Efficient Pipelined Multi-Operand Adders with High Throughput and Low Latency: Designs and Applications

Chi-Hsiang Yeh and Behrooz Parhami
Department of Electrical and Computer Engineering
University of California, Santa Barbara, CA 93106-9560, USA

Abstract

We describe several approaches for performing multi-operand addition. Our constructions are regular and modularized, and their required circuit size and depth compare favorably with previously proposed schemes. We show that the sum of n k-bit integers can be found using pipelined circuits of size $nk/d + O(k(\log n + \log k))$, latency $O(\log n + \log k + d)$, and a feedback loop with a single full-adder delay, where d can be any positive integer. We also develop several techniques to fine-tune the constructions in order to obtain circuits that are adaptive to application requirements. In particular, we generalize the block-save technique and obtain competitive constructions for multi-operand addition by combining the technique with that of ripple adder trees. we also demonstrate how to apply multi-operand adders to the computation of several useful functions.

1. Introduction

Multi-operand addition is a fundamental problem in arithmetic and algebraic computations. Because of its importance, many researchers have dealt with multi-operand addition in general or in specific application contexts [2, 4, 10, 11, 12, 17].

In 1964, Wallace proposed the well-known "Wallace tree," which is built of carry-save adders and performs the (n,n) multi-operand addition in $O(\log n)$ time, where the (n,k) multi-operand addition is the problem of computing the sum of n k-bit integers. Dadda extended Wallace's idea to obtain multi-operand addition circuits based on parallel counters [15], while Ho and Chen [4] devised circuits for multi-operand addition using column compressors. All of the above schemes first perform an (n,k) sum reduction, to reduce the number of operands from n to 2, and then use a carry-propagate adder to compute the final sum.

We introduce a unifying scheme that achieves high-throughput and low latency at the same time. We show that by using the notions of accumulative compression [9] and merged arithmetic [16], we can obtain a pipelined (n,k) multi-operand adder that has size $nk/d + O(k(\log n + \log k))$, latency $O(\log n + \log k + d)$, a feedback loop with a single full-adder delay, and constant fan-in, where d can be any positive integer. These constructions achieve asymptotically optimal latency, size, and throughput at the same time and provide effective tradeoffs between these parameters. The resulting design can operate at a higher throughput than similar-sized pipelined partial CSA trees [3, 6] which have a feedback loop with 2 full-adder delays. We also demonstrate that these results can be applied to the computation of several useful functions.

2 Low latency multi-operand adders

Given n one-bit numbers, an $(n, \lceil \log_2(n+1) \rceil)$ *counter* is a circuit that produces the $\lceil \log_2(n+1) \rceil$-bit binary representation of the sum of the n bits [15, 9]. In our construction we first use parallel counters to compress each column and then use the block-save technique to complete the (n,k) sum reduction. Finally, a carry propagate adder is employed to compute the final sum.

2.1 Outline of the basic scheme

Our proposed design scheme consists of 3 phases and is illustrated by the example in Fig. 1.

- Phase 1: The number of 1s in each of the k positions is computed using an (n,m) counter, where $m = \lceil \log_2(n+1) \rceil$.
- Phase 2: The $k+m-1$ positions are partitioned into $\lceil (k+m-1)/l \rceil$ blocks, each containing l positions (except possibly for the block containing the most significant positions), where $l = \lceil \log_2(m-1) \rceil$. Each block of the outputs from Phase 1 is added separately using an (m,l) multi-operand adder.
- Phase 3: The sums obtained from blocks 0, 2, 4,... are concatenated into a $(k+m+l-2)$-bit number S_{even} and the sums obtained from blocks 1, 3, 5,... are concatenated into a $(k+m+l-2)$-bit number S_{odd}, whose least significant l bits are filled with 0s. The sum of S_{even} and S_{odd} is computed using a binary carry-propagate adder.

Phase 1 is the same as an iteration of column compression, while Phase 2 uses the block-save technique [13].

