The Unification of Systolic Differencing Algorithms

G. M. MEGSON AND D. J. EVANS*

Department of Computer Studies, Loughborough University of Technology, Loughborough, Leicestershire, LE11 3TU

A templating method for the fast derivation of systolic arrays is presented and discussed in relation to differencing formulae and similar problems which generate tabular representations. Individual designs can be optimised and generate a full table in* O(n) *rather than the* O(n²) *operations (where* n *is the number of starting values). Finally the designs are incorporated into a single array, i.e. the Unified Systolic Array for Differencing (USAD), which can be used as a cheap chip-based 'add-on' device to accelerate algorithms involving table generation and differencing.

Received March 1986, revised July 1986

1 INTRODUCTION

In this paper systolic arrays for computing the coefficient tables for commonly used difference techniques are considered. The simplicity of the formulae such as the finite (forward and backward) differences, as well as divided and reciprocal differences together with their recurrent computational nature makes them well suited to a VLSI and systolic design approach. For introductory material on the systolic concept and array design the reader is referred to Refs 1, 2, 3 and 4.

In Refs 5 and 6 the authors have developed area-efficient and computationally fast Systolic Arrays for Extrapolation techniques for Ordinary Differential Equations and Romberg integration; there it was shown that a certain correlation exists between table generation and the matrix computations heavily used in systolic arrays. From our correlations it is possible to develop the concept of templating arrays. The main idea of an array template is to generate a sequence of designs for computationally related problems by using a global method of calculation, which freezes the abstract definition of the systolic array at a high level and significantly reduces array design time.

The subject of this paper is twofold. Firstly it defines templates for the problem of differencing algorithms, indicating that the previous results, i.e. Refs 5 and 6, are special cases of a more general systolic structure. Secondly, we illustrate the method by developing a sequence of simple array designs for commonly used difference methods and indicate that all the designs can be unified to fit a single systolic array, the Unified Systolic Array for Differencing (USAD).

1.1. Array templates

Before producing new systolic arrays for differencing functions we briefly review the techniques and results of Refs 5 and 6 for extrapolation table generation. Essentially the designs for extrapolation obeyed a few simple rules developed from the structure of the tables and a method indicated for the parallel evaluation of the table elements as shown in Fig. 1, for a type of template for all differencing or table-type algorithms, and consist of the following.

(1) An ordering of the table elements for the parallel evaluation of the table, and a computational rule relating elements in the partially constructed table to unknown elements. (Usually a column is defined in terms of columns to the left.)

(2) A linear array is defined with basic cells mapped to a column in the table, cell (i) computing column $T^{(i)}$. In Fig. 1(*a*) we derive the array of Fig. 1(*b*) with *n* cells ($n = 6$), each cell implementing the computational rule.

(3) A class of arrays for partial or full table generation, i.e.

(*a*) a linear array with a single fan-in link for generating the diagonal entries of a table only.

(*b*) A linear array (Fig. 1(*c*)) generating the full table, each cell outputting all the elements of a column.

(*c*) A systolic ring which reduces the number of cells to a minimum while leaving the computation time unchanged, leading to an area-efficient array for generating diagonal table entries.

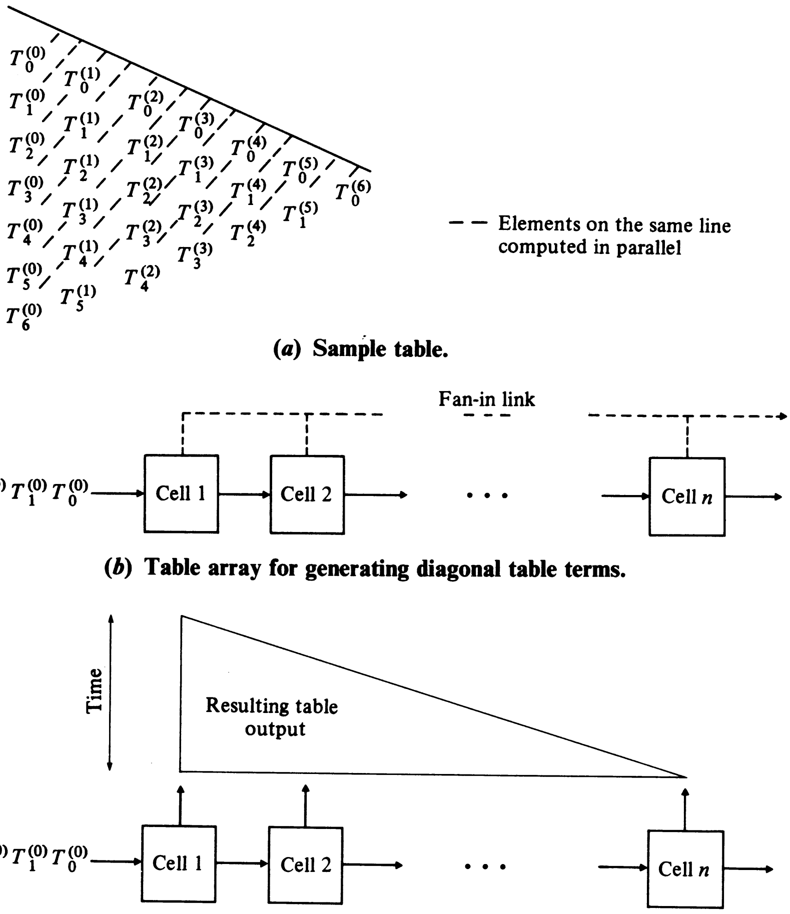
(4) A generic timing of

$T =$ (number of inputs) + (delay through the array)
$T = (n+1) \qquad\qquad + c*n$

