DESIGN OF A FAST INNER PRODUCT PROCESSOR

S. P. Smith and H. C. Torng

School of Electrical Engineering,
Cornell University,
Ithaca, New York 14853

Abstract

This paper presents the design of a fast inner product processor, with appreciably reduced latency and cost. The inner product processor is implemented with a tree of carry propagate or carry save adders; this tree is obtained with the incorporation of three innovations in the conventional multiply/add tree:

(1) The leaf-multipliers are expanded into adder subtress, thus achieving an O(logNb) latency, where N denotes the number of elements in a vector and b the number of bits in each element.

(2) The partial products, to be summed in producing an inner product, are reordered according to their "minimum alignments", bringing approximately a 20% saving in hardware.

(3) The reordering also truncates the carry propagation chain in the final propagation stage by $2logb - 1$ positions, significantly reducing the latency further.

A form of the Baugh and Wooley algorithm is adopted to implement two's complement notation with changes only in peripheral hardware.

1. Introduction

Very large scale integration offers the possibility of highly dense circuitry and, hence, large systems in small packages. Special-purpose processors for solving large numerical problems are thus made feasible. One such numerical computation having many applications is the inner product. The inner product of two vectors, $\underline{a} = (a_1, a_2, \ldots, a_N)^T$ and $\underline{d} = (d_1, d_2, \ldots, d_N)^T$, is

$$\underline{a}^T \underline{d} = \sum_{i=1}^{N} a_i d_i = a_1 d_1 + a_2 d_2 + \ldots + a_N d_N .$$

This calculation is central to digital filtering and signal processing as well as matrix computations such as matrix multiplication. VLSI offers the opportunity to implement a very fast inner product processor handling large vectors. Such a high-speed hardware implementation would be very useful, for example, in real-time signal processing[34,32,17].

Matrix Multiplication Applications

Matrix multiplication is one application of the inner product for which a number of special-purpose processor designs and algoritms have been proposed[1,11,16,19,22,20]. In the multiplication of NxN matrices, N^2 inner products, each of N-element vectors, must be computed. Each inner product consists of N independent multiplications and $N-1$ additions to sum those products. Systolic arrays[1,16,19,22] and related architectures[11,20] employ linear pipelining of this inner product operation, sequentially accumulating the sum of N products for each result. Properly interweaving N inner product pipelines enables the input bandwidth to be limited to O(N) vector elements per time instant, because the data can be reused within the array. This multiplexing of the hardware among several inner products then establishes good throughput in an efficient way. On the other hand, a disadvantage of this pipelined accumulation is that the time from the start to the finish of one inner product (the "latency") is linear in N. For long vector length N, the delay from the time data are available to the time its inner product is computed is excessive for high speed matrix computation applications, such as real-time signal processing. Speed is sacrificed for the lower I/O bandwidth permitted in sequential accumulation, and, as long as concurrency in the accumulation is not exploited, this latency cannot be decreased.

Previous Work

The problem at hand is to design a fast inner product processor that will take advantage of concurrency and efficiently act as the basic execution unit of a matrix multiplier.

Swartzlander et al. present an interesting design employing the "quasi-serial multiplier" concept[27]. The quasi-serial multiplier computes one multiplication by generating the partial product bits for each bit position, one bit position at a time, and sums them using a parallel counter. The inner product computer uses N quasi-serial multipliers (for N-vector inner products) and sums all the N sets of partial product bits in one parallel counter[28]. The latency, L, in this approach is shown to be approximately

$$L = (2b + \lceil logN \rceil)(2\lfloor logNb \rfloor - 1)t_{fa},$$

where t_{fa} is the delay of a single-bit full adder, b is the number of bits in each vector element, and N is the number of elements per vector. Although latency is logarithmic in N, the bit serial approach introduces a linear factor of b, which is undesirable for minimizing latency. Performance comparisons were made with a binary tree consisting of bit-parallel multipliers for leaves and bit-parallel adders for internal nodes. The performance comparison ([28], Table 1) shows that for b=8 and N=64 and 256 the unpipelined multiply/add binary tree has a much shorter latency than the unpipelined quasi-serial processor. Blankenbaker[5] stated that the throughput of the pipelined quasi-serial processor proposed would actually be much lower than expected. In any case, although the complexity of the tree is admittedly greater than that of the quasi-serial processor, the tree better achieves our objective to minimize latency.

Buric and Mead[6] investigated bit-serial inner product procesors for VLSI implementation, restating the multiply/accumulate linear pipeline in the context of bit-serial arithmetic. A multiply/add binary tree structure is then introduced, consisting of bit-serial multipliers for leaves and single-bit full adders for internal nodes. The latency is logarithmic in N, but again includes a linear factor of b due to bit-serial processing.

