.

Floating-point operations are critical for a large section of scientific applications. Researchers and designers have continually investigated how to migrate floating-point operations within the FPGA fabric [1]–[7]. Several floating-point libraries consisting of basic components, such as adders, multipliers and dividers, have been reported [3], [4]. In spite of the focused attention results to date are still mixed in part due to achievable latencies on FPGAs continuing to lag behind modern microprocessors with higher clock frequencies, floating-point accelerators, and better caching effects. Each generation of FPGAs have typically clocked one order of magnitude slower than their microprocessor counterparts. Memory access latencies between the DRAM and the FPGA lag cache latencies. Additionally, historical size limitations of FPGAs have required designers to create custom point designs to reduce gate counts. These effects have combined to result in very poor circuit reuse and floating-point IP portability between different platforms and applications. Recently emerging Platform FPGAs are addressing historical gate density limitations and provide additional diffused components such as multipliers and BRAMs. As the size and capabilities of Platform FPGAs grow, more flexible and programmable floating-point accelerators can be created to overcome the historical reuse and portability limitations.

Among floating-point operations, accumulation has always been of special interest [8]–[14] due to its prevalence across broad scientific application domains, e.g., sparse matrix-vector multiplication (SpMxV) [15]. SpMxV typically involves the multiplication of all the non-zero elements in a matrix by a vector, before adding the result. As the number of non-zeros is not known *a priori*, the size of the accumulations varies between rows, but it would often be important to ensure that data arrive in order in iterative methods. In this work, we propose two modular architectures that allow designers to easily create fully pipelined floating-point accumulators using fully pipelined adders and FIFOs. The modular architectures provide high throughput but with the advantage of a portable and standard interface. The architectures allow variable length vectors of either floating-point or integer operands to be input one item every clock cycle without requiring stalls between the input vectors. Our modular architectures were not designed to compete with the designs such as [8]–[14] that seek minimal gate counts. Instead, we focus on bringing high performance with new levels of reuse and portability for the logic designer. We have modeled our two modular architectures in Verilog-HDL and analyze acheivable performance within a real life application, i.e., Hessenberg reduction. Our implementation results show that the proposed designs are able to outperform all previous work by a large margin in terms of both clock frequency and latency while maintaining portability and reuse.

Our design provides the following capabilities for floating-point as well as integer accumulation:

- ▷ Full pipelining: this is the base requirement for the hardware implementation to achieve high performance;
- ▷ Ease of use: the standard interface appears as a normal primitive operator and the accumulator itself can be used as a primitive operator;
- ▷ Scalability: the accumulator operates over data sets of arbitrary sizes;
- ▷ Portability: the architecture of the accumulator is easily replicated on any hardware platforms (e.g., FPGA, ASIC) or fabrication technologies.

The remaining text is organized as follows. The related work is briefly discussed in Section II. Section III discusses the hardware architectures of two fully pipelined accumulators in detail, followed by results in Section IV. Finally, Section V concludes this work.

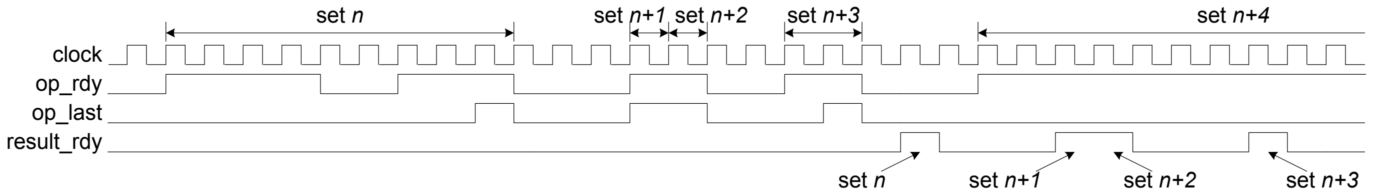


Fig. 1. The operating diagram of the proposed fully pipelined accumulators

II. Related Work

In [8], Luo and Martonosi proposed a delayed addition technique to improve the performance of floating-point accumulation. However, their design is not fully pipelined, i.e., the accumulator may need to stall internally to deal with overflow. The stall-related logic is very complicated and makes the overall approach difficult to scale. Further, as challenged by [9], its correctness and accuracy may be questionable. He *et al.* proposed a group-alignment algorithm to design an accurate floating-point accumulator [9]. There are two drawbacks in this approach. (i) A pipeline-stall is required between the processing of two consecutive data sets, which halves achievable throughput. (ii) A 1-clock-cycle-latency is required for the internal fixed-point accumulator. This requirement challenges the approach's ability to scale to double or higher precision operations. Three architectures, FCBT, DSA and SSA, consisting of adders, buffers and complex control logics are proposed in [10]. FCBT requires the knowledge of the maximum number of items in a set *a priori*, which negates the design's ability to operate in general scenarios. Both DSA and SSA produce out-of-order results when dealing with data sets of varying sizes, causing difficulties when used in hardware. Bodnar *et al.* [11] demonstrated a variant design of a floating-point accumulator based on work reported in [10]. The design in [11] produces out-of-order results as well. An application-specific and FPGA-specific design of floating-point accumulating circuit is proposed in [12]. As claimed in their work, the parameters of the design needs to be tuned for the target application, therefore limiting its broad usability. Sun and Zambreno proposed an architecture [13] where positive and negative operands in a set are summed separately into two intermediate results and then added together. As mentioned by the authors, the accuracy of their approach is a problem. Nagar and Bakos [14] attempted to reduce the complexity of control logic circuitry by integrating a coalescing reduction circuit within the low-level design of a base-converting floating-point adder. Unfortunately, the solution is currently incomplete. First, the use of a 3-stage reduction circuit is based on synthesis across two Xilinx FPGA devices, which negate its ability to be used on other platforms or technologies. Second, a minimum set size is required due to the multiple-stage reduction circuit, making the application of their solution very limited.