1058-6393/97 $10.00 © 1997 IEEE

Since the sum of a block in Phase 2 is no more than $m \cdot (2^l - 1) \leq 2^{2l} - 1$, it can be represented by a $2l$-bit number and the binary representations of all odd blocks (or even blocks) will not overlap with each other (see Fig. 1). As a result, S_{even} and S_{odd} can be obtained by simply concatenating the $2l$-bit sums of even and odd blocks, respectively, during Phase 3. Phases 1 and 2 perform the (n,k) sum reduction and Phase 3 completes the multi-operand addition process.

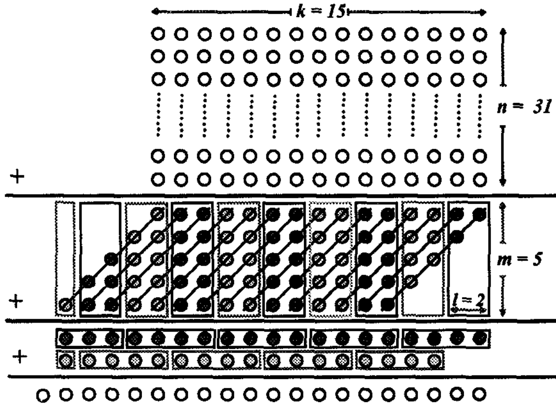

Figure 1. Illustrating the basic scheme for $(31,15)$ multi-operand addition.

2.2 Complexity analysis

It is well known that an (n,m) parallel counter can be implemented using a bounded fan-in AND-OR circuit of size $O(n)$ and depth $O(\log n)$. Various designs can be found in the literature [1, 5, 7, 9, 15]. As a result, Phase 1 can be done using a circuit of size $O(nk)$ and depth $O(\log n)$ for all the k positions. This phase usually dominates the size of the entire circuit except when $n < \log k$.

Phase 2 requires $O(k/\log n)$ copies of an (m,l) multi-operand adder, each implemented using either a tree of ripple adders or a CSA tree followed by a carry-propagate adder [15, 6] and requiring depth $O(\log\log n)$ and size $O(\log n \log\log n)$. As a result, Phase 2 needs a circuit of size $O(k \log n)$ and depth $O(\log\log n)$ for all the $O(k/l)$ blocks. This phase does not dominate the circuit complexity.

Phase 3 can be done using a wide variety of binary adders, such as carry-lookahead, conditional-sum, or carry-skip adders. The required circuit size is $O((k + \log n) \log(k + \log n))$ and the depth is $O(\log(k + \log n))$.

As a consequence, the required size is $O(nk + k \log k)$ and the depth is $O(\log n + \log k)$ for an (n,k) addition problem using this scheme.

2.3 Comparison with previous work

Compared with CSA trees for the (n,k) sum reduction problem, the construction used in this paper is more regular and modularized; compared with trees of ripple adders and the schemes proposed in [2], the latency of the previous construction is considerably smaller when k is not small.

The column compression scheme proposed in [4] for the (n,k) sum reduction problem requires $O(\log^* n)$ iterations of column compression, where $\log^* N$ is defined as the smallest integer k satisfying $\underbrace{\log \cdots \log}_{k} N \leq 1$. The construction for (n,k) sum reduction presented in [11] uses two iterations of column compression followed by $O(k/\log\log n)$ parallel copies of the circuit computing an arbitrary Boolean function with $O((\log\log n)^2)$ input bits, which requires size $O(2^{(\log\log k)^2/\log\log\log k})$ and depth $O((\log\log n)^2)$. Our method only requires one iteration of column compression followed by $O(k/\log\log n)$ parallel copies of the circuit performing $(O(\log n), O(\log\log n))$ sum reduction. In fact, using arguments similar to those used for the partitioning of the block-save technique, the $O(k/\log\log n)$ circuits, each computing Boolean functions with $O((\log\log n)^2)$ input bits, required in [11] can be reduced to $O(k/\log\log\log n)$ simpler circuits, each computing Boolean functions with $O(\log\log n \log\log\log n)$ input bits.

3 Compact accumulative or pipelined designs

In this section, we modify our construction to obtain circuits with comparable latency and considerably smaller size based on accumulative parallel counters [9].