which follows simply from the fact that there are $(n+1)$ starting values in column $T^{(0)}$ and *n* cells in the linear array. Also, *c* is the latency of each cell, which is basically the number of cycles between input to a cell and a corresponding output associated with the input element. Clearly *c* varies for the complexity of the computational rule. When $c > 1$, we can save area by using a systolic ring. Essentially, in a systolic ring we notice that after *n* cycles the last input enters the first cell of the array, but the first input has only penetrated as far as the $\lceil n/c \rceil$ cell. On successive cycles the cells at the start of the array complete all their computations and are essentially idle. We remove the last $n - \lceil n/c \rceil$ cells and wrap the output of the $\lceil n/c \rceil$th cell back to the first cell. Details of this are given in Refs 5 and 6. This results in the same computation time with only $\lceil n/c \rceil$ cells. Consequently we define a cell area in terms of a basic component of the computational rule calculation, and a cycle time equivalent to the cost of evaluating this component. The array is then clocked at this cycle time. Thus, as long as *c* is small we require only $O(n)$ cells and $O(n)$ cycles to compute the whole table. In previous designs a cycle has been at most the cost of an inner product step ($y = y + a*x$) satisfying all the assumptions above.

We now apply these techniques to the derivation of some simple differencing or table-generating algorithms. The key point to notice is that the array structures remain

* To whom correspondence should be addressed.


$T_0^{(0)}$ $T_0^{(1)}$ $T_0^{(2)}$ $T_0^{(3)}$ $T_0^{(4)}$ $T_0^{(5)}$ $T_0^{(6)}$
$T_1^{(0)}$ $T_1^{(1)}$ $T_1^{(2)}$ $T_1^{(3)}$ $T_1^{(4)}$ $T_1^{(5)}$
$T_2^{(0)}$ $T_2^{(1)}$ $T_2^{(2)}$ $T_2^{(3)}$ $T_2^{(4)}$
$T_3^{(0)}$ $T_3^{(1)}$ $T_3^{(2)}$ $T_3^{(3)}$
$T_4^{(0)}$ $T_4^{(1)}$ $T_4^{(2)}$
$T_5^{(0)}$ $T_5^{(1)}$
$T_6^{(0)}$
— — Elements on the same line computed in parallel
Fan-in link
... $T_2^{(0)} T_1^{(0)} T_0^{(0)}$
Cell 1
Cell 2
...
Cell n
Time
Resulting table output

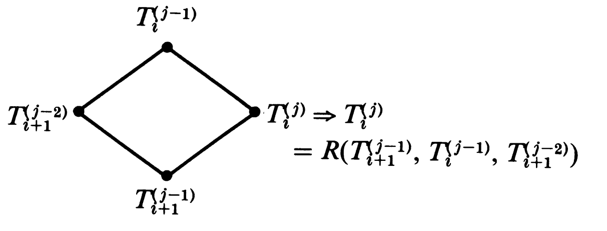
(*a*) **Sample table.**

(*b*) **Table array for generating diagonal table terms.**

(*c*) **Full table generation.**

Figure 1. Template for table generation.

essentially static, and only the design of a good basic cell using the computational rule is required. We shall denote the computational rule as a function R defining the cell; for instance, a rectangle rule of the form


$T_i^{(j-1)}$
$T_{i+1}^{(j-2)}$
$T_i^{(j)} \Rightarrow T_i^{(j)}$
$= R(T_{i+1}^{(j-1)}, T_i^{(j-1)}, T_{i+1}^{(j-2)})$
$T_{i+1}^{(j-1)}$

gives rise to a cell for computing the rule (e.g. Burlisch and Stoer extrapolation).

2. DIFFERENCE ALGORITHMS

In this section we introduce new arrays for common differencing techniques such as the forward and backward differences, divided differences and rational function approximation. All the tables presented can also be used to extract coefficient data for the construction of polynomials $P(x)$ and rational function approximation $R(x)$ of a given function $y(x)$. Many formulas are available for this, such as the Newton and Gauss forward/backward difference formulae as well as the continued fraction representation for rational functions. It is the simplicity and universal application of these methods which requires generating their coefficients (or part of them) by fast systolic arrays, which is important.

2.1. Finite difference arrays

Given a discrete function, that is a set of arguments x_k and a corresponding value y_k such that the arguments are equally spaced by the distance $h = x_{k+1} - x_k$, the difference operator Δ is defined as

first difference

$$\Delta y_k = y_{k+1} - y_k$$

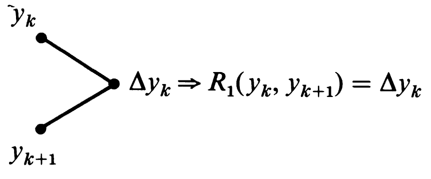
second difference

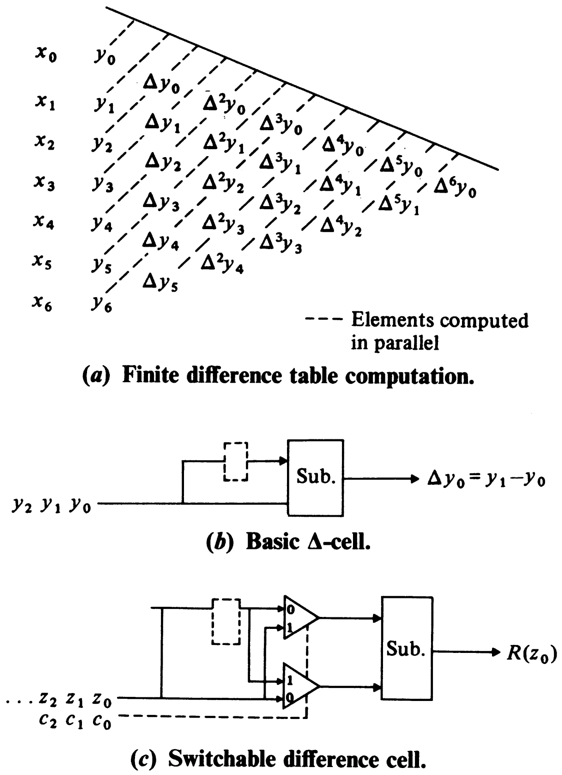
$$\Delta^2 y_k = \Delta(\Delta y_k) = \Delta y_{k+1} - \Delta y_k = y_{k+2} - 2y_{k+1} + y_k$$