Ciminiera and Serra[12] presented another pipeline scheme, consisting of a single multiplier and accumulator pair. The mul-

The research reported in this paper is supported in part by a grant from RCA/Government Systems Division.

tiplication operation is partitioned into an array of two-bit by two-bit multiplication cells. In this array, each vector element pair is multiplied in a pipelined fashion. The inner product is then produced by iterative accumulation of the multiplier array output. Although pipelining increases system throughput, latency is still linear in N.

Problem Definition

We will investigate the design of an inner product processor achieving reduced latency and cost and maintaining throughput at acceptable levels, preferably comparable with previous designs.

More specifically, consider the following definitions:

latency, L = the time from the beginning of the computation of an inner product to the completion of that product;

throughput, r = the number of inner products completed per unit time (steady state), or the inverse of the time between successive completions.

Our objective is to design an inner product processor for N-element vectors (b bits per element) such that (1) L is reduced as much as possible, at least so as not to depend linearly on N or b, (2) the number of adders and multipliers is less in comparison with the hardware requirements of other designs, and (3) r matches the throughputs of previous designs in its dependence on N and b. We will also assume no constraint exists on the availability of operands.

2. *Processor Design*

Tree

One approach to inner product computation is a binary tree. Figure 1 depicts one such tree for N=4 and b=8. Leaves are multipliers, and internal nodes are adders. (The details of the multiplier and adder nodes will be brought out below when necessary.) The tree (with bit-parallel processing) has the advantage of $O(logN)$ latency, and with pipelining can achieve high throughput.

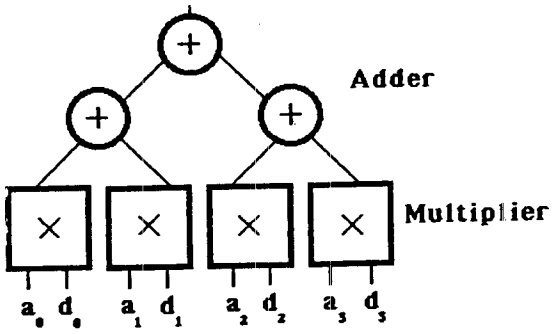

Figure 1. A multiply/add binary tree for N=4 and b=8.

In matrix multiplication, the tree has a number of advantages over the various array schemes mentioned above. N such tree processors operating in parallel can generate successively the rows (or columns) of the product of two NxN matrices. High throughput can be attained by pipelining, implemented by the addition of registers between levels of each tree. N pipelined trees can produce these matrix products at a throughput of one product every N phases of the pipe. This throughput would therefore match the throughput of square array designs previously suggested[11,20] as well as the hexagonal systolic array of Kung and Leiserson[16].

The hardware required for this set of trees is N^2 multipliers and N^2-N adders. In comparison, an NxN square array of multiplier-accumulators requires N^2 multipliers and N^2 adders[11,20]. The hexagonal array[16] requires for dense matrices $3N^2-3N+1$ multipliers and $3N^2-3N+1$ adders[26] - more hardware than the tree requires.

On the other hand, there are drawbacks to the tree scheme. The circuit layout of a tree is not as regular as an orthogonal or hexagonal array, and the interconnections between nodes may have to vary in length and, hence, in delay. The longest of these interconnections may significantly increase latency and, if there is pipelining, clock cycle time. The number and complexity of these interconnections may also demand large chip area, radically altering the above hardware comparisons. In addition, since each tree takes 2N operands, a matrix multiplier of N pipelined trees must be fed $2N^2$ operands every phase of the pipe. The square and hexagonal arrays, however, require only O(N) I/O bandwidth.

We will focus our investigation on modifying the tree structure to achieve significant savings in hardware and reductions in latency.

Modifications

Regrouping Partial Products

We will treat the multiplication of two vector elements as a summation of partial products and expand the multiplier leaves into subtrees of adders that sum these partial products. Figure 2 illustrates this expansion of the tree for N=4 and b=8. Substituting adder subtrees for multiplier leaves gives O(logb) delay for the multiplication, since each subtree has logb levels. The total latency for an inner product then becomes

$$L = (logN + logb)t_{add},$$

where t_{add} is the delay incurred at each level for addition, signal propagation, and (if pipelining is employed) register transfers. (We will henceforth assume that N and b are integral powers of two.)

