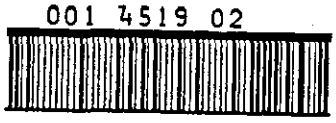


NOVEL ALGORITHMS FOR THE SOFT-SYSTOLIC PARADIGM

BY

GRAHAM MARTIN MEGSON, B.Sc.(HONS.)

A Doctoral Thesis

submitted in partial fulfilment of the requirements

for the award of Doctor of Philosophy

of the Loughborough University of Technology

1987.

Supervisor: PROFESSOR D.J. EVANS, Ph.D.,D.Sc.
Department of Computer Studies.

DECLARATION

I declare that this thesis is a record of research work carried out by me, and that the thesis is of my own composition. I also certify that neither this thesis nor the original work contained therein has been submitted to this or any other institution for a higher degree.

GRAHAM MARTIN MEGSON.

DEDICATED TO

My Wife and Love Helena,

for her constant support during

the course of this work.

ACKNOWLEDGEMENTS

The author wishes to express his thanks to Professor D.J. Evans for his guidance, suggestions and advice throughout the preparation of this thesis and in the context of research generally.

The author also acknowledges the Science and Engineering Research Council (S.E.R.C.) for their financial support.

Thanks also to my parents for giving me the incentive to start and complete this project.

Finally, thanks to Mr. R.P. Stallard for his constant revision of the Loughborough OCCAM compiler and supplying the documentation in Appendix II.

APOTHEGM

Quarendo invenietis

"By seeking, you will discover"

CONTENTS

PART III

ALGORITHMIC vs GEOMETRIC DESIGN

AND THE PRINCIPLE OF ARRAY UNIFICATION

CHAPTER 7

SYSTOLIC TABLE GENERATION

"All for one, and one for all"

The Three Musketeers.

So far this thesis has been concerned with the improvement of systolic arrays for fundamental problems in Matrix and Linear Algebra. These problems in the form of differential equations, as well as signal and image processing applications account for approximately 70% of numeric computation.

In this chapter we focus attention on systolic arrays for table based algorithms such as Interpolation and Extrapolation. Mckeown [84] shows that Aitken's Iterated Interpolation algorithm can be performed by a systolic array in O(n) time, where n is the size of the table. By extension Neville's Iterated Interpolation method can also be solved in O(n) time on a similar array.

Interpolation and extrapolation techniques also have wide uses in numerical computation and often produce results in tables of a triangular form. This triangular structure and the manner in which table elements are constructed indicate that systolic techniques for matrix problems may carry over to table based methods. Indeed, a table of elements is often represented as a matrix for easy and efficient manipulation on a computer system. Below certain similarities between matrix computations and extrapolation tables are developed to characterise table generation algorithms. The principles are then extended to table manipulation techniques for the more sophisticated simplex and assignment problems.

7.1 ROMBERG INTEGRATION USING SYSTOLIC ARRAYS

As an informative introduction to table based systolic computation we present an array to improve numerical approximations to integrals using Richardson's extrapolation procedure in the form of Romberg integration. Two designs are presented, the first an intuitive linear

array, the second, a systolic ring using approximately 1/3 the cells of the first. Both arrays have a computation time of 3n cycles (for a table of size n), a significant improvement on the $O(n^2)$ steps required to construct the extrapolation table sequentially.

The Romberg integration algorithm is well known, and is based on the Newton-Cotes formula (see Burden, Faires & Reynolds [81], Johnson and Reiss [77]). We use the particular Newton-Cotes formula known as the Trapezoidal method, which is one of the easiest to use but is usually not as accurate as required. The Romberg algorithm is widely applicable, and uses this easy-to-apply formula to obtain initial approximations to integrals, and Richardson's extrapolation to improve these approximations to gain a required accuracy.

Thus to evaluate the integral,

$$I = \int_a^b f(x).dx \ , \qquad (7.1.1)$$

for some integrable function f(x), we select an integer n>0 and apply the sequential procedure.

```
/*INPUT a,b, and integer n*/
/*OUTPUT an array R, R_nn is the approximation to I,*/
/*computed by rows*/
ROMBERG(a,b,N,f(x))
    { h=b-a; R_1,1 = h/2 (f(a)+f(b));
      OUTPUT(R_1,1);
      FOR i=2 to n
        { R_2,1 = ½[R_11 + h Σ_{k=1}^{2^(i-2)} f(a+(k-0.5)h];
          /*approximation using trapezoidal rule*/
          FOR j=2 TO i
             { R_2,j = (4^(j-1) R_2,j-1 - R_1,j-1) / (4^(j-1) - 1) ;
               OUTPUT(R_2,j);
             };
          h=½h
          FOR j=1 TO i {R_1,j=R_2,j};
        };
    }.
```

which outputs the triangular table of approximations given by,

$$
\begin{array}{lllll}
R_{11} & & & & \\
R_{21} & R_{22} & & & \\
R_{31} & R_{32} & R_{33} & & \\
\vdots & \vdots & \vdots & R_{44} & \\
\vdots & \vdots & \vdots & & \ddots \\
R_{n1} & R_{n2} & R_{n3} & \cdots\cdots & R_{nn}
\end{array}
\qquad (7.1.2)
$$

and known as the Romberg Extrapolation table, where $R_{i,1}$, i=1(1)n are approximations from the trapezoidal rule and the diagonal entries R_{ii} i=1(1)n are terms converging to an improved estimate of I in (7.1.1). In general, the sequence $\{R_{ii}\}_{i=1}^{\infty}$ converges much faster than $\{R_{m,1}\}_{m=1}^{\infty}$ and we stop when $|R_{ii}-R_{i-1,i-1}|<tol$, where tol=required accuracy. For the systolic array these factors have important consequences, firstly there must be a fixed number of cells in the array (for fabrication) and second there must be enough cells to ensure sufficiently accurate approximations. We adopt a general approach to the array and construct a finite sized table of n rows. For some problems convergence of the R_{ii} will occur before the full size n table is completed and introduces the additional problem of closing down the array prematurely, on the other hand, large problems may not converge. For the moment we assume that differences between the convergence rate of R_{ii} and R_{il}, i=1(1)n are large enough to ensure that n can be chosen to always achieve convergence subject only to area restrictions. (Later we develop a more flexible approach.

The systolic array computation is derived by partitioning the Romberg procedure into two basic steps:

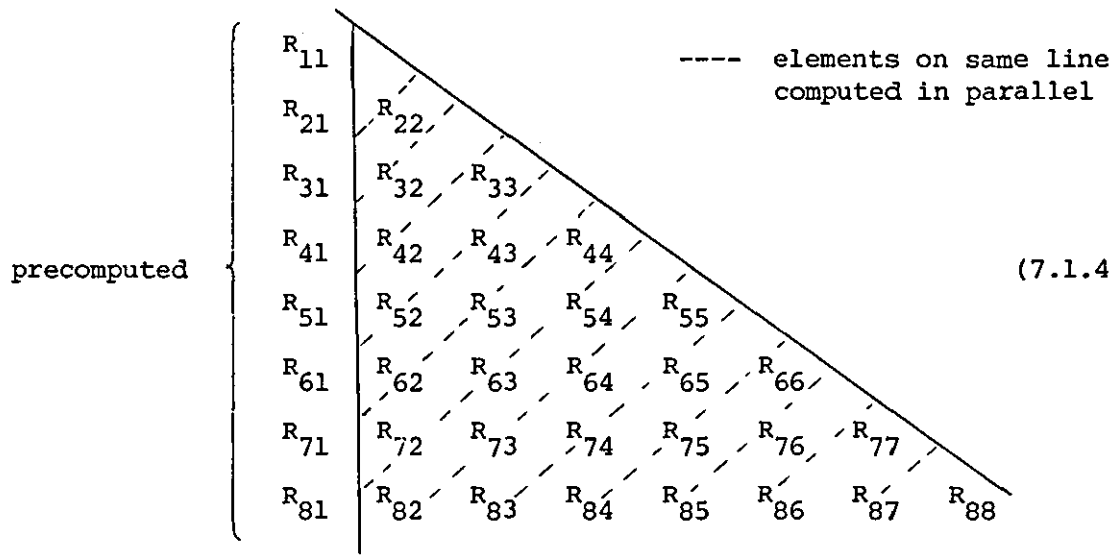
(i) approximate I using the trapezoidal rule with $m_1=1$, $m_2=2$, $m_3=4$,...

$\ldots, m_n = 2^{n-1}$ for an integer $n>0$ and stepsize $h_k = (b-a)/m_k = (b-a)/2^{k-1}$ to derive $R_{k,1}$ neglecting error terms $O(h_k^2)$.

(ii) generate table (7.1.2) using the general extrapolation relation,

$$R_{ij} = \frac{4^{j-1} R_{i,j-1} - R_{i-1,j-1}}{4^{j-1}-1} \qquad \begin{matrix} i=2(1)n \\ j=2(1)i \end{matrix} \tag{7.1.3}$$

which define a natural allocation of computation between host machine and the systolic array as follows. The host computes the terms $R_{k,1}$, $k=1(1)m$ for $m \leq n$ before waking the array, and the array constructs the table elements R_{ij}, $i=2(1)m$, $j=2(1)i$ in parallel using (7.1.3) and the evaluation ordering (for $n=8$),



(7.1.4)

This division is natural because step (i) involves evaluating the arbitrary function $f(x)$ (within the constraints of being integrable and continuous, etc.), and including it as part of the array would require an arbitrary number of complex basic array cells. Step (ii) and (7.1.4) can be constructed using a set of unidirectional linearly connected cells which implement the Richardson Extrapolation procedure (REP) of (7.1.3). The array is shown in Fig.(7.1.1) and consists of n-1 REP cells each with two inputs and three outputs, two outputs

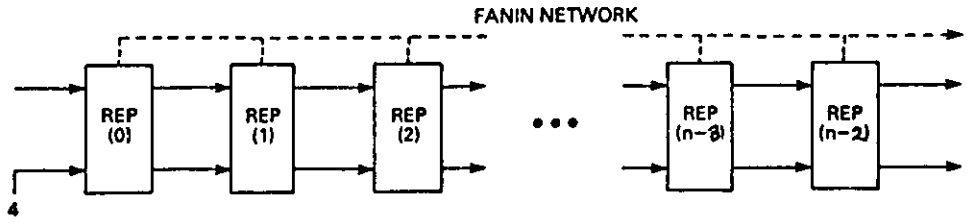


FIGURE 7.1.1: Romberg linear systolic array

connecting to the right hand adjacent cell and the third to a fanin network. The fanin network is used for filtering out the results R_{ii}, i=2(1)n which the host can use to determine convergence. Each REP cell computes a single column, with cell i evaluating column i+2, and consequently outputs only a single diagonal element. If c is the cell latency outputs on the fanin line from different cells occur at c,2c, 3c, etc. hence only a single line is required for output.

The REP cell is shown in Fig.(7.1.2) and consists of a single ips, a multiplier, subtractor and divider. The latency of the cell is c=3 ips cycles taking into account the delay through arithmetic elements and delay registers, and computes using a two-level pipelined organisation similar to H.T. Kung & Lam [84] but at a higher level of abstraction. Each cell performs (7.1.3) and also generates the power 4^j for the next cell, hence the two input and output lines. The leftmost cell of Fig. (7.1.1) accepts the elements $R_{11},R_{21},\ldots,R_{n1}$ on one input line and on the second the value 4 (which can be hardwired using a permanently register stored value). This permits the construction of the 4^j

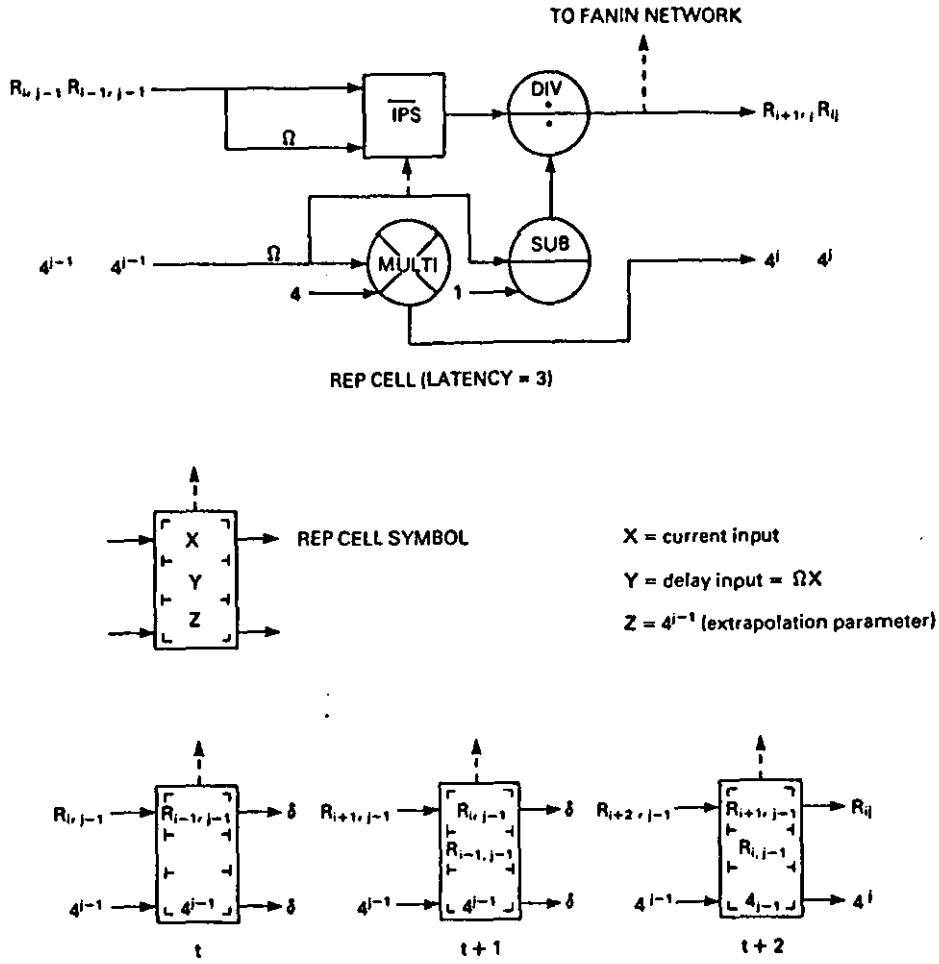


FIGURE 7.1.2: Cell operation

systolically rather than sequentially by the host, and although the additional multiplier in each cell appears extravagant we show later that it can be justified.