and generally

$$\Delta^n y_k = \Delta^{n-1} y_{k+1} - \Delta^{n-1} y_k.$$

This gives rise to the table template in Fig. 2(*a*), and the cell function R is derived from the simple computational rule




x_0 x_1 x_2 x_3 x_4 x_5 x_6
y_0 y_1 y_2 y_3 y_4 y_5 y_6
Δy_0 Δy_1 Δy_2 Δy_3 Δy_4 Δy_5
$\Delta^2 y_0$ $\Delta^2 y_1$ $\Delta^2 y_2$ $\Delta^2 y_3$ $\Delta^2 y_4$
$\Delta^3 y_0$ $\Delta^3 y_1$ $\Delta^3 y_2$ $\Delta^3 y_3$
$\Delta^4 y_0$ $\Delta^4 y_1$ $\Delta^4 y_2$
$\Delta^5 y_0$ $\Delta^5 y_1$
$\Delta^6 y_0$
--- Elements computed in parallel
$y_2\ y_1\ y_0$
Sub.
$\Delta y_0 = y_1 - y_0$
$\ldots z_2\ z_1\ z_0$
$c_2\ c_1\ c_0$
0
1
1
0
Sub.
$R(z_0)$

(*a*) **Finite difference table computation.**

(*b*) **Basic Δ-cell.**

(*c*) **Switchable difference cell.**

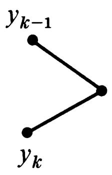
Figure 2. Finite difference template.

This in turn produces the cell for the arrays of Fig. 1, as shown in Fig. 2(*b*). The Δ-cell consists of only a single subtractor and a delay cell for synchronisation. We choose a cycle time of τ_1 = cost of subtraction. Hence, when y_0 is input it gets delayed a cycle by the delay register then synchronises with y_1 on the next cycle in the subtractor giving $c = 2\tau_1$. Normalising the cycle time to eliminate τ_1 gives the result

$$T = (n+1)+2n = 3n+1.$$

Also the systolic ring consists of $\lceil n/2 \rceil$ Δ-cells, approximately $\lceil n/2 \rceil$ subtractors.

Similarly we can define backward differences (∇) by the simple computational rule,



$$\nabla y_k \Rightarrow R_2(y_{k-1}, y_k) = \nabla y_k = y_k - y_{k_1}$$

which results in a similar cell to Fig. 2(*b*) and retains the same computation time. This is expected, as the two methods only have a minor difference in the order of computation, which is a simple reversal of the starting data input. This is convenient because it gives trivial methods which can be used to illustrate the method of unification. The cell functions R_1 and R_2 (in a mathematical sense) can be understood as defining a new computational rule R_3 for a new cell, which is a unified cell for computing both methods on the same architecture. We denote this as $R_3(R_1,(y_k, y_{k+1}), R_2(y_{k-1}, y_k)) = R_3(z)$ such that,

$$R_3(z_j) = \begin{cases} \Delta y_k & c_j = 1 \\ \nabla y_k & c_j = 0 \end{cases} j = 1(1)n+1$$

and

$$z_j = \begin{cases} y_j & c_j = 1 \\ y_{n-j+1} & c_j = 0 \end{cases} j = 1(1)n+1$$

The basic cell is shown in Fig. 2(*c*) and uses a switching control $c_j, j = 1(1)n$ to switch the order of the operands input to the subtractor, depending on whether R_1 or R_2 is computed. Notice also that the input for R_2 is reversed, and the new cell involves only a trivial amount of extra switching hardware, which essentially minimises the amount of extra hardware to implement the two designs by a single unified array.

2.2. Divided differences

The differencing algorithms defining the current template arrays have a significant drawback, for they assume equally spaced arguments. In order to fit the templating concept to more general table generators and hence for wider applications we need to consider unequally spaced arguments. For purposes of illustration, these unequally spaced arguments will take the form of divided differences and are defined as follows:

first divided differences

$$y(x_0, x_1) = \frac{y_1 - y_0}{x_1 - x_0}$$

second divided differences

$$y(x_0, x_1, x_2) = \frac{y(x_1, x_2) - y(x_0, x_1)}{x_2 - x_0}$$

higher differences

$$y(x_0, x_1, \dots, x_n) = \frac{y(x_1, \dots, x_n) - y(x_0, \dots, x_{n-1})}{x_n - x_0}$$

which would produce a template function of the form,

$$R_4(y(x_1, \dots, x_i), (y(x_0, \dots, x_{i-1}), x_0, x_i) = y(x_0, \dots, x_i)$$

and presents a further problem for generating a parallel computation order for the table elements. Essentially we have the arguments x_i represented explicitly in the cell function. Further, for two arguments x_i and x_j in R_4, successive j values with i fixed increase the distance between the elements. However, for a systolic point of view, as the distance between indices of elements increases with values in the same cell, dataflow problems also increase. The reason is simple; systolic arrays are based on nearest-neighbour connections and perform well when data elements are related only to elements with locally available indices (i.e. $i+1$, $i-1$, $i+2$, $i-2$, etc.). Attempting to use the first template of Fig. 1 for unequally spaced arguments is disastrous, and indicates that equally spaced arguments can make the explicit use of arguments implicit (see Ref. 7). Fig. 4 indicates a more general template for table algorithms which maps on to the more intuitive parallel computation of column sweep or the generation of a whole column in parallel. Fig. 4(*b*) indicates the cell for R_4. The array is linear and is shown in Fig. 5 with the new dataflow. The cell consists of two subtractors, a divider and two pre-loadable registers as well as a delay for x_{in}.