Compared with previous work, our proposed architectures offer the following three advantages. (i) They are fully pipelined, providing high performance. (ii) They bring ease of use and are scalable. (iii) They are modular architectures with trivial control logic, making them portable to different platforms.

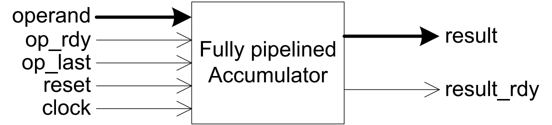


Fig. 2. The interface of the proposed fully pipelined accumulators

III. Fully Pipelined Accumulator Design

In this section, we first describe the interface and the application scenario of the modular architectures. Then we present the architectures of these two fully pipelined accumulators in detail.

A. The Interface

In the most general scenario, an accumulator sums up arbitrary numbers of items in numerous data sets. The size of a data set, i.e., the quantity of items in a set, can be arbitrary. Our generic scenario also allows items from the input data sets to be input into the accumulator continuously or sporadically. After one data set is finished, the next data set should be allowed to be input into the accumulator immediately or after some indefinite time, such as the example shown in Fig. 1. In all cases, the accumulator should accept the data as it is presented and produce correct summations in the same order.

To meet the above requirements, we design the interface of the accumulator as a primitive operator shown in Fig 2. The input and output signals of this black box operator are:

- ▷ *reset*: reset the status of internal control logic and internal registers.
- ▷ *operand*: the input data item.
- ▷ *op_rdy*: indicate the validity of an input operand.
- ▷ *op_last*: indicate the last item in a data set; should be asserted with the *op_rdy* signal of the last item.
- ▷ *result*: the summation of a data set.
- ▷ *result_rdy*: indicate the validity of the *result* signal.

An example diagram demonstrating the sequence among these control signals is given in Fig. 1. In this simple example, the multiple data sets with mixed sizes are summed and output by the accumulator in order. The summation result is marked as ready on the output after the *op_last* signal is asserted. The latency between the last input item in a data set and the result of the summation is not fixed. In other words, the user needs

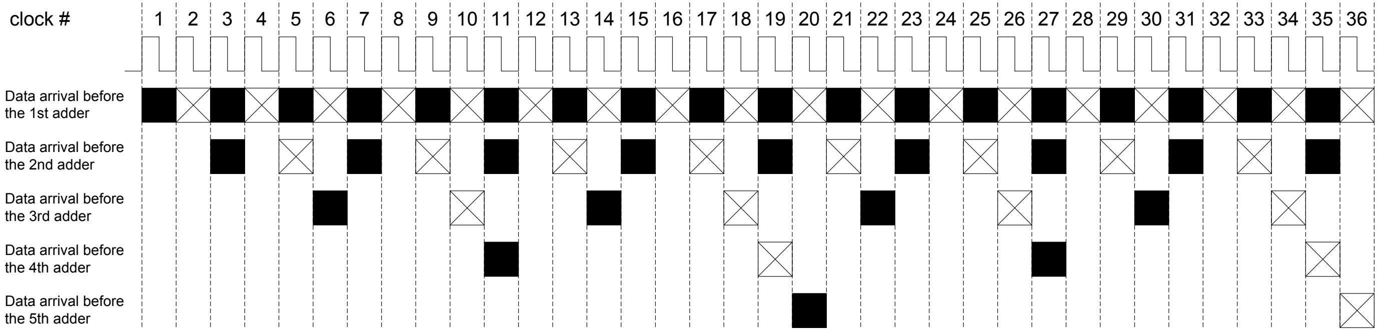


Fig. 5. The data arrival pattern inside the chain of adders (assuming the latency of each adder is 1 clock cycle; ■ and ⊠ denote two consecutive inputs to an adder)

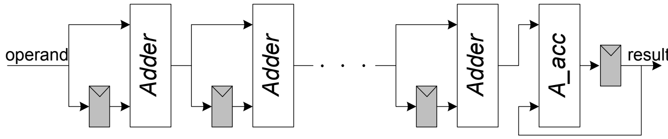


Fig. 3. Use adders to build an accumulator (*A_acc* is same to other adders)

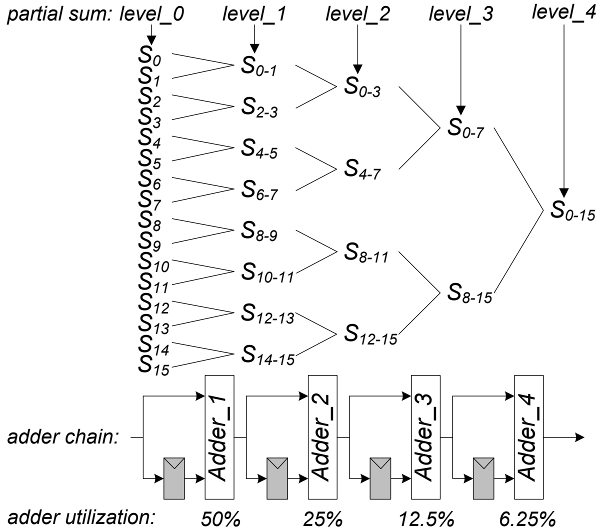


Fig. 4. Reduce items in a data set into partial sums using a chain of adders (assuming $N = \lceil \log_2 L \rceil = 4$)

to check the *result_rdy* signal and reads the result when this signal is asserted.