Remark: We could precompute the 4^j powers and preload them before the start of the computation, replacing the multiplier by a loadable register.

The operation of the REP cell is clear from Fig.(7.1.2), and a single control bit is tagged to the R_{ii}, i=1(1)m such that:

$$\text{control} = \begin{cases} 1 & \text{send cell result onto fanin network} \\ 0 & \text{normal output only} \end{cases}$$

The control tag moves systolically from cell to cell using the natural cell delay for synchronisation with newly created R_{ii} values ensuring

that cells use the fan-in line in mutually exclusive fashion. Fig. (7.1.3) illustrates the array operation as snapshots for the first seven steps of the table construction when n=6, and motivates the following theorem.

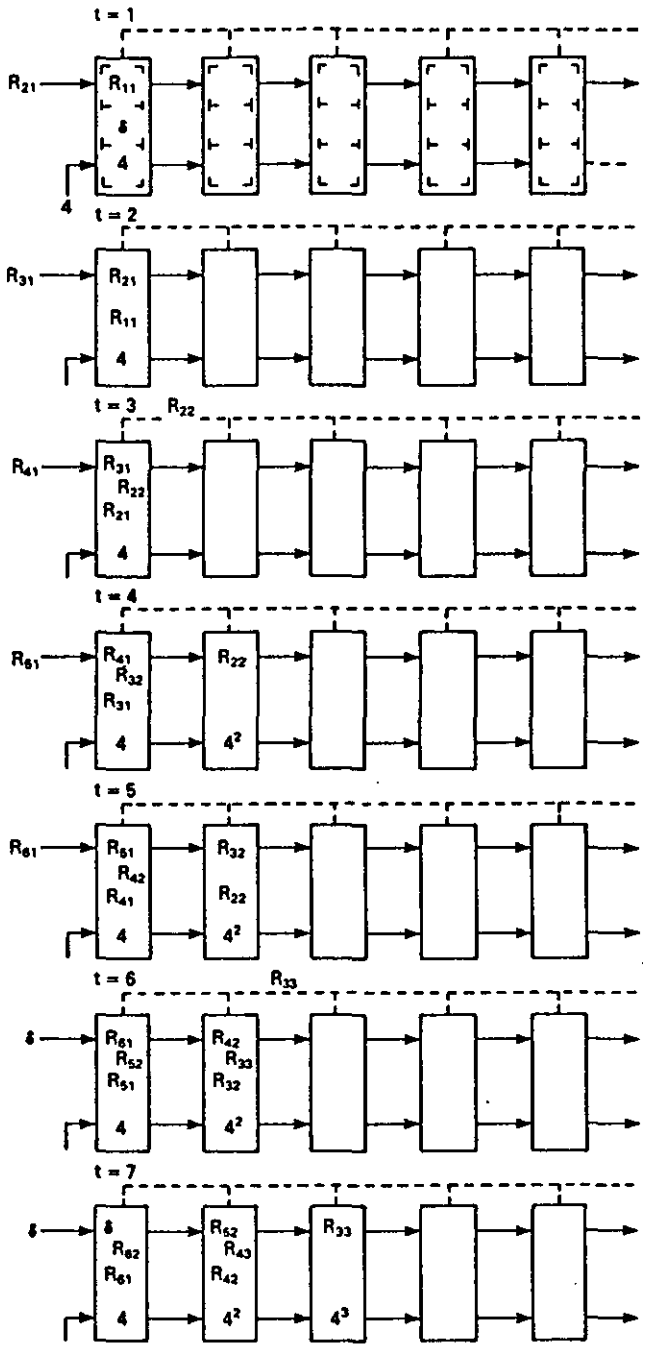


FIGURE 7.1.3: Snapshots of array operation

Theorem (7.1.1): A Romberg extrapolation table of size n can be computed in T=3(n-1) ips cycles using n-1 REP cells.

Proof:

The total time is given by,

$$T = (\text{cell latency}) * (\text{number of cells}) = c(n-1) = 3(n-1),$$

as only (n-1) columns hence cells are required to construct (7.1.2).

If a problem converges before all the rows of the maximum allowable table size by the array, we simply stop inputting R_{j1} values and reading R_{ii} values. Notice that if we run out of $R_{j,1}$ values simply pumping in zero values as neutral elements does not affect subsequent calculations, hence any table of size m<n can be computed in T=3(m-1) ips cycles. Finally, we remark that each REP cell can be considered equivalent to 2.5 ips cells and the array requires 2.5(n-1) ips equivalents.

Now, from Fig.(7.1.3) the last input R_{61} enters the array just as the second cell is about to output the value R_{33}. Generally an output leaves the $\frac{1}{3}n^{th}$ cell when the value $R_{n,1}$ is already in the first cell. It follows that once the last input has entered the first cell successive inputs will be dummy elements, consequently, only $\frac{1}{3}n$ REP cells will perform useful computations at any particular time. Hence, the size of the array can be reduced to $m' = \left\lceil \frac{1}{3}n \right\rceil$ cells, by using the two level pipelining of the REP (cell). When the last input $R_{n,1}$ has entered the first cell, the last cell computes a result in the divider, which on the next cycle will be output. We wrap the output of cell n-1 around to the input of cell 1, so that the result of cell n-1 is pipelined behind the last input in cell 1. The result computed is incorrect, but by preceding discussions will not affect results further down the

array, and will not appear on the fanin line because the control bit associated with the cell (n-1) output has not propagated through to the divider (it will arrive in another two cycles). On the next cycle cell 1 starts to compute the value which would have occurred in REP cell $(\frac{1}{3}n+1)$ of the linear array. It follows that the two cell systolic ring in Fig.(7.1.4) will compute the same table as Fig.(7.1.3) but requires $\frac{1}{3}$ the cells. As each REP cell is equivalent to 2.5 ips cells the ring requires at most n ips cells to implement it.

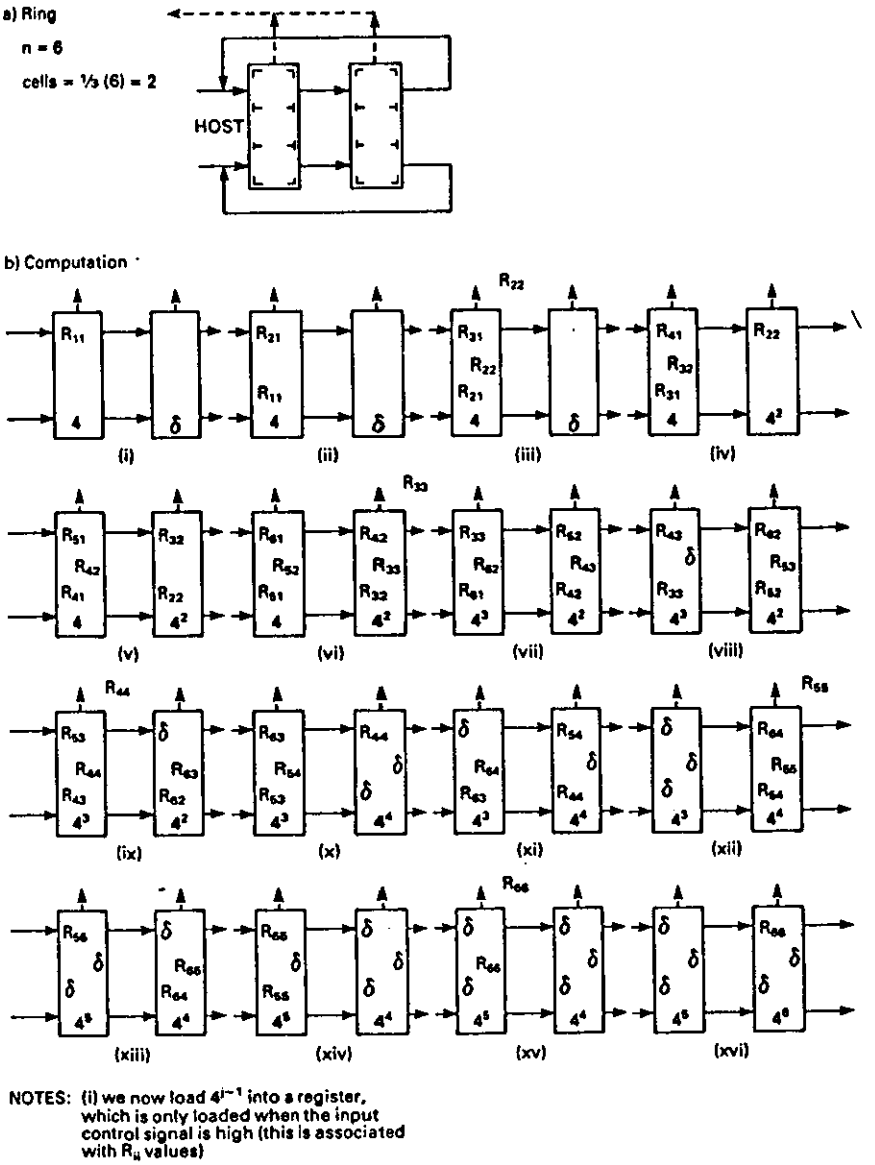


FIGURE 7.1.4: Systolic ring for Romberg Integration Table n=6
a) array b) ring computation

From H.T. Kung & Lam [84] we note that the ring has a particularly efficient cell layout depicted by Fig.(7.1.5) which requires a box of

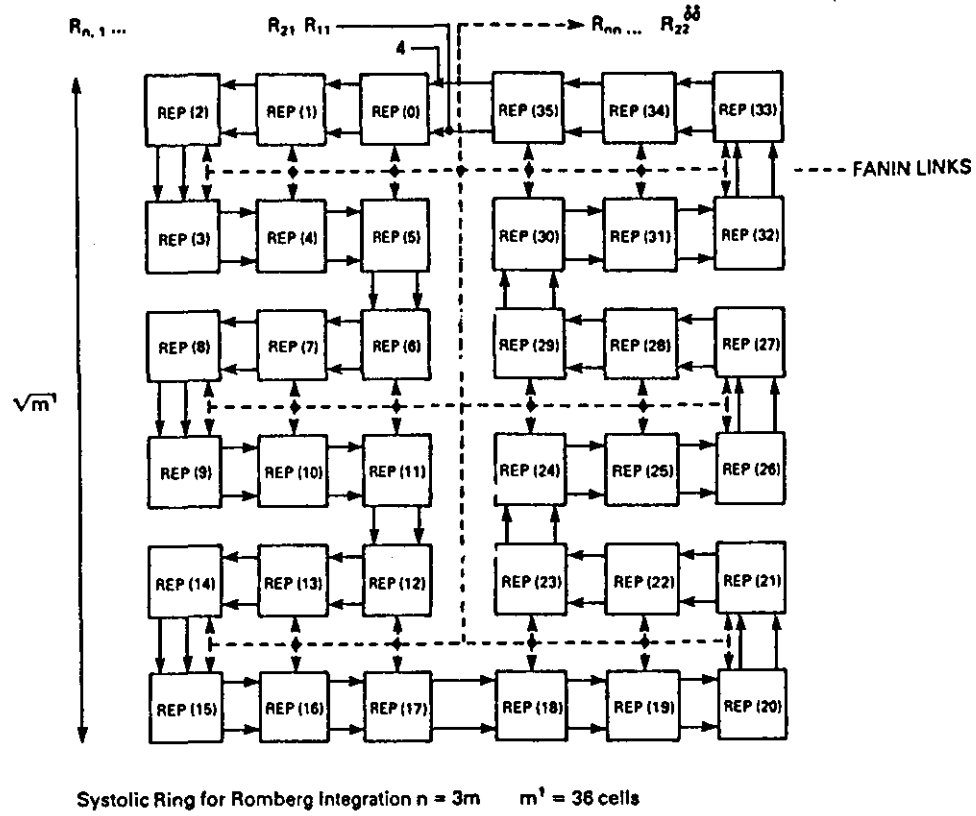


FIGURE 7.1.5: Systolic ring for Romberg Integration n=3m, m'=36 cells side $\sqrt{m'}$ (where a unit measure is the side of a square bounding the REP cell). The fanin network itself can be embedded inside the ring to minimise its length which is proportional to $(\frac{3}{2})\sqrt{m'}$, and is important for controlling skew and latch mistimings in a hard systolic frame, but not so important in a soft-systolic frame with electro-optic heuristics. The tag bits controlling the cell outputs onto the fanin network can also be utilised to the full with the ring design, as follows. First, we can justify the additional multiplier in each REP cell for computing the 4^j powers. Clearly in a systolic ring the 4^j values cannot be preloaded by the host as they would be incorrect after

the first ring cycle. Instead, notice that the control tag bit is set only with R_{ii} values which for the linear array implies that an unused cell is entered, making any previous 4^j value invalid. Consequently the tag bit can be used to set the new 4^j values as it cycles around the ring, by loading a register in each cell. The loaded value is retained until the tag bit completes a ring circuit, when it is over-written. The tag bit can also be used to control the connections between the host input/ring, and the ring connections between cell 1 and cell m^1. On startup cell REP(0) in Fig.(7.1.5) accepts values from the host, including the control. After a single cycle of the ring the control bit leaves the m^{1}th cell switching in the ring connections. On subsequent cycles no switching occurs.