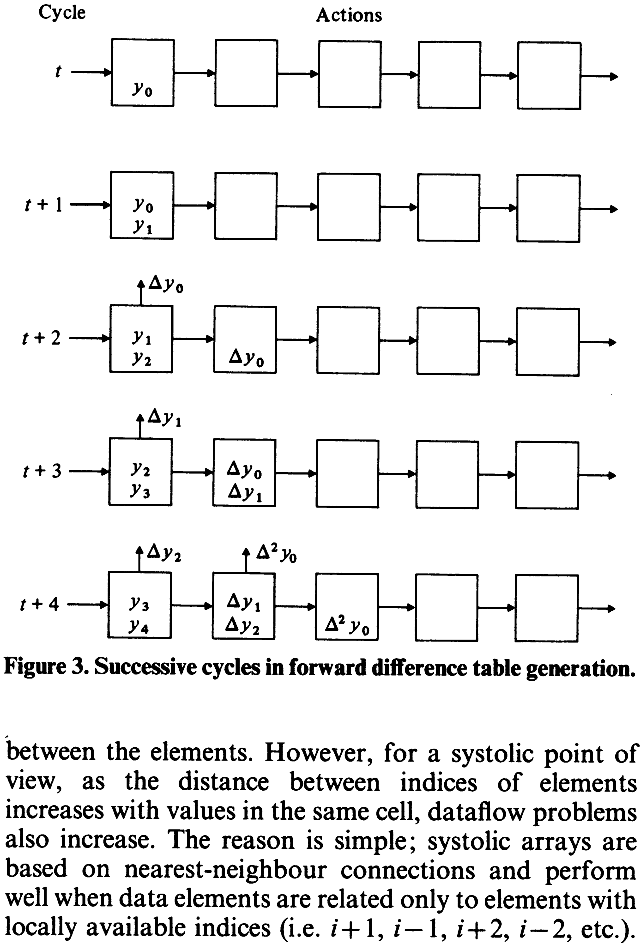


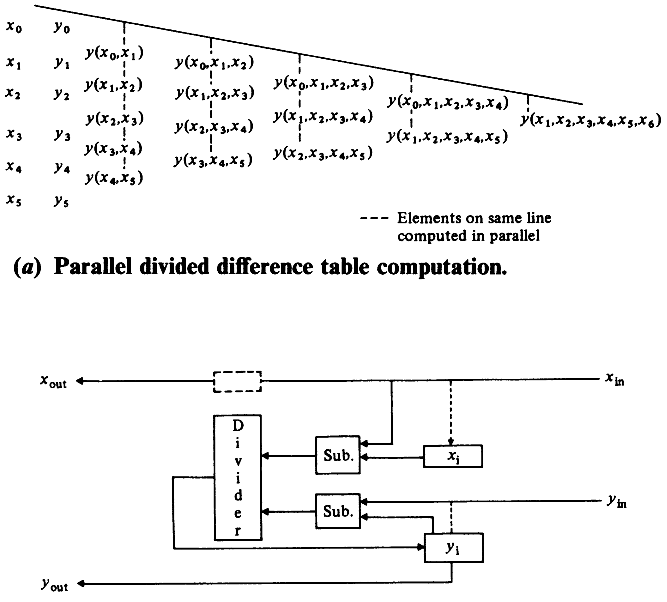
Figure 3. Successive cycles in forward difference table generation.

The cell operation is divided into two parts, i.e.

(*a*) pre-load cell i with x_i and y_i the starting values;

(*b*) compute the next column element as follows: (i) evaluate $x_{\text{in}} - x_i$, and $y_{\text{in}} - y_i$ in parallel; (ii) compute the divided difference using the divider, and overwrite y_i with the new divided difference.

Thus for a linear array of cells we still require n cells, which are pre-loaded before the computation starts. On the successive cycles after pre-loading, the array computes one complete column of the table every cycle, with each cell contributing a single element to the column. Hence we can seen that the template definition of Section 1 remains essentially correct, we have introduced a new table evaluation order, and, as we shall show later, the use of systolic rings is prohibited due to changes in the dataflow.



(*a*) **Parallel divided difference table computation.**

(*b*) **Divided difference cell.**

Figure 4. Divided difference template ---- lines used for pre-loading.

The timing of the divided-difference arrays is given by

$$T = (\text{pre-load time}) + (\text{time through array}) = n + c * n.$$

Notice that the pre-load time replaces the length of input, which is of the same order of magnitude due to the geometry of the table. It follows that $c = \tau_1 + \tau_2$, where τ_2 = cost of divide. So $T = 2n$ after normalising the cycle time c. Notice that the cycle time must be equivalent to c because of the feedback by y_i. It follows that $c = 1$ and so no benefit is gained from a systolic ring even if it could be used. In any event a systolic ring is not practical because of the initial pre-loading and subsequent data movement. It should also be clear that for divided differences the new cell computes forward differences and backward differences by making the $x_{\text{in}} - x_i$ subtractor always produce a value 1. We shall show how to perform this when we present the unified array. However, a comparison with the finite difference array gives

$$T_1 = 3\tau_1 n \quad \text{(finite difference)}$$

$$T_2 = 2(\tau_1 + \tau_2) n \quad \text{(divided difference)}$$

Hence the dedicated finite-difference algorithm and array will be $[2(\tau_1 + \tau_2)/3\tau_1]$ faster than the unified array. (*Remark*. An alternative way of viewing this new template is that the first method allocated each cell to compute a single column, whereas the new method