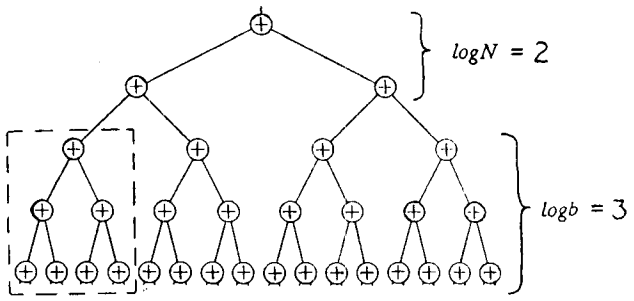

Figure 2. The multiply/add tree for N=4 and b=8 with multipliers expanded into adder subtrees. The box encloses one multiplier.

The key advantage in expanding the multiplier leaves is that *we now have access to the partial products before they enter the tree.* We can then rearrange the partial products, as we will discuss below, so as to reduce overall latency as well as hardware. In order to clearly describe these rearrangements and the resulting advantages, we must introduce some terminology.

We define the *alignment* of a binary number as the number of right-hand zeros it has, expressed in units of bits. Thus, 101100000 has an alignment of five (or five bits), and 110100 an alignment of two (or two bits). We will say that numbers having the same number of right-hand zeros are of equal alignment. In the discussion that follows, many of the numbers under consideration have a predetermined number of

right-hand zeros introduced independently of the values of the remaining bits in the higher order bit positions. For example, we may deal with some unspecified four-bit number. If that number is shifted left three bits, we know that whatever the specific number is, it has an alignment of at least three. Therefore, we define the number of right-hand zeros that a number is forced to have independent of the specific data as the *minimum alignment* (a_{min}) of the number. In our example, consequently, the shifted number has $a_{min} = 3$.

With these definitions, we see that we originally (Fig. 2) expanded a multiplier leaf into a subtree of adders summing partial products of staggered minimum alignments. We now list all the partial products involved in an inner product and regroup them in sets of equal minimum alignment. Figure 3 gives an example of this regrouping for N=4 and b=8. These new collections of partial products are then supplied to the tree. We redefine a *subtree* to include only enough adders to sum the N (=4) partial products of equal minimum alignment, instead of the b (=8) partial products of staggered alignment (generated in one multiplication). Notice that none of the a_{min} right-hand zeros for any partial product need to be included in the summations computed in the subtrees, since the sum of numbers of equal minimum alignment has the same minimum alignment. All the subtrees can therefore receive b-bit numbers (b=8 in our running example), by excluding the a_{min} insignificant right-hand zeros, thus maintaining all subtrees identical.

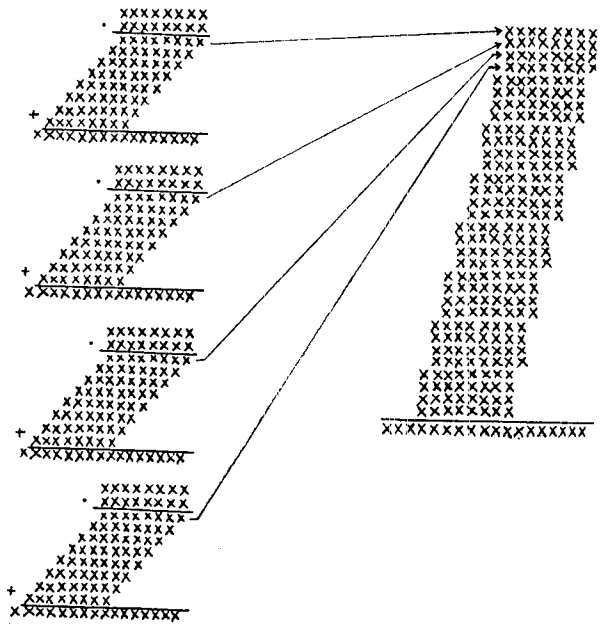

Figure 3. Rearranging partial products into groups of equal minimum alignment for N=4 and b=8. Each group will feed into separate subtrees (unnecessary righthand zeros omitted) before combining with numbers of different minimum alignment.

The modified tree structure for N=4 and b=8 is shown in Figure 4. (Logic to generate the partial products is omitted.) The minimum alignments associated with partial products input into the tree increase from zero at the left-most subtree (enclosed in a box) to seven at the right-most subtree. Once each group of partial products of equal minimum alignment is summed in its respective subtree, the remainder of the additions in the tree must accomodate unequal minimum alignments among the subtree outputs. For example, the output of the subtree for $a_{min} = 0$ (subtree 0) is added to that for $a_{min} = 1$ (subtree 1). Since we have already shown that only the uppermost b-bits of the partial products must be fed into the tree, the output of subtree 1 does not reflect its true minimum alignment, but appears to have $a_{min} = 0$. To compensate, we must shift the bits output by subtree 1 left one position. This shift is simply accomplished in the placement of wires and is signified by $\overleftarrow{1}$ associated with the shifted data path. The outputs of subtrees 0 and 1 can then be correctly added.

