N94-18349

An Improved Distributed Arithmetic Architecture

X. Guo and D. W. Lynn
NASA Space Engineering Research Center for VLSI System Design
University of Idaho
Moscow, Idaho 83843

Abstract- **Speed requirements have been, and will continue to be a major consideration in the design of hardware to implement digital signal processing functions like digital filters and transforms like the DFT and DCT. The conventional approach is to increase speed by adding hardware and increasing chip area. The real challenge is to save chip area while still maintaining high speed performance. The approach we propose is based on the distributed arithmetic implementation (DA) of digital filters. The improvement is based on two observations. Firstly, a single memory element can replace several identical memory elements in a fully parallel DA implementation. Secondly, truncation or rounding may be introduced into the computation at strategic points without increasing error unduly. Both of these approaches can be used to attain area savings without impairing speed of operation.**

1 Introduction

Finding the inner product between two vectors is an operation that commonly arises in signal processing as well as in general data processing. Digital convolution and correlation are directly described as inner products. Other operations such as the discrete Fourier transform and other common transforms can be implemented as a sequence of inner products. Consider the inner product

$$y = \sum_{k=1}^{K} A_k x_k \tag{1}$$

In the case of a FIR digital filter, A_k represents a set of fixed weights, and x_k represents the current and past $K-1$ filter inputs. The inner product can be implemented directly by using a single multiplier and an accumulator in a serial one product at a time manner, as in Figure 1, or in a fully parallel manner by using K multipliers and a multi-input adder or adder tree, as in Figure 2. Obviously, the fully parallel architecture will always be faster than the serial approach.

The distributed arithmetic (DA) approach to computing the inner product was developed in the early seventies [1,2,3,4,5,6,7,8]. In this approach, combinations of the A_k are precomputed and stored in memory. Input data are used to identify which memory words are to be fetched, shifted and added to produce the final result. Without loss of generality,

This research was supported in part by NASA under grant NAGW-1406.

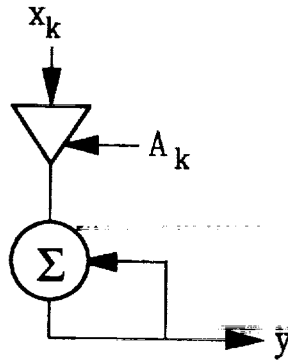

Figure 1: Multiplier-Accumulator Implementation

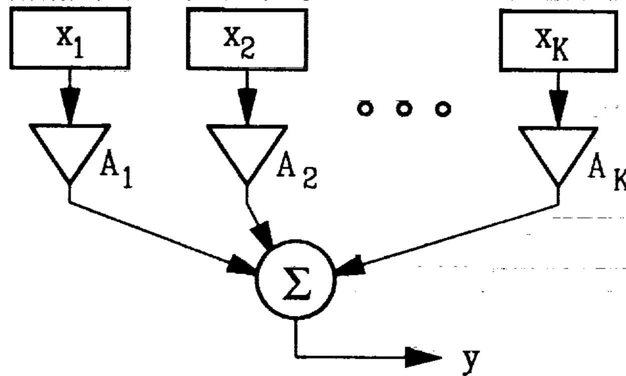

Figure 2: Fully Parallel Direct Implementation

first assume that the x_k are scaled such that $|x_k| < 1$. In two's complement form

$$x_k = -b_{k0} + \sum_{m=1}^{M-1} b_{km} 2^{-m} \tag{2}$$

where the b_{km} represent the individual bits in x_k with b_{k0} the sign bit. Substituting (2) into (1) and rearranging the order of summation gives

$$y = \sum_{m=1}^{M-1} \left\{ \sum_{k=1}^{K} A_k b_{km} \right\} 2^{-m} - \sum_{k=1}^{K} A_k b_{k0} \tag{3}$$

Since the bits b_{km} are either 0 or 1, the term $\sum_{k=1}^{K} A_k b_{km}$ can be precomputed for all 2^K possible combinations of b_{km}. These values are then stored in a ROM or RAM. The actual combinations of b_{km}, arising out of the input data, are used to address one of the precomputed terms stored in the memory. Note that these combinations are formed by selecting the mth bit from each of the K M-bit input words. The mth term so addressed is then shifted by m bits to the right before being added to the other M terms. The only exception to this is when $m = 0$. In this case, which corresponds to using the sign bits of the input data to form the address, the addressed term is subtracted from the other terms.

As with the direct implementation of the inner product, there are two approaches to implementing DA. The inner product can be computed by using only one memory and a single accumulator as shown in Figure 3, or in a fully parallel manner by using M memories,

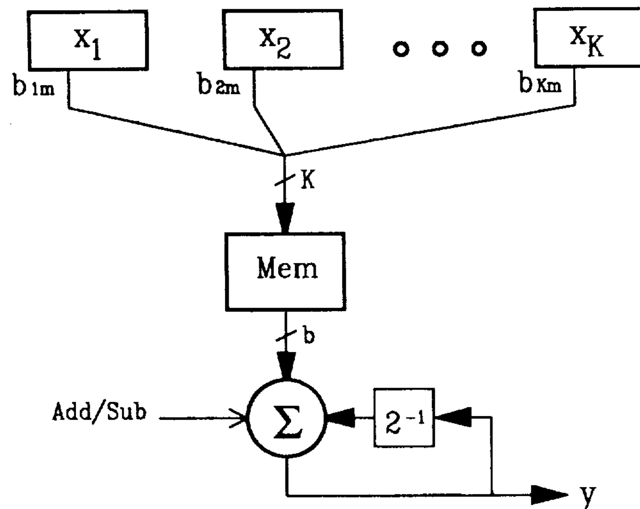

Figure 3: Single Memory DA Implementation

each with identical contents, as shown in Figure 4. Again, the fully parallel approach will always be the fastest. In Figure 4, we note that the shifting is actually accomplished by connecting the memory outputs to appropriate positions on a multi-input adder.