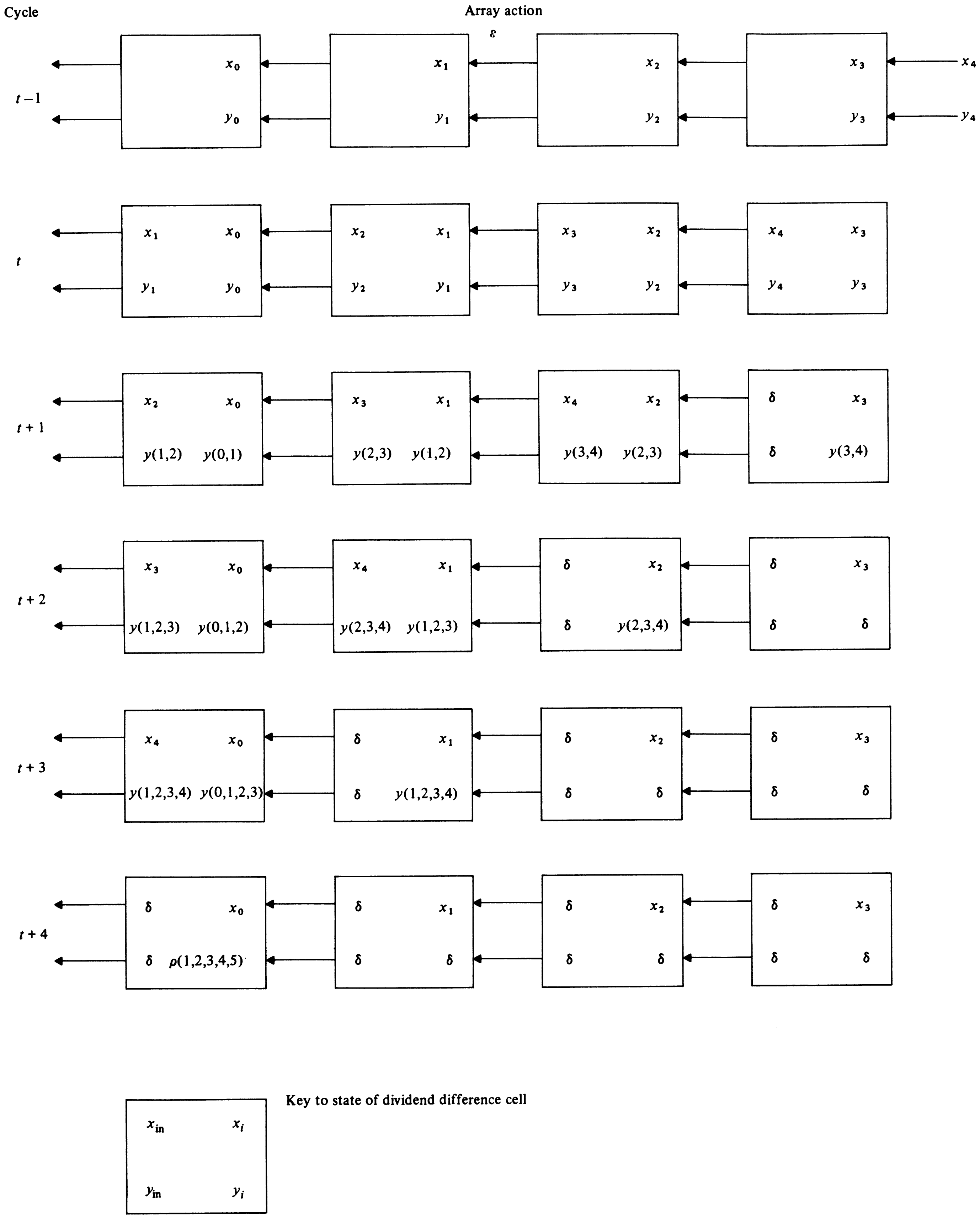
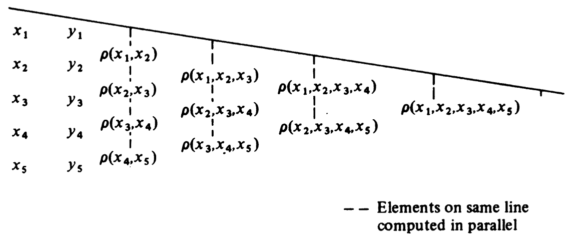
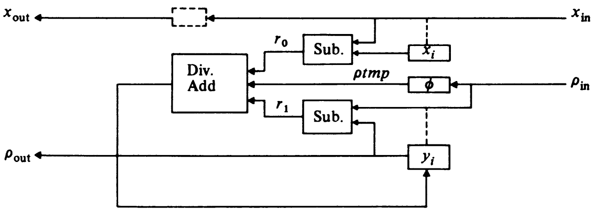

Cycle
Array action
ε
$t-1$
x_0 x_1 x_2 x_3 x_4
y_0 y_1 y_2 y_3 y_4
t
x_1 x_0 x_2 x_1 x_3 x_2 x_4 x_3
y_1 y_0 y_2 y_1 y_3 y_2 y_4 y_3
$t+1$
x_2 x_0 x_3 x_1 x_4 x_2 δ x_3
$y(1,2)$ $y(0,1)$ $y(2,3)$ $y(1,2)$ $y(3,4)$ $y(2,3)$ δ $y(3,4)$
$t+2$
x_3 x_0 x_4 x_1 δ x_2 δ x_3
$y(1,2,3)$ $y(0,1,2)$ $y(2,3,4)$ $y(1,2,3)$ δ $y(2,3,4)$ δ δ
$t+3$
x_4 x_0 δ x_1 δ x_2 δ x_3
$y(1,2,3,4)$ $y(0,1,2,3)$ δ $y(1,2,3,4)$ δ δ δ δ
$t+4$
δ x_0 δ x_1 δ x_2 δ x_3
δ $p(1,2,3,4,5)$ δ δ δ δ δ δ
x_{in} x_i
y_{in} y_i
Key to state of dividend difference cell

Figure 5. Dataflow/computation in divided difference array.



(*a*) Reciprocal difference table generation.



(*b*) Reciprocal difference cell.

Figure 6. Reciprocal difference template.

assigns each cell the job of computing a single row throughout the table.)

2.3. Reciprocal differences

Although divided differences can be used to substitute derivatives in formulas like those of Taylor and Newton, only polynomials can be approximated. However, rational functions represent a much wider class of functions as they are quotient polynomials. A function like tan (x) cannot be accurately approximated around its asymptotes by a polynomial, whereas a rational function can. A rational function has the form $R(x) = P(x)/Q(x)$ with $P(x)$ and $Q(x)$ polynomials. When $Q(x) = 1$ we generate all the polynomial approximations.