B. The Core Idea

Our proposed accumulator is implemented with fully pipelined adders to increase throughput. If the latency of the primitive adder is 1 clock cycle, the adder itself is an accumulator. However, today's floating-point adders are pipelined and thus incur latencies of dozens of clock cycles to carry out a single addition. Even for integer addition, it can take multiple clock cycles to finish an operation when the precision of the operands is quite large (e.g., 128-bit or 256-bit). Even though it is possible to use a single adder to perform accumulation, the user would have to wait for L clock cycles before pushing the next item into the adder, where L is the latency of the adder. This L-clock-cycle latency would require decreasing the incoming rate of operands to the single adder that performs the accumulation. Fortunately, addition itself is a reduction operation, which reduces two inputs to one output. In other words, the data rate is reduced to half after one addition. So, a simple solution to build an accumulator is based on multiple adders that form a chain as shown in Fig. 3. The chain then feeds the last adder (i.e., *A_acc*), which accumulates the partial results. By adopting a technique similar to log-sum [16], an N-adder chain reduces a block of 2^N items into a partial sum and lowers the data rate at the same time, as shown in Fig. 4. Fig. 5 shows the input data arrival patten in front of each adder in which we assume the latency of each adder is 1 clock cycle. If we assume that the original data rate is one item per clock cycle, the data rate drops to one item per 2^N clock cycles after the N-adder chain. By concatenating $N = \lceil \log_2 L \rceil$ adders into a chain and putting one additional adder at the end, we build an accumulator that is fully pipelined and is capable of handling an arbitrary length data set. Unfortunately, this simple design is not able to deal with the general case shown in Fig. 1. Two more capable designs are discussed in the following text.

C. The Modular Fully Pipelined Accumulator (MFPA)

In [17] we reported a *quasi*-fully pipelined accumulator in which the user can feed a new operand into the accumulator every clock cycle. However, that approach required waiting L clock cycles after the last item was input to produce the result. The user was required to wait for this delay before entering a new data set. In this work, we are able to remove this limitation by adding an internal FIFO into the architecture such that the final design is a *genuine* fully pipelined accumulator.

The internal architecture of the fully pipelined accumulator is shown in Fig. 6. It consists of $\lceil \log_2 L \rceil + 1$ fully pipelined adders, one FIFO and associated control logic. Logically, the overall architecture is divided into two parts; *partial sum reduction* and *accumulation*. The first part consists of $\lceil \log_2 L \rceil$ adders that reduce the original items in a data set into partial sums. The second part accumulates these partial sums. The constituent adder has an interface similar to Fig. 2, i.e., two

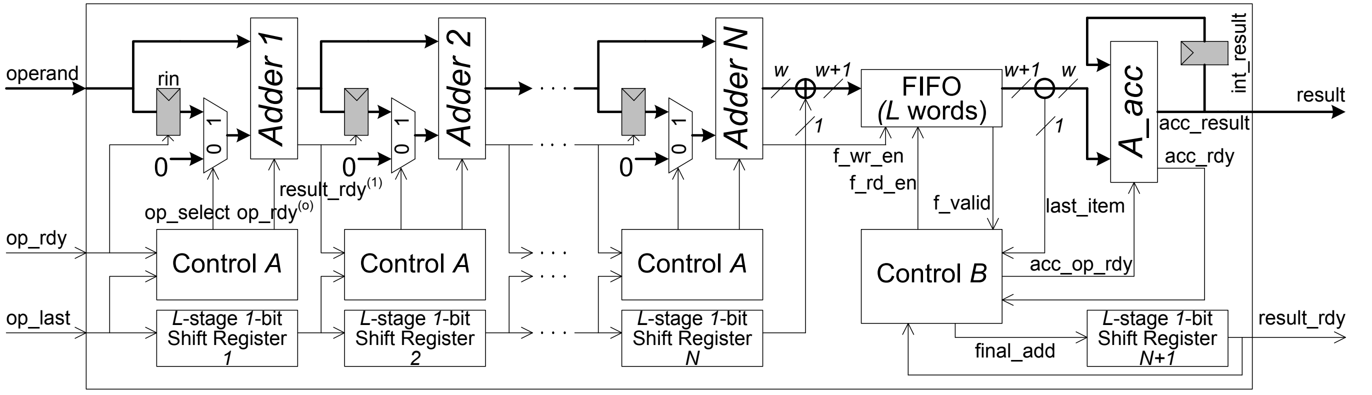
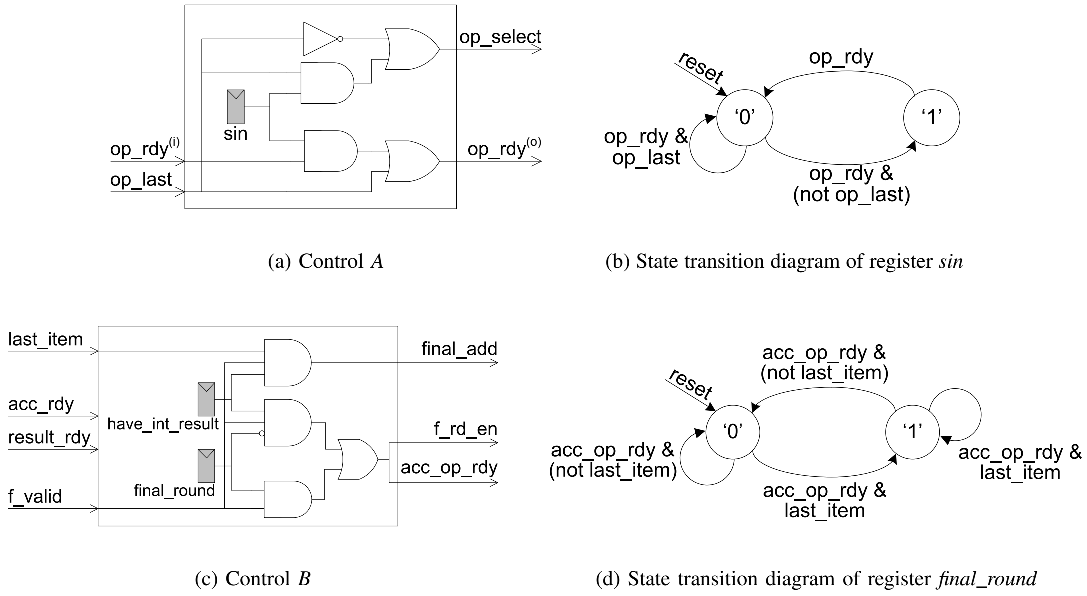