Comparing Figures 4 and 2 is instructive. We note that where the input data words are M bits wide, M memories are always used in the DA implementation, independent of K, the number of multiplies in the inner product. However, each memory must store 2^K terms. So, increasing K will increase the required size of the memories. Also, as K increases, the number of stored bits per term must increase in order to maintain accuracy. The direct implementation, by comparison, uses K multipliers. As M increases, the width and depth[1] of the multipliers must increase to preserve accuracy. Thus, depending on the word size, number of products, and required accuracy, one approach may have size advantages over the other.

In terms of speed, DA does have one clear advantage over the direct implementation. Increasing the accuracy of the inner product by increasing the number of bits in the input data words *and* in the coefficients will not degrade the speed performance of a DA implementation. The number of memories and the width of each will increase, but the number of stored terms in each memory will not. In a direct implementation, however, not only the width of the multipliers increase, but so will their depth resulting in slower performance. Increasing K does not decrease the speed of the multipliers, but it will increase the depth of the adder tree in the direct implementation resulting in some loss of performance. In a DA implementation, increasing K will slow down the memories, but it does not increase the depth of the adder tree.

While the structure of the fully parallel DA implementation is very regular and hence attractive for VLSI implementation, it appears to be very inefficient in terms of its use of space. That is, for each inner product computed, only one of the 2^K terms stored in each memory is used. Further, the contents of each of the M memories is *identical.* Our

[1]This assumes that the width of the coefficients also increases proportional to M.

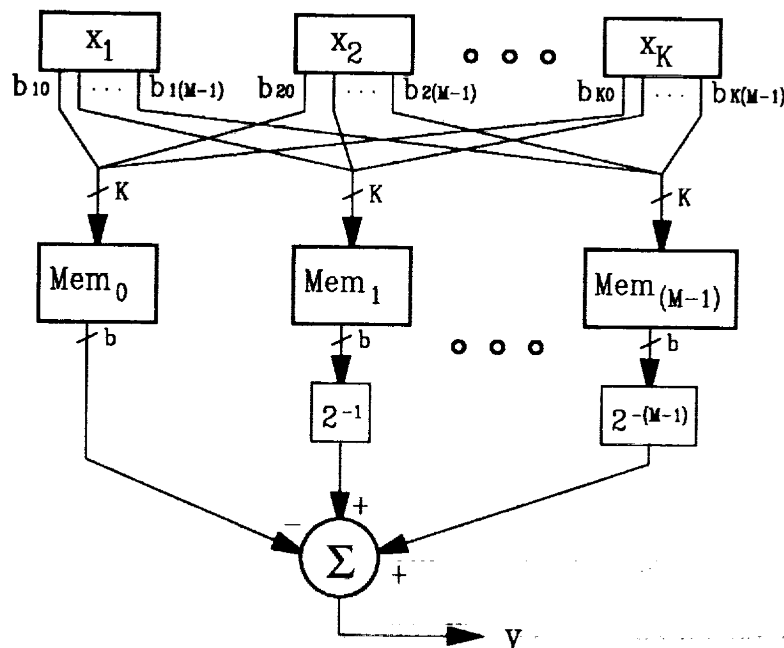

Figure 4: Fully Parallel DA Implementation

first observation about how the fully parallel DA architecture may be improved involves replacing the M memories with just one memory unit that provides M data access paths.

2 An Improved DA Architecture

A ROM, using one transistor per storage bit, is shown in Figure 5. The stored bits are zero or one depending on whether the drain of the associated transistor is connected to the data line or not. Note that the address decoder and the data line sense amplifiers are not shown. Not counting these components, the number of transistors required for the memories in a ROM based fully parallel DA implementation is

$$n_t = M(b2^K + 2^K + b) \tag{4}$$

where b represents the number of bits stored in each word of the memory. Next, consider Figure 6 which represents one plane of a M-way multi-access memory. Each plane stores one word and 2^K planes together make up the complete memory unit as shown in Figure 7. Each plane has M sets of b control transistors that are used to route the stored word to the appropriate output register. Each set of b control transistors is controlled by a single control line. The data bits associated with control line m in each plane are connected to a bus which connects with output register m. Which of the 2^K control lines is asserted is determined by address decoder m. Since this circuit effectively addresses the output registers instead of the stored words, there is no need for address lines for the stored words themselves. Further, a transistor is not required for each stored bit. A zero is stored simply with a shorted line, a one with an open. The control transistors assume the function of the

