IMPLEMENTATIONS OF THE CONVOLUTION OPERATION

Per-Erik Danielsson

INTERNAL REPORT
LiTH-ISY-I-0546
Implementations of the convolution operation

Per-Erik Danielsson

Internal Report

LiTH-ISY-I-0546

Abstract

The first part of this article surveys a large number of implementations of the convolution operation (which is also known as the sum-of-products, the inner product) based on a systematic exploration of index permutations. First we assume a limited amount of parallelism in the form of an adder. Next, multipliers and RAM:s are utilized. The so called distributed arithmetic follows naturally from this approach.

The second part brings in the concept of pipelining on the bit-level to obtain high throughput convolvers adapted for VLSI-design (systolic arrays). The serial/parallel multiplier is analyzed in a way that unravels a vast amount new variations. Even more interesting, all these new variations can be carried over to serial/parallel convolvers. These novel devices can be implemented as linear structures of identical cells where the multipliers are embedded at equidistant intervals.
## Content

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0. Preface</td>
<td>3</td>
</tr>
<tr>
<td>1. Introduction</td>
<td>4</td>
</tr>
<tr>
<td>2. Parallel adder implementations</td>
<td>8</td>
</tr>
<tr>
<td>3. Parallel multiplier implementations</td>
<td>13</td>
</tr>
<tr>
<td>4. &quot;Distributed arithmetic&quot;</td>
<td>18</td>
</tr>
<tr>
<td>5. Iterative (systolic) arrays</td>
<td>21</td>
</tr>
<tr>
<td>6. Serial/parallel multipliers</td>
<td>27</td>
</tr>
<tr>
<td>7. A catalogue of serial/parallel-multipliers</td>
<td>35</td>
</tr>
<tr>
<td>8. The serial/parallel convolver</td>
<td>39</td>
</tr>
<tr>
<td>9. Conclusions</td>
<td>50</td>
</tr>
<tr>
<td>10. Acknowledgement</td>
<td>51</td>
</tr>
<tr>
<td>11. References</td>
<td>51</td>
</tr>
</tbody>
</table>
Initially, this paper was intended to be a tutorial survey. It came about as a side-interest when the author was investigating bit-serial multiprocessor architecture for image processing. I became more intrigued by the subject when I discovered that the problem of implementing the sum of a number of products, however old, still seemed to have several unexplored dimensions.

Particularly, I claim novelty for

- Figure 2 below that sums up and clarifies the variations that appear when indices are permuted in the basic convolution formula.

- Figure 4b that shows a way to greatly simplify a design that has been used in the IBM RSP signal processor.

- Figure 16 which is a suggestion for a highly parallel convolution chip. Because of the extreme pipelining involved, it could be expected to be very fast. A 3x3 convolution on 8-bit data in 25 ns seems to be in reach.

However, the main contributions are in the last sections of the paper that deals with serial/parallel multipliers and convolvers. A set of equations is established that allows the design of a serial/parallel convolver of any choice. A whole family of previously unheard serial/parallel multipliers are presented and, most important, the serial/parallel concept is carried over from the bit level in the single multiplication to the word level for the convolution itself. Formulas are developed that provide an almost effortless design of modular bit/serial programmable convolvers as well as whole range of convolvers tailored to a certain precision and kernel size.
1. INTRODUCTION

One of the most common operations in signal processing is convolution. In the discrete space the convolution takes the form

$$Y(i) = \sum_{l=1}^{L} A_l X_l^{(i)}$$

(1)

where

$$X_l^{(i)}$$

is one of L input sample values to be used for computation of the output Y at point (i) and $$A_l$$ is the corresponding weight (coefficient) in an L-point convolution kernel. When not needed we will subsequently drop the superscript (i). Both $$X_l$$ and $$A_l$$ are binary numbers which without loss of generality can be assumed to be fractions. Although not particularly important to the following discussion let us also assume that all negative numbers are represented by 2-components. We will use the following notation.

$$X = \sum_{n=0}^{N-1} x_{2n} 2^{-n} = x_{20}2^0 + ... + x_{2n}2^{-n} + ... + x_{2N-1}2^{-N+1}$$

(2)

$$A = \sum_{k=0}^{K-1} a_{2k} 2^{-k} = a_{20}2^0 + ... + a_{2k}2^{-k} + ... + a_{2K-1}2^{-K+1}$$

(3)

where $$x_{20}$$ and $$a_{20}$$ take their values from the set (0,-1) while all the other $$x_{2n}$$ and $$a_{2k}$$ take their values from (0,+1).
The expressions (2) and (3) unfold (1) into

\[ Y = \sum_{\lambda=1}^{L} \sum_{k=0}^{K-1} a_{\lambda k} \cdot 2^{-k} \cdot \sum_{n=0}^{N-1} x_{\lambda n} \cdot 2^{-n} = \sum_{d=0}^{D-1} y_d \cdot 2^{-d} \]  

(4)

The expression (4) corresponds to and motivates Figure 1 where the bit-contributions \( a_{\lambda k} x_{\lambda k} \) are ordered in the manner that is customary for paper and pencil multiplication. It is readily seen from Figure 1 that the size of the convolution operation is in the order of \( O(L \cdot K \cdot N) \). For simplicity, in Figure 1 we have chosen these parameters to be \( L = K = N = 3 \) which brings the total number of bit-contributions to 27 for the total sum. For negative numbers \( A_{\lambda} \) in 2-complement representation, the lower-most row in each group is to be fed by the representation of \(-X_{\lambda}\) instead of \(X_{\lambda}\). Also, for negative numbers \(X_{\lambda}\), each multiplier has to be extended with one extra "staircase" of guard bits.

![Figure 1](image-url)
In the following sections we will present several computational schemes employing various degrees of parallelism. The surprisingly large number of different algorithms and corresponding hardware implementations has the following two main reasons.

i) The total sum (4) consist of bit contributions along the three "index axes" $J$, $k$ and $n$. Several possibilities of parallelism and carry propagation arise by simply permutating the order of these indices.

ii) Since the operands $A_x$ are constants it is possible to exploit this as an a priori knowledge to shorten microprogram sequences or to store precalculated combinations of these constants in fast RAM:s for table look-up.

The permutation of indices gives us 6 possibilities since there are three indices involved. The six variations are depicted in Figure 2. Accumulation takes place from top to bottom, right to left in all cases and each dot is one bit contribution determined by data $x_{J,n}$. The blanks are zero-contributions due to a zero bit in the coefficient. The arbitrarily chosen coefficients (the constants) in Figure 2 are

$$A_1 = 1101 \quad A_2 = 0010 \quad A_3 = 0101$$

The reader is urged to trace the movements of the bit-contributions when going from one scheme to the next.
2. PARALLEL ADDER IMPLEMENTATIONS

In traditional multiplications schemes the contributions depicted in Figure 1 use to be accumulated row-wise from top to bottom which is the order given by expression (4) and further illustrated by Figure 2a. The corresponding implementation employing a limited form of parallelism along the n-axis is an N-bit parallel adder with or without carry acceleration. See Figure 3a. To indicate the parallelism also in the mathematical expression we may transform (4) to

$$Y = \sum_{\lambda=1}^{L} \sum_{k=0}^{k-1} a_{\lambda k} 2^{-k} x_{\lambda}$$

One bit $a_{\lambda k}$ of the multiplier $A_{\lambda}$ determines whether the multiplicand $X_{\lambda}$ or 0 is added to the accumulator. In each clock-cycle the accumulator is right-shifted one step to be ready for the next cycle controlled by the next bit in $A_{\lambda}$.