Fig. 6. The internal architecture of the modular fully pipelined accumulator (Note: (1) all the adders, including *Adder 1* to *Adder N* and *A_acc*, are the same; (2) $\oplus$ and $\ominus$ are concatenation and de-concatenation operations respectively)



(a) Control *A*

(b) State transition diagram of register *sin*

(c) Control *B*

(d) State transition diagram of register *final_round*

Fig. 7. The control logic used in the synchronous architecture

control signals, *op_rdy* and *result_rdy*, besides *operand* and *result*.

The first $\lceil \log_2 L \rceil^*$ adders form a chain that reduces the frequency of the inputs to *A_acc*, the adder after the FIFO carrying out the accumulation. The N adders in the adder-chain reduce the number of items by half at each level as shown in Fig. 4. Each adder takes two inputs and produces one output. For each pair of items, the first item is saved in a register *rin* before the arrival of the second item. The status of register *rin* is indicated by register *sin*, whose state transition diagram is illustrated in Fig. 7(b). In the normal case, the adder at each level will perform the addition of two items once both become available at the input ports, which is indicated by the *op_rdy*$^{(o)}$ signal. However, as soon as the last item in a set arrives, it will be added to either the previous item or zero depending on whether the last item is even-numbered or odd-numbered. This selection is realized using a 2-to-1 multiplexer with an *op_select* signal. As mentioned before, the *op_last* signal indicates the arrival of the last item in a data set to be accumulated. Internally, this signal will travel down through the shift registers to indicate the last partial sum at each level.

Since $2^N \geq L$, it is guaranteed that the data arrival interval to *A_acc* is greater than or equal to its latency most of the time. If the number of items in the original data set is p, then the adder-chain will reduce the number of items (to be accumulated) to $P = \lceil \frac{p}{2^N} \rceil$. In other words, the items $\{x_1, x_2, \ldots x_p\}$ in the original data set are reduced to $\{X_1, X_2, \ldots X_P\}$ after the adder-chain.

Given a sequence of P items, $\{X_1, X_2, \ldots X_P\}$, the interval between two items in the first $P-1$ items is guaranteed to be 2^N clock cycles since $X_j = \sum_{i=(j-1)\cdot 2^N+1}^{j \cdot 2^N} x_i$, $j = 1, 2, \ldots, P-1$. However, the last partial sum X_P in the

*We use N to denote $\lceil \log_2 L \rceil$ in the following text.

```verilog
always @ (posedge clock) begin
    if (reset) begin
        int_result      <= 0;
        have_int_result <= 1;
    end
    else begin
        if (!final_round) begin
            if (acc_rdy && (!result_rdy))
                have_int_result <= 1;
            else if (accu_op_rdy)
                have_int_result <= 0;
            else
                have_int_result <= have_int_result;
        end
        else begin
            if (accu_op_rdy && (!last_item))
                have_int_result <= 0;
            else
                have_int_result <= 1;
        end
        //////////////////////////////
        if (!final_round) begin
            if (acc_rdy)
                int_result <= acc_result;
            else
                int_result <= int_result;
        end
        else
            int_result <= 0;
    end
end
```

Fig. 8. The Verilog description for *have_int_result* and *int_result*

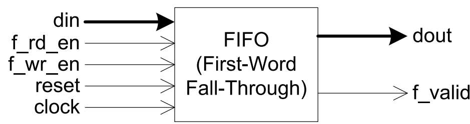


Fig. 9. The interface of the FIFO

same set may arrive any time between 1 clock cycle and 2^N clock cycles after X_{P-1} arrives because the size of the original set is arbitrary and the architecture is not supposed to stall. Assuming that the several data sets following the current set (being accumulated) are all very short and their sizes are less than 2^N, then there is only one X_i (i.e., X_1) after the adder-chain for these short data sets. If $X_i^{(m)}$ denotes X_i in set m, $X_P^{(m)}, X_1^{(m+1)}, X_1^{(m+2)}, \ldots$ may arrive when *A_acc* is performing $X_{P-1}^{(m)} + \sum_{i=1}^{P-2} X_i^{(m)}$. In order to solve this data hazard and keep the results in order, these data items are saved in the FIFO first. Since the latency of *A_acc* is L clock cycles, at most L storage cells are required to save these data temporarily.