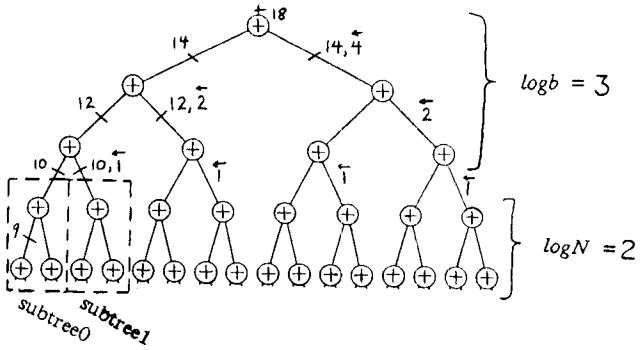

Figure 4. Expanded tree for N=4 and b=8 taking reordered partial products.

Since, throughout the rest of the tree, addends are combined that are nearest to each other in minimum alignment, only the smallest shifts necessary are made. As a result, at each level requiring shifting all shifts identical, as shown in Figure 4. In addition, the shifts double at each higher level, because the difference between the minimum alignments of addends to be combined increases by a factor of two with each successive level.

Carry-Save Adder Implementation

Another possible alteration to the tree will be the substitution of carry-save adders for carry-propagate adders. Carry propagation has hitherto occurred in every addition in our tree, introducing carry propagation delay at every level. A tree of carry-save adders (CSA's) will postpone all carry propagation until the very last stage, thus reducing overall latency[31,15,10].

Since CSA's are three-input, two-output adders, they do not inherently produce binary trees. We can, however, maintain a binary tree structure by applying the following mapping. Combine pairs of adjacent carry-propagate adders having the same parent in the original tree. Notice that such a pair can be considered as a new type of adder node, having four inputs and two outputs. The resulting tree of such nodes is also a binary tree. We will now implement each adder node with two CSA's. The mappings from carry-propagate adders to adder nodes with CSA's is shown in Figure 5. An *adder node* (or *node*) is then defined by the mapping in Figure 5 as a four-input, two-output adder circuit composed of two CSA's connected in sequence.

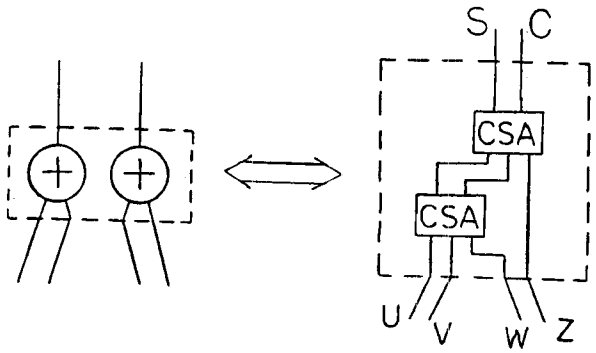

Figure 5. Mapping from carry-propagate adders(left) to adder node consisting of two CSA's in series.

Lastly, we define a *type I* tree as one in which partial products of staggered minimum alignment (associated with one multiplication) are grouped together and a *type II* tree as one in which partial products of equal minimum alignment are grouped.

3. Design Evaluation

Hardware

To explain clearly how a saving in hardware arises, we will first consider trees I and II as implemented with carry-propagate adders. The carry-propagate versions of trees I and II, along with the widths of all data paths, are shown in Figure 6. Notice that data paths are narrower in the type II tree. In the type I tree, the number of data paths we must add to the initial b-bit width at first increases rapidly with the number of levels: b=8 bits input to the leaves grow to 2b=16 bits, an increase of b=8 bits in $logb + 1 = 4$ levels. Further additions to path widths grow linearly with the number of levels: $2b$=16 bits expand to $2b + logN = 18$ bits, an increase of $logN = 2$ bits in $logN = 2$ levels. In the type II tree, on the other hand, the extension to path widths that we must add to the initial b-bit width increases at first linearly with the number of levels: b=8 bits expand to $b + logN = 10$ bits, an increase of $logN = 2$ bits in $logN = 2$ levels. Further additions to path widths then grow exponentially with the number of levels: $b + logN = 10$ bits expand to $2b + logN = 18$ bits, an increase of b=8 bits in $logb + 1 = 4$ levels. The final path width is the same, but *the regrouping of partial products postpones the fast growth of the path widths.* Hence, adders handle fewer bits, reducing hardware cost, simply because we reordered these partial products.