One "difficulty" with Figure 3a is that the accumulator result has to be left-shifted $K$ steps when we increment the outer index $\lambda$. Thus, we have to take care of carries over the total number of output bits which is

$$D = N + K + \log_2 L \text{ bits}$$

and which is also the necessary word-length for the adder/accumulator. Therefore, we could just as well do as shown by Figure 3b where we are left-shifting the multiplicand while keeping the accumulator in place.
Now, as have been shown by Peled [1] we can use our a priori knowledge of the constants $A_k$ and completely skip those cycles for which the corresponding bits $a_{nk}$ are zero. See Figure 4a. For speed reasons the shifter now has to be a full $N$-bit $K$-way combinatorial so called barrel shifter. The microprogram determines the number of shifts so that every clock cycle is used for accumulating new contributions

$$X_k 2^{-k}$$

Actually, the constants $A_k$ are no longer visible as data but incorporated into the microprogrammed control sequence. When all non-zero coefficients in $A_k$ are exhausted the microprogram immediately proceeds with the next data item $X_{kn}$.
It was also shown in [1] that the procedure in Figure 4a could benefit greatly by using Canonical Signed Digit Code (CSD) for the coefficients $A_\ell$. Hereby, for a positive or negative integer of $K$ bit accuracy the average number of non-zero binary digits decreases from $K/2$ to $K/3$. And for a typical set of filter coefficients which normally has several elements of near zero magnitude, the number of non-zero digits often averages to $K/4$. With the mechanism of Figure 4a the corresponding speed-up factor (up to 4) is achieved since in average each $X_\ell$ is used in only $K/4$ ADD/SUB cycles.

Now, if we change the order of the two summations in (5) we obtain the expression

$$ Y = \sum_{k=0}^{K-1} 2^{-k} \sum_{\ell=1}^{L} a_{\ell k} \cdot X_\ell $$

(6)

which is further illustrated by Figure 2b), index order $(k, \ell, n)$. It should be immediately clear that we can proceed with the accumulations in the manner of Figure 4b. For each $k$ the inner loop consists of cycles where a new $X_\ell$ is fetched and accumulated. The microprogram selects only those $X_\ell$'s for which the corresponding $a_{\ell k}$ is non-zero. Thus, the number of cycles in the inner loop (index $l$) is dependent on the actual set of $a_{\ell k}$'s. The inner loop is always terminated by incrementing index $k$ and right-shifting the accumulator, hereby chopping off one bit in the final result.

The solution in Figure 4b benefits from a high frequency of zeros in the $A_\ell$'s just as the scheme of Figure 4a. However, Figure 4b is considerably cheaper since its adder width is $K$ bit less and, most important, the barrel shifter is no longer needed. Instead, the accumulator content is combinatorially shifted one step simultaneously to the last accumulation in the inner loop. In those rare cases where all $a_{\ell k}$ are zero for a certain $k$ this gives a time penalty of one cycle for Figure 4b.

Since Figure 4b requires a new $X_\ell$ for each cycle this scheme might seem to put a higher strain on the bandwidth of the data memory. Occasionally, however, also in Figure 4a must the data memory be ready to bring a new data item every cycle. The processor of
Figure 4b is inherently faster since the carry propagation path is shorter. As a principal solution, the design of Figure 4b seems to be superior to the one in Figure 4a which is the basis of the IBM RSP-processor [16].

So far we have utilized the following orderings of the three indices:

- \( \lambda, k, n \) in Figure 3, Figure 4a and expression (5)
- \( k, \lambda, n \) in Figure 4b and expression (6)

In all cases the index axis \( n \) has been covered by a parallel adder, indices \( \lambda \) and \( k \) by a time sequence. A rather obvious variation would be to reverse the order of \( k \) and \( n \) in the basic expression (4), i.e. to exchange multiplicand and multiplier, which gives us the expression

\[
Y = \sum_{\lambda=1}^{L} \sum_{n=0}^{N-1} x_{\lambda n} 2^{-n} A_{\lambda}
\]

with the index ordering \( \lambda, n, k \). By reversal of \( \lambda \) and \( n \) in (7) we get

\[
Y = \sum_{n=0}^{N-1} \sum_{\lambda=1}^{L} x_{\lambda n} 2^{-n} A_{\lambda}
\]

using the index ordering \( n, \lambda, k \). The expression (7) and (8) correspond to the Figure 2c and 2d respectively. With the basic components in the previous Figures 3 and 4 the expressions (7) and (8) can be implemented as shown by Figure 5a and 5b respectively. In Figure 5b we are now proceeding monotonously from lower to higher significant bits just as in Figure 4b. The databits \( x_{\lambda n} \) should therefore be fetched not by words \( X_{\lambda} \) but as bit-vectors \( X_{n} \).

Cycle skipping decisions in the manner of Figure 4 are not possible in Figure 5a or 5b since we are now using the data bits as control signals in the processor and no a priori knowledge of these can be anticipated. In summary, there seems to be no specific advantages in the schemes of Figure 5 compared to Figure 4.
The remaining index orderings are \( k, n, \lambda \) and \( n, k, \lambda \). These cannot be exploited here since the incrementing of index \( \lambda \) does not follow increasing powers of two. Thus, the bit-vector

\[
a_{1k} x_{1n} , a_{2k} x_{2n} , \ldots , a_{jk} x_{jn} , \ldots , a_{Lk} x_{Ln}
\]

found as a vertical column of bit-contributions in Figure 2e cannot be used as an input operand to a conventional adder.

The possibilities for using a conventional adder unit for implementation of the convolution operation then seems to be exhausted.
3. PARALLEL MULTIPLIER IMPLEMENTATIONS

In this section we will investigate combinational networks with higher degrees of parallelism than in the previous section. The expression (1) suggests a rather obvious implementation that uses one combinatorial (= direct) multiplier as in Figure 6a or a whole set of multipliers as in Figure 6b. The present size limit of a one chip multiplier seems to be 12x12 (12+12 inputs and 24 bit result) or at most 16x16 (32 bit result).

In figure 6 as in many of the following Figures, pipe-lining is an option for increased throughput rate. However, when pipe-lining is trivially obvious as in Figure 6 we will not specifically comment on this fact.

Fig. 6
Since one of the operands to the multiplier \((A_x)\) is a constant we could replace the direct multiplier with a RAM where all possible \(2^N\) outcomes of \(A_x X_x\) are prestored. The memory size for such a look-up table is \(2^N (N + K)\) bits and for the total convolution operation we need \(L\) such tables which of course amounts to a memory space of \(L \cdot 2^N (N + K)\) bits.

The implementation is shown in Figure 7. While the RAM has \(K\) fewer inputs than the direct multiplier in Figure 6 the RAM size grows exponentially with the parameter \(N\) which prohibits its use for high precision multiplication.
Unfortunately we need one table for each \( A_j \). Thus, the RAM-approach of Figure 7a with serial accumulation of the partial result seems more costly than the single direct multiplier of Figure 6a. Figure 6b and Figure 7b are more likely to be comparable in complexity although it is always difficult to compare so different structures as a RAM and a direct multiplier. Intuitively, in the 8-bit precision case it seems that a 256x16 bit RAM would compete rather well with an 8x8 multiplier while for 12-bit precision a 4Kx24 bit RAM seems to be more costly than a 12x12 multiplier. It is worth noting that the RAM is probably faster than the direct multiplier since the RAM involves no carry propagation.

The complexity of Figure 7 is possible to reduce by recycling each data item \( L \) times as shown by Figure 8. Effectively, we will then change the ordering between the index \( \lambda \) and the hitherto hidden index \( i \) that traverses the output data points. By using expression (1) in Figures 6 and 7 we have employed the ordering \( i, \lambda, (k,n) \) where indices inside parentheses indicate parallel computation. Figure 8 now suggests the ordering \( \lambda, i, (k,n) \). The smaller RAM size is achieved at the expense of approximately 5 times higher bandwidth requirements for the data memories: For each accumulation not only DO Read \( X_\lambda \) but also DO Read and Store partial results \( Y(i) \) having double word-length.

