Synthesis of Area-Efficient VLSI Architectures for Vector and Matrix Multiplication

S. G. Smith
P. B. Denyer

University of Edinburgh
Department of Electrical Engineering
King's Buildings
Mayfield Rd.
Edinburgh EH9 3JL
Scotland

Abstract

A methodology is presented for synthesis of area-efficient, high-performance VLSI modules for vector and matrix multiplication. Three fundamental computational elements are employed in the composition of these architectures: memory register, multiplexer (1-from-2 data selecter), and carry-save add-shift computer. Two's complement serial/parallel carry-save accumulation provides performance, while the use of symmetric-coded distributed arithmetic eliminates redundant computation to effect area-savings.

1. Introduction

Digital signal processing is a prominent applications area which stands to benefit from VLSI realisations of high-performance integer arithmetic. Computation of vector and matrix products - fundamental operations in modern signal processing - is usually executed on dedicated arrays of scalar multiply/accumulators. This functional partitioning is forced on the designer by available standard parts.

We demonstrate that an alternative functional partitioning, using symmetric-coded distributed arithmetic (DA), exists to scalar multiplication in the VLSI computation of vector products, bringing demonstrable area savings with no loss of performance. DA provides the facility to compute the sum of several products concurrently, in architectures which exhibit the same structure, regularity and modularity as do scalar multipliers. In fact a scalar multiplier is a trivial case of a DA architecture.

Index of abbreviations

In the interests of brevity, the following acronyms and abbreviations are used:

CSAS	carry-save add-shift	**PP**	partial product
DA	distributed arithmetic	**PPS**	partial product sum
DSP	digital signal processing	**PIP**	partial inner-product
IP	inner-product	**PIPS**	partial inner-product sum
LS	least-significant	**PISO**	parallel-in-serial-out register
LSB	least-significant bit	**SIPO**	serial-in-parallel-out register
MS	most-significant	**S/P**	serial/parallel
MSB	most-significant bit	**VLSI**	very large scale integration
OB	offset binary	**2C**	two's complement

2. Overview of vector computation

A multiplication is an unconstrained 1-D sum of (weighted) PPs. The dimension represents one of the two input operands - the weight of whose bits is a function of dimension index. We refer to the input operands as *data* and *coefficient* respectively, and note two common differences of usage in DSP applications. Firstly, the coefficient is often known *a priori* (unlike data which either arrive from external sources or are freshly-derived from previous computations). Secondly, the precision of representation may differ (the coefficient is often represented by fewer bits). In S/P architectures, one computational dimension (usually the data-bit index) lies along the time axis.

A fundamental form of vector product is the *dot*, or *inner* product (IP) [1]. The IP of two vectors is formed by summing the pairwise products of the vector elements. An IP is then an unconstrained two-dimensional sum of PPs (the 2nd dimension being vector length). Once again, we note the difference of usage between data and coefficient (in this case vectors). By permuting and/or factoring summation indices, several different approaches are made possible. Classical multiply-accumulate techniques [2] put the vector index outermost, while DA [3, 4] has the data-bit index outermost. By factoring the index of vector length, architectures may be realised which yield an optimal mixture of these two techniques.

Matrix-vector multiplication extends the list of favourable properties associated with coefficients, in that there are often further properties of symmetry in the coefficient matrix to be exploited. We shall demonstrate these advantageous properties in later examples.

2.1. Some carry-save approaches to vector computation

The properties of carry-save arithmetic seem particularly well suited to the computation of vector products, i.e. the unconstrained summation of single-bit products in three dimensions. Bit-level systolic arrays [5] are an efficient means of implementing such architectures. Here computation is executed concurrently on processor arrays tailored to the problem, and communication between processors remains local.

Denyer and Myers [6] proposed arrays of carry-save adders which accumulated inner-products across each bit plane in bit-parallel carry-save fashion (MS-plane first), using the free inputs of the next-plane computer for accumulation. Cappello & Steiglitz [7] formalised this concept. Danielsson noted that families of convolvers could correspond to S/P multipliers, as convolution has the same structure at word-level as multiplication has at bit-level [8].

These architectures follow good VLSI practice, and result in area-efficient realisations of vector computers. However they exploit neither symmetries and redundancies in computation, nor preknowledge of coefficients, as do distributed arithmetic architectures. Although a combination of techniques might yield interesting results, we eschew the systolic architectures, in deference to the bit-serial 'building-block' approach [9, 10].

3. Two's complement serial/parallel multiplication

We begin our development of matrix-vector architectures by discussing the S/P multiplier. We will demonstrate that a slightly-modified version of this well-known structure may serve as an architectural basis for vector and matrix computations.

S/P multipliers in their simplest form represent a 'collapsed' mapping of the basic CSAS parallel rectangular array multiplier [11] into a linear CSAS array (one spatial dimension becoming temporal). Coefficient bits are distributed *spatially* (in parallel fashion), and data bits *temporally* (in LSB-first serial fashion). Hardware consists mostly of a linear array of m gated CSAS cells (each with a resident coefficient bit), which forms the product by shifting accumulation of PPs. Assuming the m-bit coefficient is available in parallel form (this is easily arranged should the coefficient be in serial form and known m clock cycles in advance [12]), multiplication of n-bit serial data can be executed in n clock cycles. The LS n bits of the $n + m - 1$ bit product are output from the right-hand side of the array in serial form.