Finally, we remark that the array in Fig.(7.1.5) has 36 cells, allowing any table m<108 to be constructed, which would probably be adequate for most applications. The ring also has the useful property of 1 host input and 1 host output (with the value 4 hardwired) making it attractive for a hard systolic approach to implementation. Soft-systolic versions of the arrays can be found in the Appendix as programs 9 (linear) and 10 (ring). The designs both require O(3n) ips cycles compared with $O(n^2)$ time on sequential machines, a significant improvement.

7.2 THE CONSTRUCTION OF GENERIC ARRAYS FOR EXTRAPOLATION TABLE GENERATION

It can be readily appreciated that the above Romberg integration array is a special case of a generalised (or generic) table generating array. For example, above we assumed only the diagonal table entries were required for output simplifying the host/array interface and allowing the use of the area efficient ring structure. In wider

applications part or all of the table interior may also be required, and this demands a more flexible array structure. To develop our generic array we consider the construction of extrapolation tables used in the solution of Ordinary Differential Equations (ODEs) associated with initial value type problems of the form,

$$\left.\begin{aligned} y' &= f(t,y),\ a \leq t \leq b \\ \text{initial condition} \quad y(a) &= \alpha \quad . \end{aligned}\right\} \tag{7.2.1}$$

The generalised method is examined first for a low order formula (i.e. Eulers method) combined with a suitable extrapolation formula, and is then extended to the Burlisch & Stoer [66] method as an example of array construction. Finally, before the development of any systolic arrays two fundamental restrictions must be observed:

(i) The systolic array can only be applied to existing ODE's by construction of extrapolation tables.

(ii) The array must be of fixed size, implying a limit to the number of levels or step divisions allowable, in order to keep the table size fixed and manageable.

The first point indicates that any portion of the ODE algorithm which involves the evaluation f(t,y) must be placed outside the array. This follows because in general f(t,y) can be arbitrary providing it is integrable, and the systolic array would become arbitrarily complex if it were included. (This is an extension of the restrictions of the Romberg array). The second rule observes that the function values used to estimate y(t) at every level must be evaluated before the array can be used. This appears to defeat the object of extrapolation, because we would normally stop when convergence is reached, ignoring the computation at lower levels altogether.

We now consider how an extrapolation procedure can be incorporated into the solution of an ODE using an algorithm attributed to Gragg. Although the algorithm is simple it provides a suitable vehicle by which to illustrate the above points and relate extrapolation techniques to matrix computations. Applying Eulers' method with stepsize h>0 to (7.2.1) the ODE solution is approximated as follows:

$$\left.\begin{aligned} & w_0 = \alpha \\ & w_{i+1} = w_i + hf(t_i, w_i), \quad i=0(1)n-1 \\ \text{with} \quad & n = (b-a)/h \text{ and } t_i = a+ih, \quad i=0(1)n-1 \end{aligned}\right\} \tag{7.2.2}$$

and the approximation error $y(t_i)-w_i$ leads to a function $\delta(t)$ such that,

$$y(t_i) = w_i + h\delta(t_i) + O(h^2), \quad i=1(1)n. \tag{7.2.3}$$

Now let w(t,h) denote the approximation to y(t) with stepsize h. For example, choose two step levels h_0 and h_1 ($<h_0$) and consider evaluating y(b) with,

$$q_0 = (b-a)/h_0 \quad \text{and} \quad q_1 = (b-a)/h_1$$

applying (7.2.2) and (7.2.3) twice, once with $h=h_0$ and again with $h=h_1$ yielding,

$$\left.\begin{aligned} y(b) &= w(b,h_0) + h_0\delta(b) + O(h_0^2) \\ y(b) &= w(b,h_1) + h_1\delta(b) + O(h_1^2) \end{aligned}\right\} \tag{7.2.4}$$

which after simple manipulation produces,

$$y(b) = \frac{h_0 w(b,h_1) - h_1 w(b,h_0)}{h_0 - h_1} + O(h_0^2) \ . \tag{7.2.5}$$

If the difference method like (7.2.2) has a particular type of error expansion (see Burden, Faires & Reynolds [81]) it can be generalised to construct an extrapolation table, with diagonal elements converging to a good (accurate) approximation of y(t). For example, with three

levels, h_0, h_1, h_2 we obtain,

$$y_{1,1} = w(t,h_0)$$

$$y_{2,1} = w(t,h_1), \quad y_{22} = \frac{h_0^2 y_{2,1} - h_1^2 y_{11}}{h_0^2 - h_1^2}$$

$$y_{3,1} = w(t,h_2), \quad y_{32} = \frac{h_1^2 y_{31} - h_2^2 y_{21}}{h_1^2 - h_2^2}, \quad y_{33} = \frac{h_0^2 y_{3,2} - h_2^2 y_{22}}{h_0^2 - h_2^2}$$

which results in Gragg's extrapolation algorithm defined below, where,

INPUT = end points a,b, initial condition α, tolerance TOL, and level limit $p \leq 8$

OUTPUT = T,w,h, where w approximates y(t) at stepsize h or a message indicating that the minimum stepsize was exceeded.

```
Gragg
  { NK=(2,3,4,6,8,12,16,18);
    T_0=a; w_0=α; h=h_max;
    FOR i=1 TO 7
      { FOR j=1 TO i {Q_ij=(NK_{i+1}/N_kj)^2};};
    WHILE (T_0<b) DO
      { k=1; FLAG=0;
        WHILE (k≤p AND FLAG=0) DO
          { HK=h/NK_k; T=t_0; w_2=w_0;
            w_3=w_2+HK*f(T,w_2); /*Euler step*/
            T=t_0+HK;
            FOR j=1 TO N_{k-1}
              { w_1=w_2; w_2=w_3;
                w_3=w_1+2*H_k*f(T,w_3); /*mid point method*/
                T=t_0+(j+1)*HK;
              };
            y_k=[w_3+w_2+HK*f(T,w_3)]/2;
            /*smooth for y_{k,1}*/
            IF k≥2 THEN
                {j=k; v=y_1; /*save y_{k-1,k-1}*/
                WHILE (j≥2) DO
                   { y_{j-1}=y_j + (y_j-y_{j-1})/(Q_{k-1,j-1}-1)
                     j=j-1;
                   };
                IF |y_1-v|<TOL THEN FLAG=1
                /*accept y_1 as new w*/
              };
            k=k+1;
          };
        k=k-1;
        IF FLAG=0 THEN
          { h=h/2; IF h<h_min THEN{OUTPUT 'minimum h exceeded'; STOP}
        ELSE
          { w_0=y_1; T_0=T_0+h; OUTPUT(t_0,w_0,h);
            IF (k≤3) AND (h<h_max/2) THEN h=2h
          };
  }};
```

The extrapolation table can now be represented as a lower triangular p×p matrix,

$$Y = \begin{bmatrix} y_{11} & & & \\ y_{21} & y_{22} & & \\ \vdots & \vdots & \ddots & \\ y_{p1} & y_{p2} & \cdots & y_{pp} \end{bmatrix} . \qquad (7.2.6)$$

Next we define a constant p×p lower triangular matrix Q with Q_{ij} the ratio of two step sizes squared.

$$Q = \begin{bmatrix} Q_{11} & & & \\ Q_{21} & Q_{22} & & \\ \vdots & \vdots & \ddots & \\ Q_{p1} & Q_{p2} & \cdots & Q_{pp} \end{bmatrix} , \qquad (7.2.7)$$

and define E_p as an extrapolation operator. The extrapolation given by the Gragg algorithm is then formulated by,

$$Q(E_p)Y_1 = Y , \qquad (7.2.8)$$

where, $Y_1 = [y_{11}, y_{21}, \ldots, y_{p,1}]^t$ such that,

$$[Q_{11}, \ldots, 0](E_p) \begin{bmatrix} y_{11} \\ y_{21} \\ \vdots \\ y_{p1} \end{bmatrix} = \begin{bmatrix} y_{11}^1 \\ y_{21}^1 \\ \vdots \\ y_{p1}^1 \end{bmatrix} \quad \text{or } y_{11}^1 = \frac{h_0^2 y_{21} - h_1^2 y_{11}}{h_0^2 - h_1^2} = y_{21} + \frac{y_{21} - y_{11}}{Q_{11}^{-1}}$$

and,

$$[Q_{21}, Q_{22}, \ldots, 0](E_p) \begin{bmatrix} y_{11}^1 \\ y_{21} \\ \vdots \\ y_{p1} \end{bmatrix} = \begin{bmatrix} y_{11}^2 \\ y_{21}^1 \\ \vdots \\ y_{p1} \end{bmatrix} , \text{ or } \quad y_{21}^1 = y_{31} + \frac{y_{31} - y_{21}}{Q_{22}^{-1}}$$

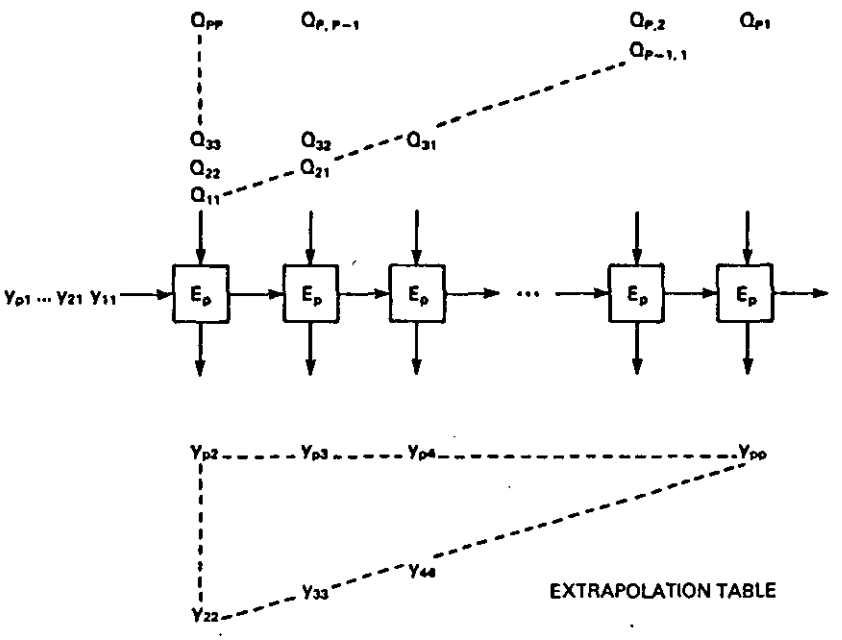
$$y_{11}^2 = y_{21}^1 + \frac{y_{21}^1 - y_{11}^1}{Q_{21}^{-1}} .$$

... etc.

The form of computation is similar to dot product calculations, hence using the same pipelining ideas developed in Section (3.2) Fig.(3.2.1.2), extrapolation can be interpreted as a matrix-vector type computation

$$Q_1 y_1^{(1)} = y_1^{(2)}, \quad Q_2 y_1^{(2)} = y_1^{(3)}, \ldots, Q_p y_1^{(p)} = y_1^{(p+1)} , \qquad (7.2.9)$$

with Q_i row i of Q and $y_1^{(i)}$ column i of the extrapolation table, and the non-commutative operation implied by juxta-position the formula associated with E_p rather than the inner (dot) product. Thus, we have obtained a generic recurrence structure which leads directly to the linear array in Fig.(7.2.1) which adopts a diagonal input format for Q, and a column output form for y, and implements operator E_p as a basic cell computation.



Ep ≡ extrapolation cell, computes extrapolation procedure

Time unit = latency (cost of) extrapolation computation for cell

Total time = (P − 1) C

P = number of step levels

C = latency of Ep cells

FIGURE 7.2.1: Generic extrapolation array

The construction of Q must be performed before array operation removing the explicit representation of the sequence $h_0,h_1,\ldots,h_{p-1}$ for p steps in the array. Consequently for specially chosen sequences simplification in Fig.(7.2.1) occurs. For example, the sequence,

$$h_0,\ h_1 = \frac{h_0}{2}\ ,\ h_2 = \frac{h_0}{4}\ ,\ h_3 = \frac{h_0}{8}\ ,\ldots,h_{p-1} = \frac{h_0}{2^{p-1}}$$

produces the step size relation matrix,

$$Q = \begin{bmatrix} (2)^2 & & & \\ & (4)^2 & & \\ & & \ddots & \\ & & & 2^{2(p-2)} \\ 2^{2(p-1)} & 2^{2(p-2)} & \cdots & (2)^2 \end{bmatrix} \qquad (7.2.10)$$

and with the Q_{ij} preloaded into array cells produces the array in Fig. (7.2.2a). The Q_{ij} in (7.2.10) are easily constructed 'on-the-fly' and we can save preloading expenses by augmenting the E_p cell with additional hardware to generate the required power for the next cell (to the right). If the E_p cell latency is c(>1) cycles (where a cycle is some basic calculation like an inner product step) and the cell can be structured to allow two level pipelining, values can be input on basic cycles rather than every c cycles, and the total number of array cells reduced to $m=\lceil p/c \rceil$. This produces the systolic ring in Fig.(7.2.2b), which by wrapping the m^{th} cell output back to cell 1 generates the table computation sequence in Fig.(7.2.3). Hence,

Theorem 7.2.1: The generation of an extrapolation table of level p can be computed in T=c(p-1) ips cycles where c is the E_p cell latency in ips equivalents, and requires at most p cells.