![Diagram of a data flow](image)
One advantage of Figure 8 is the straightforward addressing of the datapoints. In some implementations where speed is bound by address computation and index incrementing, this may outweigh the drawback of high memory bandwidth.

For the above RAM-implementations it is important to note that set-up time for the operations can be rather extensive. Most sensitive to this limitation is the case in Figure 7b since the actual computation time is only one cycle per output data point while no less than $L \cdot 2^N (N+K)$ bits has to be preloaded into the RAMs.

The size limitations of both the direct multiplier and the RAM bring forward the question of possible partitioning of the previous schemes. A partitioning would also facilitate a trade-off between time and space complexity. In the multiplier case partitioning means e.g. employing four 8x8 multipliers instead of one 16x16 (or four 256 words RAM:s instead of one 65 kword RAM) in the manner shown by Figure 9. The final accumulation of the partial results turns out to be rather costly.

![Diagram](image-url)
Since a RAM has a size that is exponentially dependent on the number of address bits partitioning can be expected to have a great impact. See Figure 10. A single (unrealistically large) memory of 64Kx32 bits for doing a 16x16 multiply is reduced to 2 x 256 x 24 bit by simply splitting the input in two halves each half doing an 16x8 multiply. More precisely, the two halves of Figure 10 produce the two terms of (9)

\[ A_7 X = \sum_{k=0}^{7} x_{j_k} 2^{-k} A_{15} + \sum_{k=7}^{15} x_{j_k} 2^{-k} A_{15} \]  

(9)

Note that the two RAM:s have identical content. This facilitates the time-shared solution of Figure 11.
Although the previous schemes show that modularization of the RAM:s save considerable memory space we still have to produce one look-up table for each one of the L kernel points. In the next section we will see how it is possible to reduce the problem so that only one look-up table is needed. Furthermore, for a given computation time this table has only the size of each single table in Figure 7. Likewise, when modularized this table will be smaller than each one of tables in Figure 10 and 11. The trick to do it is the same as before: use another order of the indices in the basic summations.

4. "DISTRIBUTED ARITHMETIC"

In the two previous sections we have deliberately used the bit-accumulation structure given by the adder and the multiplier respectively. This is not really necessary. Figure 2d suggests that we could use the formula

\[ Y = \sum_{n=0}^{N-1} \sum_{\lambda=1}^{L} 2^{-n} x_{ln} A_{\lambda} \tag{8} \]

Note that all the N blocks of Figure 2d have the same pattern of dots. This gives us an ideal situation for accumulation by RAM table look-up. Each of the N blocks are the sum of a subset of the same set of A:s, the subset n to be specified by the bit-vector

\[ x_n = (x_{1n}, x_{2n}, \ldots, x_{Ln}) \]

and the RAM to be preloaded with all \(2^L\) subsums. The whole procedure proceeds monotonously from least to most significant bit which facilitates simple accumulation between the blocks. See Figure 12.
Note that there is only one table in Figure 12a sized

\[ 2^L (K + \log L) \text{ bits} \]

which in the fully parallel case of Figure 12b has to be duplicated \( N \) times.

Modularization is of course also possible as shown by Figure 13 where we have partitioned the memory into two parts each one addressed by half as many bits from the \( L \) bit vector \( X_L \). It is relatively easy to design various parallel/serial combinations from Figures 12 and 13 for a given problem size \( L \cdot K \cdot N \) and a given speed requirement. As an example let \( L = 64 \), \( K = 12 \), \( N = 12 \) and
the speed requirement be one output value per 2 μs (equivalent to 32 12x12 mul/sec). Estimating that a RAM cycle could easily be as short as 80 ns the solution is given by Figure 14. Buffer registers between the RAM:s and the adders, necessary for proper pipe-lining of the data flow, have been omitted for the sake of simplicity.

Fig. 13

Fig. 14
Input data in Figures 12, 13 and 14 have to be delivered as bit-vectors.

\[ X_n = (x_{1n}, x_{2n}, \ldots, x_{Kn}, \ldots, x_{Ln}) \]

which requires some bit formatting that can be accomplished with shift-register technique. This might be a draw-back in some applications but in a VLSI-design these shift-registers are easily incorporated.

All in all, the RAM-implementation of the accumulation scheme of Figure 2e (index sequence \( n, (\lambda, k) \)) with the proper amount of partitioning and modularization seems to be a very powerful implementation of the convolution operation. It was first proposed by Croisier [2] and (independently) by Peled and Liu [3], the latter paper being responsible for bringing this technique to the attention of the scientific communities. For some reason it has lately become known as "distributed arithmetic" [4]. Several implementations are under way, either as an integrated part of a general signal processor or as a separate specialized VLSI chip for video rate pipe-lined signal processing [5].

5. **ITERATIVE (SYSTOLIC) ARRAYS**

The rather recent concept systolic arrays has acquired considerable attention with the book [6] by Mead and Conway where the section 8.3 was written by H.T Kung and Leiserson, Carnegie-Mellon University. Actually, systolic arrays are a subset of the more general and older concept iterative arrays (= cellular automata) where F.C. Hennie wrote the pioneering work [7]. Usually, systolic arrays are limited to networks without feedback loops. Unlike the previous purely theoretical works on cellular automata the concept of systolic arrays seems to be applied by different VLSI-designers for very practical algorithm implementation.
The word systolic implies a heart beat and a pulsed flow. Translated to the digital design domain it means a clocked and pipelined system, the final output result being computed (accumulated) in a number of processor stages. In addition, the input data and control parameters are also allowed to move from input to different stages. No buses are allowed which means that the only global signals on the chip are ground, power and clock(s). In fact, in a systolic array different input and output data streams may very well flow in separate and/or opposite directions forming a linearly, orthogonally or hexagonally connected network. Still, no inter-cellular closed cause-event loops are allowed (according to the interpretation of the present author). But the moving of 1:s and 0:s in different directions make up the impression of something similar to the cardiovascular system in a living being.

It is important to note that while the convolvers of the previous sections can be used for both FIR- and IIR-filters (non-recursive and recursive), the pipe-lining principle as such excludes recursive filtering.

Recently, Kung [8] suggested a 2D-convolution chip design based on the systolic idea. Each cell in the iterative structure is a rather crude arithmetic unit containing only a shift mechanism and a serial accumulator capable of adding a constant multiplied with a power of two. The communication between cells are bit-serial and constitute the "systolic" part of the design.

In the approach shown by Figure 15 we take the more radical step of using iterative cells at the bit level. Each cell (see Figure 16) is performing the full-adder function which is the basic accumulation step in the convolution operation. The number of cells in this array are \( N \times L \times (K + \log L) \). The throughput rate is equal to the clock rate which can be extremely high since the logic depth of all cells are only two (or four, depending on actual design of the full adder). 40 MHz seems to be a rather conservative estimate.
Then, in every 25 ns the array delivers a new result $y^{(i)}$. A five point kernel for 8-bit data and 8-bit coefficients ($L = 5$, $N = 8$, $K = 8$) requires approximately 440 cells, approximately 35 pins (8 for data in, 20 for data out) and would perform the equivalent of 200 Mmul/sec.

The array for the case $L = 3$, $K = 3$, $N = 5$ is shown by Figure 15. The basic index order is $(n, l, k)$ that is the same as the one used in expression (8) and in Figure 2d.

As seen by Figure 2, there are basically four layouts of a pipelined parallel accumulation, namely: i) Figure 2a, ii) Figure 2b, 2e, iii) Figure 2c, iv) Figure 2d, 2f.