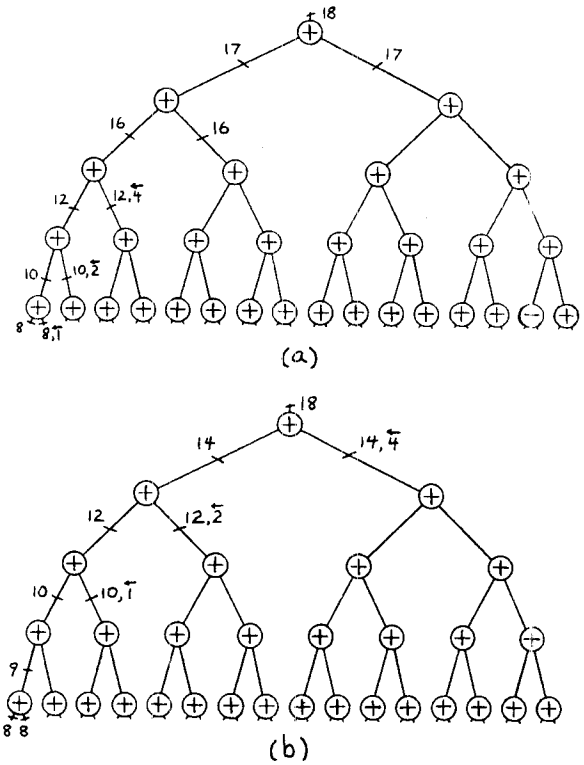

Figure 6. A comparison of data path widths in trees consisting of carry-propagate adders (N=4 and b=8). (a) The standard multiply/add tree (I). (b) The altered tree (II).

The total number of carry-propagate adder bits necessary in tree I[26] is

$$\beta_I = Nb(b + logb + 1) + 2(N - b - 1) + logN .$$

The total number of carry-propagate adder bits in tree II[26] is

$$\beta_{II} = Nb(b + 2) + (2b - 1)logN + b(logb - 3) .$$

Values for β_I, β_{II}, and the hardware gain, $\frac{\beta_{II}}{\beta_I}$, are shown in Table 1 for b = 8, 16, and 32, and N = 8, 16, 32, 64, and 128.

Table 1: β_{II} / β_I for carry-propagate adder trees.

N	b				
	4	8	16	32	64
256	.805	.821	.855	.895	.930
128	.809	.824	.858	.897	.931
64	.817	.831	.862	.899	.933
32	.829	.841	.869	.904	.936
16	.850	.858	.881	.912	.940
8	.884	.886	.901	.925	.948
4	.941	.927	.930	.944	.960

Similarly, Table 2 shows values for β_I, β_{II}, and the hardware gain, $\frac{\beta_{II}}{\beta_I}$, for trees I and II implemented with adder nodes according to the mapping of Figure 5. These two trees and their path widths are shown in Figure 7. The values in Table 2 were obtained numerically by a program that traversed the tree of adder nodes and computed the sum of CSA output path widths. This hardware improvement has come without any sacrifice in speed.

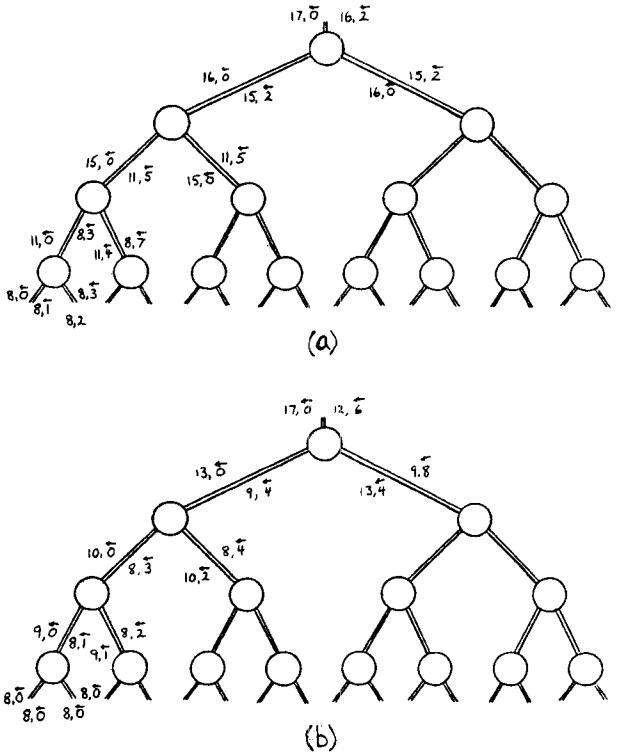

Figure 7. A comparison of data path widths in trees consisting of adder nodes (N=4 and b=8). (a) The standard multiply/add tree (I). (b) The altered tree (II).