Proof: [see Fig.(7.2.1)].

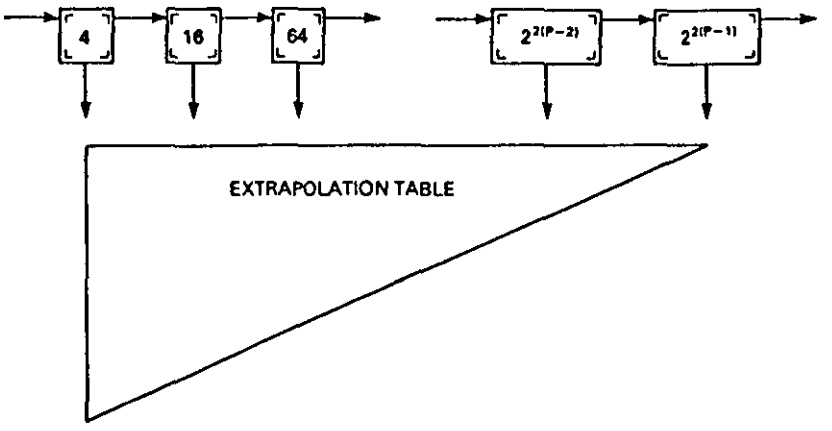


FIGURE 7.2.2a: Extrapolation array for $h_0, h_1 = \frac{h_0}{2}$, $h_2 = \frac{h_0}{4}$ ...

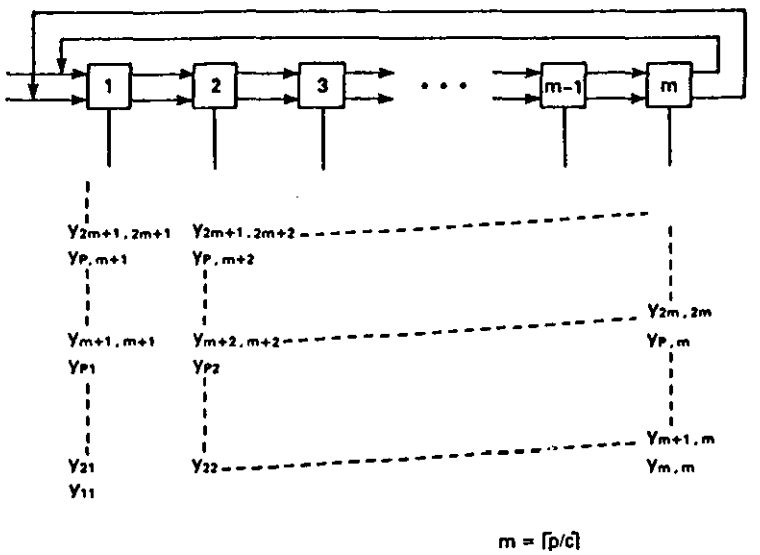
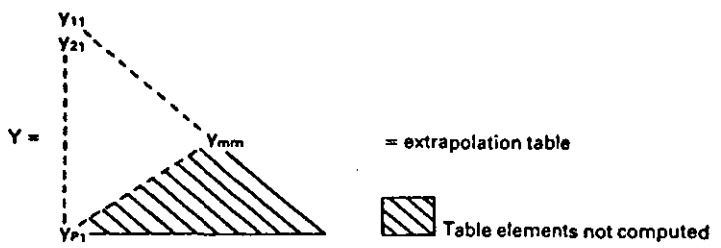


FIGURE 7.2.2b: Systolic ring computation of extrapolation table



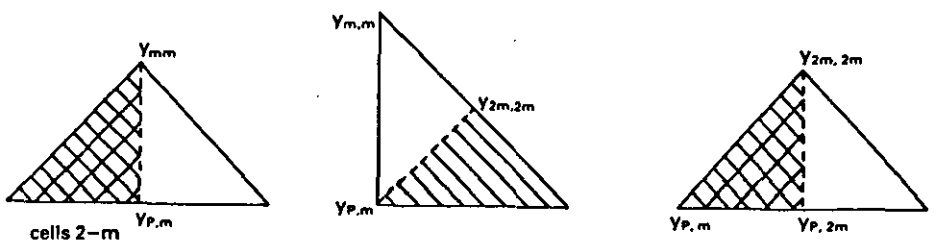


FIGURE 7.2.3: Cycles in systolic ring

Corollary 7.2.1: If the E_p procedure can be implemented with the equivalent of c ips cells incorporating two level pipelining we require $m=\lceil p/c \rceil$ E_p cells requiring area proportional to p ips cells.

The array for Romberg integration using (7.2.10) and (7.1.3) as the E_p operator follows trivially. With the further assumption that only diagonal values are output Fig.(7.2.2b) reduces further to the compact ring layout of Fig.(7.1.5) form. Likewise when formula (7.2.5) is adopted as the E_p operator the following cell structure is obtained:

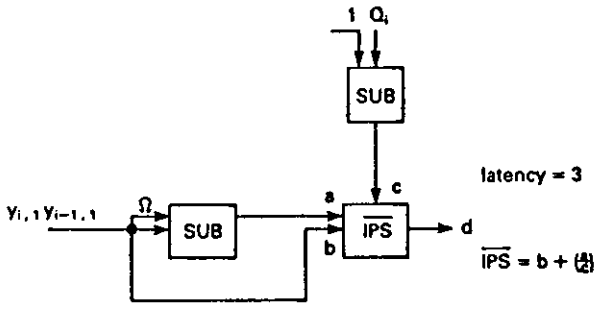


FIGURE 7.2.4: Gragg E_p operator cell structure

Hence with c=3, and a sub-cell assumed bounded by an ips equivalent theorem (7.2.1) gives the timing T=3(p-1).

Next, we consider a more complicated E_p function derived from the Bulirsch & Stoer extrapolation method using rational function approximations to y(t). The motivation being that the function f in (7.2.1) can be approximated by polynomials or quotients of polynomials (rational functions) which not only give a wide range of approximating functions, but also allow larger step sizes hence smaller tables and arrays than preceding methods. In addition, polynomials can be evaluated efficiently using Horners method reducing the time for a host machine to generate the required starting values. The Bulirsch and Stoer algorithm requires an

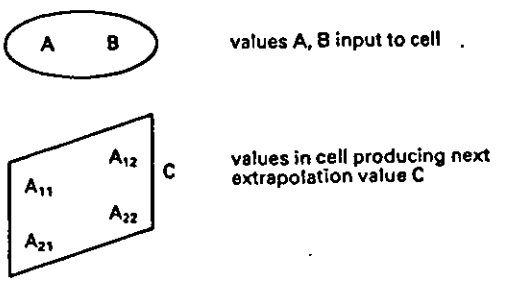
E_p function of the form,

$$T_k^{(i)} = T_{k-1}^{(i+1)} + \frac{T_{k-1}^{(i+1)} - T_{k-1}^{(i)}}{\left(\frac{h_i}{h_{i+k}}\right)^2 \left[1 - \frac{T_{k-1}^{(i+1)} - T_{k-1}^{(i)}}{T_{k-1}^{(i+1)} - T_{k-2}^{(i+1)}}\right] - 1} \qquad (7.2.11)$$

relating the elements $T_{k-2}^{(i+1)}$, $T_{k-1}^{(i)}$, $T_{k-1}^{(i+1)}$ and $T_k^{(i)}$ together by a Rhombus rule. Fig.(7.2.5a) indicates an intuitive order of parallel computation similar to (7.1.4) but with the key,

Key to (7.2.5a)

The cell that results is unnecessarily complex involving strange delay arrangements, non-planarity and three inputs to produce the correct cell output sequence. A less intuitive ordering is the skew Rhombus rule which requires just two inputs and two outputs and calculates according to Fig.(7.2.5b) using the key,



Key to (7.2.5b)

If we define $A=T_{k-1}^{(i+1)}-T_{k-2}^{(i+1)}$, $B=T_{k-1}^{(i+1)}-T_{k-1}^{(i)}$, $Q=\left(\frac{h_i}{h_{i+k}}\right)^2$ (7.2.11) becomes,