Figure 2a and 2c simply correspond to multipliers serially connected. Figure 2b and 2e require that the same x-bits are active in different blocks. We believe that Figure 2d (and 2f) are most suited for iterative and pipelined design.

The basic cell is described in detail by Figure 16. It is seen to have a static 1-bit storage for the coefficient bits $a_{jk}$ and these are distributed over the array as shown in Figure 15. The leftmost data bit $x$ is the sign bit of the 2-complement represented number which explains why the negative counterparts of the coefficients are stored in the three last rows of the array.
The actual loading of the coefficients can be simplified if the memory cells for the bits $a_{\lambda}$ are chained together into one long meandering shift register. However, to avoid graphical overcrowding these connections have been omitted from Figure 15.

Now, the task of the device is to compute

$$
\gamma(i) = \sum_{\lambda=1}^{L} A_{\lambda} x_{\lambda}(i)
$$

in a running window fashion over the one-dimensional string of samples $x(i)$. New $X$-values are continuously entering at the top at each clock cycle setting in motion a flow of $Y$-values ($Y$-waves) down the array. All output values $\gamma(i)$ start to be accumulated at the very top right cell with the contribution.

$$
a_{12} x_{14}^{(i)}
$$

In the vertical direction, this value is propagated downward so that in the next clock cycle is produced

$$
a_{12} x_{14}^{(i)} + a_{22} x_{14}^{(i-1)}
$$

which is equal to

$$
a_{12} x_{14}^{(i)} + a_{22} x_{24}^{(i)}
$$

since

$$
x_{\lambda n} = x_{\lambda+1,n}
$$

So, to let the correct $x$-bits coincide with its proper wave of accumulating $Y$-value, the $x$-bits at the border of the array are vertically delayed two units before propagated horizontally into the array. This same principle is used by Kung [8]. The vertical $x$-pace is half of the pace of the $Y$-wave. As we shall see in the following sections this is only one of several possible relations between the $x$- and the $Y$-wave.
In the horizontal direction the carry signals belonging to an accumulating Y-value is propagated with one unit delay over the cells to produce, e.g. in the first row

\[ a_{12} x_{14} + 2 a_{11} x_{14} \]

So, for the correct x-bit to coincide with the proper Y-wave the x-bits should be propagated horizontally at the same pace. The front of the Y-waves will make a 45 degree slope, the x-waves will make a slope of arctg (1/2). Every x-bit combines with all the a-bits before its wave dies out at the left border of the block. Note that the vertically flow of data is right-shifted one step after every lth row. In Figure 15 we have \( L = 3 \). This requires \( x_3^{(i)} \) to be delayed \( 3 + 1 = 4 \) units before it gets into play since it starts its contribution to \( Y^{(i)} \) three levels down and one steps to the left.

By the same reason should \( x_2^{(i)} \) be delayed \( 6 + 2 = 8 \) time units. More generally; bit \( x_n^{(i)} \) has to be delayed \( (N-1-n)(L+1) \) time unit before entering its proper block in the array. These delays make up for approximately half of the delay elements to the right of the Figure 15. The rest of these elements are used for time alignment of the output bits. The last bit \( y_0^{(i)} \) leaves the array 23 time units after it started to be computed at the upper right corner. Thanks to the shift-register system of delay elements all the other bits in \( Y^{(i)} \) leave at the same time.

A closer look at the delay conditions reveals that what is labeled \( Y^{(i)} \) in Figure 15 actually is composed of

\[ A_1 x^{(i)} + A_2 x^{(i-1)} + A_3 x^{(i-2)} \]

where again \( (i) \) is the sample number of the entering data.

The design of Figure 15 seems to be very advantageous over previous attempts in the same direction e.g. by Swartzlander and
Gilbert [9] who describes a pipelined convolver that follows the basic scheme of Figure 6b. Hence, this solution requires an adder tree which destroys the modularity of the total solution. Also the design of [9] requires many more input/output pins for a comparable problem size.

An approach similar to the one presented here is presented by Denyer and Myers [15], the difference being that their array is organized with index order \((k, \ell, n)\) as in Figure 2b (or \((k, n, \ell)\) as in Figure 2e) instead of \((n, \ell, k)\) as in Figure 2d (or \(n, k, \ell\) as in Figure 2f).

6. SERIAL/PARALLEL MULTIPLIERS

Serial/parallel multipliers involve three bitstrings for

- the input variable \((x_0, x_1, x_2, \ldots, x_{N-1})\)
- the input constant \((a_0, a_1, a_2, \ldots, a_{K-1})\)
- the output \((y_0, y_1, y_2, \ldots, y_{D-1})\)

representing three binary numbers in, say 2-complement form. The a priori assumption is that the \(x\)-string and the \(a\)-string are serially fed into the computational unit and that the resulting \(y\)-string is likewise serially shifted out. The \(y\)-string is successively computed so that an original value of 0 0 0 0 0 0 during motion of the string is converted to the final value \(y_0, y_1, y_2, y_3, y_4\).

Thus we have three strings that move into and out from the computational unit. Since the \(a\)-string consists of constants we could possibly assume already now that this string is preloaded and static. However, we will postpone this decision to avoid loss of generality. The computational unit is assumed to consist of one linear array of cells and can be visualized as shown by Figure 17 although the three strings do not necessarily move in the same direction. Note that we do not assume any globally distributed signals besides the clock.
In a somewhat less puristic design the input data are allowed to be fanned out over several or maybe over all of the cells. However, this will violate some of the virtues of the cellular design of Figure 17: Complete modularity, no concern about delays in signal propagation (except for clock-skew) no extra boosters for high fan-out signals. From literature the four serial/parallel multipliers of Figure 18 are known [5], [10], [11], [12], [13], [14]. For simplicity, at this point we assume that all quantities are positive numbers.

From Figure 18a we see that by feeding the x-string, least significant bit $x_3$ first, we can produce the y-string serially. Figure 18b is a simple derivation of 18a by introducing extra delay elements in both the x- and the y-data path. Figure 18c on the other hand is completely different since only the x-string is delayed. Note that the ordering of the a-bits are reversed. Without any pipe-lining in the y-path we will suffer from long signal propagation times in Figure 18c. This defect is compensated for in Figure 18d where a delay is inserted for each cell in both x- and y-path.
Fig. 18
Actually, the serial/parallel multipliers of Figure 18 is only a few examples of a much larger family that hitherto seems to have escaped attention. For instance, one unknown family member is shown by Figure 19. The $x$-string and the $y$-string are entering/leaving from the same end. The delay elements are alternatively positioned in the upper and the lower path.

![Diagram](image)

We will now endeavour to identify the whole family of serial/parallel multipliers. For each of the three strings we define the velocities

$$v_x, v_a, v_y$$

to be the number of cell distances each string is displaced in one single time unit (clock cycle). Let us call the space axis along the linear array of cells the $z$-axis. For each string we have at each $z$ an index value.
that for any given instant \( t \) equals the weight (negative power of two = index) of the bit at cell position \( z \) along the array. We also define three \( \text{slopes} \)

\[ w_x, w_a, w_y \]

for the static strings that equals the increase of index per cell distance.

The basic equations are

\[ i_x(z,t) = w_x (z - v_x t) \quad (9) \]

\[ i_a(z,t) = w_a (z - v_a t) \quad (10) \]

\[ i_y(z,t) = w_y (z - v_y t) \quad (11) \]

Note that we for the moment treat these new variables as if they were defined in continuous space \( z \) and continuous time \( t \).

Actually the indices \( i_x(z) \), \( i_a(z) \) and \( i_y(z) \) are identical to the previously used indices \( n, k \) and \( d \) respectively in equations (2), (3) and (4). Three examples of moving strings are shown in Figure 19 which also introduces some notations that will be used in the following.