Table 2: β_{II} / β_I for carry-save adder trees.

N	b				
	4	8	16	32	64
256	.778	.763	.796	.844	.891
128	.778	.764	.797	.845	.892
64	.778	.765	.798	.846	.893
32	.778	.767	.800	.848	.894
16	.779	.771	.805	.852	.897
8	.784	.780	.815	.861	.903
4	.795	.801	.836	.878	.916

Regarding other hardware concerns, circuit layout becomes no worse here than for the tree before rearranging partial products, and certain patterns in trees emerge that simplify design. From Figure 7 it can be seen that both approaches present similar patterns in their tree layouts. Note that every node acts as a root to a tree within the entire tree structure. To avoid confusion with the previous discussion concerning subtrees, we will term these trees *inner trees*. Observe now that all inner trees having roots at the same level are identical. Thus, in Figure 7, the first level has sixteen identical leaves in either approach, the second level has eight nodes all roots of identical trees, the third level four such root nodes of identical trees, and so on. So only $logNb - 1 = logN + logb - 1$ unique nodes (one for each level) need to be designed for either approach, the remaining nodes being merely duplicates of these. These patterns should be exploited to simplify circuit layout.

Latency

The latency of the CSA tree (types I and II) is $L = 2(logN + logb - 1)t_{CSA}$, where t_{CSA} is the delay for signal propagation from one CSA to another, including any latching delay. Now, however, we must add the delay from a carry-lookahead adder (CLA)[15,10,31] needed to add the final sum and carry terms of the last CSA. The delay of the CLA may form a bottleneck on the system, because carry propagation does not occur except in the CLA. By reordering the partial products by equal minimum alignment, however, we can truncate the chain of carry propagation computed in the CLA and therefore reduce overall latency.

To reduce carry propagation in the CLA, *we increase as much as possible the number of right-hand zeros in one addend*, because carry propagation will not begin until the bit position where both operands are non-zero. By guaranteeing that the minimum alignment of one addend is large relative to the other (the number of bits of the final sum remaining unchanged) we ensure that the number of bit positions through which a carry may propagate is reduced. By the standard multiply/add tree (tree I), the final C and S addends (the two CLA inputs) have minimum alignments equal to one and zero, respectively, yielding the maximum propagation possible. The above reordering of the partial products, however, will minimize this propagation. The largest minimum alignment of the final C (C_{final}) is derived and found to be 2logb[26]. (The final S must have $a_{min} = 0$.) By rearranging partial products, therefore, we increase $a_{min}(C_{final})$ from one in tree I to 2logb in tree II. For b=8 and N=4, the final result has $2b + logN = 18$ bits, and $2logb = 6$, so the carry propagation chain drops 29.4 percent from 17 bits (tree I) to 12 bits (tree II).

In short, merely reordering data both cuts down latency by reducing carry propagation in the CLA and achieves this improvement with a reduction in hardware.

4. Two's Complement Implementation

With minor alterations, the processor described above can accept data in two's complement notation. We have applied the scheme presented by Baugh and Wooley[9] and, in particular, have employed an intermediate form of their algorithm described as follows.

For simplicity, we will consider the case where N=1 and b=4, that is, the multiplication of two four-bit numbers. Let $x = x_3x_2x_1x_0 = -x_3 2^3 + x_2 2^2 + x_1 2^1 + x_0 2^0$ (two's complement notation). Let y be similarly defined. The product, xy, may then be written as an array of partial product bits:

$$
\begin{array}{ccccccc}
 & & & -y_3x_0 & y_2x_0 & y_1x_0 & y_0x_0 \\
 & & -y_3x_1 & y_2x_1 & y_1x_1 & y_0x_1 & \\
 & -y_3x_2 & y_2x_2 & y_1x_2 & y_0x_2 & & \\
+y_3x_3 & -y_2x_3 & -y_1x_3 & -y_0x_3 & & &
\end{array}
$$

From the three columns that contain negative bits we may derive two 3-bit negative numbers: $-y_3x_2 - y_3x_1 - y_3x_0$ and $-y_2x_3 - y_1x_3 - y_0x_3$. To convert these to two's complement notation, we complement and add one:

$$
\begin{array}{cccccc}
1 & 1 & \overline{y_3x_2} & \overline{y_3x_1} & \overline{y_3x_0} & \\
 & & & & 1 & \\
1 & 1 & \overline{y_2x_3} & \overline{y_1x_3} & \overline{y_0x_3} & \\
 & & & & 1 &
\end{array}
$$

The 1's in the upper bit positions constitute sign extensions necessary to add these two's complement numbers into the total sum correctly. The two's complement equivalent of the above array of partial products is therefore given below:

$$
\begin{array}{cccccccc}
 & & & \overline{} & \overline{y_3x_0} & y_2x_0 & y_1x_0 & y_0x_0 \\
 & & \overline{} & \overline{y_3x_1} & y_2x_1 & y_1x_1 & y_0x_1 & \\
 & & \underline{y_3x_2} & \underline{y_2x_2} & \underline{y_1x_2} & y_0x_2 & & \\
 & y_3x_3 & y_2x_3 & y_1x_3 & y_0x_3 & & & \\
1 & 1 & & & 1 & & & \\
1 & 1 & & & 1 & & &
\end{array}
$$

For cases with $N > 1$, we can look upon each row as representing N rows of partial product bits of the same form.