It follows that rational functions have a wider range of applications and would also incorporate other designs. The link with difference algorithms and rational functions is via the reciprocal differences. Rational functions can be represented by a continued fraction, which is itself composed of reciprocal difference components.

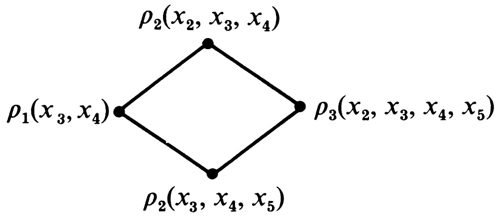
A continued fraction is of the form

$$y(x) = y_1 + \cfrac{(x-x_1)}{\rho_1 + \cfrac{(x-x_2)}{\rho_2 - y_1 + \cfrac{(x-x_3)}{\rho_3 - \rho_1 + \cfrac{(x-x_4)}{(\rho_4 - \rho_2)\atop \vdots}}}}$$

where ρ_i are reciprocal differences and

$$\rho_1 = \frac{(x_2-x_1)}{(y_2-y_1)} = \frac{1}{y(x_2, x_1)}$$

in the above fraction. Fig. 6 indicates the table template and basic cell, which is very similar to the divided difference, and the R function for the cell definition now becomes



$$\Rightarrow \frac{x_5-x_2}{\rho_2(x_3, x_4, x_5) - \rho_2(x_2, x_3, x_4)} + \rho_1(x_3, x_4).$$

The cell hardware has also increased. We now require an adder, divider, two subtractors, three pre-loadable registers and a delay for the x_{in} value.

In a particular cycle the cell computes as follows,

$$t\colon \quad r_0 = x_{\text{in}} - x_i;\, r_1 = \rho_{\text{in}} - y_i$$

$$t+\tfrac{1}{2}\colon \quad y_i = \left\{\frac{r_0}{r_1}\right\} + ptmp$$

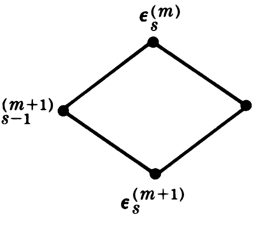
$$t+1\colon \quad \rho_{\text{out}} = y_i,\, ptmp = \rho_{\text{in}}$$

Snapshots of the array computation are shown in Fig. 7 for clarification. The timing of the array is given by $T = 2cn$ after normalisation with $c = 1$. No systolic ring can be used, and the cycle time due to the feedback loop to the y_i register is $\tau_2 + 2\tau_1$ (taking the cost of add = cost of subtract). Hence the finite difference array and the divided difference array run $[(\tau_2+2\tau_1)/3\tau_1]$ and $[(\tau_2+2\tau_1)/(\tau_2+\tau_1)]$ faster, respectively.

3. THE EPSILON ALGORITHM

The template used in Refs 5 and 6 for table generation can also be extended to the finite differences. However, to deal with unequally spaced arguments a second template for divided and reciprocal differences was introduced. The main change between the two templates was the explicit representation of the arguments in the computational rule and the change from cell-column generation to cell-row generation. This next algorithm indicates that the two templates can be used to implement a single algorithm.

The epsilon algorithm[8, 9] is a powerful technique for accelerating a slowly convergent sequence. The basic computational rule and the cell function R is given as



$$\epsilon_{s+1}^{(m)} \Rightarrow \epsilon_{s+1}^{(m)} = \epsilon_{s-1}^{(m+1)} + \frac{1}{\epsilon_s^{(m+1)} - \epsilon_s^{(m)}}$$

The table template is shown in Fig. 8 (*a*). Notice that this uses the first array template rules and the cell in Fig. 8 (*b*) is suitable for systolic ring implementation. The amount of hardware is less than that required by reciprocal differences, yet the epsilon algorithm has a very similar structure to the reciprocal formula, or R function. It follows from this that the epsilon algorithm can be implemented by both templates and hence by both the systolic ring-type cell and the pre-loading schemes. Later