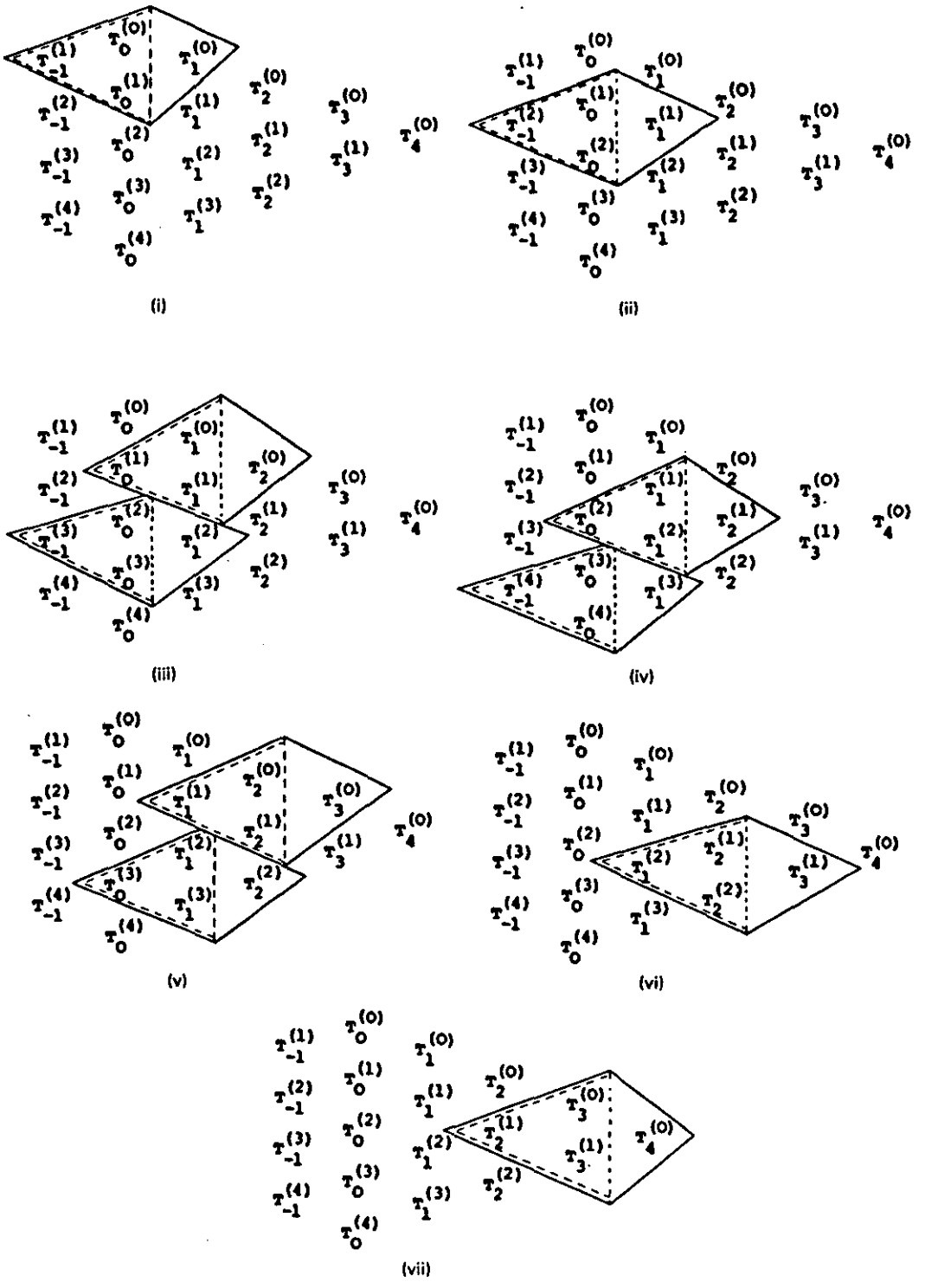


FIGURE 7.2.5a: Rhombus computation of the tableau

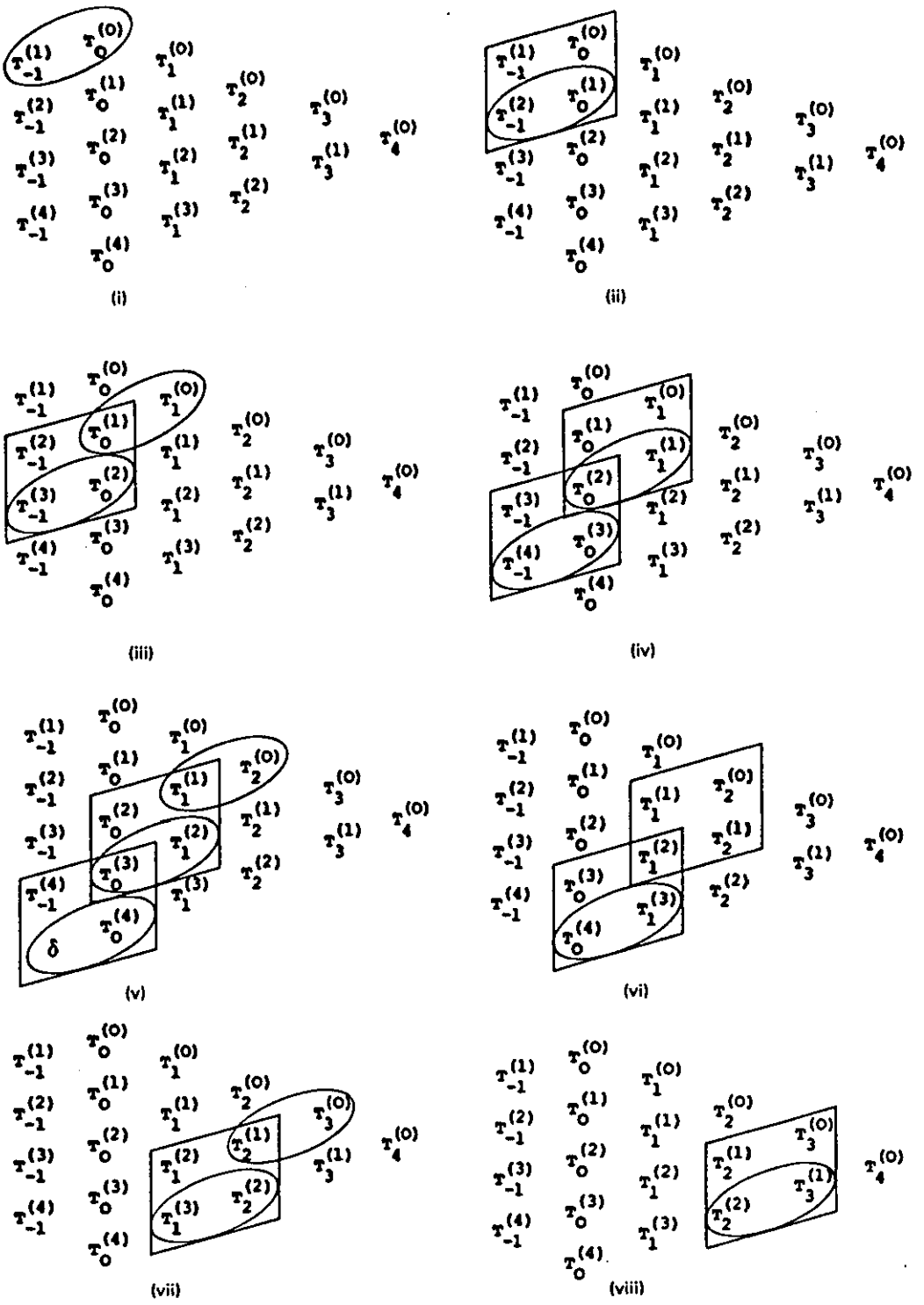


FIGURE 7.2.5b: Skew Rhombus computation of the tableau in parallel

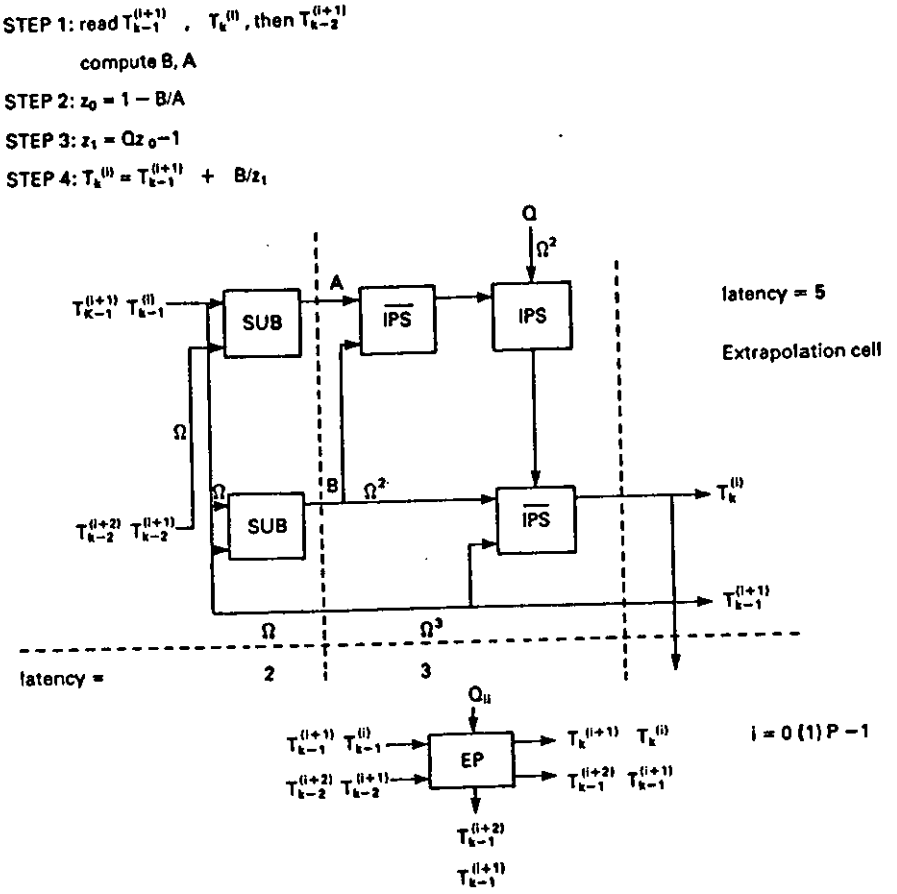


FIGURE 7.2.5c: Bulirsch & Stoer extrapolation cell

$$T_k^{(i)} = T_{k-1}^{(i+1)} + \frac{B}{Q[1-\frac{B}{A}]-1} \quad , \tag{7.2.12}$$

matching the form in the generic array, and yielding the E_p cell in Fig.(7.2.5c). Notice that the cell latency is c=5 ips cycles and that the hardware requirement is bounded by 5 ips cells. Thus Theorem (7.2.1) and Corollary (7.2.1) are applicable yielding T=5(p-1) and $m=\lceil p/5 \rceil E_p$ cells for a ring structure. The addition of an extra input and output for each cell is a trivial alteration to the array in Fig.(7.2.1) and does not complicate the host array interface as $T_{-1}^{(i)}=0$ is usual (and can be hardwired) while the output $T_{k-1}^{(i+1)}$ of the last cell can be simply discarded.

The extrapolation arrays so far have always assumed that:

a) The diagonal elements of the table converge to the required value y(t) at point t.

b) Convergence is achieved for some level $p' \leq p$.

Now suppose we had an array of p E_p cells and for some table the diagonal entries converged on level $p' < p$, we have no way of knowing in advance that the table will converge early and must compute all p starting values and their associated Q_{ij} elements. Likewise, if the table elements diverge we can detect it (by monitoring diagonal elements output) and close down the array prematurely, but still have all the starting values to compute initially. The computation of unused starting values, especially as they will tend to occur at lower levels where more function evaluations are required, represents a significant overhead. It follows that table generation would be more efficient if we could compute only the necessary starting values. A naive approach to solving this problem would be to try and pipeline the starting value evaluations and array operation. Clearly for arbitrary f(t,y) functions in (7.2.1) and only a small number of levels (hence table sizes) this is impossible. Basically, the value of f(t,y) must be found during the periodicity time of an E_p cell, for the above arrays this is a single ips, as we move down the table more f(t,y) evaluations are required per starting value providing a simple contradiction. Instead we consider an 'Adaptive' systolic array for extrapolation based on the work in Murphy [78] where more flexible sequential table construction algorithms were considered. The array is based loosely on the systolic priority queue (Leiserson [81]) for keeping real time order statistics, in which starting values can be input at intermittent intervals with the array

having no real knowledge of when the next input will arrive. On cycles between inputs the priority queues continue to compute, we suggest a modified form in which no computation occurs between inputs delayed by significant time. We define two measures of time, array time and host time such that the total computation time T=(array time) +(host time). Now suppose we compute p' starting values with an array of size p' the full table is computed making host time zero. If we compute $p'' < p'$ starting values we construct a table of size p'' decide on convergence and freeze the array while the remaining $p'-p''$ starting values are computed which contributes to the host time giving $T=c(p'-1)+$ (freeze time). Thus, freeze time is the cost associated with evaluating starting values which were previously precomputed. The decision to freeze the array must be made by the host on the basis of whether the tables diagonal values are smoothly converging or not according to the following criteria.

(i) The detection of smooth convergence: which by decreasing errors decides:

   a) To abandon the table because it is not converging
   b) Convergence will occur with the already computed starting values
   c) The estimated size of table required to provide convergence, using extra starting values.

(ii) In case of an incorrect prediction:

   a) Whether to change the stepsize (increase or decrease)
   b) To re-run the array with more starting values.

(iii) Closedown of the array; because convergence has already been achieved.

(iv) Raise an exception: that no more starting values can be used. (i.e. all array cells occupied).

From these conditions the freeze command can be generated. The implementation of the freeze depends on the type of clocking mechanism, for asynchronous control operation is essentially dataflow and stalling the handshaking protocol in the host is sufficient. For synchronous arrays freeze is implemented by gating the array cell cycle clock as follows:-

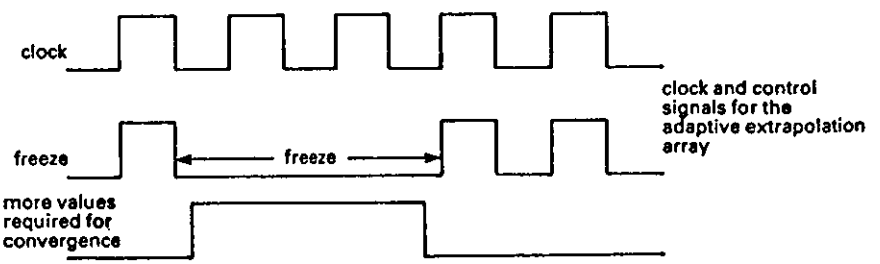
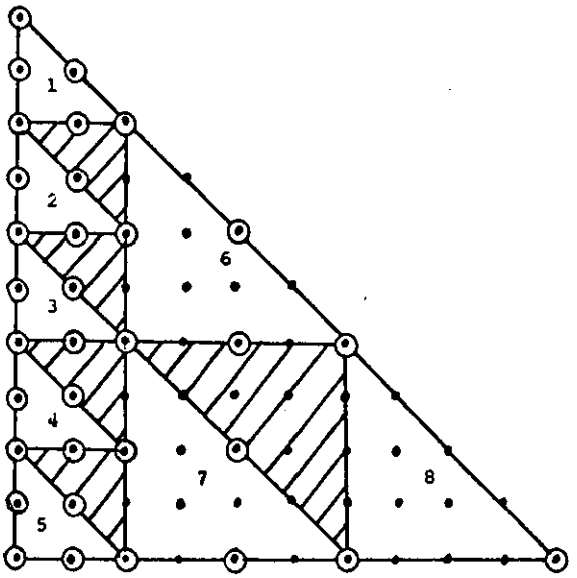


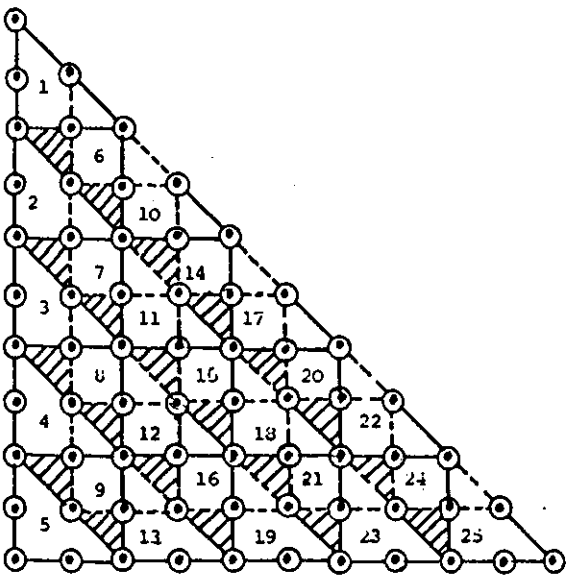
FIGURE 7.2.6

Normally, gating a clock would be bad practice, but as the array is globally affected no problems should be encountered.