This last array of partial product bits is the intermediate form of the Baugh and Wooley algorithm that we will employ. We will generate this array and sum the bits using the tree. As a result of this scheme, the adder node tree of Figure 7(b) for non-negative numbers can serve without alteration to sum these bits.

Latency will increase somewhat, because of the introduction of extra 1's to be added. The value of these 1's summed together (N and b being powers of two) is

$$\alpha = 2^{2b-1+logN} + 2^{2b-2+logN} + 2^{b-1+logN} .$$

This constant can be added to the output of the CLA stage by an additional CLA stage. In this case, although throughput can remain unchanged via pipelining, latency will increase by the delay incurred in the new CLA.

Furthermore, NAND gates must be introduced into the partial product generation logic. Whereas for only positive numbers, Nb^2 AND gates were employed, now $2N(b-1)$ NAND and $N(b-1)^2$ AND gates are required. This change is minor, and although it affects the design process, no new delay is added.

5. Conclusion

Our objective has been to design a fast inner product processor with reduced latency and reduced cost, while maintaining throughput comparable with other reported structures. We achieve the stated objective by introducing three modifications to the conventional multiply/add tree:

(1) The leaf-multipliers are expanded into adder subtrees and thus achieving a resulting O(logNb) latency;

(2) The partial products are reordered according to "minimum alignment"; this reordering brings about approximately a 20% saving in hardware;

(3) The partial products are also reordered to truncate the carry propagation chain in the final carry propagation stage by 2logb - 1 positions, further reducing latency.

A simple adjustment is identified to accommodate operands expressed in two's complement notation.

Acknowledgment

We thank Dr. D. Gandolfo and Dr. T. Martin for their assistance and encouragement.

References

[1] G. Alia, "VLSI Systolic Arrays for Band Matrix Multiplication", *Integration*, Vol. 1, No's. 2 & 3, Oct. 1983, pp. 233-250.

[2] D. Agrawal and T. Rao, "On Multiple Operand Addition of Signed Binary Numbers", *IEEE Trans. Computers*, C-27, No. 11, Nov. 1978, pp. 1068-1070.

[3] W.C. Booth, "Approaches to Radar Signal Processing", *Computer*, Vol. 16, June 1983, pp. 32-42.

[4] G. Bilardi, M. Pracchi, and F. Preparata, "A Critique of Network Speed in VLSI Models of Computation", *IEEE J. Solid-State Circuits*, SC-17, No. 4, Aug. 1982, pp. 696-702.

[5] John V. Blankenbaker, "Comments on 'Inner Product Computers'", *IEEE Trans. Computers*, C-28, No. 12, Dec. 1979, p. 944.

[6] Misha R. Buric and Carver A. Mead, "Bit-Serial Inner Product Processors in VLSI", *Caltech Conference on VLSI*, Jan. 1981, pp. 155-164.

[7] S.A. Browning, "Computations on a Tree of Processors", *Caltech Conference on VLSI*, Jan. 1979. pp. 453-478.

[8] K. Bromley, J.J. Symanski, J.M. Speiser, and H.J. Whitehouse, "Systolic Array Processor Developments", in *VLSI Systems and Computations* (cited above), 1981, pp. 273-284.

[9] C.R. Baugh and B.A. Wooley, "A Two's Complement Parallel Array Multiplication Algorithm", *IEEE Trans. Computers*, C-22, No. 12, Dec. 1973, pp. 1045-1047.

[10] J.J.F. Cavanaugh, *Digital Computer Arithmetic: Design and Implementation*, New York: McGraw-Hill Book Co., 1984.

[11] M. Chen and T. Murata, "Efficient Matrix Multiplications on a Concurrent Data-Loading Array Processor", *Proc. IEEE Int. Conf. Parallel Processing*, 1983, pp. 90-94.

[12] L.Ciminiera and A. Serra, "Arithmetic Array for Fast Inner Product Evaluation", *Proc. IEEE 5th Symposium Computer Arithmetic*, 1981, pp. 207-214.