3.1 Accumulative parallel counters

An accumulative parallel counter (APC) is a true generalization of serial counters, and can be used to replace the parallel counters in the previous algorithm in order to obtain size-efficient multi-operand adders with small latency. An (n,q) APC is defined as a sequential circuit with a single q-bit word of memory holding, at the end of each counting cycle, the sum of its previous content and its n single-bit inputs. An example $(16,6)$ APC is shown in Fig. 2. The example $(16,6)$ APC uses a pipelined $(15,4)$ counter followed by several stages of latches and a pipelined 6-bit ripple adder. The latches following the parallel counter schedule its outputs so that they arrive at the ripple adder at the same time. Sixteen single-bit signals $x_0, x_1, \ldots, x_{15}$ can be supplied to the APC at each clock cycle, and the sum of these inputs accumulates and is stored in the accumulative registers in carry-save form. At any cycle, the sum of the two output numbers in carry-save form is equal to the sum of all inputs that have arrived at the ripple adder. We refer the reader to [9] for details.

3.2 High throughput accumulative adders

The construction introduced in Section 2 can be easily pipelined by adding latches at each level if maximum bandwidth is desired, or after every two (or more) levels to reduce the number of latches and the pipeline latency.

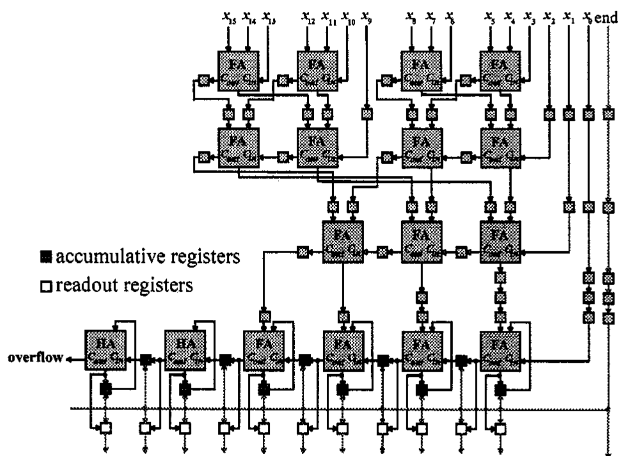

Figure 2. A pipelined $(16,6)$ **APC with "end" signal for counting the bits in one position for Phase 1 of the basic scheme.**

Although the previous design achieves competitive latency and size compared to previous designs, the circuit cost is prohibitively high when n and k are very large, and the very high throughput after being pipelined may not be required. In this subsection, we show that by increasing the latency by $d - O(\log d)$, where d can be any positive integer, the overall circuit size can be reduced by a factor of d.

For any integer $d \geq 1$, we partition the n input numbers into $\lceil n/d \rceil$ groups. Phase 1 of the basic scheme is implemented using $(\lceil n/d \rceil, \lceil \log_2(n+1) \rceil)$ APCs with proper control mechanism. At cycle 1, the first group of numbers is input, at cycle 2, the second group is input, at cycle 3 the third group, and so on. From cycle 1 (if $d > 1$) to cycle $d-1$, the input "end" signal (see Fig. 2) is 0 and at cycle d, the input "end" single is 1.

The "end" signal goes through the same pipeline levels as the other data bits of the multi-operand adder circuit. When the "end" signal propagates to the "readout" registers and the "accumulative" registers, the outputs from the last level of FAs and HAs are loaded into the "readout" registers and the "accumulative" registers are cleared.

During Phase 2, the data bits in the readout registers are partitioned into odd and even blocks and added using pipelined $(2\lceil \log_2(n+1) \rceil - 1, \lceil \log_2 \lceil \log(n+1) \rceil \rceil + 1)$ multi-operand adders. Note that the number of operands during Phase 2 is approximately doubled compared to that in the non-pipelined construction since they are now stored in carry-save form. Phase 3 requires a pipelined carry-propagation adder to find the final sum.

If Phases 2 and 3 are pipelined with the same cycle time as that of Phase 1, we can select any integer $1 \leq q \leq d$ as required to perform the $(q\lceil n/d \rceil, k)$ multi-operand addition. In fact, the multi-operand adder can be used for any number of operands not exceeding $2^{\lceil \log_2(n+1) \rceil} - 1$. When the "end" signal propagates to the last stage, the result of the multi-operand addition is available at the output register of the circuit. Note that the next multi-operand addition can be initialized after one idle cycle, following the last input group of the previous execution.