The output of Shift Register *N* is concatenated with the result from *Adder N* and fed into the FIFO. In other words, the data width of the FIFO is $w + 1$-bit in which w is the operand precision of the accumulator. The extra one bit is used to tell whether the associated partial sum is the last item in one data set. The *result_rdy* signal of *Adder N* serves as the *f_wr_en* signal to the FIFO, whose interface is illustrated in Fig. 9. The FIFO applies First-Word Fall-Through policy, i.e., the first word is always put at the output port and its validity is indicated by signal *f_valid*, which is equivalent to *empty* signal of a traditional FIFO (with opposite meaning). The *full* signal is not used in our architecture because it is guaranteed that the FIFO will never become full in our design.

The *A_acc* performs the computation of $\sum_{i=1}^{P} X_i$. The addition of $X_j + \sum_{i=1}^{j-1} X_i$ $(j = 1, 2, \ldots, P)$ is triggered by signal *acc_op_rdy*, which serves as the *f_rd_en* signal as well. When there is a valid item at the output port of the FIFO, it has to check whether this item and the item that is previously pushed into the *A_acc* belong to the same data set. This decision is made by checking the status flag signal *final_round*, which indicates the last addition in the accumulation of a data set, i.e., $X_P^{(m)} + \sum_{i=1}^{P-1} X_i^{(m)}$ for some m. As mentioned before, the last item in a data set is indicated by an extra bit, *last_item*, along the data item. The *last_item* signal will trigger the status change of the *final_round* signal, as shown in Fig. 7(d). If the previous addition in the pipeline of *A_acc* is not the final addition in a data set, the addition of the current output of the FIFO with int_result has to wait until int_result becomes valid, which is indicated by *have_int_result*. Otherwise, the current output of the FIFO can be pushed into the FIFO right away by adding zero onto it. The status change of *have_int_result* and the value assignment to *int_result* are illustrated in Fig. 8, coded in Verilog. Once the final addition in a data set is pushed into the *A_acc*, a *final_add* signal is generated at the same moment and pushed into a shift register whose output will indicate the availability of the summation of a data set.

D. The Area-efficient Modular Fully Pipelined Accumulator (AeMFPA)

A careful evaluation of the prior MFPA architecture shows that the $N = \lceil \log_2 L \rceil$ adders are not fully utilized. The utilization of the adders is shown in Fig. 4. The utilization of the first adder is only $\frac{1}{2}$. The sum of the utilization of these N adders is shown in (1).

$$\sum_{i=1}^{N} \left(\frac{1}{2}\right)^i = 1 - \left(\frac{1}{2}\right)^N < 1 \quad (1)$$

Thus, it is possible to combine the N adders into a single adder to decrease the transistor count but still provide the same throughput. The work in [10] demonstrates the use of one single adder for accumulation, but does require complex scheduling control logic and out-of-order output. In our second architecture, we keep the same simple interface but make the accumulator more area efficient. We call the second design *area-efficient modular fully pipeline accumulator (AeMFPA)*, as shown in Fig. 10.

The second part of the AeMFPA architecture is identical to the MFPA design. The difference comes from within the first part of the MFPA design. We use only a single fully pipelined adder to perform the reduction operations. The challenge of this reduction is to guarantee that the overall architecture remains robust in its ability to deal with a random data input pattern. Particularly, two techniques are applied to realize this objective.