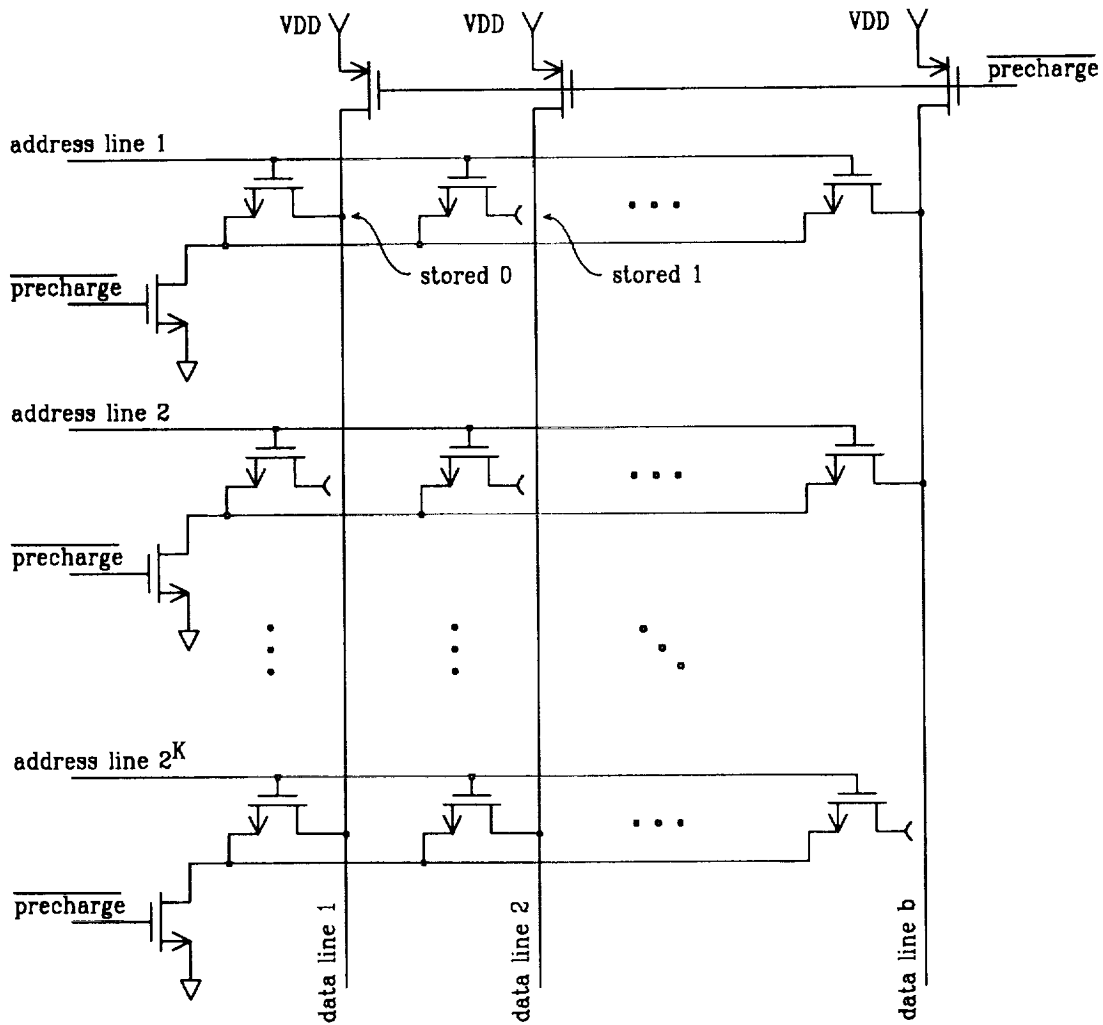

Figure 5: ROM Architecture

storage transistors in the ROM architecture, and provide a path between every stored word in the memory unit and every output register. The total number of transistors required to implement the memory unit, again excluding address decoders and output registers is

$$n_t = Mb2^K + b2^K + 2^K \tag{5}$$

Note that both approaches use M K to 2^K decoders and identically sized adder trees. The ratio of the number of transistors in the storage sections of the Multi-Access memory unit and the memories in a fully parallel DA architecture give an estimate of the area savings potential presented by one approach over the other. Dividing (5) by (4) gives

$$R_{area} = \frac{Mb2^K + b2^K + 2^K}{Mb2^K + M2^K + Mb} = \frac{Mb2^K + (b+1)2^K}{Mb(2^k + 1) + M2^K} \tag{6}$$

Since b will usually be greater than M, $R_{area} > 1$. That is, the multi-access memory architecture presents no area savings, despite the fact it replaces M copies of each stored