Unfortunately, after the final computational step the top $m - 1$ bits of the product remain as residual sums and carries along the array (as in the parallel case), and must somehow be recombined to form the true product. One way round this is to pack $m - 1$ sign-repetitions on the input data [12] - the full $n + m - 1$ bits of the product then appear in serial form and the parallel residues may be discarded. The harmful effects of incorrect 2C data-MSB interpretation do not propagate to the serial output. Although the $m - 1$ sign-repetitions represent a high overhead in terms of either throughput (words per second) or dynamic range (useful bits per word), in situations where such an overhead may be tolerated, this approach is commendable - the resultant hardware (dubbed the S/P 'flush' multiplier [13]) is compact and fast.

The 'fractional' S/P multiplier [14] (where the data input and product output formats are identical) requires extra hardware for 2C data-MSB treatment, and for recombination of the product from residual carry-save form into a single 2C word. While the techniques presented in this paper are applicable to any multiplication architecture (serial or parallel), we choose the S/P flush multiplier as demonstration vehicle. Fig. 1 shows the 2C coded S/P flush multiplier.

3.1. Internal details

As sum and carry outputs are latched on each clock cycle, shifting is accomplished by simple cell abutment. While PPS signals are transferred to the neighbouring cell, 'saved' carry signals recirculate locally - weight is consistent with the increased weight of the following PP-bit. Use of the carry-save technique avoids carry propagation delays, resulting in high potential throughput. The AND-gating on the CSAS cell top-input is for PP-formation, and on others for clearing the accumulator (sum and carry) on commencement of a product calculation. A control signal (low to mark data LSB, else high) performs the latter function.

Depending on the broadcast data bit, each PP is either the coefficient word, or zero (PPS weighting results from the shifting operation of the accumulator). Note that PP-formation may be pipelined to improve performance. To comply with 2C bit-weighting in the coefficient, the MS-cell subtracts the local PP-bit. Being the leading boundary processor of the array, it has a free input which may absorb a second data operand to support multiply-addition. Logical modularity may be improved (at the expense of this free input) by installing an adder with fed-back sum in the leading stage [14] - this structure acts as a 2C negator. The multipliers illustrated (Figs. 1 & 3) use the latter negation strategy.

3.2. External issues

We are concerned mainly with internal details of these architectures, but make the following observations on operational environments. S/P structures find application in both parallel-data and serial-data architectures.

Parallel-data operation

In the former, datapath environment (Fig. 2(a)), one of the input operands enters a PISO to be broadcast to the array. When computation is completed, the carry-save product residue loads a parallel adder to be merged into a single word (this may be overlapped with the next product calculation). Here computation is 'fractional' - the m sign-extensions are not present. However the final (MS) PP must be subtracted to account for 2C MSB-weighting in the data - a control signal and some extra circuitry must somehow be provided for this purpose.

Serial-data operation

In the latter case (Fig. 2(b)), the coefficient must enter a SIPO in advance of data (computation cannot commence until coefficient bits are resident in their respective locations). On completion, the parallel sum and carry residues may be discarded, as the full-precision product appears in serial form at the end of the array. For fractional operation (i.e. no data sign-extensions), the sum and carry residues must load a double PISO, to be merged in a bit-serial adder to form high-order product bits. Once again, extra hardware and a control signal are required to effect subtraction of the MS-PP.

For the sake of simplicity, but with no loss of generality, we proceed on the assumption that environment is serial-data, data contain m sign-extensions, and coefficients reside in parallel registers. Thus the CSAS computer is hardware-minimal, at the cost in throughput of processing the m sign-extensions.

3.3. Free inputs

Along with the free serial input mentioned earlier, a favourable property of S/P multipliers is the free input-pair exhibited by each adder cell in the CSAS array. Normally the PPS and carry inputs are cleared on commencement of computation - however a pair of low-precision bit-parallel words may be inserted instead and accumulated 'on the fly'. This opens the possibility of 'free accumulation' and multi-precision operation [15], as intermediate results may be passed between modules *in carry save form*.

4. The symmetric-coded serial/parallel multiplier

Orthodox S/P multipliers, as just described, use 2C coding throughout. As we know of no better data-coding for the addition (accumulation) operation, this results in minimal hardware and maximal throughput. To assist in the development of matrix-vector architectures, we choose to code the data word in symmetric, offset-binary (OB) form, where logical 0 is interpreted as -1 [16]. This alters the role of data bits in the computation - logical 0 now effects the subtraction of the coefficient from the PPS, instead of the addition of zero.

Data conversion to OB is easily accomplished by MSB inversion. 2C data word A consisting of bits a_i converts to OB coded word A' as shown:

$$A = -a_0 + \sum_{i=1}^{n-1} a_i 2^{-i}, \qquad a_i \in \{0,1\} ,$$

$$A' = \sum_{i=0}^{n-1} a_i 2^{-i-1}, \qquad a_i \in \{-1,1\}$$

Error compensation by parallel load of the coefficient

It can be seen that

$$A' = A + 2^{-n} \qquad 1$$

i.e. a small representational error results from the change of code to OB.