Now, as soon as the \( x \)-string overlaps the \( a \)-string over a cell \( z \) we will have a contribution to the \( y \)-string. The index number of this contribution equals the sum of \( a \)-index an \( x \)-index so that

\[ i_y(z,t) = i_x(z,t) + i_a(z,t) \quad (12) \]

Equation (12) must hold for all \( z \) and all \( t \). With identification of parameters in (9), (10) and (11) equation (12) gives us the basic relation between the string-defining parameters.
\[ w_y = w_a + w_x > 0 \]  
\[ v_y = \frac{w_a v_a + w_x v_x}{w_a + w_x} > 0 \]  
(13)  
(14)

For obvious reasons we must have \( v_y > 0 \). Otherwise the bits would stay for ever inside the array. Also, we must have \( w_y > 0 \). Otherwise a signal has to propagate over a long string of cells as in Figure 18c.

The speeds and the slopes of \( x \) and \( a \) has to meet the special criterion

\[ |v_x - v_a| < \left| \frac{1}{w_a} w_x \right| \]  
(15)

which has to do with the fact that the relative speeds of the two strings must not be higher than that all the bits of one string get a chance to combine with all the bits of the other string. Figure 20 illustrates a few cases of this problem. In fact, if the "strictly less than" condition of (15) is valid we will have a situation where several bits of one string combines with the same bit in in the other string more than once. To avoid such inefficiencies we sharpen the inequality to

\[ |v_x - v_a| = \left| \frac{1}{w_a} w_x \right| \]  
(16)

There is also a lower limit or the sum of the slopes

\[ |w_a| + |w_x| > 1 \]  
(17)

below which two neighboring cells may be doing the same computation. For example \( w_a = +1/2, w_x = +1/2 \) and \( w_a = +1/3, w_x = +2/3 \) meet the criterion (17), while \( w_a = +1/3, w_x = +1/3 \) does not.
For $v_a = 0$, i.e. a static string of constants we get from (13) and (14)

$$w_y \cdot v_y = w_x \cdot v_x$$

(18)

and from (16)

$$|v_x| = \frac{1}{w_a \cdot w_x}$$

(19)

Furthermore, the static string of a-bits must not conceal any of its bits to the logic cells. Nor must it display its bits to the full-adders in an uneven fashion. If one bit in the static a-vector is more exposed to the computational part than another we cannot be expected to do the job with a regular iterative structure. This implies that $w$ must have the form
\( w_a = 1/h \quad h = \pm 1, \pm 2, \pm 3, \ldots \) \hfill (20)

For instance
\[ w_a = 1/3 \quad \text{i.e. a 111222333 is acceptable} \]

while
\[ w_a = 2/3 \quad \text{i.e. a 112334556 is uneven and unacceptable} \]

like
\[ w_a = 3/2 \quad \text{i.e. a 134679} \]

Equations (19) and (20) combines into
\[
\begin{vmatrix} w_x \ y_x \end{vmatrix} = \begin{vmatrix} h \end{vmatrix} \hfill (21)
\]

The totality of equations and constraints with \( v \perp = 0 \) is
\[
w_y = w_a + w_x \gg 0 \hfill (13)
\]
\[
\begin{vmatrix} w_a \end{vmatrix} + \begin{vmatrix} w_x \end{vmatrix} > 1 \hfill (17)
\]
\[
w_y \cdot v_y = w_x \cdot v_x \hfill (18)
\]
\[
w_a = 1/h \quad h = \pm 1, \pm 2, \pm 3, \ldots \hfill (20)
\]
\[
\begin{vmatrix} w_x \cdot v_x \end{vmatrix} = \begin{vmatrix} h \end{vmatrix} \hfill (21)
\]

Note that the constant \( h \) equals the bit rate \( w \cdot v \) for the input string as well as for the output string.

Next section will present a whole catalogue of solutions to the equation system (13), (17), (18), (20) and (21).
7. A CATALOGUE OF SERIAL/PARALLEL-MULTIPLIERS

The complete catalogue can only be indicated by the following table since the number of solutions are unlimited. However, the solutions consist of rational numbers and the more complex the ratios are the less regular the implementation. The table contains all the simplest solutions plus some that are relatively regular. Solution 8 is not really acceptable since the x-signal has to be fanned out over the whole array \( w_x = 0 \). "Solution" 26 violates condition (17) and is included in the table to indicate that a negative slope \( w_x \) is not possible for \( h = +2 \).

Figure 21 shows implementations in a stylized form for a subset of solutions in the table. For \( h = +1 \) and \( h = -1 \) it is relatively easy to see the different solutions simply as movements of the delay elements in a given structure. Note that solution #4 was implemented already in Figure 19.