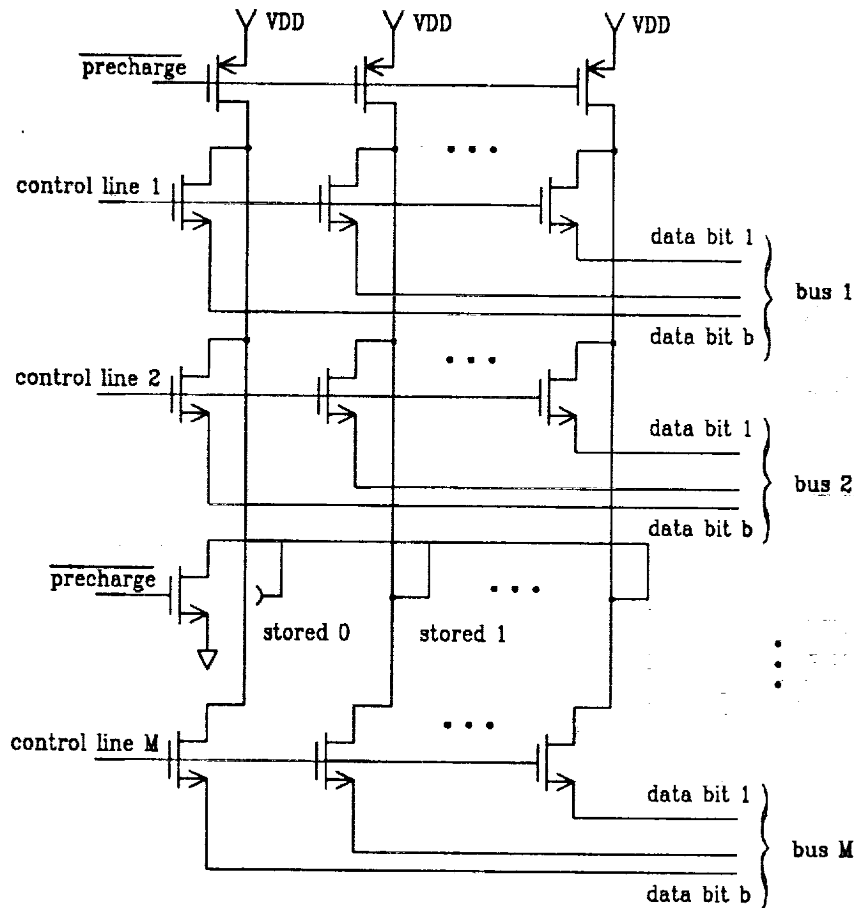

Figure 6: Memory Unit Plane

word with just one copy. This is because the expense of bus controls erases the savings of memory transistors. In fact the number of transistors associated with stored bits in the fully parallel implementation is $Mb2^K$. This is also the number of bus control transistors in the multi-access memory. However, for memory architectures where there is more than one transistor per stored bit, the savings in storage transistors will not be absorbed by bus control transistors. To see this, consider the case that 4 transistor static RAM cells are used as memory elements. Static RAM may be required in cases where the inner product is to be configurable, in the sense that the coefficients may be changed from time to time, requiring the memory contents to be rewritten. The conventional static RAM architecture is shown in Figure 8 and one plane of the multi-access memory architecture is shown in Figure 9.

A fully parallel implementation of DA using M static RAMs of the type shown in Figure 8 would use $4Mb2^K$ transistors and $Mb2^K$ cell select transistors. The static RAM multi-access memory unit would use $4b2^K$ transistors for storage and $Mb2^K$ bus control transistors. Thus, when static RAM cells are used,

$$R_{area} = \frac{(M+4)b2^K}{5Mb2^K} = \frac{M+4}{5M} \tag{7}$$

Here there will be an area savings so long as M (the number of bits in the input data words and the number of memory units in the fully parallel DA implementation) is greater then 1.

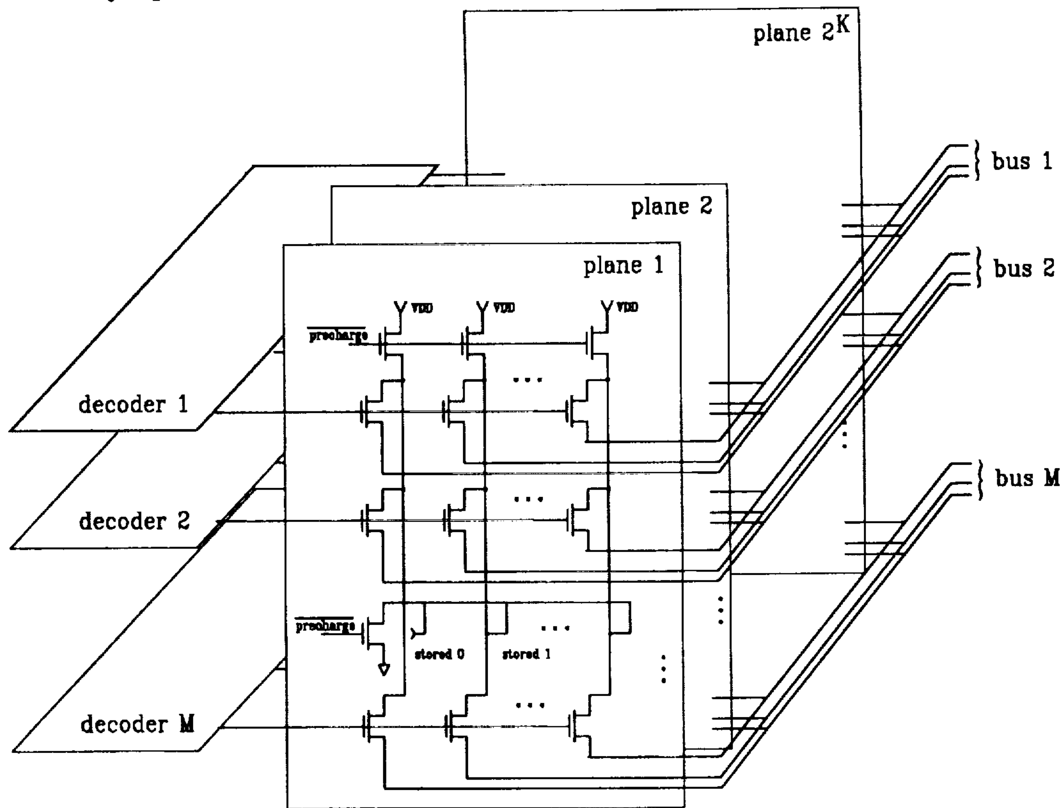

Figure 7: Multi-Access Memory Unit

If $M = 8$, $R_{area} = .30$. Thus the area savings can be significant. These observations have been made in the context that the number of transistors corresponds to area requirements. The same results hold whenever the area required for the storage cell (be it transistor based or not) requires more area to implement than do bus control transistors. Again, it must be noted that while the area required for address decoders and the adder tree are the same for both implementations, these requirements are not included in the R_{area} computation. So, R_{area} only reflects the savings potential in the storage section of the implementation. To the degree that the storage section dominates the other elements of the implementation this may translate into significant savings. Not only do the other elements need to be included in the computations, but an actual VLSI layout of a multi-access memory based DA circuit needs to be attempted to make sure that the connection complexity of the multi-access memory unit does not overwhelm what appears to be a significant area savings potential in the case of static RAM memory cells.