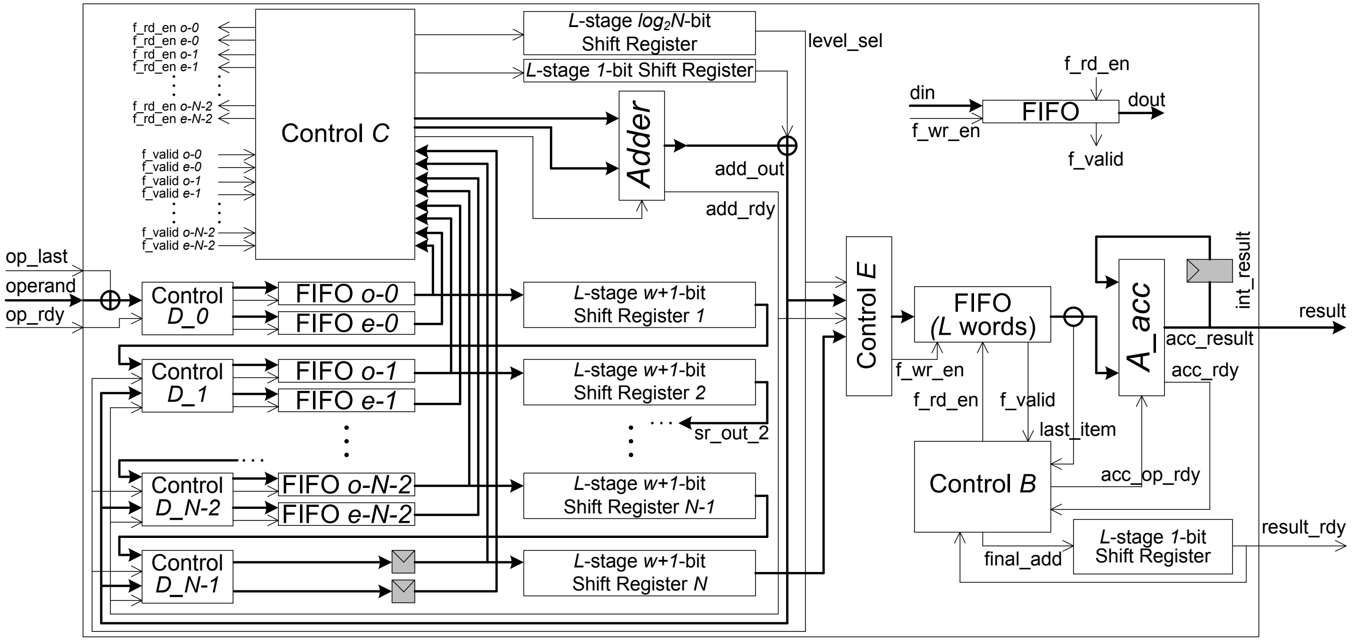


Fig. 10. The internal architecture of the area-efficient modular fully pipelined accumulator (Note: (1) both adders, *Adder* and *A_acc*, are the same; (2) ⊕ and ⊖ are concatenation and de-concatenation operations respectively)

- ▷ Put the partial sums of different levels separately in different FIFOs;
- ▷ Add by-pass shift registers to solve resource contention regarding the first adder.

As shown in Fig. 4, the data reduction part takes N levels, each of which generates partial sums. Except the partial sums generated by level N, the partial sums at one level are the inputs of the following level. In MFPA, there is one adder at each level. Therefore there is no resource contention and the output of an adder can be directly connected to the inputs of the follow-on adder. In the AeMFPA architecture, all levels share the same adder and compete for this sole resource. Therefore data storage is required to save partial sums before they are fed back into the adder. To keep final results in order and simplify control logic, the partial sums as well as the original items are put into different pairs of FIFOs corresponding to various levels, as shown in Fig. 10. A pair of FIFOs are used at each level to allow two operands to be read simultaneously.

Since N pairs of FIFOs or registers ($N-1$ pairs of FIFOs from level 0 to level $N-2$ and a pair to registers at level $N-1$) are connected to the same *Adder*, a policy is necessary to determine which pair of operands to pick up. In our design, we assign decreasing priorities onto different levels, with level $N-1$ having the highest priority. When two pairs of FIFOs both have items to be added, the two items in the pair of FIFOs of higher priority are selected for processing. In this way, the pair of FIFOs at level $N-1$ only need to have depth of 1, and can be replaced by the pair of registers shown in Fig. 10. This selection process is implemented as *Control C* in AeMFPA. All valid signals of the FIFOs and the status of the pair of registers at level $N-1$ are checked to select one pair of operands for the *Adder*. Besides the two operands, *Control C* needs to generate the following two extra attributes with the outgoing partial sum:

- ▷ The level of the outgoing partial sum, i.e., the following level. This attribute is pushed into a shift register that is parallel to the *Adder*. Because there are N levels, the width of the shift register is $\lceil \log_2 N \rceil$ bits.
- ▷ If the two selected operands are the last two items in the current level, the outgoing partial sum will be the last item in the following level. This attribute is pushed into the other 1-bit-wide shift register parallel to the *Adder*. The output of this shift register and the outgoing partial sum will be saved into FIFOs, which are all $w+1$-bit wide.

If the size of all data sets is the multiple of 2^N, the *Adder* will never be overused due to (1). However, it is mentioned before that the input pattern is random. In an extreme case, when a multiple-item set is followed by many single-item sets, those single items need to be pushed down without delay. In the MFPA architecture, the continuity of single-item sets will cause the utilization of all adders in the adder-chain to be 1. Under this extreme case, the utilization sum of all adders will be N, causing the overuse of the single *Adder* in AeMFPA architecture. In order to solve this resource contention issue, N layers of shift register are added into AeMFPA. At each level, if the last partial sum in a data set is odd-numbered, it will be pushed into the corresponding shift register. For example, assuming there are 11 partial sums at level 1, the first 10 partial sums will be added together by the *Adder* 2-by-2. The last partial sum will be pushed into the Shift Register 2 instead.

The valid output of Shift Register i (indicated by its most

TABLE I
COMPARISON WITH PREVIOUS WORK

Design*	Adders	Buffer Size	Frequency	Total Latency per Set	Latency per Set	In-order	Fully Pipelined	Scalable	Control Logic
PCBT [10]	$\lceil \log_2 p \rceil$	$2\lceil \log_2 p \rceil$	Decrease with p	$p + L\lceil \log_2 p \rceil$	Predictable	Yes	Yes	No	Simple
FCBT [10]	2	$3\lceil \log_2 p \rceil$	Decrease with p	$\leq 3p + (L-1)\lceil \log_2 p \rceil$	Predictable	Yes	No	No	Complex
DSA [10]	2	$L\lceil \log_2 L + 1 \rceil$	Stable	$p + L\lceil \log_2 L + 1 \rceil$	Not Predictable	No	Yes	Yes	Complex
SSA [10]	1	$2L^2$	Stable	$\leq p + 2L^2$	Not Predictable	No	Yes	Yes	Complex
FAAC [13]	$< \lceil \log_2 k \rceil + 2$	$2k$	Stable	$p + k + L\lceil \log_2 L + 1 \rceil$	Predictable	Yes	Yes	Yes	Medium
MFPA	$\lceil \log_2 L \rceil + 1$	L	Stable	$\leq p + L\lceil \log_2 L + 2 \rceil$	Predictable	Yes	Yes	Yes	Simple
AeMFPA	2	$< 2L$	Stable	$\leq p + L\lceil \log_2 L + 2 \rceil$	Predictable	Yes	Yes	Yes	Simple

*p: number of items in a set; L: the latency of the constituent adder; k: the latency of the simplified adder in [13].