<table>
<thead>
<tr>
<th>Solution #</th>
<th>h</th>
<th>( w_a )</th>
<th>( w_y )</th>
<th>( v_y )</th>
<th>( w_x )</th>
<th>( v_x )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>+1</td>
<td>+1</td>
<td>0</td>
<td>( \infty )</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>2</td>
<td>+1</td>
<td>+1</td>
<td>+1/4</td>
<td>+4</td>
<td>-3/4</td>
<td>-4/3</td>
</tr>
<tr>
<td>3</td>
<td>+1</td>
<td>+1</td>
<td>+1/3</td>
<td>+3</td>
<td>-2/3</td>
<td>-3/2</td>
</tr>
<tr>
<td>4</td>
<td>+1</td>
<td>+1</td>
<td>+2/5</td>
<td>+5/2</td>
<td>-3/5</td>
<td>-5/3</td>
</tr>
<tr>
<td>5</td>
<td>+1</td>
<td>+1</td>
<td>+1/2</td>
<td>+2</td>
<td>-1/2</td>
<td>-2</td>
</tr>
<tr>
<td>6</td>
<td>+1</td>
<td>+1</td>
<td>+3/5</td>
<td>+5/3</td>
<td>-2/5</td>
<td>-5/2</td>
</tr>
<tr>
<td>7</td>
<td>+1</td>
<td>+1</td>
<td>+2/3</td>
<td>+3/2</td>
<td>-1/3</td>
<td>-3</td>
</tr>
<tr>
<td>8</td>
<td>+1</td>
<td>+1</td>
<td>+3/4</td>
<td>+4/3</td>
<td>-1/4</td>
<td>-4</td>
</tr>
<tr>
<td>9</td>
<td>+1</td>
<td>+1</td>
<td>+1</td>
<td>+1</td>
<td>0</td>
<td>( \infty )</td>
</tr>
<tr>
<td>10</td>
<td>+1</td>
<td>+1</td>
<td>+5/4</td>
<td>+4/5</td>
<td>+1/4</td>
<td>+4</td>
</tr>
<tr>
<td>11</td>
<td>+1</td>
<td>+1</td>
<td>+4/3</td>
<td>+3/4</td>
<td>+1/3</td>
<td>+3</td>
</tr>
<tr>
<td>12</td>
<td>+1</td>
<td>+1</td>
<td>+3/2</td>
<td>+2/3</td>
<td>+1/2</td>
<td>+2</td>
</tr>
<tr>
<td>13</td>
<td>+1</td>
<td>+1</td>
<td>+5/3</td>
<td>+3/5</td>
<td>+2/3</td>
<td>+3/2</td>
</tr>
<tr>
<td>14</td>
<td>+1</td>
<td>+1</td>
<td>+2</td>
<td>+1/2</td>
<td>+1</td>
<td>+1</td>
</tr>
<tr>
<td>15</td>
<td>+1</td>
<td>+1</td>
<td>+3</td>
<td>+1/3</td>
<td>+2</td>
<td>+1/2</td>
</tr>
<tr>
<td>16</td>
<td>-1</td>
<td>-1</td>
<td>+1/4</td>
<td>+4</td>
<td>+5/4</td>
<td>+4/5</td>
</tr>
<tr>
<td>17</td>
<td>-1</td>
<td>-1</td>
<td>+1/3</td>
<td>+3</td>
<td>+4/3</td>
<td>+3/4</td>
</tr>
<tr>
<td>h</td>
<td>w_a</td>
<td>w_y</td>
<td>v_y</td>
<td>w_x</td>
<td>v_x</td>
<td></td>
</tr>
<tr>
<td>----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>-1</td>
<td>-1</td>
<td>+2/5</td>
<td>+5/2</td>
<td>+7/5</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>-1</td>
<td>-1</td>
<td>+1/2</td>
<td>+2</td>
<td>+3/2</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>-1</td>
<td>-1</td>
<td>+2/3</td>
<td>+3/2</td>
<td>+5/3</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>-1</td>
<td>-1</td>
<td>+3/4</td>
<td>+4/3</td>
<td>+7/4</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>-1</td>
<td>-1</td>
<td>+1</td>
<td>+1</td>
<td>+2</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>-1</td>
<td>-1</td>
<td>+5/4</td>
<td>+4/5</td>
<td>+9/4</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>-1</td>
<td>-1</td>
<td>+3/2</td>
<td>+2/3</td>
<td>+5/2</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>-1</td>
<td>-1</td>
<td>+2</td>
<td>+1/2</td>
<td>+3</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>+2</td>
<td>+1/2</td>
<td>+1/4</td>
<td>+8</td>
<td>-1/4</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>+2</td>
<td>+1/2</td>
<td>+1</td>
<td>+2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>+2</td>
<td>+1/2</td>
<td>+4/3</td>
<td>+3/2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>+2</td>
<td>+1/2</td>
<td>+3/2</td>
<td>+4/3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>+2</td>
<td>+1/2</td>
<td>+2</td>
<td>+1</td>
<td>+3/2</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>+2</td>
<td>+1/2</td>
<td>+5/2</td>
<td>+4/5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>-2</td>
<td>-1/2</td>
<td>+1/2</td>
<td>+4</td>
<td>+1</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>-2</td>
<td>-1/2</td>
<td>+1</td>
<td>+2</td>
<td>+3/2</td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>-2</td>
<td>-1/2</td>
<td>+3/2</td>
<td>+4/3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>-2</td>
<td>-1/2</td>
<td>+2</td>
<td>+1</td>
<td>+5/2</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>-2</td>
<td>-1/2</td>
<td>+5/2</td>
<td>+4/5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>-2</td>
<td>-1/2</td>
<td>+3</td>
<td>+2/3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>+3</td>
<td>+1/3</td>
<td>+1</td>
<td>+3</td>
<td>+2/3</td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>+3</td>
<td>+1/3</td>
<td>+4/3</td>
<td>+9/4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>+3</td>
<td>+1/3</td>
<td>+2</td>
<td>+3/2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>-3</td>
<td>-1/3</td>
<td>+2/3</td>
<td>+9/2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>-3</td>
<td>-1/3</td>
<td>+1</td>
<td>+3</td>
<td>+4/3</td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>-3</td>
<td>-1/3</td>
<td>+5/3</td>
<td>+9/5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>-3</td>
<td>-1/3</td>
<td>+2</td>
<td>+3/2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>+4</td>
<td>+1/4</td>
<td>+1</td>
<td>+4</td>
<td>+3/4</td>
<td></td>
</tr>
<tr>
<td>46</td>
<td>+4</td>
<td>+1/4</td>
<td>+5/4</td>
<td>+16/5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>47</td>
<td>+4</td>
<td>+1/4</td>
<td>+2</td>
<td>+5/4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>-4</td>
<td>-1/4</td>
<td>+3/4</td>
<td>+16/3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>49</td>
<td>-4</td>
<td>-1/4</td>
<td>+1</td>
<td>+4</td>
<td>+5/4</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2lb

Figure 18d

Figure 21c

Figure 21d
Solution \#2 \[ \begin{align*}
  w_0 &= 1 \\
  v_0 &= 0 \\
  y &= \frac{1}{2} \\
  y &= \frac{3}{2} \\
  w_x &= -\frac{3}{2} \\
  v_x &= -\frac{3}{2}
\end{align*} \]

| t = 0 | x | \begin{tabular}{c}
  0
\end{tabular} | \begin{tabular}{c}
  1
\end{tabular} | \begin{tabular}{c}
  2
\end{tabular} | \begin{tabular}{c}
  3
\end{tabular} | \begin{tabular}{c}
  4
\end{tabular} | \begin{tabular}{c}
  5
\end{tabular} | \begin{tabular}{c}
  6
\end{tabular} |
|-------|---|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| y | \begin{tabular}{c}
  0
\end{tabular} | \begin{tabular}{c}
  1.2
\end{tabular} | \begin{tabular}{c}
  3.9
\end{tabular} | \begin{tabular}{c}
  6.5
\end{tabular} | \begin{tabular}{c}
  9.3
\end{tabular} | \begin{tabular}{c}
  12.5
\end{tabular} | \begin{tabular}{c}
  15.9
\end{tabular} |

| t = 1 | x | \begin{tabular}{c}
  0.4
\end{tabular} | \begin{tabular}{c}
  4.3
\end{tabular} | \begin{tabular}{c}
  8.9
\end{tabular} | \begin{tabular}{c}
  13.4
\end{tabular} | \begin{tabular}{c}
  18.5
\end{tabular} | \begin{tabular}{c}
  23.8
\end{tabular} |
| y | \begin{tabular}{c}
  0.8
\end{tabular} | \begin{tabular}{c}
  3.2
\end{tabular} | \begin{tabular}{c}
  6.7
\end{tabular} | \begin{tabular}{c}
  10.6
\end{tabular} | \begin{tabular}{c}
  14.7
\end{tabular} | \begin{tabular}{c}
  19.6
\end{tabular} | \begin{tabular}{c}
  24.7
\end{tabular} |

| t = 2 | x | \begin{tabular}{c}
  0.8
\end{tabular} | \begin{tabular}{c}
  4.8
\end{tabular} | \begin{tabular}{c}
  9.7
\end{tabular} | \begin{tabular}{c}
  15.4
\end{tabular} | \begin{tabular}{c}
  20.9
\end{tabular} | \begin{tabular}{c}
  26.7
\end{tabular} |
| y | \begin{tabular}{c}
  1.6
\end{tabular} | \begin{tabular}{c}
  5.6
\end{tabular} | \begin{tabular}{c}
  10.6
\end{tabular} | \begin{tabular}{c}
  15.6
\end{tabular} | \begin{tabular}{c}
  20.6
\end{tabular} | \begin{tabular}{c}
  26.6
\end{tabular} | \begin{tabular}{c}
  32.6
\end{tabular} |

| t = 3 | x | \begin{tabular}{c}
  1.2
\end{tabular} | \begin{tabular}{c}
  5.2
\end{tabular} | \begin{tabular}{c}
  10.2
\end{tabular} | \begin{tabular}{c}
  17.0
\end{tabular} | \begin{tabular}{c}
  23.8
\end{tabular} | \begin{tabular}{c}
  30.7
\end{tabular} |
| y | \begin{tabular}{c}
  2.8
\end{tabular} | \begin{tabular}{c}
  5.6
\end{tabular} | \begin{tabular}{c}
  10.6
\end{tabular} | \begin{tabular}{c}
  16.3
\end{tabular} | \begin{tabular}{c}
  22.3
\end{tabular} | \begin{tabular}{c}
  29.3
\end{tabular} | \begin{tabular}{c}
  36.6
\end{tabular} |