Clearly for large tables results can be constructed by alternating freeze/computation phases requiring a minimal number of starting values to be constructed. A final warning to the practicality of the adaptive array is that we must collect enough diagonal estimations to perform a prediction. The Bulirsch & Stoer E_p cell with latency c=5 requires 10 starting values to create two diagonal estimates. Consequently we may have to compute more starting values than necessary. The problem arises because of the two level pipelining of cells, and can be avoided by redesigning the cell without pipelining at the expense of reduced throughput.



a) Straight forward partition



b) Overlapped partitions

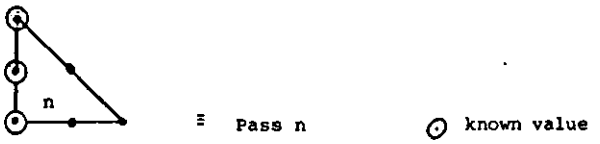
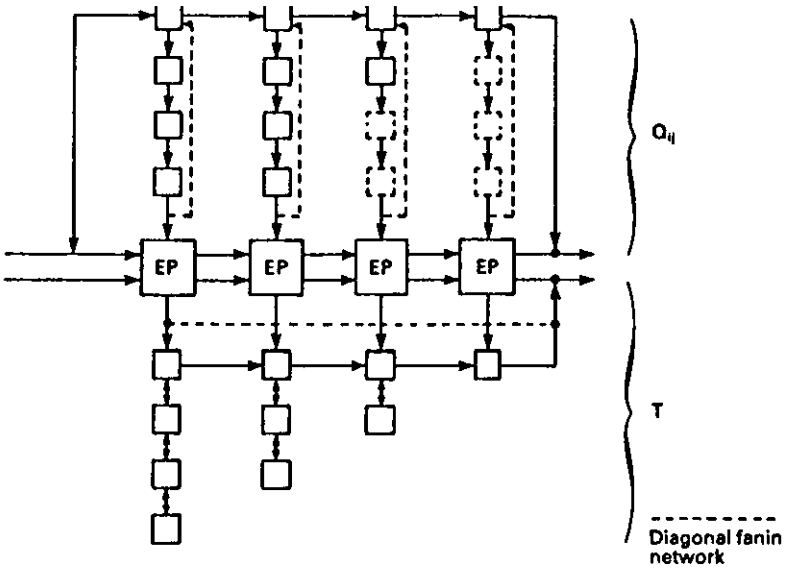


FIGURE 7.2.7: Multipass table construction



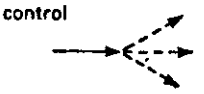
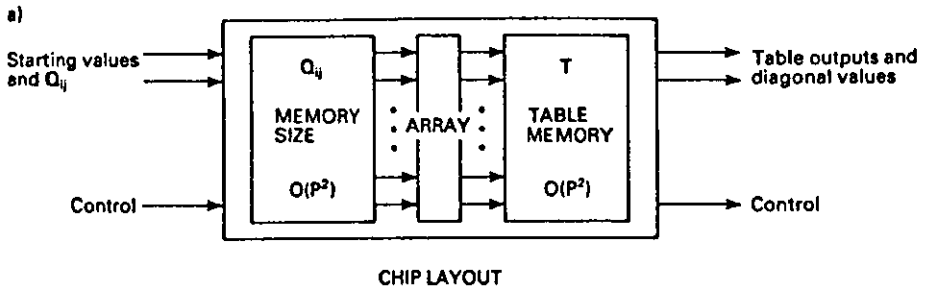
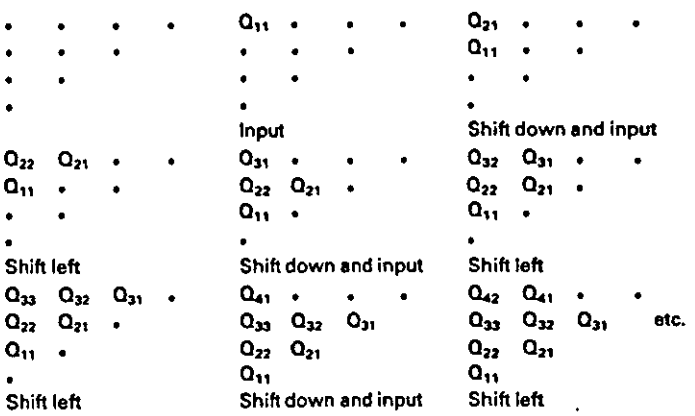


FIGURE 7.2.8: Internal memory extrapolation array

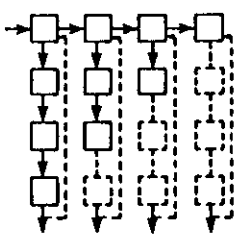


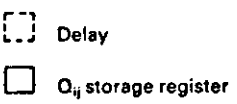
b)

Loading Q_{ij} with simple shifts



c) Memory structure





Q_{ij} memory p = 4 with systolic delay synchronisation

FIGURE 7.2.9: Chip organisation

Control		Action	
0	0	Shift down	Load Q_{ij}
0	1	Shift right	
1	0	Shift left	
1	1	Shift up	Read T

Next consider the problem indicated by case (iv) of the prediction criteria, under these circumstances we need a table larger than the array can compute. The solution is to use multipass table construction on a fixed array of size p as illustrated in Fig(7.2.7). Notice that not all the table values are computed, and could affect the convergence (or its detection) forcing a larger table, hence more starting values, to be evaluated than required if all the table elements were known. As f(t,y) function evaluations increase for deeper levels the rise in host computation can be significant. This increase to some extent is outweighed by the fact that we can compute a table of any size, and where area is a premium a small array can be constructed. Figs.(7.2.8) and (7.2.9) indicate the possible structure of a self-contained extrapolation array which minimises the host/array interface by storing the Q_{ij} and (table) y_{ij} values for sequential input and output. The feedback loop of the Q_{ij} memory allow the Q_{ij} to circulate once loaded to solve a number of consecutive table problems. The table memory incorporates a fanin line so that just the diagonals can be output, allowing the table to be overwritten for consecutive problem solutions, as well as full table output for single problems. Clearly a single chip device will require a small p and multipass which requires constant reloading of the Q_{ij} memory. We conclude that a small compact extrapolation array is feasible.

7.3 THE UNIFICATION OF SYSTOLIC DIFFERENCING ALGORITHMS

The above fast systolic arrays for extrapolation techniques have shown that a certain correlation exists between table generation and matrix computations. From these correlations it is possible to develop

the concept of array templating. The templating method allows the fast derivation, (by a sequence of designs), of systolic arrays for computationally related problems which freeze the abstract definition of the array at a high level using a global method of calculation. The subject of this section is twofold. Firstly, it defines templates for the problem of differencing algorithms, indicating that the above methods are special cases of a more general structure. Secondly, we introduce the concept of unification and illustrate the method by showing that our differencing templates can be unified to fit a single array the Unified Systolic Array for Differencing (USAD).

A prototype template for table generation which has been used implicitly in the previous sections consists of the following four components:

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

(ii) A linear array is defined with basic cells mapped onto a column of the table. Cell i computes column i and implements the computational rule. (see Fig.(7.2.1)).

(iii) A class of arrays for partial or full table generation (as indicated by (7.2.2) and (7.1.1)).

(iv) A generic timing given by,

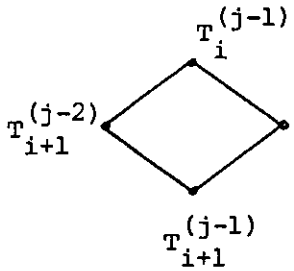
$$T = (\text{number of inputs}) + (\text{delay through array}) = (n+1) + cn \ , \qquad (7.3.1)$$

for cell latency c and table of size n+1.

We have observed that c varies for the complexity of the computational

rule and when c>1 we can save area by using a systolic ring. (7.3.1) is a maximal timing and requires some explanation, intuitively, the array time must be bounded by the time for the last starting value to enter and pass completely through the array. To derive the timings in theorems (7.1.1) and (7.2.1) we notice that the structure of the table demands only one output (the last diagonal) from the last cell. After all starting values have been input the diagonal element of the $\lceil n/c \rceil$th cell has been computed, and requires the array delay $c(n-\lceil \frac{n}{c} \rceil)$ assuming n is divisible by c substitution into (7.3.1) yields T=cn+1.

We can now distinguish between the ideas of templating and unification. The key point is that the template defines an array structure which remains static, while the design of a basic cell varies with the computational rule. We shall denote computational rules as a function R defining the cell as a black box. For instance, a rectangle rule of the form,



$$\Rightarrow \quad T_i^{(j)} = R(T_{i+1}^{(j-1)}, T_i^{(j-1)}, T_{i+1}^{(j-2)}) \qquad (7.3.2)$$

is the computational rule for the Bulirsch and Stoer extrapolation table. A number of rules R_1, R_2, etc. which fit the same template and have similar geometric properties can be unified if a common cell structure with minimal area can be identified.

Next we identify a class of R_i functions which can be unified from common differencing techniques such as forward, backward, divided differences, and rational function approximation. All the tables considered can be used to extract co-efficient 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 approximation such as the Newton forward/backward difference formulae and the continued fraction representation for rational functions. It is the simplicity and universal application of these methods which makes the generation of their coefficients (or part of them) by fast systolic arrays important.

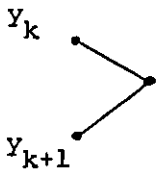
Now given a discrete function, i.e., a set of arguments x_k and a corresponding value y_k such that arguments are equally spaced by the distance $h=x_{k+1}-x_k$. The difference operator Δ is defined as,

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

2nd 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.(7.3.1a) and the cell function R is derived from the computational rule



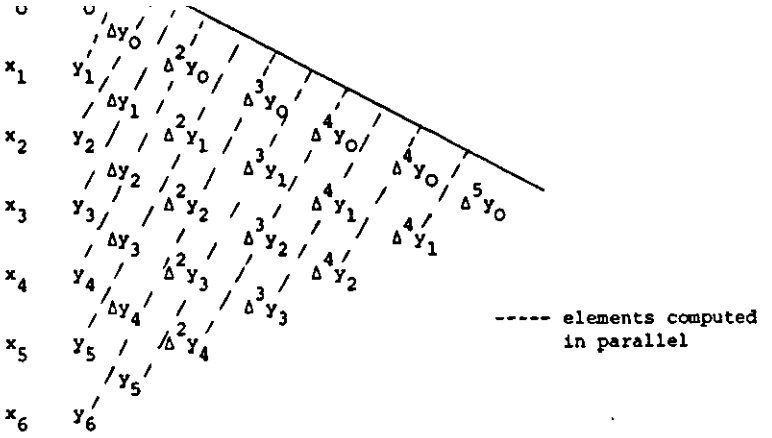
$$\Delta y_k \Rightarrow R_1(y_k, y_{k+1}) = \Delta y_k \ , \qquad (7.3.3)$$

defining the Δ-cell in Fig.(7.3.1b) consisting of a single delay register and subtracter. The array operation is shown in Fig.(7.3.2) and with cycle time τ_1=cost of subtraction the cell latency $c=2\tau_1$. By normalising the cycle time to eliminate τ_1 (7.3.1) gives the maximal timing,

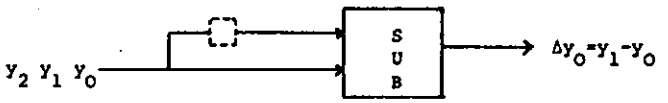
$$T = (n+1)+2n = 3n+1 \ , \qquad (7.3.4)$$

which can be reduced to 2n+1 cycles when the diagonal output is observed, and a systolic ring configuration requires only $\lceil n/2 \rceil$ subtracters.

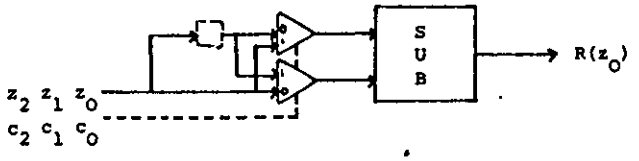
Similarly the backward difference (∇) is defined by the simple computational rule,



a) Finite difference table computation



b) Basic Δ-cell



c) Switchable difference cell

FIGURE 7.3.1: Finite difference template

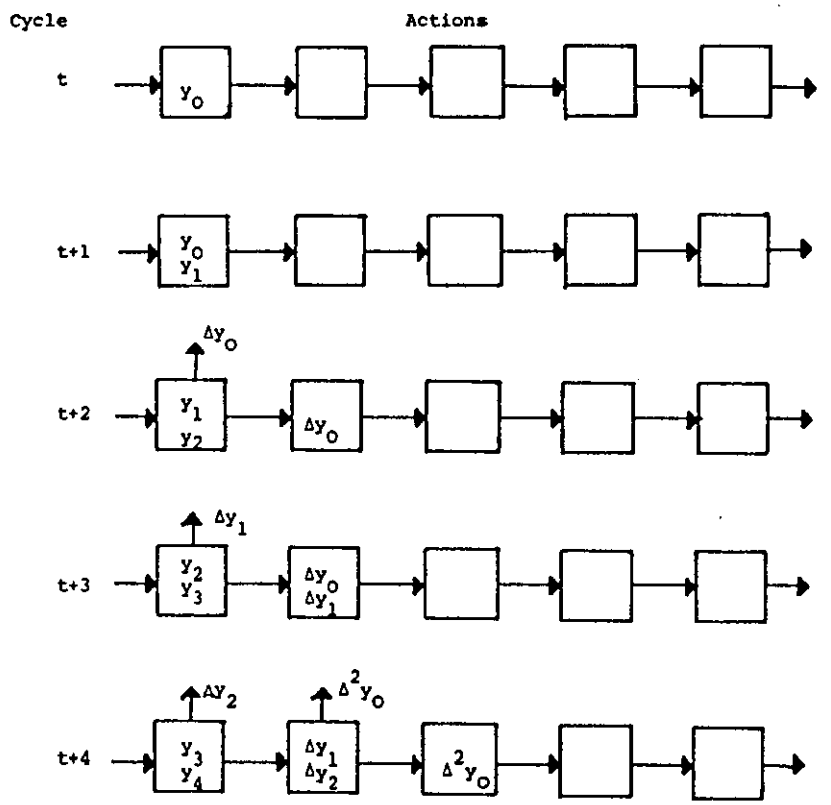
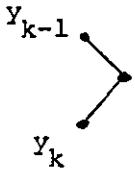


FIGURE 7.3.2: Successive cycles in forward difference table generation



$$\nabla y_k \Rightarrow R_2(y_{k-1},y_k) = \nabla y_k = y_k - y_{k-1} \qquad (7.3.5)$$

producing a similar cell to Fig.(7.3.1b) and retaining the computation time (7.3.4). This is expected as the template computation orders are isomorphic requiring only the reversal of the starting input data, and these trivial methods can be used to illustrate the principle of unification. In a mathematical sense the cell functions R_1 and R_2 define R_3 a new function and rule which unifies both methods on the same minimal area architecture. We denote this,

$$R_3(R_1(y_k,y_{k+1}),\ R_2(y_{k-1},y_k)) = R_3(z)\ , \qquad (7.3.6)$$

such that,

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

and

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

deriving the unified basic cell of Fig.(7.3.1c) controlled by a switch c_j to swap the input operands of the subtracter to select R_1 or R_2. Notice that the input for R_2 is reversed.