[13] L. Dadda, "Some Schemes for Parallel Multipliers", *Alta Frequenza*, Vol. 34, March 1965, pp. 349-356.

[14] D. Gordon, I. Koren, and G. Silberman, "Embedding Tree Structures in VLSI Hexagonal Arrays", *IEEE Trans. Computers*, C-33, No. 1, Jan. 1984, pp. 104-107.

[15] K. Hwang, *Computer Arithmetic: Principles, Architecture, and Design*, New York: John Wiley and Sons, 1979.

[16] H.T. Kung and C.E. Leiserson, "Algorithms for VLSI Processor Arrays", *Introduction to VLSI Systems*, C. Mead and L. Conway, Reading, MA: Addison-Wesley, 1980, pp. 271-291.

[17] H.T. Kung, "Special-Purpose Devices for Signal and Image Processing: An Opportunity in Very Large Scale Integration (VLSI)", *SPIE Real-Time Signal Processing III*, Vol. 241, 1980, pp. 76-84.

[18] H.T. Kung, L.M. Ruane, and D. Yen, "A Two-Level Pipelined Systolic Array for Convolutions", in *VLSI Systems and Computations*, H.T. Kung, B. Sproul, and G. Steele, eds., Rockville, Maryland: Computer Science Press, 1981, pp. 119-123.

[19] H.T. Kung, "Why Systolic Architectures?", *Computer*, Vol. 15, No. 1, Jan. 1982, pp. 37-45.

[20] S.Y. Kung, K. Arun, R. Gal-Ezer, and D. Rao, "Wavefront Array Processor: Language, Architecture, and Applications", *IEEE Trans. Computers*, C-31, No. 11, Nov. 1982, pp. 1054-1066.

[21] C.E. Leiserson, *Area-Efficient VLSI Compuation* (PhD dissertation), Cambridge, MA: The MIT Press, 1983.

[22] J.V. McCanny and J.G. McWhirter, *Bit-Level Systolic Array Circuit for Matrix Vector Multiplication*, IEE Proceedings, Vol. 130, Pt. G, No. 4, August 1983, pp. 125-130.

[23] C. Mead and M. Rem, "Minimum Propagation Delays in VLSI", *IEEE J. Solid-State Circuits*, SC-17, No. 4, Aug. 1982, pp. 773-775.

[24] J.G. Nash, S. Hausen, and G.R. Nudd, "VLSI Processor Arrays for Matrix Manipulation", in *VLSI Systems and Computations*, H.T. Kung, B. Sproul, and G. Steele, eds., Rockville, Maryland: Computer Science Press, 1981, pp. 367-378.

[25] W. Ruzzo and L. Snyder, "Minimum Edge Length Planar Embeddings of Trees", in *VLSI Systems and Computations* H.T. Kung, B. Sproul, and G. Steele, eds., Rockville, Maryland: Computer Science Press, 1981, pp. 119-123.

[26] S.P. Smith and H.C. Torng, "Design of a Fast Inner Product Processor", Technical Report EE-CEG-84-5, School of Electrical Engineering, Cornell University, Ithaca, NY, Oct. 1984.

[27] E.E. Swartzlander, Jr., "The Quasi-Serial Multiplier", *IEEE Trans. Computers*, C-22, No. 4, April 1973, pp. 317-321.

[28] Earl E. Swartzlander, Jr., Barry K. Gilbert, and Irving S. Reed, "Inner Product Computers", *IEEE Trans. Computers*, C-27, 1, Jan. 1978, pp. 21-31.

[29] S. Singh and R. Waxman, "Multiple Operand Addition and Multiplication", *IEEE Trans. Computers*, C-22, No. 2, Feb. 1973, pp. 113-120.

[30] J. Vanaken and G. Zick, "The X-Pipe: A Pipeline for Expression Trees", *Proc. 1978 Int. Conf. Parallel Processing*, pp. 238-245.

[31] C.S Wallace, "A Suggestion for a Fast Multiplier", *IEEE Trans. Electronic Computers*, EC-13, Feb. 1964, pp. 14-17.

[32] S. Waser, "High-Speed Monolithic Multipliers for Real-Time Digital Signal Processing", *Computer*, Vol. 11, No. 10, Oct. 1978, pp. 19-29.

[33] S. Waser and M.J. Flynn, *Introduction to Arithmetic fro Digital Systems Designers*, New York: Holt, Rinehart and Winston, 1982.

[34] D.M. Wilcox and R.J. Nichols, "Optical Sensor Signal Processing Requirements for Ballistic Missile Defense", *SPIE Real-Time Signal Processing III*, Vol. 241, 1980, pp. 2-10.