Consider the product P of 2C-coded data word A with coefficient C, and product P' of OB-coded data word A' with coefficient C.

$$P = AC, \qquad P' = A'C$$
$$= AC + 2^{-n}C$$
$$= P + 2^{-n}C$$

We see that the change of code from 2C to OB results in a representational error of $2^{-n}C$, which may be removed by subtracting the coefficient word at LSB-time (on commencement of product computation). As the CSAS computer is incapable of explicit subtraction, this may be accomplished by adding the 2's complemented coefficient, i.e. by bit-inversion and incrementing (implicit subtraction). To this end, the inverted bits of the coefficient are used to load the free carry-loops in the main array at LSB-time (cf. clearing in the 2C version). The PPS input to the stage occupied by the coefficient LSB (i.e. the last stage) must be set at LSB-time to perform the necessary increment (all others are cleared).

As data is OB-coded, the coefficient word is either added to or subtracted from the PPS, depending on the broadcast data bit. Again, coefficient subtraction is performed implicitly (the incrementing bit is simply delayed, inverted data). Thus the OB S/P multiplier contains an XNOR-gate for bit-product formation (instead of the AND gates of the 2C version), and an extra CSAS cell at the end of the array for incrementing. Fig. 2 shows the OB S/P multiplier.

Note that the weight of data-bits is different between 2C and OB (the former is twice the latter). Care should be taken in aligning the addend input, and interpreting the weight of the coefficient and the output product.

Error compensation by left-shift and decrement

The above scheme necessitates loading of the coefficient into the carry loops at LSB-time. This causes a considerable increase of logical complexity in the basic computational cell, with corresponding area and performance costs. A second method of coding-error compensation cancels the data error directly, by pre-processing data bits before broadcast. Manipulation of eqn. 1 produces the expression:

$$A = A' - 2^{-n}$$

$$\text{i.e.} \quad 2A = 2A' - 2^{-n+1}$$

Thus preprocessing takes the form of a left-shift and decrement operation. Output data must be right-shifted for subsequent correct interpretation of the product.

This technique increases the latency of the OB multiplier, requires a guard-bit on input data thereby reducing dynamic range, and affects modularity in adverse manner. These points make it a less likely candidate for implementation. Pipelining of bit-product formation in the former scheme can be used to reduce cell complexity, leading to a more compact, modular architecture.

5. The serial/parallel inner-product computer

So far we have described a modification to the 2C S/P multiplier, allowing it to handle OB-coded data inputs. The price paid for this is the extra low-precision adder on the output of the main array, and increased cell complexity. However the S/P multiplier is now in the form where, with a little further modification [17], it can compute inner products directly, using DA.

DA [4] replaces the multiplications involved in an IP computation with a series of memory look-ups. A set of 'partial inner products' (PIPs) made by convolving the coefficient vector with all possible bit-patterns from the data vector is precomputed and stored in memory. The PIPs are accessed (addressed) by the actual bit-pattern across each bit-plane of the data vector, and accumulated (with the correct binary weight) to form the IP.

If data are coded in OB, the full PIP-set exhibits negative symmetry [16]. To exploit this property, we designate an arbitrary coefficient word as 'master', also referring to the data word associated with this coefficient in the IP computation as master. The master data-bit may then be removed from the address word and the memory-size halved. This bit instead serves as an 'add/subtract' instruction to the accumulator. Viewed in this light, the OB S/P multiplier contains a single-word 'memory', accessed by a 'zero-bit address word', i.e. look-up is trivial.

We now describe the conversion of the OB S/P multiplier into a 2-point IP computer. Instead of storing one coefficient word C, we introduce a second coefficient word D, and store the 2 PIPs K and K', where

$$K = \frac{C + D}{2}, \qquad K' = \frac{C - D}{2}$$

The factor of 2 prevents word-growth in PIPs, and compensates for the factor of 2 weight difference between OB and 2C data codes. We take 2 serial data words A and B as input, choosing (say) A as master. A is then broadcast as data to the CSAS array, and (A XNOR B) is used to select either K or K'. If the bits of A and B are equal in any bit-plane, the 'sum-PIP' K is selected - if unequal, the 'difference-PIP' K' is selected. If the 'master' bit is 1, the PIP is added, if not it is subtracted. Thus the modified S/P multiplier is capable of computing the inner-product step AB + CD + E (where E is the optional serial addend), at little extra hardware cost.

Through a simple, recursive procedure, this principle can be extended to compute longer IPs:

For each additional data-coefficient pair:

Replace each register with a multiplexer and register-pair,

Load the register-pair with the old PIP ± the new coefficient,

Select multiplexer output by XNOR of master data-bit and new data-bit.

Error compensation

Consider the IP $\sum P_x$ of 2C-coded data vector A_x with coefficient vector C_x, and product $\sum P'_x$ of OB-coded data vector A'_x with C_x.

$$\sum P_x = \sum A_x C_x, \qquad \sum P'_x = \sum A'_x C_x$$
$$= \sum (A_x C_x + 2^{-n} C_x)$$
$$= \sum P_x + 2^{-n} \sum C_x$$

Thus the 'sum-PIP' $\sum C_x/2$ loads the carry-loop on commencement to compensate for OB data-coding.

Fig. 3 summarises the evolution of this class of DA architectures via the recursive composition procedure, starting from the S/P multiplier. Some abstraction is necessary to contain detail - for instance we are not concerned with loading/unloading operations, carry-setting and cascadability (these are the same in all cases). We restrict ourselves to 3 architectural elements: registers, multiplexers and CSAS computers. The architectures are viewed 'end on', i.e. data flow is out of the page. Master bits are shown beside computers, while selection functions point at selecters.