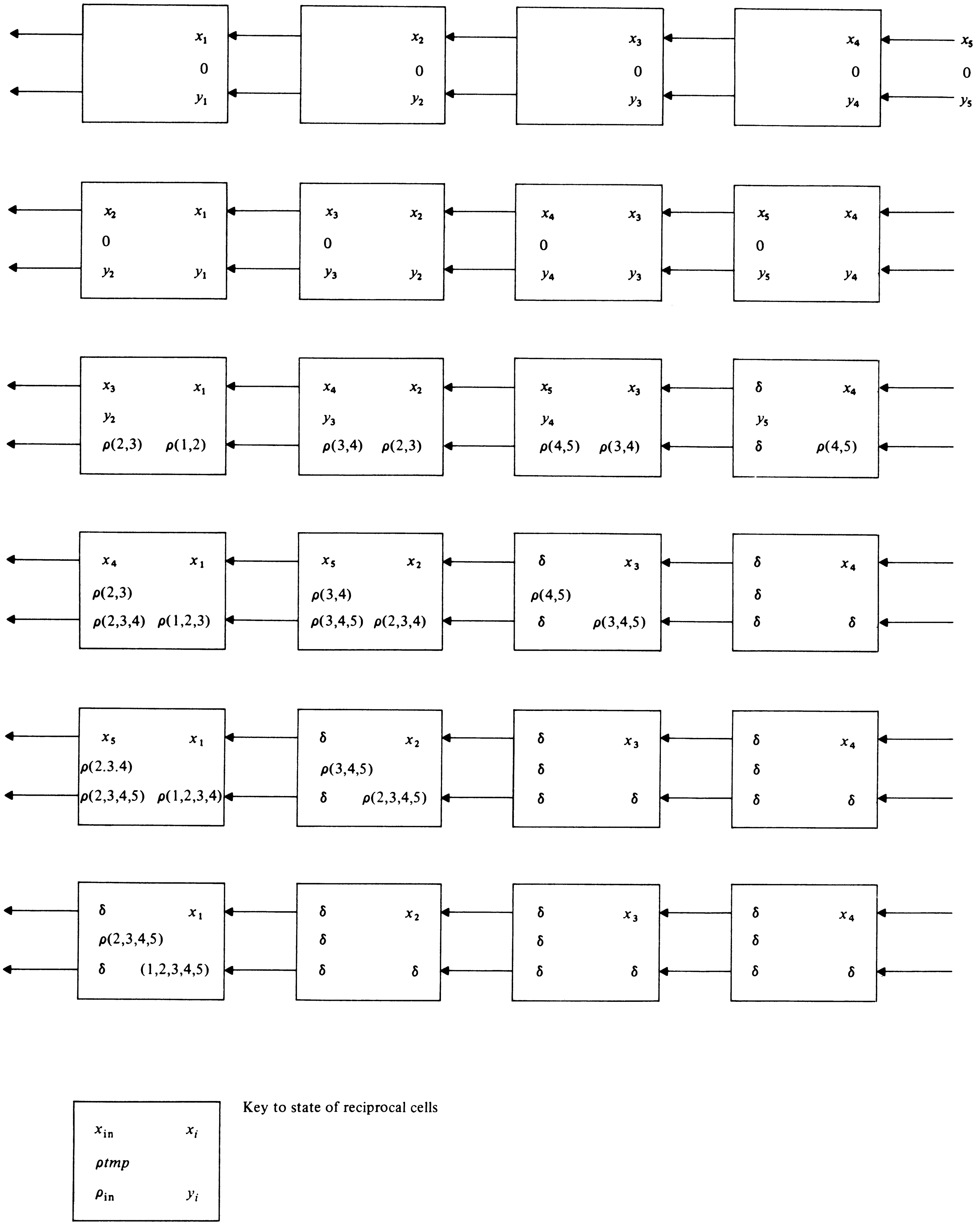

x_1 0 y_1
x_2 0 y_2
x_3 0 y_3
x_4 0 y_4
x_5 0 y_5
x_2 x_1 0 y_2 y_1
x_3 x_2 0 y_3 y_2
x_4 x_3 0 y_4 y_3
x_5 x_4 0 y_5 y_4
x_3 x_1 y_2 $\rho(2,3)$ $\rho(1,2)$
x_4 x_2 y_3 $\rho(3,4)$ $\rho(2,3)$
x_5 x_3 y_4 $\rho(4,5)$ $\rho(3,4)$
δ x_4 y_5 δ $\rho(4,5)$
x_4 x_1 $\rho(2,3)$ $\rho(2,3,4)$ $\rho(1,2,3)$
x_5 x_2 $\rho(3,4)$ $\rho(3,4,5)$ $\rho(2,3,4)$
δ x_3 $\rho(4,5)$ δ $\rho(3,4,5)$
δ x_4 δ δ δ
x_5 x_1 $\rho(2.3.4)$ $\rho(2,3,4,5)$ $\rho(1,2,3,4)$
δ x_2 $\rho(3,4,5)$ δ $\rho(2,3,4,5)$
δ x_3 δ δ δ
δ x_4 δ δ δ
δ x_1 $\rho(2,3,4,5)$ δ (1,2,3,4,5)
δ x_2 δ δ δ
δ x_3 δ δ δ
δ x_4 δ δ δ
x_{in} x_i $ptmp$ ρ_{in} y_i
Key to state of reciprocal cells

Figure 7. Dataflow in reciprocal difference array.

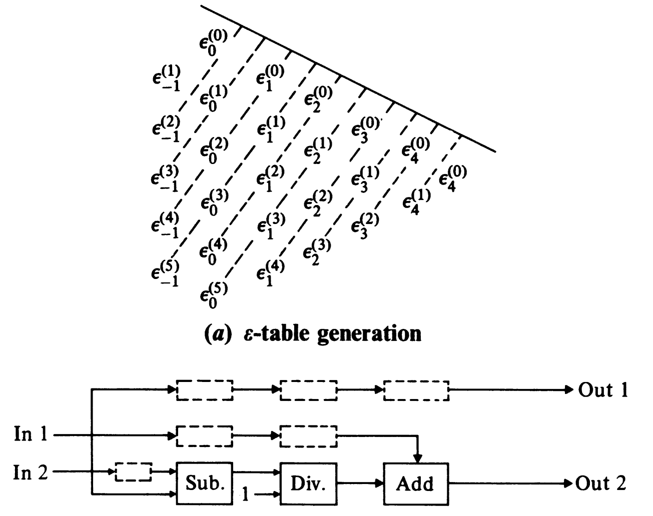
we will show that the second template introduced is more general and that the template used in Refs 5 and 6 and epsilon scheme is a special case when argument spacing can be eliminated from the R function in the cell. For completeness we state the computation order for Fig. 8(b).

$$t\text{: } IN1\ \varepsilon^{(0)}_{-1}, \quad IN2\ \varepsilon^{(0)}_{0}$$

$$t+\tfrac{1}{4}\text{: } IN1\ \varepsilon^{(1)}_{-1},\ IN2\ \varepsilon^{(1)}_{0},\ r_0 = \varepsilon^{(1)}_{0} - \varepsilon^{(0)}_{0}$$

$$t+\tfrac{1}{2}\text{: } IN1\ \varepsilon^{(2)}_{-1},\ IN2\ \varepsilon^{(2)}_{0},\ r_1 = 1/r_0,\ r_0 = \varepsilon^{(2)}_{0} - \varepsilon^{(1)}_{0}$$

$$t+\tfrac{3}{4}\text{: } IN1\ \varepsilon^{(3)}_{1},\ IN2\,\varepsilon^{(3)}_{0},\ r_2 = \varepsilon^{(0)}_{-1} + r_1, r_1 = 1/r_0,$$
$$r_0 = \varepsilon^{(3)}_{0} - \varepsilon^{(2)}_{0}.$$



(*a*) ε-table generation

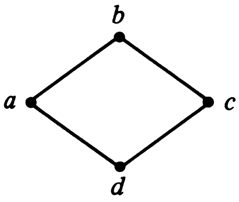
(*b*) ε-cell structure

Figure 8. Epsilon ε-algorithm template.