3 Truncation and Rounding

Once each term is fetched from the memory, they are shifted and added to form the final result. This operation is diagramed in Figure 10. If each memory in a fully parallel implementation stores terms that are b bits wide, then the resulting inner product will occupy

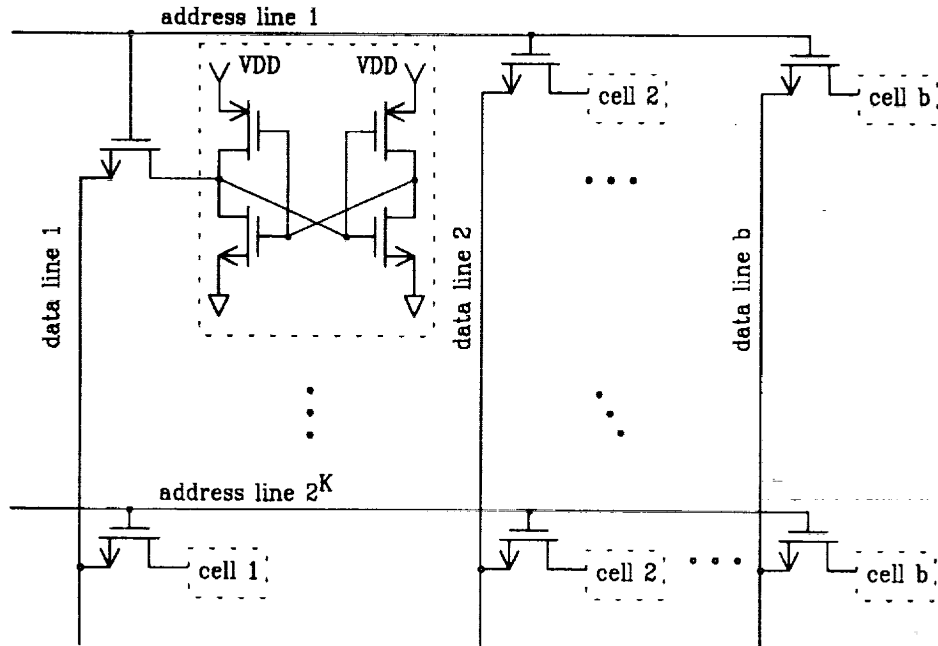

Figure 8: Static RAM

at most $M + b$ bits.[2] Suppose, however, that the product only needs to be determined to an accuracy of f significant bits. In this case it may be possible to truncate or round the individual terms before adding. Doing so would not only reduce the amount of hardware required in the adder tree, but would also reduce the size of some of the memories. This is obvious in the case of the fully parallel DA implementation and, as we will see later, it is also true for the multi-access memory unit based implementation. First let us consider what the impact of truncating or rounding will be on the accuracy of the final result.

Truncating the individual terms and discarding bits that fall in column $f + e$ and to the right (as shown in Figure 10) will give a maximum worst case error of

$$E_{tt} = 2^{-e}((M + b - (f + e)) - 1 + 2^{-(M+b-(f+e))}) \tag{8}$$

where we have normalized the result so that the binary point falls just to the left of column f.[3] The worst case truncation error is calculated by considering that all the truncated bits are ones.

When we round the individual terms and then discard bits that fall in column $f + e$ there are two worst case error situations. If the bits in column $f + e$ are all ones, and all bits to the right are zeros. In this case the error is

$$E_{rta} = 2^{-(e+1)}(M + b - (f + e)) \tag{9}$$

[2] This can be shown by temporarily treating the terms as whole integers and assuming that all M terms take on the maximum value $(2^b - 1)$. The final sum will then be $(2^b - 1) * (2^M - 1)$ which can be written as $((2^{b+M} - 1) - (2^M - 1) - 2^b) + 1$. When written this way and assuming $M < b$ it is easy to see that the result occupies at most $M + b$ bits.

[3] We also need $f \geq b$.