Fig. 3(a) is the abstracted version of the S/P multiplier of Fig. 2, with a single register to hold coefficient word C and a single CSAS computer to form the product from data word A.

Fig. 3(b) shows the modification of the S/P multiplier to form a 2-point IP $A_1A_1 + A_2C_2$. We replace the coefficient register with a multiplexer and register-pair. Here A is master, and the function A_1 XNOR B_1 drives the multiplexer, which selects one of the PIPs (these are represented as $C_1 + C_2$ and $C_1 - C_2$, although PIPs are actually stored as half these values).

Fig. 3(c) shows the extension of this technique to form the 3-point IP $A_1A_1 + A_2C_2 + A_3C_3$. Here A_1 is master and A_2 and A_3 are used for data selection, by XNOR with A_1.

It should be noted however, that while savings in CSAS elements are linear, PIP storage costs grow exponentially. Depending on the technological implications of adding storage, a point will soon be reached where the DA approach is less attractive than the conventional [18]. For this reason, we propose a mixture of DA and conventional techniques for longer IP computations - this is effected by factoring the vector length index as described earlier.

6. Architectural case studies

Armed with the knowledge of how to construct IP computers, we may now review some of the matrix and vector architectures which can be synthesised with these techniques. Here we treat coefficients as known matrices operating on vectors of incoming data, exploiting where possible the properties of symmetry exhibited by these matrices.

Matrix-vector multiplication

The 2-point IP computation described above may be expressed in matrix notation as shown.

$$E = \begin{bmatrix} C & D \end{bmatrix} \begin{bmatrix} A \\ B \end{bmatrix}$$

While general matrix-vector computation of the form:

$$\begin{bmatrix} G \\ H \end{bmatrix} = \begin{bmatrix} C & D \\ E & F \end{bmatrix} \begin{bmatrix} A \\ B \end{bmatrix}$$

may be executed on a pair of unrelated 2-point IP computers, the centrosymmetric matrix computation:

$$\begin{bmatrix} E \\ F \end{bmatrix} = \begin{bmatrix} C & D \\ D & C \end{bmatrix} \begin{bmatrix} A \\ B \end{bmatrix}$$

may be performed on a simple variation of the architecture of Fig. 3(b) (the addition of a second CSAS computer). Fig. 4(a) depicts this architecture. Due to centrosymmetry, PIP-selection is mutually *inclusive*, i.e. the selected PIP is used in both computers. Note that B is master in the second computer, as B is associated with the master coefficient in the implicit IP computation which produces output F. While this structure finds limited application in DSP (hyperbolic rotation is one example of its use [19]), a further slight modification transforms it into a form of matrix-vector computer which is very common indeed.

Complex multiplication

The operation of complex multiplication (plane rotate and/or scale) occurs frequently in digital signal processing, for example in Fourier transformation [2], orthogonal filtering [20], and waveform generation [21]. Dedicated complex multipliers are relatively rare, and usually evaluate the complex product using four real multipliers, minimising communication and sharing storage of operands in parallel [22] or serial [23, 24] 2C architectures. Just as the FFT uses the commonality of coefficients to make computational savings over the DFT by combining before rotating, so it is possible to perform addition before multiplication to reduce the number of real multiplies in the complex multiplier. The 3-multiplier solutions of Golub and Buneman [25, 26] reduce computation - however these approaches increase storage costs and adversely affect dynamic range in bit-serial realisations [10].

The four carry-propagate adders required by the conventional bit-parallel approach may be reduced to two using 'merged' arithmetic [27]. Alternatively, a reduced form of binomial expansion may approximate the trigonometric functions [28], reducing the number of required shifts and adds to effect the transformation. However none of these approaches make full use of the cross-symmetry of operands to simplify calculation of the complex product.

CORDIC processors [29, 25] have been suggested as an alternative to complex multipliers as a vector rotation medium - however (like 2's complement dividers) they contain 'conditional' operations which hamper performance. Limited success has been achieved in pipelining CORDIC processors [30] - nevertheless CORDIC offers the flexibility to tackle computational areas such as advanced function generation and transformation [19].

In the more down-to-earth problem of vector plane-rotation, the most common solution is the complex multiplier with unity-modulus coefficient. White [31] suggested an area-efficient symmetric-coded distributed arithmetic solution to the complex multiplication problem, and embedded it in an FFT processor. This architecture has since reappeared in bit-parallel [32] and serial-pipeline [33] form. We may derive White's model from the previous centrosymmetric matrix-vector computer.

If we designate data word-pair A,B and coefficient word-pair C,D as real and imaginary components of a complex numbers $\boldsymbol{A}$ and $\boldsymbol{C}$ respectively, a further slight modification to the architecture of Fig. 4(a) may perform complex multiplication. The complex product $\boldsymbol{E} = \boldsymbol{CA}$, where $\boldsymbol{E}$ = E + jF may be evaluated by the matrix computation:

$$\begin{bmatrix} E \\ F \end{bmatrix} = \begin{bmatrix} C & -D \\ D & C \end{bmatrix} \begin{bmatrix} A \\ B \end{bmatrix}$$

Although the centrosymmetric property no longer holds, the coefficient matrix now displays the equally useful property of mutually *exclusive* PIP-selection [34]. Here the sum-PIP in the implied imaginary IP computation equals the difference-PIP in the implied real computation (and vice-versa). Instead of selecting one PIP via a multiplexer for use in both computers, we steer both PIPs through a commutator (2-from-2 data selecter).