This gives a latency of $c = 4\tau_2$, hence $T = n + 4n, = 5n$, after normalisation, and a systolic ring requires $\lceil n/4 \rceil$ ε-cells.

4. A UNIFIED SYSTOLIC ARRAY FOR DIFFERENCES (USAD)

We have considered briefly the problem of unification in the combination of the forward and backward finite differences. Now we indicate that all the algorithms discussed in this paper can be implemented on a single array (USAD). The basic principle is that all the computational rules used for the table-generation and cell functions are similar to the rhombus rule.



For instance, by omitting vertex a, the forward and backward formulas are represented, while divided, reciprocal differences and Wynn's algorithm all fit the full rule with only slight variations on the computation. What is required is a minimised architecture cell for all the cells and a set of controlled switches to dictate the type of computation rule. Careful consideration shows that the reciprocal-difference cell contains all the hardware necessary to compute all the formulae; only the controls are to be added. We augment the reciprocal-difference cell with three control bits c_i, $i = 1(1)3$ with the interpretation shown in Table 1.

Table 1

c_1	c_2	c_3	Cell function	Array format
0	0	0	r_0/r_1	Divided difference
0	0	1	r_1	Forward difference
0	1	0	—	—
0	1	1	r_1	Backward difference
1	0	0	$r_0/r_1+\rho_{\text{in}}$	Reciprocal difference
1	0	1	$1/r_1+\rho_{\text{in}}$	ε-Algorithm
1	1	0	—	—
1	1	1	—	—

We can interpret the controls as commands to the reciprocal cell such that,

$$c_1 = \begin{cases} 0 & r_0/r_1 \\ 1 & r_0/r_1+\rho_{\text{in}} \end{cases},$$

$$c_2 = \begin{cases} 0 & \text{Normal operand order} \\ 1 & \text{switch operand order} \end{cases},$$

$$c_3 = \begin{cases} 0 & r_0 \text{ output valid} \\ 1 & \text{set } r_0 = 1 \end{cases}$$

while

$$s = \overline{c_1} \wedge c_3 = \begin{cases} 0 & \text{inputs to divider unchanged} \\ 1 & \text{input to divider swapped.} \end{cases}$$

c_2 now performs the swapping of operands discussed with the forward/backward differences, while c_3 provides a neutral value so that the divider can be effectively masked out for forward and backward differences, and acts as a reciprocal cell for the epsilon method. c_1 masks out the adder, while the special command s permits r_1/r_0 to be computed when necessary. The pre-loading of the arrays is trivial and is not discussed here.

The timing of the array is identical to that of the reciprocal-difference array. The additional switching logic and hardware are trivial, consisting of simple combinational logic, and adds no time to the algorithm.

5. CONCLUSIONS

We have investigated the use of array-templating techniques for the derivation of systolic arrays for computationally related problems. In particular we have examined a collection of differencing and extrapolation techniques when Refs 5 and 6 are considered. The concept of array unification has been introduced to combine similar systolic arrays with similar cells, which is achieved by combining cell functions to produce a minimal hardware arrangement. Two templates were discovered, indicating that earlier work on extrapolation systolic arrays is a special case of a general template developed in this paper. In particular, we have shown that equally spaced arguments allow the arguments themselves to be inferred in a cell function. Unequally spaced arguments must remain explicit in the cell computation, and this results in a change in the method of computation. Equally spaced arguments introduce cells which compute columns, unequally spaced arguments computing rows of the table to be generated.

Although the algorithms discussed in this paper are computationally simple, the unified array and the method by which the algorithms are analysed are important for future systolic array design. We have allowed a number of problems to be implemented on the same cell architecture and so have a very cost-effective VLSI design. Recent trends in systolic array development and particular the CMU WARP processor[7] are aimed at more flexible systolic array design. The frequent use of the methods examined here should make the USAD device an interesting alternative for the fast computation of approximating functions.

6. REFERENCES

1. C. E. Leiserson, Area-efficient VLSI computation. *Ph.D. Thesis* (1981). C.M.U. Pittsburgh.
2. H. T. Kung, The structure of parallel algorithms. *Advances in Computers* **19** (1980).
3. H. T. Kung and M. S. Lam, Wafer-scale integration and two-level pipelined implementation of systolic arrays. *J. Parallel and Distributed Computing* **1**, 32–63 (1984).
4. D. J. Evans and G. M. Megson, A Systolic Array for the QD Algorithm. Internal Report C.S. 254, L.U.T. (submitted to IEE for publication).
5. D. J. Evans and G. M. Megson, Romberg integration using systolic arrays. *Parallel Computing* **3**, 289–304 (1986).
6. D. J. Evans and G. M. Megson, Construction of extrapolation tables by systolic arrays for solving ordinary differential equations. *Parallel Computing* **4**, 33–48 (1987).
7. H. T. Kung, *Systolic Algorithms for the CMU Warp Processor*. CMU-CS-84-158 (1984).
8. Y. Saad and A. Sameh, Iterative methods for the solution of elliptic difference equations on multiprocessors. In *CONPAR 81 Lecture Notes in Computer Science*, edited G. Goos and J. Hartmanis, Springer-Verlag, Heidelberg, pp. 395–411.
9. P. Wynn, Acceleration techniques for iterated vector and matrix problems. *Mathematics of Computation* **16**, 301–322 (1962).
10. F. Scheid. Theory and Problems of Numerical Analysis. Schaum's Outline Series, McGraw-Hill, New York (1968).