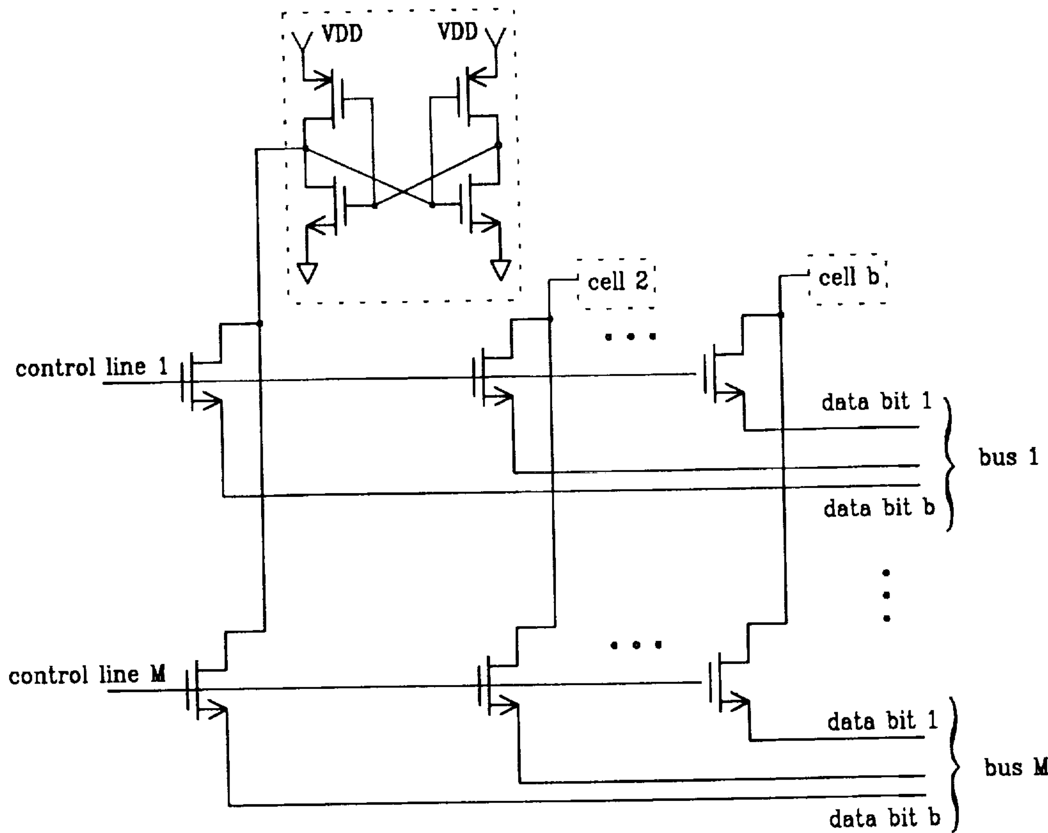

Figure 9: Static RAM Multi-Access Memory Plane

If the bits in column $f+e$ are zeros and all the bits to the right are ones, the error is

$$E_{rtb} = 2^{-(e+1)}((M+b-(f+e+1)) - 1 + 2^{-(M+b-(f+e+1))}) \tag{10}$$

After the individual terms have been truncated or rounded to $f+e$ bits, the final sum is computed and then either truncated or rounded to f bits. This second truncation or rounding will add to the total error. In the case of truncation, the worst case error will be

$$E_{tf} = 1 - 2^{-(e-1)} \tag{11}$$

In the case of rounding, the additional worst case errors are

$$\begin{aligned} E_{rfa} &= 2^{-1} \\ E_{rfb} &= 2^{-1} - 2^{-(e-1)} \end{aligned} \tag{12}$$

Noting that $E_{rta} > E_{rtb}$ and $E_{rfa} > E_{rfb}$, we will use E_{rta} and E_{rfa} when referring to rounding. There are four possible approaches to arriving at the final result depending on which of truncation or rounding is applied to the individual terms and which is applied to the final sum. The four possibilities are summarized in Figure 11. From the graph, we see that as few as five or six extra bits beyond f are required in order to arrive at errors that are very near what we would expect if we retained all the bits in the individual terms,

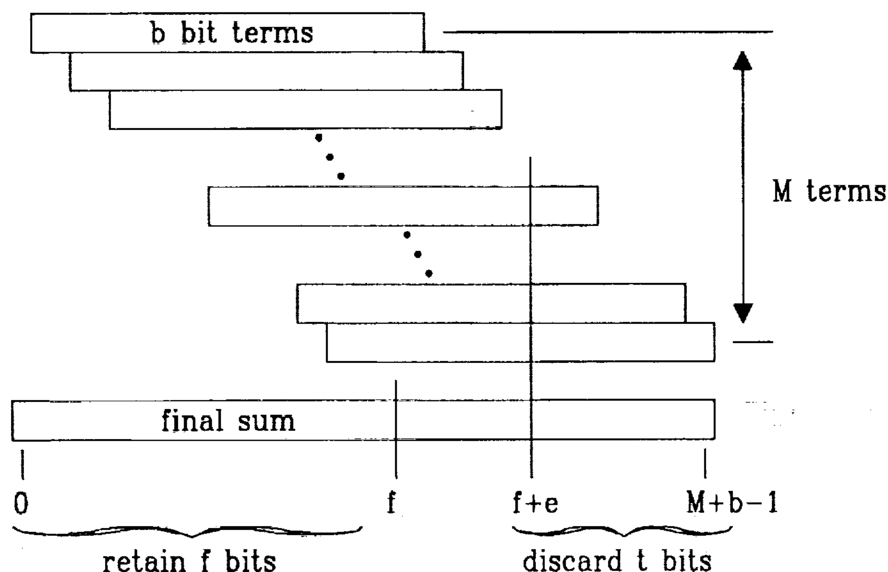

Figure 10: The Final Sum

formed the sum and rounded or truncated the result to f bits. We also note that this is true independent of whether the individual terms are first truncated or rounded. When $e < 4$, the error resulting from rounding is similar to the error resulting from truncating one less bit. These observations suggest that there is not a great deal to be gained by rounding individual terms over truncating them. In the case of a fully parallel DA implementation, the rounding of individual terms can be precomputed and only the rounded terms stored, so there is no cost in doing so. However in the multi-access memory based implementation, rounding of individual terms would have to be performed upon access. As we shall see shortly, the area requirement of the multi-access memory for rounding will be the same for a memory the truncates one less bit. This, coupled with the above observations on error, indicate that rounding individual terms does not provide a very great advantage over truncation in the multi-access memory.