Complex inner-product

We may express the previous computation in complex notation:

$$E = CA$$

and extend the above concept to the complex inner-product computation:

$$E = \begin{bmatrix} C & D \end{bmatrix} \begin{bmatrix} A \\ B \end{bmatrix}$$

In similar fashion to the real arithmetic case, we form an IP computer by removing the register of a multiplier, and replacing it with a register-pair and multiplexer. Here registers are complex, i.e. they contain *pairs* of numbers. Thus a 'register-pair' in this case comprises 4 real registers, while a 'multiplexer' is a 1-from-4 data selecter.

Extending the selection methods of the complex multiplier, the PIP-set may be split into a 'sum-set' and a 'difference-set', with mutually exclusive selection. However selection functions within these sets are complicated by the fact that selection *within sets* depends on the target computer. This follows from the asymmetry of the scalar coefficient matrix:

$$\begin{bmatrix} e \\ f \end{bmatrix} = \begin{bmatrix} c_1 & -d_1 & c_2 & -d_2 \\ d_1 & c_1 & d_2 & c_2 \end{bmatrix} \begin{bmatrix} a_1 \\ b_1 \\ a_2 \\ b_2 \end{bmatrix}$$

By judicious arrangement of PIP storage, we may at least ensure that selection functions are shared between PIP-sets. These functions are 'quasi-exclusive', so-called because, like the exclusive-OR and -NOR functions, they form diagonals in the Karnaugh map (Fig. 5). While exclusive functions form alternate diagonals, quasi-exclusive functions divide the map into two diagonal regions. Fig. 6 shows the 2-point complex IP computer, with selection functions appropriate to the PIP arrangement.

A 3-point complex computer may be formed by replacing each register with a 4-register-3-multiplexer combination, and so on. However register count and multiplexing depth begin to dominate beyond the 2-point complex IP, as does the fan-in (hence area cost) of selection functions.

7. Architectural synthesis

We have seen how 3 simple elements may be combined in different ways to form various architectures for matrix and vector computation. Building on a single CSAS array, the construction rules for IP computers are procedural and recursive. Matrix computation follows easily, by addition of further CSAS computers and association of master data and coefficient in each implicit IP computation.

Procedural construction lends itself readily to computer automation in a silicon compilation environment [10]. With construction rules encapsulated in composition procedures, we envisage assembly of DA architectures in response to a single-line call in high-level language, rather than explicit calls to component modules.

8. Comparison with conventional approaches

To illustrate the area savings afforded by the DA approach, we compare the example architectures with the standard approach (SA), where storage is shared whenever possible (Table 1). S/P hardware costs are O(m), where m is coefficient/PIP wordlength. We neglect O(1) costs.

Table 1: Comparison of standard approach and DA

function	register		mux		CSAS	
	SA	DA	SA	DA	SA	DA
2-point IP	2	2	-	1	2	1
3-point IP	3	4	-	3	3	1
2-pt. matrix-vector	2	2	-	1	4	2
complex multiply	2	2	-	1	4	2
2-pt. complex IP	4	8	-	7	8	2

The DA solution uses multiplexers and sometimes extra storage, but always less CSAS computers. The hardware costs of these elements vary with technology, nonetheless we suggest that the reduction in CSAS hardware afforded by the DA approach outweighs the increase in storage/selection costs over conventional approaches.

Although area-efficient, DA appears to be 'storage heavy'. However if bit-serial throughput enhancement techniques [15] are employed, then the ratio of active logic to storage increases, accentuating the area savings of the DA approach.

DA requires the generation of some logic functions of input data bit-planes, and extra wires for their broadcast. Word-growth may occur *in PIPs*, necessitating perhaps one or two extra stages to maintain the accuracy of conventional realisations. Also it should be noted that the DA approach requires 'precomputed' PIPs to be available. If coefficients can only be provided in standard form, then a network of bit-serial adders and subtracters must be provided for PIP calculation, with attendant time and area penalties.

9. Distributed arithmetic in context

DA was proposed as a method for avoiding the use of standard-part multipliers in FIR/IIR filtering [4, 35]. From the outset the two approaches were classed as diametrically opposite - they have been directly compared on several occasions, e.g. [36, 37]. This has led to a general perception of DA as a 'ROM-accumulator' technique, involving memory technology in VLSI realisations with the area overhead of control and addressing logic.

We have demonstrated that 'memory addressing' (at least for small problem sizes) in VLSI is merely a data-steering operation, governed by simple logical functions on the incoming data bit-planes. Computation is performed on CSAS arrays, exactly as done in multipliers. Thus DA is more a *modification* than a *replacement* of multiplier technology.

Complexity of DSP algorithms is often expressed in terms of 'multiplier count'. We suggest that hardware partitioning into registers and CSAS computers (rather than multipliers) coupled with the use of DA in synthesis might result in improved complexity analysis techniques.