The differencing algorithms defining the current template arrays have a significant drawback for they assume equally spaced arguments. To develop the templating concept for more general table generators (and hence wider applications) unequally spaced arguments must be considered and, for purposes of illustration shall take the form of divided differences which are defined as follows:-

1st divided differences $$y(x_o,x_1) = \frac{y_1-y_o}{x_1-x_o}$$

2nd divided differences $$y(x_o,x_1,x_2) = \frac{y(x_1,x_2)-y(x_o,x_1)}{x_2-x_o}$$

Higher differences $$y(x_o,x_1,\ldots,x_n) = \frac{y(x_1\ldots x_n)-y(x_o\ldots x_{n-1})}{x_n-x_o}$$

which modifies the template computation to that shown in Fig(7.3.3a) and uses the template function,

$$y(x_o\ldots x_i) = R_4(y(x_1,\ldots,x_i),y(x_o,\ldots,x_{i-1}),x_o,x_i) \qquad (7.3.7)$$

producing the cell in Fig.(7.3.3b). This time the arguments x_i not just the starting values are represented explicitly in the cell function. Furthermore, for two arguments x_i and x_j in (7.3.7), successive j values for i fixed become spatially separated (i.e. $d=|i-j|$ increases). From a systolic viewpoint as d increases the problems of synchronising the correct element with purely non-stationary dataflows also increases. Thus, attempting to use the template for equally spaced arguments is disastrous and indicates the effect on array dataflow as the argument representation changes from explicit to implicit. Operation of the divided difference array is shown in Fig.(7.3.4), with each cell consisting of two subtracters, a divider, two preloadable registers, and an additional delay for x_{in}. The cell computation is divided into two parts i.e.,

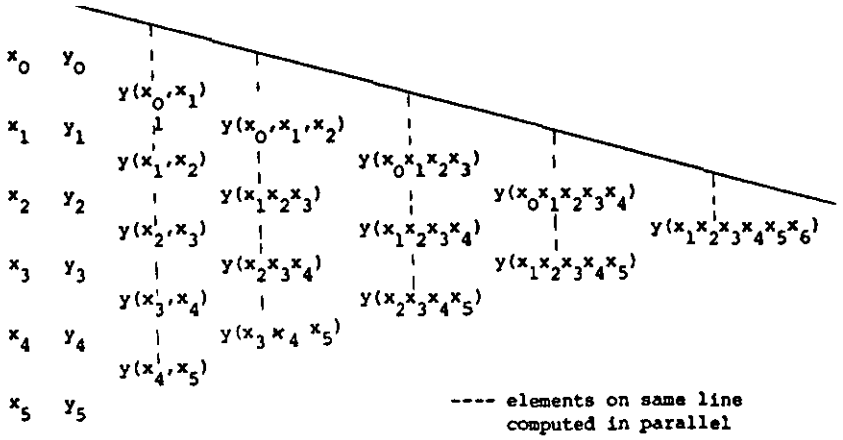
i) preload cell i with x_i and y_i the starting values i=1(1)n

ii) compute the next column element as follows:

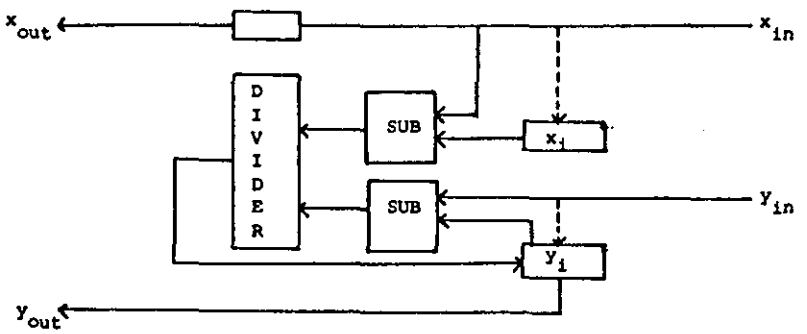
  a) evaluate $a=x_{in}-x_i$, $b=y_{in}-y_i$ in parallel

  b) set $y_i=b/a$ the new divided difference

The linear array requires the preloading of the starting values, and



a) Parallel divided difference table computation



b) Divided difference cell

Notes: ----- lines used for preloading.

FIGURE 7.3.3: Divided difference template

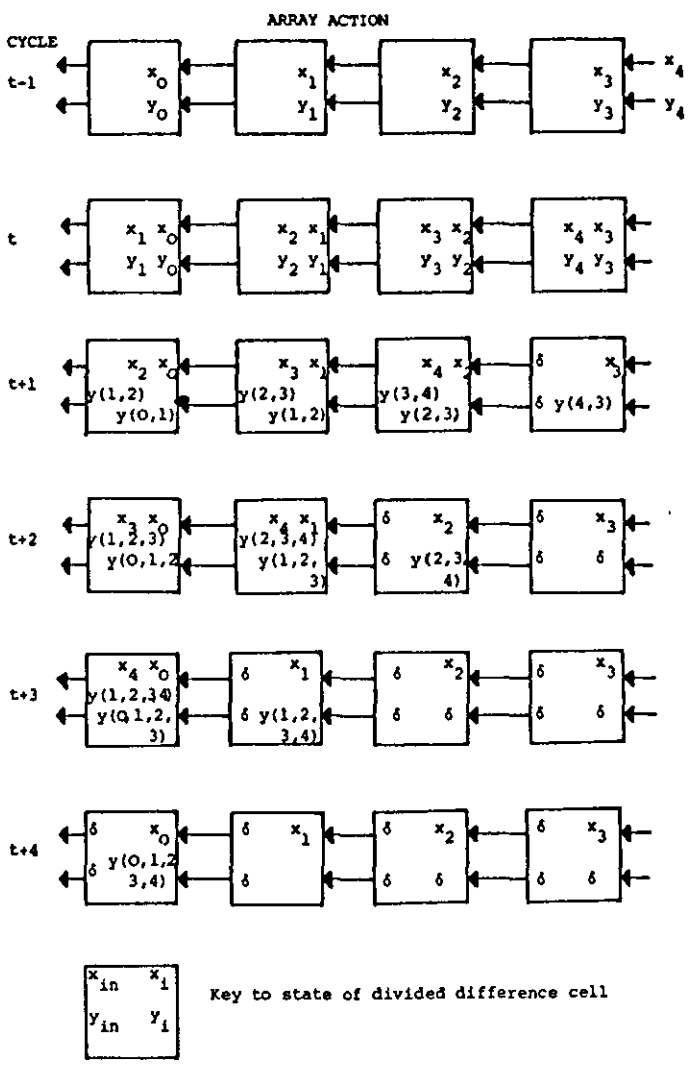


FIGURE 7.3.4:

Dataflow/computation in divided difference array

on successive cycles after preloading computes one complete column of the table every cell cycle. Compare this to the row ordering of the previous templates and we see a change from row sweep to column sweep orientated calculation. Alternatively in the old template array each cell computed a column, in the new template the same cell computes a row. The timing of the divided difference array is given by,

$$T = (\text{preload time})+(\text{time through array}) = n+c*n \qquad (7.3.8)$$

Notice that preload time replaces the length of input. It follows that $c=\tau_1+\tau_2$, where τ_2=cost of a divide, so that $T=2n$ after normalising the cycle time (in this case c). No two level pipelining of cell sub-calculations is possible due to the y_i feedback in Fig.(7.3.3b) and this together with the preloading prohibits the use of systolic rings. REMARK: notice that forcing the $x_{in}-x_i$ subtracter to produce the result 1 causes the divided difference array to form forward and backward differences.

Although divided differences can be used as substitutes for derivatives in formulas like the Taylor and Newton formulas only polynomials can be approximated. Rational functions represent a much wider class of functions as they are quotients of polynomials. A function like tan(x) for example cannot be accurately approximated around its asymptotes by a polynomial whereas with a rational function it can. Generally, 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 the class of polynomial approximations. It follows that rational functions not only have a wider range of applications but also incorporate other array designs. The link with difference algorithms and rational functions is

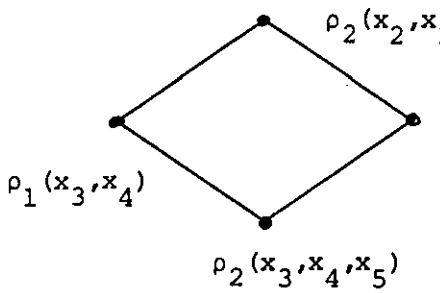
via reciprocal differences. Rational functions can be represented by continued fractions which themselves are composed of reciprocal difference components. A continued fraction has the form,

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

where ρ_i are reciprocal differences such that

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

Fig.(7.3.5) indicates the required table template and the basic cell which has a similar R function to divided differences with the rule,



$$\rho_3(x_2,x_3,x_4,x_5) \Rightarrow \rho_1(x_3x_4) + \frac{(x_5-x_2)}{\rho_2(x_3,x_4x_5)-\rho_2(x_2,x_3,x_4)}$$

denoted,

$$R_5(\rho_2(x_2,x_3,x_4),\rho_2(x_3,x_4,x_5),\rho_1(x_3,x_4),x_5,x_2) = \rho_3(x_2,x_3,x_4,x_5) \tag{7.3.9}$$

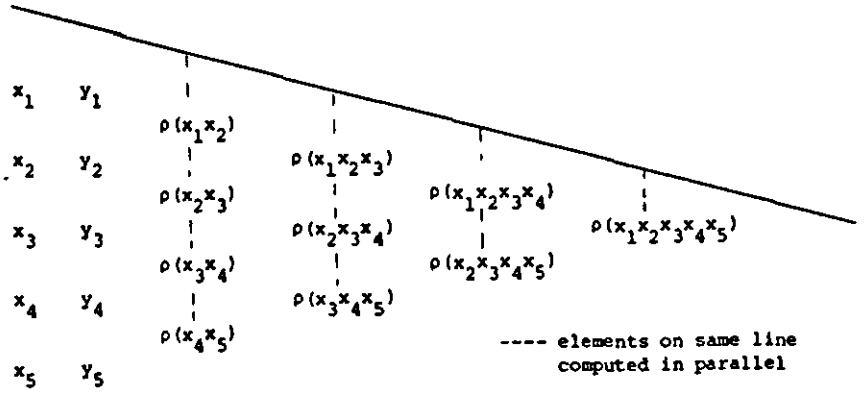
The cell now requires an adder, divider, two subtracters, three pre-loadable registers and a delay for the x_{in} value, and computes as follows,