4 Implementing Truncation and Rounding

First we consider the transistor cost of a ROM based fully parallel DA implementation. From Figure 10 we see that if we desire to compute the final sum to $f + e$ bits, we will need $(M - t)$ ROMs storing b bit words, and t ROMs that each store one bit less in succession where $t = (M + b) - (f + e)$. Note that for consistency, $t < b$ and $t \leq M$. If $t \geq b$, M should be reduced and if $t > M$, b should be reduced. Now, referring to Figure 5 we see that for each bit truncated, $2^K + 1$ transistors are saved. Since $\sum_{i=1}^{t} i = t(t + 1)/2$, the cost of the implementation is

$$n_t = M(b2^K + 2^K + b) - \frac{t(t+1)}{2}(2^K + 1) \qquad (13)$$

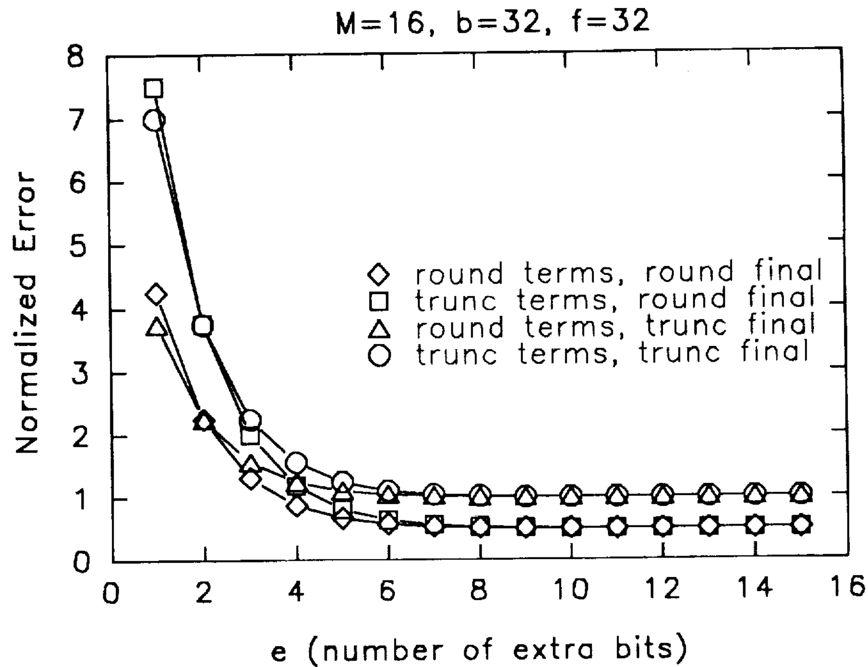

Figure 11: Errors from Truncation and Rounding

Again, K to 2^K decoders are required for all M ROMs. We also note that this equation applies equally well to both truncation and rounding of individual terms. Next we consider the transistor cost of the multi-access ROM based implementation. We note that while each stored term must have the full b bits, the width of the last t data paths decreases by one bit for each path. This is shown in Figure 12 where the storage cells are implemented by connections (or the lack thereof) to ground through a precharge transistor as in Figure 7. Thus, we save $t(t+1)/2$ bus control transistors in each plane so, the overall cost of the implementation is

$$n_t = Mb2^K + b2^K + 2^K - \frac{t(t+1)}{2}2^K \tag{14}$$

Comparing the savings of the two approaches, we see, not surprisingly, that the multi-access ROM continues to loose ground against the fully parallel implementation. The disadvantage is further amplified when we consider applying rounding to individual terms. In the fully parallel approach, the rounding is precomputed, but in the multi-access approach the rounding must be computed on-line. An extra bit in each of the terms to be rounded is required. If the bit is a one, the one is to be added to the next more significant bit. This could be achieved by routing the extra bit to an appropriate place in the adder tree. Another approach might be to truncate so that a final sum of $f + e + 1$ bits is computed, resulting in an equivalent error. In either case, an extra bit would be needed for each of the M data paths in each of the 2^K planes.

Extending the comparison to the use of static RAM, from Figures 8 and 10, we see that truncating or rounding so that the final sum is computed to $f + e$ bits would save $(5t(t+$