Solution \#3 \[ \begin{align*}
  w_0 &= -1 \\
  v_0 &= 0 \\
  y &= \frac{1}{2} \\
  y &= \frac{3}{2} \\
  w_x &= \frac{3}{2} \\
  v_x &= \frac{3}{2}
\end{align*} \]

| t = 0 | x | \begin{tabular}{c}
  0.3
\end{tabular} | \begin{tabular}{c}
  2.3
\end{tabular} | \begin{tabular}{c}
  1.8
\end{tabular} | \begin{tabular}{c}
  0.7
\end{tabular} | \begin{tabular}{c}
  2.7
\end{tabular} | \begin{tabular}{c}
  0.6
\end{tabular} |
| y | \begin{tabular}{c}
  0.3
\end{tabular} | \begin{tabular}{c}
  2.3
\end{tabular} | \begin{tabular}{c}
  1.8
\end{tabular} | \begin{tabular}{c}
  0.7
\end{tabular} | \begin{tabular}{c}
  2.7
\end{tabular} | \begin{tabular}{c}
  0.6
\end{tabular} |

| t = 1 | x | \begin{tabular}{c}
  0.6
\end{tabular} | \begin{tabular}{c}
  4.8
\end{tabular} | \begin{tabular}{c}
  3.6
\end{tabular} | \begin{tabular}{c}
  2.6
\end{tabular} | \begin{tabular}{c}
  4.6
\end{tabular} | \begin{tabular}{c}
  3.6
\end{tabular} |
| y | \begin{tabular}{c}
  0.6
\end{tabular} | \begin{tabular}{c}
  4.8
\end{tabular} | \begin{tabular}{c}
  3.6
\end{tabular} | \begin{tabular}{c}
  2.6
\end{tabular} | \begin{tabular}{c}
  4.6
\end{tabular} | \begin{tabular}{c}
  3.6
\end{tabular} |

| t = 2 | x | \begin{tabular}{c}
  0.9
\end{tabular} | \begin{tabular}{c}
  7.2
\end{tabular} | \begin{tabular}{c}
  5.4
\end{tabular} | \begin{tabular}{c}
  4.4
\end{tabular} | \begin{tabular}{c}
  6.4
\end{tabular} | \begin{tabular}{c}
  5.4
\end{tabular} |
| y | \begin{tabular}{c}
  0.9
\end{tabular} | \begin{tabular}{c}
  7.2
\end{tabular} | \begin{tabular}{c}
  5.4
\end{tabular} | \begin{tabular}{c}
  4.4
\end{tabular} | \begin{tabular}{c}
  6.4
\end{tabular} | \begin{tabular}{c}
  5.4
\end{tabular} |

| t = 3 | x | \begin{tabular}{c}
  1.2
\end{tabular} | \begin{tabular}{c}
  9.6
\end{tabular} | \begin{tabular}{c}
  7.8
\end{tabular} | \begin{tabular}{c}
  6.8
\end{tabular} | \begin{tabular}{c}
  8.8
\end{tabular} | \begin{tabular}{c}
  7.8
\end{tabular} |
| y | \begin{tabular}{c}
  1.2
\end{tabular} | \begin{tabular}{c}
  9.6
\end{tabular} | \begin{tabular}{c}
  7.8
\end{tabular} | \begin{tabular}{c}
  6.8
\end{tabular} | \begin{tabular}{c}
  8.8
\end{tabular} | \begin{tabular}{c}
  7.8
\end{tabular} |

---

Fig 21a

Fig 21b
Solution #30 \[ \begin{align*}
  w_o &= \frac{1}{2} \\
  v_o &= 0 \\
  w_y &= +2 \\
  v_y &= +1 \\
  w_x &= +\frac{3}{2} \\
  v_x &= +\frac{3}{2}
\end{align*} \]

<table>
<thead>
<tr>
<th>t=0</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>.5</td>
<td>.5</td>
<td>.5</td>
<td>.5</td>
<td>.5</td>
<td>.5</td>
</tr>
<tr>
<td>y</td>
<td>.5</td>
<td>.5</td>
<td>.5</td>
<td>.5</td>
<td>.5</td>
<td>.5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>t=1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>.6</td>
<td>.6</td>
<td>.6</td>
<td>.6</td>
<td>.6</td>
<td>.6</td>
</tr>
<tr>
<td>y</td>
<td>.6</td>
<td>.6</td>
<td>.6</td>
<td>.6</td>
<td>.6</td>
<td>.6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>t=2</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>y</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>t=3</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>y</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

Solution #32 \[ \begin{align*}
  w_o &= -\frac{1}{2} \\
  v_o &= 0 \\
  w_y &= -\frac{1}{2} \\
  v_y &= 4 \\
  w_x &= +1 \\
  v_x &= +2
\end{align*} \]

<table>
<thead>
<tr>
<th>t=0</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>y</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>t=1</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>y</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>t=2</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>y</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

Fig. 21c

Fig. 21d
8. THE SERIAL/PARALLEL CONVOLVER

The serial/parallel multiplier is in fact a convolution of two
bit-strings in the sense that

\[ y_i = \sum_{k=0}^{K-1} x_{i-k} a_k + \text{carries} \]

\[ y_{i+1} = \sum_{k=0}^{K-1} x_{i+1-k} a_k + \text{carries} \]

etc.

This fact indicates that the whole serial/parallel concept can be
carried over to the higher level of our computational problem. In
other words we can compute the sums

\[ y(i) = \sum_{\lambda=1}^{L} x(i) A_{\lambda} = \sum_{\lambda=1}^{L} x(i-\lambda) A_{\lambda} \]

\[ y(i+1) = \sum_{\lambda=1}^{L} x(i+1) A_{\lambda} = \sum_{\lambda=1}^{L} x(i+1-\lambda) A_{\lambda} \]

etc.

with a structure on the word level that repeats the
serial/parallel structure of the bit level. We call such
implementations serial/parallel convolvers. This entirely new idea
will be examined below.

In Figure 22 is shown the same schemes as in Figure 21a, the only
difference being that the bit-multiplying AND-gates and bit delay
elements of Figure 21a are replaced by serial/parallel multipliers
(SP) and word delays (D) respectively.
It should be noted that we are completely free to use one solution from the above catalogue for the serial/parallel multipliers and a completely different one for the convolver on the word level. However, by utilizing our knowledge about the internal structure of the S/P units in Figure 22 we can serialize the whole convolver as shown by Figure 23.

The linear array consists of identical cells and the S/P multipliers are imbedded in this structure at equidistant intervals. The generality of this procedure is further emphasized by the following theorem.
Theorem:

A (long) serial/parallel multiplier of any type having sufficient number of cells can be used for computation of a convolution sum by placing the constants $A_1, A_2, \ldots, A_L$ as bands at equidistant positions surrounded by bands of 0's. The total number of cells per pitch is $D \cdot h$ where $D$ is the number of bits in the output variable and $h$ is the number of times each bit in the constants is replicated.

Proof:

See Figure 24 for a visualization of the essence of the theorem. The input variables $X$ of this example are moving to the right twice as fast as the output variables $Y$.

While passing $A_1$, $Y_3$ is incremented by the amount $A_1 \cdot X_2$. The distance to $A_2$ should be such that when $X_2$ enters this non-zero band, $Y_2$ should have reached the same position as $Y_3$ had relative to $A_1$. Assume that $Y$ contains $D$ bits.