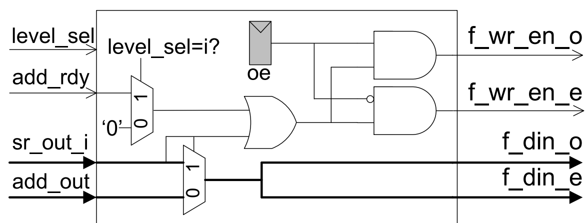


Fig. 11. Control D logic

TABLE II
IMPLEMENTATION RESULTS OF TWO MODULAR ARCHITECTURES

Design	XC2VP30		XC5VLX110T	
	Slices	Frequency (MHz)	Slices	Frequency (MHz)
FAAC [13]	6,252	162	2,269	244
Constitute Adder	978	221	344	429
MFPA	4,991	207	1,692	367
AeMFPA	3,130	204	1,234	321

significant bit) and a valid output (indicated by add_rdy) of the *Adder* for level i (indicated by $level_sel$) are put into the pair of FIFOs in a ping-pong fashion. In other words, the partial sums in a data set are put into FIFO-o and FIFO-e one after the other indicated by Register oe, as shown in Fig. 11. The very first partial sum in a data set is always put into FIFO-o. Control D_0 is simpler because it only deals with the original items in a data set. Because we use FIFOs to keep all intermediate results in order, the partial sums belonging to the same level always arrive in a sequence. In Fig. 11, either *sr_out_i* or *add_out* can be valid for level i at one moment, not both.

Control E is a simplified version of Control D. Whenever there is either a valid output from Shift Register N or a valid output from the *Adder* for level N, the output is pushed into the FIFO.

The resource requirement of the $N-1$ pairs of FIFOs is trivial. Based on (1), the single *Adder* has the capacity to process the outputs from different levels of FIFOs in time. Therefore, the depth of these FIFOs is of single-digit.

IV. EVALUATION

A. Analysis

The two proposed modular architectures are compared with other previous architectures in Table I. Both PCBT and FCBT are not scalable designs because they require to know the maximal size of the data set in advance, which is impossible in general case. Although DSA and SSA remove this limitation, they bring in another inconvenience, the out-of-order output sequence. This feature makes them difficult to be used in real hardware implementation. The complex control logic used in DSA and SSA further renders them unattractive. The FAAC architecture is able to handle arbitrary size of data set, produce the results in the order, and have a comparably reasonable control logic. However, it breaks the sequence of addition among the items in a data set, which brings in computation error because floating-point operations are non-associative. Further, FAAC architecture requires to use three different types of floating-point operators, which may increase the implementation effort. In contrast, our designs deliver a modular architecture with trivial effort to implement, and bring in the ease of use, the high performance and the accuracy.

TABLE III
IMPLEMENTATION RESULTS COMPARISON BETWEEN DIFFERENT DESIGNS ON XILINX XC2VP30

Design*	Adders	Slices	BRAMs	Frequency (MHz)	Total Latency†		Slices
					clock cycles	μs	$\times \mu s$
PCBT [10]	7	6,808	–	165	226	1.370	9,327
FCBT [10]	2	2,859	10	170	≤ 475	≤ 2.794	7,988
DSA [10]	2	2,215	3	142	232	1.634	3,619
SSA [10]	1	1,804	6	165	≤ 520	≤ 3.152	5,686
FAAC [13]	4	6,252	0	162	176	1.086	6,790
MFPA	5	4,991	2‡	207	198	0.957	4,776
AeMFPA	2	3,130	14‡	204	198	0.970	3,036

*$p = 128$, $L = 14$, $k = 4$. DSA, SSA, FAAC, MFPA and AeMFPA can deal with arbitrary size of data sets.
†Accumulate a data set of 128 items.
‡All the Block RAMs that implement FIFOs are deeply underutilized.

Algorithm 1: Hessenberg reduction (vector-based)

Input: A square complex matrix A with rank n
Output: The Hessenberg matrix H
1.1 **for** k=*0* **to** $n-3$ **do**
1.2 $v_k =$ **House**$(A_{k+1:n-1,k})$; /*Step 1*/
1.3 $A_{k+1:n-1,k:n-1} =$
$A_{k+1:n-1,k:n-1} - 2v_k(v_k^* A_{k+1:n-1,k:n-1})$; /*Step 2*/
1.4 $A_{0:n-1,k+1:n-1} =$
$A_{0:n-1,k+1:n-1} - 2(A_{0:n-1,k+1:n-1} v_k)v_k^*$; /*Step 3*/

B. Implementation

We have modeled our architectures on both Xilinx XC2VP30 and XC5VLX110T FPGA devices using a fully pipelined double precision floating-point accumulator. The constituent adder and the FIFO are generated using Xilinx Core Generator 10.1. A single adder consists of 14 pipeline stages. The FIFOs are generated using BRAM. The resource requirement of both the primitive adder and the two designs on two platforms is listed in Table II.