10. Cascading

The concept of DA has been introduced, and its usefulness in the computation of short vector and matrix products demonstrated. In many cases, these structures will be used in long cascades, e.g. in the computation of vector inner products [1]. Cascading issues relate closely to those of the conventional approach, as we propose to sum globally over the index of vector length. The only difference is that we have factored this index and nested one of those factors inside the bit-index - the other is outermost as usual.

Recall that these structures have one free input in the serial (temporal) dimension, and two potentially free inputs in the parallel (spatial) dimension, all of which can find use in cascading. The carry-loop could be freed up in all but the initial DA processor of

the cascade, by lumping all the sum-PIPs local to each DA calculation into one global sum-PIP which represents the entire IP calculation. However there is only one free serial input, and as the addend word invariably extends up into the 'fractional' part of the addend, there is no obvious way to exploit the free carry-input. Thus local sum-PIPs will be loaded as before.

When cascading the flush IP computer, no free inputs may be exploited as the format of the output product is different from the input addend. That is not to say that the flush IP computer cannot be cascaded for computation of inner-products - on the contrary, it is best suited to such computation, if target data rates permit the computational inefficiencies resulting from the guard-bit requirement. Accumulation must be carried out on dedicated adders in this case.

Word growth can occur in long IP calculations - this may be accommodated in higher-order bit-serial accumulators [10]. Several architectural possibilities exist for inner-product calculations [38, 39], which have direct relevance to serial-data realisations. Often the choice is decided by the allowable transform latency - two FIR filtering case studies using the *FIRST* silicon compiler [10] (matched filtering and adaptive filtering) yielded markedly different multiplexed architectural solutions. The former architecture was a pipelined, forward flowing cascade, whilst the latter, minimal-latency architecture fanned-in sums through a binary addition tree [10].

Formal mechanisms for *specifying* cascaded inner-product architectures are presented in [10]. Systems designers must take many factors into account when making these specifications - these include tolerable transform latency, signal statistics, signal bandwidth (hence multiplexing scheme), accuracy requirements, etc. However a formal mechanism is also required for *implementing* cascaded inner-product architectures. We present two such mechanisms, one which exploits the free input and one which does not.

Free accumulation

We envisage a cascade of IP computers, with the double-precision product output from each connecting to the addend input of the subsequent. The low-order m bits of this word are accommodated in bit-parallel form, and the remaining $qn - m$ bits in multi-precision serial form. One extra SIPO is required to convert the low-order bits into parallel form for loading at the free parallel input. The operation of this SIPO is identical to the SIPO used for coefficient loading, except that in this case no holding register is required. Word-growth beyond the range of double-precision may be accommodated in higher-order serial-data adders.

Adder-based accumulation

Here we make no use of the free input - double-precision outputs are fed directly to a multi-precision accumulator. The action of this accumulator beyond the double-precision range is identical to its action in the free-accumulation structure. The flush IP computer finds application in this cascading environment.

11. Conclusions

A distributed arithmetic architecture for computation of small matrix-vector products has been described, and a general architectural methodology for matrix-vector computers outlined. The evolution of the architecture from a basis serial/parallel multiplier through real inner-product computers and the complex multiplier to complex inner-product computers has been charted, and a substantial reduction in computational hardware over conventional multiply/add solutions demonstrated. These modules are easily cascadable for longer calculations, and form ideal function library components for silicon compilation.

Acknowledgement

The authors acknowledge the support of the UK Science & Engineering Research Council.

References

1. E. E. Swartzlander Jr., B. K. Gilbert, and I. S. Reed, "Inner Product Computers," *Trans. IEEE* **C-27** pp. 21 - 31 (January 1978).

2. L. R. Rabiner and B. Gold, *Theory and Application of Digital Signal Processing*, Prentice-Hall (1975).

3. A. Croisier et al., *Digital Filter for PCM Encoded Signals*, U.S. Patent 3 777 130 (December 4, 1973).

4. A. Peled and B. Liu, "A New Hardware Realisation of Digital Filters," *Trans. IEEE* **ASSP-22** pp. 456 - 462 (December 1974).

5. J. G. McWhirter and J. V. McCanny, "Novel Multibit Convolver/Correlator Chip based on Systolic Array Principles," *SPIE Real-Time Signal Processing V* **341** pp. 66 - 73 (1982).

6. P. B. Denyer and D. J. Myers, "Carry-Save Adders for VLSI Signal Processing," pp. 151 - 160 in *VLSI '81*, ed. J. P. Gray, Academic Press (1981).

7. P. R. Cappello and K. Steiglitz, "A Note on 'Free Accumulation' in VLSI Filter Architectures," *Trans. IEEE* **CAS-32** pp. 291 - 296 (March 1985).

8. P. E. Danielsson, "Serial/Parallel Convolvers," *Trans. IEEE* **C-33** pp. 652 - 667 (July 1984).

9. R. F. Lyon, "A Bit-Serial VLSI Architectural Methodology for Signal Processing," pp. 131 - 140 in *VLSI 81*, ed. J. P. Gray, Academic Press (1981).

10. P. B. Denyer and D. Renshaw, *VLSI Signal Processing - A Bit-Serial Approach*, Addison-Wesley (1985).

11. S. Waser, "High-Speed Monolithic Multipliers for Real-Time Digital Signal Processing," *Computer* **11** pp. 19 - 29 (October 1978).