The construction can be improved in several ways. When many multi-operand additions are usually queued up and q is small on the average, it is desirable to eliminate the idle cycle between two executions of multi-operand additions. This can be done by inserting an AND gate following each of the outputs of accumulative registers. The complemented "end" signals ANDed with the uncleared accumulative registers serves to supply zero inputs to the full and half adders. When the "end" signal is 0, the accumulative register contents are passed to the adders unmodified.

When the minimum possible q is not 1, we can improve the subcircuits for Phases 2 and 3 by using fewer pipelining stages. The load signals for these stages can be easily provided by the propagated "end" signal. The numbers of latches required for Phases 2 and 3 are reduced by a factor of $q_{\min}$, where $q_{\min}$ is the minimum possible number of groups that will be used. The latency is also improved.

An alternative method is to accumulate the sum at the end of Phase 2. The subcircuits between the Phases may also be improved. Details of these modified constructions are omitted in this paper.

3.3 Comparison with previous work

Hallin and Flynn [3, 6] proposed a pipelined partial CSA tree to perform multi-operand addition with high throughput and small size. This method considerably improves the throughput of the naive partial CSA tree whose feedback path traverses the entire tree by limiting the feedback loop to only two levels of FAs. Our method further reduces the feedback loop to only one level of FAs based on APCs, increasing the throughput in [3, 6] significantly. Moreover, our constructions are more regular.

Compared to the method that "accumulates" the sum at the output of the $(n/d, k)$ multi-operand adder, which achieves the same throughput using the same technique as APCs and has latency $O(k)$, our construction has significantly smaller latency. If the sum is accumulated by feeding back the output to the carry-propagate adder, the feedback loop requires at least $O(\log k)$ levels of FAs.

4 Unified design of multi-operand adders

Although the basic algorithm introduced in Subsection 2.1 and its pipelined implementations compare favorably with previous approaches theoretically, and usually achieve good performance in practice, the construction of best multi-operand adders depends heavily on the combination of parameters n, k, d and the required throughput, latency, and hardware complexity. As a result, in order to obtain circuits

that fit the need of specific applications, it is desirable to have flexibility to fine-tune the constructions. In this section, we extend the basic algorithm to obtain a unified approach for the design of multi-operand adders and present several efficient special cases.

4.1 Extending the basic scheme

The basic algorithm can be easily extended as follows:

- Phase 1: If the number of operands is not small enough, each column is compressed using a parallel counter.
- Phase 2: The positions are partitioned into $\lceil k'/l' \rceil$ blocks, each containing l' positions, where k' is the maximum length of operands, m' is the number of operands, and $l' = \lceil \log_2(m'-1) \rceil$. The $2l'$-bit sum of each block is computed separately.
- Phase 3: The sums obtained from even and odd blocks are concatenated into S_{even} and S_{odd}, respectively. The sum of S_{even} and S_{odd} is computed using a binary carry-propagate adder.

Using this extended scheme, Phase 1 can be skipped to obtain the block-save-tree (BST) adder, while Phase 2 can apply the scheme recursively to compute the sum of a block. We illustrate the algorithm with several special cases in the following subsections.

4.2 Block-save-tree adders

When the number of operands is not very large, or for some specific combinations of parameters n and k, directly applying Phase 2 can lead to efficient designs.

An efficient design for this special case uses the tree of ripple adders to compute the sum of each block. Since the maximum length of the sums is $O(\log n)$, latency of Phase 2 is $O(\log n)$. Thus, the overall circuit size is $O(nk)$ and the depth is $O(\log n + \log k)$ for (n,k) multi-operand addition. Note that directly applying a single tree of ripple adders to perform the (n,k) multi-operand addition will result in a circuit of size $O(nk)$ and depth $O(\log n + k)$. The improvement offered by our approach is achieved via combining the block-save technique and the tree of ripple adders.

4.3 Generalizing the block-save technique

In this subsection, we generalize the block-save technique to provide further flexible design parameters.