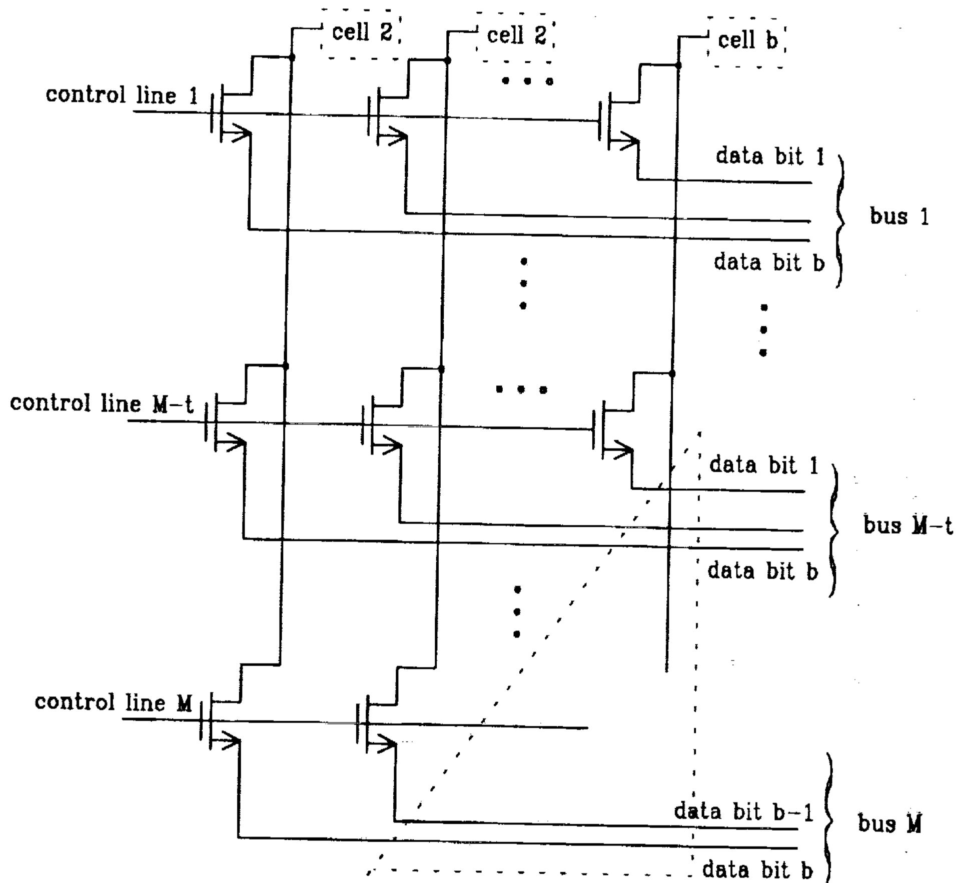

Figure 12: Multi-access Plane with Truncation

$1)/2)2^K$ transistors in the fully parallel implementation. Letting the storage elements in Figure 12 be the 4 transistor cells used in Figure 9, we see that the savings from truncation in the multi-access RAM based implementation is the same as it was in the multi-access ROM, namely $2^K t(t+1)/2$. Not surprisingly, we find that the fully parallel RAM based implementation benefits more from truncation than does the multi-access based RAM architecture. We note however, that it still possesses a significant advantage. The ratio of the number of transistors becomes

$$R_{area} = \frac{(M+4)b - t(t+1)/2}{5(Mb - t(t+1)/2)} \tag{15}$$

With $t = M = 8$ and $b = 16$, $R_{area} = 0.34$ as compared to the .30 ratio that arises if $t = 0$. We also note that the reduced number of bus control transistors and reduced bus widths reduces the connection complexity of the multi-access architecture.

5 Conclusions

We have shown that the multi-access architecture requires significantly less area than a fully parallel architecture when the number of transistors per stored bit is greater than one, as it will be when static RAM cells are employed. Since this observation is based on the assumption that the transistors used for storage are the same size as those used

for bus control, we can say more abstractly that the multi-access architecture will save space whenever the area required to implement each storage cell is greater than the area required to implement a bus control or routing transistor. The savings estimates do not include the cost of decoders and the cost of the adder tree (which will be the same in both cases). The area requirements of these elements must be added in so we can truly asses the area savings advantage of our approach. Both approaches appear to be fairly regular, both lending themselves well to VLSI implementation. Again this observation is made independently of the implementation of the decoders and the adder tree. The connection complexity between these elements in both architectures also needs to be considered. In short, a VLSI layout of both architectures needs to be done in order to be able to accurately compare the two.

We have also presented the errors associated with truncating or rounding individual terms and the area savings that can result in both architectures from doing so. These errors need to be reconsidered, placing them in the overall context of the inner product. In particular we have not considered what b should be given M and K.

References

[1] A. Peled and B. Liu, "A New Approach to the Realization of Nonrecursive Digital Filters," *IEEE Trans. Audio and Electroacoustics*, Vol. AU-21, No.6, pp. 477-485, December 1973.

[2] S. Zohar, "New Hardware Realization of Non-recursive Digital Filters," *IEEE Trans. on Computers*, Vol. C-22, pp. 328-338, April 1973.

[3] A. Peled and B. Liu, "A New Hardware Realization of Digital Filters," *IEEE Trans. on A.S.S.P.*, Vol. ASSP-22, pp. 456-462, December, 1974.

[4] C.S. Burrus, "Digital Filter Structures Described by Distributed Arithmetic," *IEEE Trans. on Circuits and Systems*, Vol. CAS-24, No.12, pp. 674-680, December, 1977.

[5] F.J. Taylor, "An Analysis of the Distributed Arithmetic Digital Filter," *IEEE Trans. on A.S.S.P.*, Vol. ASSP-34, No.5, pp. 1165-1170, October 1986.

[6] M. Hatamian, et al., "Parallel Bit-Level Pipelined VLSI Designs for High-Speed Signal Processing," *Proceedings of IEEE*, Vol.75, No.9, pp.1192-1202, September 1987.

[7] S. Zohar, "A VLSI Implementation of a Correlator/Digital Filter Based on Distributed Arithmetic," *IEEE Trans. on A.S.S.P.*, Vol. ASSP-37, No.1, pp. 156-160, January 1989.

[8] S.A. White, "Applications of Distributed Arithmetic to Digital Signal Processing: A Tutorial Review," *IEEE ASSP Magazine*, Vol.6, No.3, pp. 4-19, July 1989.

[9] Xiaoyi Guo, "An Improvement on Distributed Arithmetic Implementation of High Speed Digital Filters", Masters Thesis, University of Idaho, May 1991.