12. D. J. Myers and P. A. Ivey, "Circuit Elements for VLSI Signal Processing," *Br. Telecom Technol. J.* **2** pp. 67 - 77 (July 1984).

13. S. G. Smith and P. B. Denyer, "Serial/Parallel Architectures for Area-Efficient Vector Multiplication," *Proc. IEEE ICASSP'87*, (Dallas, TX, April 1987).

14. R. Gnanasekaran, "A Fast Serial-Parallel Multiplier," *Trans. IEEE* **C-34** pp. 741 - 745 (August 1985).

15. S. G. Smith, M. S. McGregor, and P. B. Denyer, "Techniques to Increase the Computational Throughput of Bit-Serial Architectures," *Proc. IEEE ICASSP'87*, (Dallas, TX, April 1987).

16. C. F. N. Cowan, S. G. Smith, and J. H. Elliott, "A Digital Adaptive Filter using a Memory-Accumulator Architecture: Theory and Realisation," *Trans. IEEE* **ASSP-31** pp. 541 - 549 (June 1983).

17. S. G. Smith, "Efficient Serial/Parallel Inner-Product Computation," *Electronics Letters* **22** pp. 750 - 752 (July 3, 1986).

18. S. A. White, "On Mechanization of Vector Multiplication," *Proc. IEEE* **63** pp. 730 - 731 (April 1975).

19. H. M. Ahmed, J. -M. Delosme, and M. Morf, "Highly Concurrent Computing Structures for Matrix Arithmetic and Signal Processing," *Computer* **15** pp. 65 - 82 (January 1982).

20. S. K. Rao and T. Kailath, "Orthogonal Digital Filters for VLSI Implementation," *Trans. IEEE* **CAS-31** pp. 933 - 945 (November 1984).

21. S. G. Smith, "Modelling Musical Instruments in the Digital Domain," *Proc. IEEE ICASSP'84*, pp. 19.7.1 - 19.7.4 (San Diego, March 1984).

22. J. Deverell, "Multiplication of Complex Numbers Using Iterative Arrays," *Electronics Letters* **7** pp. 205 - 207 (6th May, 1971).

23. R. W. Linderman et al., "CUSP: a 2-μm CMOS Digital Signal Processor," *J. IEEE* **SC-20** pp. 761 - 769 (June 1985).

24. A. F. Murray and P. B. Denyer, "A CMOS Design Strategy for Bit-Serial Signal Processing," *J. IEEE* **SC-20** pp. 746-753. (June 1985).

25. A. M. Despain, "Fourier Transform Computers using CORDIC Iterations," *Trans. IEEE* **C-23** pp. 993 - 1001 (October 1974).

26. P. S. Moharir, "Extending the Scope of Golub's Method Beyond Complex Multiplication," *Trans. IEEE* **C-34** pp. 484 - 487 (May 1985).

27. E. E. Swartzlander Jr., "Merged Arithmetic," *Trans. IEEE* **C-29** pp. 946 - 950 (October 1980).

28. R. F. Eschenbach and B. M. Oliver, "An Efficient Coordinate Rotation Algorithm," *Trans. IEEE* **C-27** pp. 1178 - 1180 (December 1978).

29. J. E. Volder, "The CORDIC Trigonometric Computing Technique," *Trans. IRE* **EC-8** pp. 330 - 334 (August 1959).

30. E. D. Deprettere, P. Dewilde, and R. Udo, "Pipelined Cordic Architectures for Fast VLSI Filtering and Array Processing," *Proc. IEEE ICASSP'84*, pp. 41A.6.1 - 41A.6.4 (San Diego, March 1984).

31. S. A. White, "A Simple FFT Butterfly Arithmetic Unit," *Trans. IEEE* **CAS-28** pp. 352 - 355 (April 1981).

32. I. R. Mactaggart and M. A. Jack, "A Single Chip Radix-2 FFT Butterfly Architecture Using Parallel Data Distributed Arithmetic," *J. IEEE* **SC-19** pp. 368 - 373 (June 1984).

33. S. G. Smith and P. B. Denyer, "Efficient Bit-Serial Complex Multiplication and Sum-of-Products Computation Using Distributed Arithmetic," *Proc. IEEE-IECEJ-ASJ ICASSP'86*, pp. 2203 - 2206 (Tokyo, April 1986).

34. S. G. Smith, "Serial/Parallel Modules for Complex Arithmetic," *Electronics Letters* **22** pp. 1256 - 1257 (November 6, 1986).

35. C. S. Burrus, "Digital Filter Structures Described by Distributed Arithmetic," *Trans. IEEE* **CAS-24** pp. 674 - 680 (December 1977).

36. T. A. C. M. Claasen, W. F. G. Mecklenbrauker, and J. B. H. Peek, "Some Considerations on the Implementation of Digital Systems for Signal Processing," *Philips Res. Repts.* **30** pp. 73 - 84 (1975).

37. H. J. De Man, C. J. Vandenbulcke, and M. M. Van Cappellen, "High-Speed NMOS Circuits for ROM-Accumulator and Multiplier Type Digital Filters," *J. IEEE* **SC-13** pp. 565 - 572 (October 1978).

38. D. Cohen, "Mathematical Approach to Iterative Computational Networks," *Proc. 4th IEEE Symp. on Computer Arith.*, pp. 226 - 238 (Santa Monica, CA, October 1978).