The basic idea of the original block-save technique is to partition the positions into blocks of width small enough so that the sums of even (or odd) blocks will not overlap. We first observe that the positions can be partitioned into more than two classes of blocks. For example, the positions can be partitioned into 3 classes of blocks of width l. If the sum of a block is no more than $3l$, then the sums of the blocks can be concatenated into 3 numbers, which can then be reduced to 2 numbers using $O(k)$ FAs, adding 1 level to circuit depth. Then Phase 3 will generate the correct result. Actually the length of the blocks can be arbitrarily chosen, and the block-save reduction is followed by one or more circuit stages performing further reduction if required. This algorithm can thus be fine-tuned to obtain efficient designs for particular requirements.

BST adders and their variants based on the generalized block-save technique can be pipelined using approaches similar to those used in section 3. All these accumulative adders have size $O(nk/d + k(\log n + \log k))$, latency $O(\log n + \log k + d)$, and a feedback loop with a single full-adder delay. When $d = \Omega(\log n + \log k)$ and $d = O(n/\log n + n/\log k)$, the circuit size is $O(nk/d)$ and latency is $O(d)$.

4.4 Other designs

Phase 2 of the proposed scheme can use any method to perform the multi-operand addition. If we recursively use the scheme to compute the multi-operand sums for Phase 2, the resultant design will be similar to the construction given in [4] in the extreme case, and can terminate the column compression operations earlier in general. We can also use CSA trees followed by a carry-propagate adder for this purpose. Furthermore, either the CSA tree method or the tree of ripple adders scheme can be generalized to use counters rather than FAs to reduce the number of operands.

5 Some applications

Multi-operand adders can be used as basic building blocks for a wide class of important computations. Since our tradeoff method leads to size-efficient designs for multi-operand adders, the cost of such computations can be greatly reduced while keeping the latency small.

5.1 Multiplication

Multiplication of two n-bit numbers can be transformed into $(n, 2n-1)$ multi-operand addition of the n partial products following a depth-1 subcircuit of n^2 AND gates. Thus, the efficient designs of multi-operand adders result in efficient circuits for multiplication.

5.2 Residue computation (or modular reduction)

Reduction of a k-bit number modulo a known r-bit number p can be transformed into $(k-r+1, r)$ multi-operand addition followed by the reduction of the obtained k'-bit sum mod p, where $k' \le r + \lfloor \log_2(k-r+1) \rfloor$. An example is shown in Fig. 3. If the resultant k'-bit number is not small enough, we can perform more iterations of multi-operand addition to further reduce the maximum possible value of the dividend; if it is small enough, we can find the result by subtracting $p, 2p, 3p, \ldots$ from it or with the aid of fast table-lookup schemes [8].

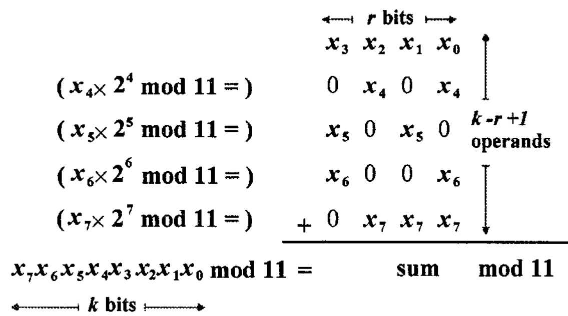

Figure 3. Residue computation based on multi-operand addition.

5.3 RNS-binary conversion

The conversion from binary to RNS representations can be done by several modular reduction circuits in parallel. The conversion from RNS to binary representations can be performed using Chinese remaindering [6], which can be implemented using modulo circuits and multi-operand adders. As a result, we can use RNS arithmetic to compute various functions utilizing multi-operand adders.

5.4 Computation of other useful functions

Many functions, such as exponentiation, division, and multi-operand multiplication, sine, logarithm, etc., can be computed using unbounded fan-in threshold circuits of constant depth, based on RNS arithmetic, multi-operand addition, multiplication, modular reduction, and the computation of an arbitrary Boolean function with $O(\log N)$ inputs [14, 18]. Since all of the above building blocks, except for the last one, can be implemented based on multi-operand addition except for the last one, these functions can be decomposed into several multi-operand adders and some glue logic. Moreover, an unbounded f-fan-in AND/OR gate can be replaced by a $O(\log f)$-depth tree composed of $O(f)$ bounded fan-in AND/OR gates. Using the accumulative technique, a $O(\log f + d)$-latency pipelined circuit with $O(f/d)$ AND/OR gates is required.