Table III lists the implementation details of a double precision floating-point accumulator using 7 different architectures on Xilinx XC2VP30. The results for PCBT, FCBT, DSA, SSA, and FAAC are referenced from [10] and [13] respectively. Our architectures are synthesized by Xilinx XST and place-and-routed by Xilinx ISE 10.1 with default setting. Apparently, our architecture enjoys a higher frequency and a lower latency. Both MPFA and AeMPFA operate at frequencies that are 20%$\sim$46% faster than other architectures, respectively. Although the total latency of our architectures are bigger than that of FAAC in terms of clock cycle, both MFPA and AeMFPA outperform FAAC by 13% and 12% in terms of latency in μs. In the meantime, the resource requirement of MFPA and AeMFPA is only 80% and 50% of FAAC, respectively. Overall, our AeMFPA enjoys the smallest *area*$\times$*latency* in all 7 architectures.

The correctness and accuracy of our proposed architectures have been verified in a real application, QR eigenvalue algorithm [17] in which the double precision floating-point accumulator is used in the Hessenberg reduction step as shown in Alg. 1. The Hessenberg reduction involves the matrix/vector and vector/vector multiplications at various length of arbitrary order. By providing a simple interface, the accumulator can be integrated into the whole design with little effort. Being fully pipelined, it can generate the desired performance of a hardware implementation.

V. Conclusion

In this work, we propose two novel and modular architectures to construct fully pipelined accumulators comprising constituent fully pipelined adders, trivial storage requirement and simple control logic. Both architectures are capable of performing efficient accumulation over data sets of arbitrary sizes, and generating results in-order. Due to the modular design and the simple interface, it is convenient for both the implementer and the user to adopt these two architectures in various real life applications. Since these two architectures can be used to deal with both integer and floating-point operations with arbitrary precision, they have the widest impact although the proposed architectures are more attractive for floating-point accumulation. Implementation results on both Virtex II and Virtex 5 show that both architectures have higher frequency and lower latency than all previous work.

References

[1] G. Lienhart, A. Kugel, and R. Männer, "Using floating-point arithmetic on FPGAs to accelerate scientific N-body simulations," in *Proc. the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'02)*, Apr. 2002, pp. 182–191.

[2] S. Paschalakis and P. Lee, "Double precision floating-point arithmetic on FPGAs," in *Proc. The 2003 IEEE International Conference on Field-Programmable Technology (FPT'03)*, Dec. 2003, pp. 352–358.

[3] G. Govindu, R. Scrofano, and V. K. Prasanna, "A library of parameterizable floating-point cores for FPGAs and their application to scientific computing," in *Proc. The International Conference on Engineering Reconfigurable Systems and Algorithms (ERSA'05)*, Jun. 2005, pp. 137–145.

[4] K. S. Hemmert and K. D. Underwood, "Open source high performance floating-point modules," in *Proc. the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'06)*, Apr. 2006, pp. 349–350.

[5] L. Zhuo and V. K. Prasanna, "Scalable and modular algorithms for floating-point matrix multiplication on reconfigurable computing systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 18, no. 4, pp. 433–448, Apr. 2007.

[6] J. Detrey and F. de Dinechin, "Parameterized floating-point logarithm and exponential functions for FPGAs," *Microprocessors and Microsystems*, vol. 31, no. 8, pp. 537–545, Dec. 2007.

[7] J. Sun, G. D. Peterson, and O. O. Storaasli, "High-performance mixed-precision linear solver for FPGAs," *IEEE Trans. Comput.*, vol. 57, no. 12, pp. 1614–1623, Dec. 2008.

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

[9] C. He, G. Qin, M. Lu, and W. Zhao, "Group-alignment based accurate floating-point summation on FPGAs," in *Proc. The International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'2006)*, Jun. 2006.

[10] L. Zhuo, G. R. Morris, and V. K. Prasanna, "High-performance reduction circuits using deeply pipelined operators on FPGAs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 18, no. 10, pp. 1377–1392, Oct. 2007.

[11] M. R. Bodnar, J. R. Humphrey, P. F. Curt, J. P. Durbano, and D. W. Prather, "Floating-point accumulation circuit for matrix applications," in *Proc. 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'06)*, Apr. 2006, pp. 303–304.

[12] F. de Dinechin, B. Pasca, O. Cret, and R. Tudoran, "An FPGA-specific approach to floating-point accumulation and sum-of-products," in *Proc. The 2008 IEEE International Conference on Field-Programmable Technology (FPT'08)*, Dec. 2008, pp. 33–40.

[13] S. Sun and J. Zambreno, "A floating-point accumulator for FPGA-based high performance computing applications," in *Proc. The 2009 IEEE International Conference on Field-Programmable Technology (FPT'09)*, Dec. 2009, pp. 493–499.

[14] K. K. Nagar and J. D. Bakos, "A high-performance double precision accumulator," in *Proc. The 2009 IEEE International Conference on Field-Programmable Technology (FPT'09)*, Dec. 2009, pp. 500–503.

[15] J. Sun, G. Peterson, and O. Storaasli, "Sparse matrix-vector multiplication design on FPGAs," in *Proc. the 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'07)*, Apr. 2007, pp. 349–352.

[16] P. M. Kogge, *The Architecture of Pipelined Computers*. Taylor & Francis, Jan. 1981.

[17] M. Huang and O. Kilic, "Reaping the processing potential of FPGA on double-precision floating-point operations: an eigenvalue solver case study," in *Proc. the 18th Annual International IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2010)*, May 2010, pp. 95–102.