39. H. T. Kung, "Why Systolic Architectures?," *Computer* **15** pp. 37 - 46 (January 1982).

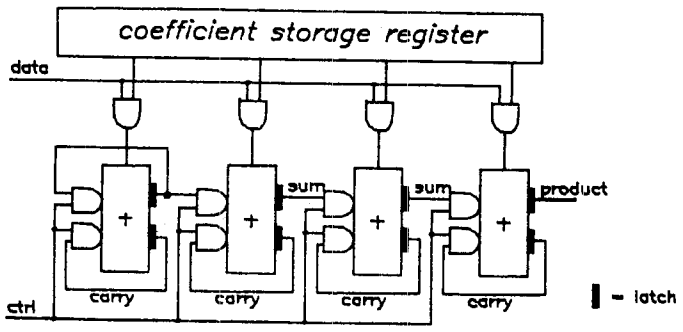

Figure 1: S/P multiplier with 2C data-coding

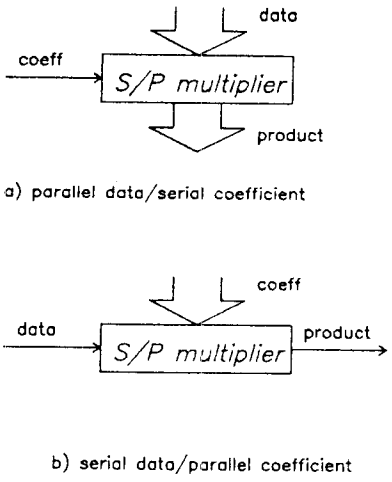

Figure 2: Operational environments of S/P multiplier

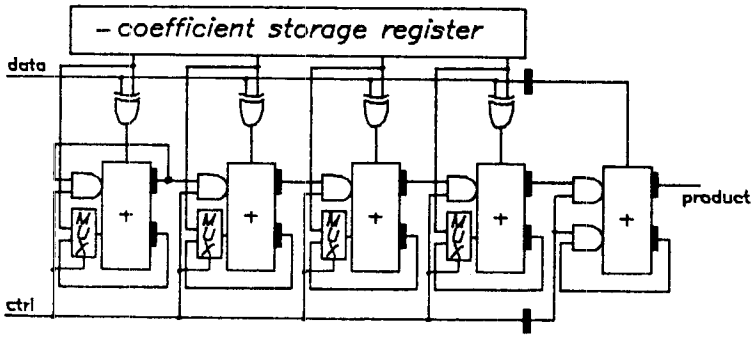

Figure 3: S/P multiplier with OB data-coding

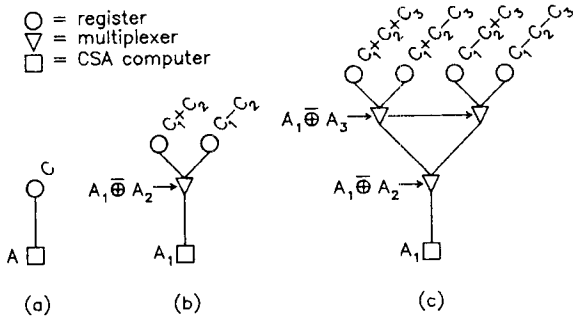

Figure 4: real IP computer evolution

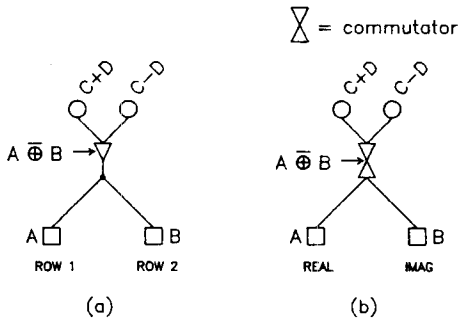

Figure 5: matrix-vector computer evolution

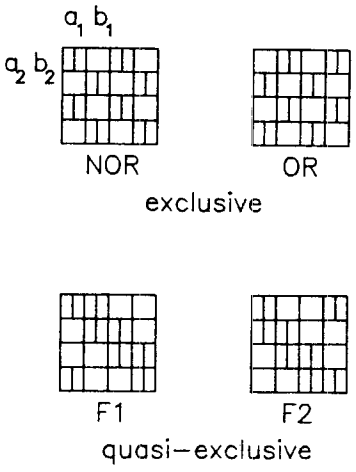

Figure 6: Karnaugh maps of exclusive and quasi-exclusive functions

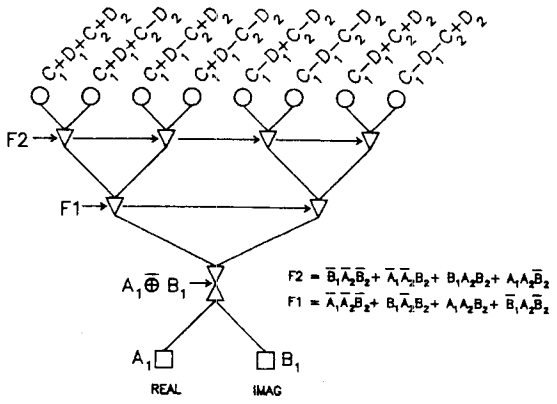

Figure 7: complex IP computer evolution

*** ERRATUM ***
Figs. 3 - 7 are wrongly named in main text as Figs. 2 - 6.