In our treatment, the threshold logic used in [14, 18] is essentially viewed as an abstract computation model that facilitates the initial description of the design. After required replacements and minor modifications, $O(d + \log N)$-latency circuits for these problems can be obtained, where N is the number of inputs. The details are omitted.

6 Conclusion

We have introduced a unifying scheme for modular construction of multi-operand adders. The resulting designs achieve high throughput, low latency, small cost, and provide effective tradeoffs among these parameters. We generalized the block-save technique and presented several efficient spacial cases. Application to several important problems was also addressed.

Future research may deal with many issues, including refinement of the constructions, detailed comparison of various approaches to previous work, application to other problems, and optimization of the resultant designs.

References

[1] L. Dadda, "Composite parallel counters," *IEEE Trans. Comput.*, Vol. 29, pp. 940-946, Oct. 1980.

[2] L. Dadda and V. Piuri, "Pipelined adders," *IEEE Trans. Comput.*, Vol. 45, No. 3, pp. 348-356, Mar. 1996.

[3] T.G. Hallin and M.J. Flynn, "Pipelining of arithmetic functions," *IEEE Trans. Comput.*, Vol. C-21, pp. 880-886, Aug. 1972.

[4] I.T. Ho and T.C. Chen, "Iterated addition by residue threshold functions and their representation by array logic," *IEEE Trans. Comput.*, Vol. C-22, pp. 762-767, 1973.

[5] R.F. Jones Jr. and E.E. Swartzlander, Jr., "Parallel counter implementation," *J. VLSI Signal Processing*, pp. 223-232, 1994.

[6] I. Koren, *Computer Arithmetic Algorithms*, Prentice Hall, Englewood Cliffs, New Jersey, 1993.

[7] B. Parhami, "Parallel counters for signed binary signals," *Proc. Asilomar Conf. Signals, Systems, and Computers*, pp. 513-516, 1989.

[8] B. Parhami, "Analysis of tabular methods for modular reduction," *Proc. Asilomar Conf. Signals, Systems, and Computers*, pp. 526-530, Oct./Nov. 1994.

[9] B. Parhami and C.-H. Yeh, "Accumulative parallel counters," *Proc. Asilomar Conf. Signals, Systems, and Computers*, pp. 966-970, Oct. 1995.

[10] S.J. Piestrak, "Design of residue generators and multioperand modular adders using carry-save adders," *IEEE Trans. Comput.*, Vol. 43, pp. 68-77, Jan. 1994.

[11] N. Pippenger, "The complexity of computations by networks," IBM *J. Res. Develop.*, Vol. 31, No. 2, pp. 235-243, Mar. 1987.

[12] S. Singh and R. Waxman, "Multiple operand addition and multiplication," *IEEE Trans. Comput.*, Vol. 22, No. 2, pp. 113-119, Feb. 1973.

[13] K.Y. Siu, V.P. Roychowdhury, and T. Kailath, "Depth-size tradeoffs for neural computation," *IEEE Trans. Comput.*, Vol. 40, No. 12, pp. 1402-1412, Dec. 1991.

[14] K.Y. Siu, J. Bruck, T. Kailath, and T. Hofmeister, "Depth-efficient neural networks for division and related problems," *IEEE Trans. Inform. Theory*, Vol. 39, pp. 946-956, May 1993.

[15] E.E. Swartzlander, Jr., "Parallel counters," *IEEE Trans. Comput.*, Vol. 22, pp. 1021-1024, Nov. 1973.

[16] E.E. Swartzlander, Jr., "Merged arithmetic," *IEEE Trans. Comput.*, Vol. 29, pp. 946-950, Oct. 1980.

[17] E.E. Swartzlander, Jr., *Computer arithmetic*, IEEE Computer Society Press, Los Alamitos, CA, 1990.

[18] C.-H. Yeh and E.A. Varvarigos, "New efficient majority circuits for the computation of some basic arithmetic functions," *J. Computing and Information*, http://phoenix.trentu.ca/jci/, Vol. 2, No. 1, pp. 114-136, 1996.