Typically, $D = \log L + K + N$ and the length of the $Y$-string in terms of cell units is $D/w_y$. From Figure 24 we conclude that

$$ (v_x - v_y) \cdot t = D/w_y \quad (22) $$

and

$$ t = \frac{D}{w_y v_x - w_y v_y} = \text{elapsed time} $$

for the $X$-vector to move from one computation stage to the next. From (13), (17) (18), (20) and (21) we get

$$ v_x \cdot t = \frac{D}{w_y v_x - w_y v_y} = \frac{D}{w_y - w_x} = D/w_a = D \cdot h $$

which proofs the theorem.
Figure 25 shows several examples of events that illustrates the theorem, the important corollary being that any serial/parallel multiplier of sufficient length can be used as a full convolver by allowing for correct amount 0-space between the coefficient bands. The cellular hardware is absolutely modular and can be extended indefinitely. It is programmed for a certain problem size \( L \cdot K \cdot N \) as soon as the coefficient bits are loaded into their positions.
\[ Y_1 = A_1 \cdot X_0 + A_2 \cdot X_1 + A_3 \cdot X_2 \]
\[ Y_2 = A_1 \cdot X_1 + A_2 \cdot X_2 + A_3 \cdot X_3 \]
\[ Y_3 = A_1 \cdot X_2 + \ldots \]

\[ X_0 = \frac{1}{2} \]

\[ w_0 = w_A = +1 \]
\[ w_y = w_Y = 3 \]
\[ w_x = w_X = +2 \]

Solution #14

Fig. 25a
\[ Y_1 = A_1 \cdot X_0 + A_2 \cdot X_1 + A_3 \cdot X_2 \]
\[ Y_2 = A_1 \cdot X_1 + A_2 \cdot X_2 + A_3 \cdot X_3 \]
\[ Y_3 = \ldots \]

\[ K_0 = \frac{1}{2} \]

\[ w_0 = w_A = \pm 1 \]
\[ w_y = w_y = \frac{3}{4} \]
\[ w_x = w_x = -\frac{1}{4} \]

Solution #7

Fig. 25b
Since the word limits for the static preloaded constants $A$ as well as for the moving variables $Y$ and $X$ are no longer fixed, the cells of the programmable array has to be slightly more complex than the simple cell of Figure 16. See Figure 26.
The y-signal is accompanied by a "clear carry" signal which at the beginning of each Y-word resets a carry that may have been set to 1 (which happens if the preceding Y-word was negative). The x-signal is accompanied by a word limit signal that defines a certain bit to be the sign bit. In Figure 26 x and y are travelling in opposite directions since we are using solution # 4. However, the basic cell design is the same for all serial/parallel multipliers as long as 2-complements representation is assumed. The verification of the design through formal proofs and examples are left out for the sake of brevity. Similar cell designs are found e.g. in [13] and [14].

Fig. 26
Now, in a case where the convolver does not have to be programmable for different problem sizes one would like to compress the whole structure and avoid the waste of space taken up by the bands of 0-bits between the coefficients.

Figure 27a illustrates that it is possible to compress the convolver of Figure 25a in this manner. However, we will now take a closer look at what happens in general when the 0-bands are cut out. Let

\[ w_A, w_X, w_Y \] (with upper case subscripts)

denote the slopes proper for the word-strings A, X and Y respectively. As can be seen from Figures 24 and 25 before cutting out the 0-bands these slopes are the same as the corresponding slopes at the bit-level. Note that the cell distance on the word level is the pitch from one multiplier to the next.

Without 0-bands we have the slopes

\[ w_A = w_a \] (23)

\[ w'_X = w_X \frac{K}{D} \]

\[ w_Y = w_y \frac{K}{D} \] (24)

since the pitch has decreased from D to K.

\[ w'_X \] is not what we want since \( w_A + w'_X = w_Y \) does not hold. The correct slope \( w_X \) is

\[ w_X = w_Y - w_A = w_Y \cdot \frac{K}{D} - w_A = w_X + w_Y \left( \frac{K}{D} - 1 \right) \] (25)

This means that the slope of the X-string has to be modified with a relative amount that is

\[ w_X/w'_X = D/K + w_Y/w_X \left( 1 - D/K \right) \] (26)
Fig. 27a

\[ Y_j = A_1 \cdot X_0 + A_2 \cdot X_1 + A_3 \cdot X_2 \]

\[ Y_2 = A_1 \cdot X_1 + \ldots \]

Fig. 27b

\[ Y_0 = A_1 \cdot X_0 + A_2 \cdot X_1 + A_3 \cdot X_2 \]

\[ Y_1 = A_1 \cdot X_1 + A_2 \cdot X_1 + A_3 \cdot X_2 \]

Fig. 27c

\[ Y_1 = A_1 \cdot X_0 + A_2 \cdot X_1 + A_3 \cdot X_2 \]

\[ Y_2 = A_1 \cdot X_1 + A_2 \cdot X_2 + A_3 \cdot X_3 \]
For the transformation from 25a) to 27a) we get

\[ \frac{w_x}{w'_x} = 2 + 3/2(1 - 2) = 1/2 \]

which "explains" why the x-path takes shortcuts over half of the multiplier bands. Figure 27b) and c) demonstrate the generality of the principle.

In most cases it would seem more appealing to accelerate the x-path via short-cuts (as in Figure 27a) rather than decelerate by adding extra delay elements as in Figure 27b and 27c. Acceleration takes place if

\[ |w'_x| > |w_x| \]  

(27)

For \( w'_x > 0 , w_x > 0 \) (25), (27) combines to

\[ w_x K/D > w_y K/D - w_a = (w_x + w_a)K/D - w_a \]

or

\[ w_a (1 - K/D) > 0 \]

For \( w'_x < 0 , w_x < 0 \) we get

\[ w_a (1 - K/D) < 0 \]

and for \( w'_x > 0 , w_x < 0 \) we need acceleration for

\[ w_x K/D - w_a + w_y K/D > 0 \]  

(28)

When the left hand side of inequality (28) equals 0 we have a kind of perfect situation where no extra delay elements have to be inserted, nor do we have to accelerate any x-signals. This occurs for a very small number of cases. For \( K/D = 1/2 \) it occurs uniquely for \( w = 3/2, w = 1/2 \) (solution # 11) and the resulting configuration is shown in Figure 28.
9. CONCLUSIONS

The attempt to investigate convolution implementations on the bit level led us to the illustrative schemes of Figure 2 where the three indices gave us six possible permutations. From this we first found a systematic way to implement convolvers with adders and multipliers. The so-called distributed arithmetic is also neatly explained as the index permutation \((n, xk)\), Figure 2d: Take all the bits \((k = 0, 1, \ldots, K-1)\) of each constant and do all the combinations over the kernel \((x = 1, 2, \ldots, L)\). Then traverse the bit index \(n\). It is our belief that this point of view is very clarifying.
Pipe-lining in the extreme is the theme of the second half of this paper. The schemes of Figure 2 are applicable also in this case as long as we use a two-dimensional array structure. However, it seems that the bit-serial one-dimensional arrays are more suited for VLSI-designs. We started by investigating serial/parallel multipliers and were able to capture the interplay between the moving bitstrings in a set of equations and inequalities. Then, we showed that the basic relations for the serial/parallel multiplier could be used for designing the convolver itself with the different multipliers embedded in one single linear structure. Hereby, a totally modular, very fast, highly programmable, extremely pin-saving VLSI-design can be obtained.

One price to be paid for the programmability is that half of the structure is idle. By giving up the programmability feature in the last section of the paper we achieved a more compact design, introducing a certain amount of shortcuts or delay elements. However, for any given wordlength relation K/D there is a "perfect solution" that allows the structure to be folded and the data to flow in a highly regular manner.

10. ACKNOWLEDGEMENT

This paper was originally conceived and written in June 1981 while the author was a visiting scientist with IBM Research Division, San Jose, CA 95193.

11. REFERENCES