t: $r_0=x_{in}-x_i$; $r_1=\rho_{in}-y_i$

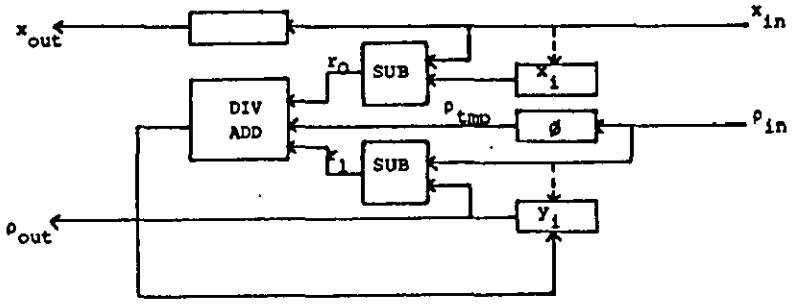
t+½: $y_i=\{r_0/r_1\}+\rho_{tmp}$

t+1: $\rho_{out}=y_i$; $\rho_{tmp}=\rho_{in}$

Snapshots of array operation are shown in Fig.(7.3.6), again no



a) Reciprocal difference table generation



b) Reciprocal difference cell

FIGURE 7.3.5: Reciprocal difference template

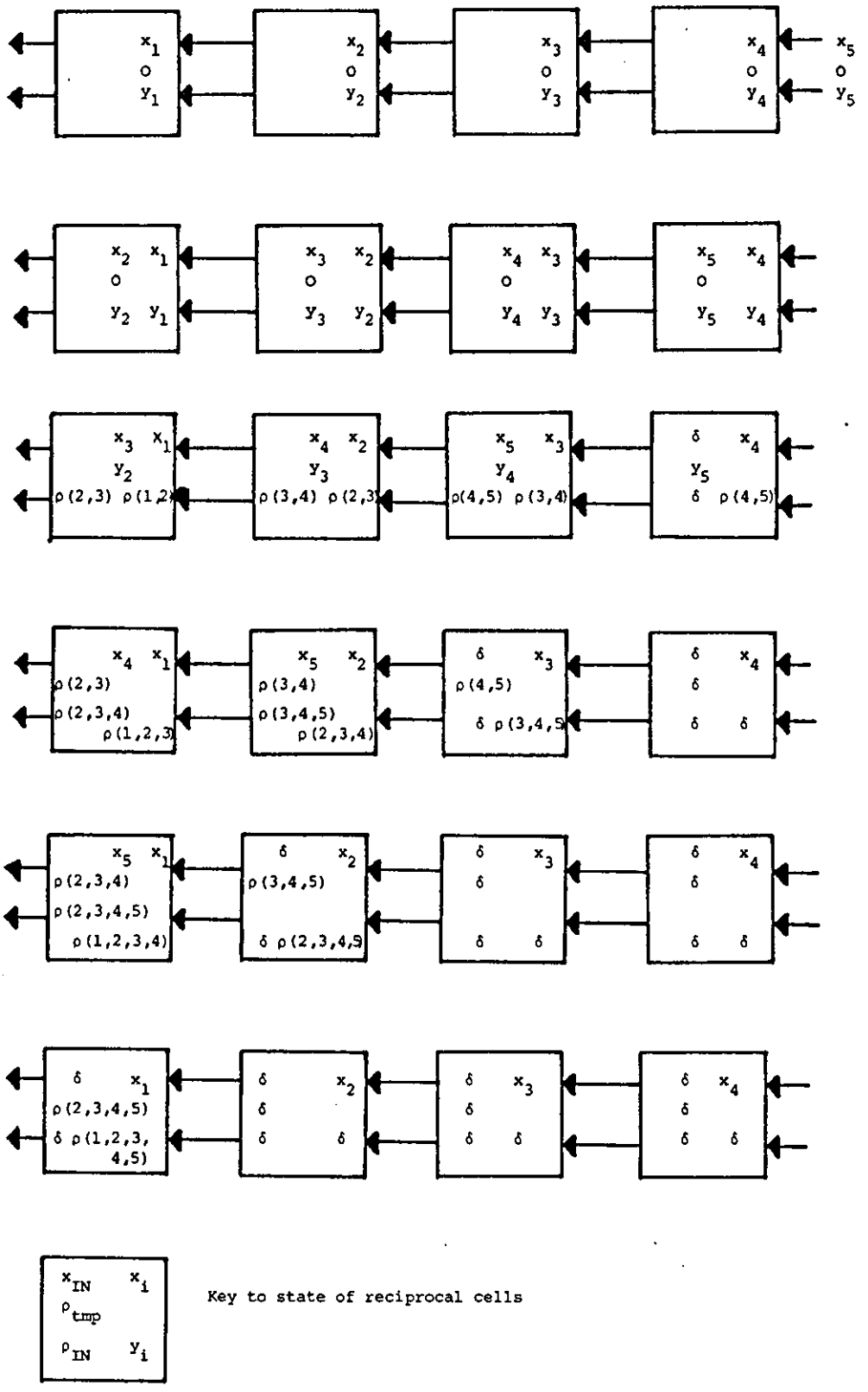
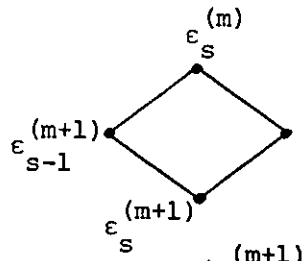


FIGURE 7.3.6: Dataflow in reciprocal difference array

systolic rings can be used and the computation time is T=2cn where $c=\tau_2+2\tau_1$ (with the cost of add equivalent to subtract).

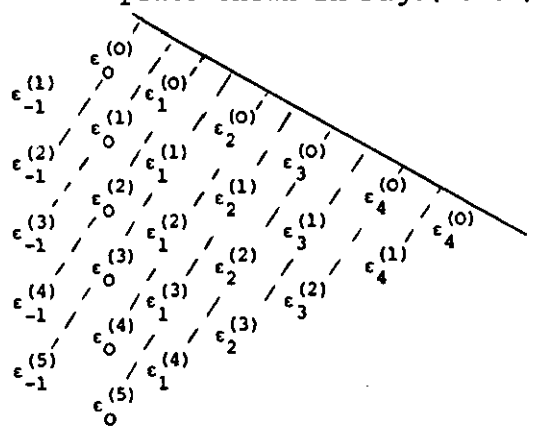
We now have two templates for table generation, one orientated to cell column generation the other to cell row generation. The final algorithm we consider is the Epsilon algorithm (Wynn [62]) a powerful technique for accelerating a slowly convergent sequence of values. The basic computational rule and cell function R_6 is given by,



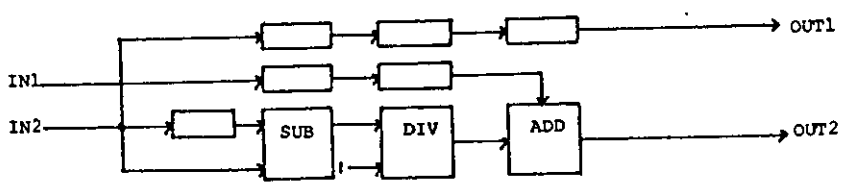
$$\varepsilon_{s+1}^{m} \Rightarrow \varepsilon_{s+1}^{(m)} = \varepsilon_{s-1}^{(m+1)} + \frac{1}{(\varepsilon_s^{(m+1)} - \varepsilon_s^{(m)})}$$

or
$$R_6(\varepsilon_s^{(m+1)}, \varepsilon_{s-1}^{(m+1)}, \varepsilon_s^{(m)}) = \varepsilon_{s+1}^{(m)} \qquad (7.3.10)$$

and the associated table template shown in Fig.(7.3.7). Notice that



a) ε-table generation



b) ε-cell structure

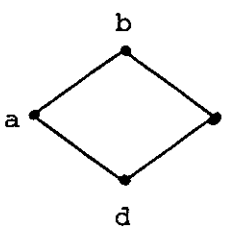
FIGURE 7.3.7: Epsilon ε-algorithm template

due to the lack of x_i arguments R^6 fits the cell row oriented template, and that the associated cell structure is pipelined permitting systolic rings. Yet the epsilon method has a very similar structure to reciprocal differences and hence can be implemented by both templates. It follows that the second unequally spaced argument template is more general, with the equally spaced problems producing a special form of R function which allows further pipelining. For completeness we state the computation order for Fig.(7.3.7b).

$$
\begin{aligned}
t:\quad & IN1\ \ \varepsilon_{-1}^{(0)},\ IN2\ \ \varepsilon_{0}^{(0)} \\
t+\tfrac{1}{4}:\quad & IN2\ \ \varepsilon_{-1}^{(1)},\ IN2\ \ \varepsilon_{0}^{(1)},\ \ r_0=\varepsilon_{0}^{(1)}-\varepsilon_{0}^{(0)} \\
t+\tfrac{1}{2}:\quad & IN1\ \ \varepsilon_{-1}^{(2)},\ IN2\ \ \varepsilon_{0}^{(2)},\ \ r_1=1/r_0,\ \ r_0=\varepsilon_{0}^{(2)}-\varepsilon_{0}^{(1)} \\
t+\tfrac{3}{4}:\quad & IN1\ \ \varepsilon_{-1}^{(3)},\ IN2\ \ \varepsilon_{0}^{(3)},\ \ r_2=\varepsilon_{-1}^{(1)}+r_1,\ r_1=1/r_0,\ \ r_0=\varepsilon_{0}^{(3)}-\varepsilon_{0}^{(2)}
\end{aligned}
$$

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

Finally we are in a position to discuss a useful application of unification the USAD array. The basic principle is to derive a cell function of the form,



$$\Rightarrow R_7(R_3(R_1,R_2),R_4,R_5,R_6) \qquad (7.3.11)$$

All the cell functions R_3-R_6 fit this rhombus pattern. For instance omitting vertex a) produces forward and backward differences, while divided, reciprocal differences and Wynn's algorithm fit the full rule. We require a minimal cell architecture which computes (7.3.11) using a set of switches to control the type of computation rule. Careful

consideration indicates that the reciprocal difference cell contains all the hardware necessary and only the controls need to be added. We augment the reciprocal difference cell with three control bits, c_i, i=1(1)3 with the following interpretations.

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 "
0	1	0	-	-
0	1	1	r_1	Backward "
1	0	0	$r_0/r_1 + \rho_{in}$	Reciprocal "
1	0	1	$1/r_1 + \rho_{in}$	ε-Algorithm
1	1	0	-	-
1	1	1	-	-

where controls can be interpreted as commands to the reciprocal cell such that,

$$c_1 = \begin{cases} 0 & r_0/r_1 \\ 1 & r_0/r_1 + \rho_{in} \end{cases}, \quad c_2 = \begin{cases} 0 & \text{Normal operand order} \\ 1 & \text{switch operand order} \end{cases}$$

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

while,

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

c_2 now performs the switching tasks for R_3, while c_3 provides a neutral value to mask out the divider for forward and backward differences, or acts as a reciprocal cell for the epsilon method. c_1 masks out the adder, while the command S allows r_1/r_0 to be computed when necessary.

The array preloading is trivial and is not discussed here, while the timing of the array is identical to the reciprocal difference array. The additional switching and hardware is simple combinational logic and does not add significant time to the algorithm.

Although the algorithms discussed in this section are computationally simple, the unified array and the method by which they are analysed (using templates) is important for future systolic array designs. We have allowed a number of problems to be implemented on the same cell architecture to produce a cost effective VLSI design from a soft systolic starting point. Recent trends in systolic array development (in particular the CMU WARP processor) are aimed at more flexible systolic array design. The frequent use of the methods examined here should make a USAD device an interesting alternative for fast computation of approximating functions.

7.4 A SYSTOLIC ARRAY FOR THE QUOTIENT DIFFERENCE ALGORITHM

Finally, to finish this study of linear arrays for table generation we consider another important area of numerical analysis; finding the roots of polynomial equations. Many methods are available, and the choice of technique depends upon whether all the roots are required or only a few, whether roots are real or complex, simple or multiple, or if first approximations are available. In all cases the rapid production of the required roots is a primary concern. We consider a systolic design for producing all the roots of a polynomial by a table generating procedure called the quotient-difference (QD) algorithm.

This new design complements the above arrays because firstly it

allows the analysis of the effect of triangular versus rectangular table construction, and second it examines the problem of constructing an open ended (potentially infinite) table on a finite sized array.

Now let,

$$P(x) = a_0x^n + a_1x^{n-1} + \ldots + a_n \tag{7.4.1}$$

be a polynomial with all its roots distinct (i.e. none with the same absolute value), denote the dominant root r_1. This root may be found by computing the solution sequence of the associated difference equation,

$$a_0x_k + a_1x_{k-1} + \ldots + a_nx_{k-n} = 0 \tag{7.4.2}$$

and setting,

$$r_1 = \lim_{k\to\infty} \left(\frac{x_{k+1}}{x_k}\right) . \tag{7.4.3}$$

This follows because $p(x)=0$ is the characteristic equation of (7.4.2) and has a solution which can be written in the form,

$$x_k = c_1r_1^k + c_2r_2^k + \ldots c_nr_n^k , \tag{7.4.4}$$

where r_i, $i=2(1)n$ are the remaining roots of $p(x)$. If $c_1\neq 0$ simple manipulation yields,

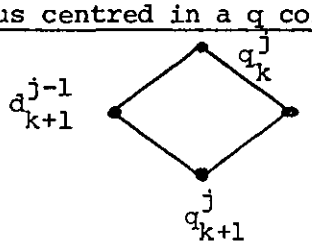
$$\frac{x_{k+1}}{x_k} = r_1\left\{\frac{1+(c_2/c_1)(r_2/r_1)^{k+1}+\ldots+(c_n/c_1)(r_n/r_1)^{k+1}}{1+(c_2/c_1)(r_2/r_1)^{k}+\ldots+(c_n/c_1)(r_n/r_1)^{k}}\right\} \tag{7.4.5}$$

and as r_1 is the dominant root,

$$\lim(r_i/r_1) = 0, \quad i=2(1)n , \tag{7.4.6}$$

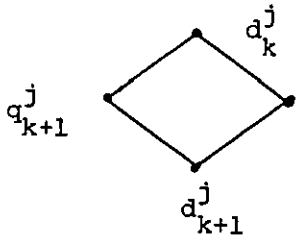
and (7.4.3) follows from (7.4.5) immediately. By extension of (7.4.5) if we define $q_k^1=\frac{x_{k+1}}{x_k}$ and $d_k^0=0$, and construct Table (7.4.1) using the following rhombus rules:

(i) <u>A rhombus centred in a q column</u>



$$\Rightarrow \quad q_k^j+d_k^j = d_{k+1}^{j-1}+q_{k+1}^j$$

(ii) A rhombus centred in a d column



$$\Rightarrow \quad d_k^j q_k^{j+1} = q_{k+1}^j d_{k+1}^j$$

producing two alternative forms,

$$\text{a)} \quad d_k^j = q_{k+1}^j - q_k^j + d_{k+1}^{j-1} \quad \text{or} \quad \text{b)} \quad q_{k+1}^j = q_k^j + d_k^j - d_{k+1}^{j-1} \tag{7.4.7}$$

$$\text{a)} \quad q_k^{j+1} = (q_{k+1}^j * d_{k+1}^j)/d_k^j \quad \text{or} \quad \text{b)} \quad d_{k+1}^j = (d_k^j * q_k^{j+1})/q_{k+1}^j \tag{7.4.8}$$

respectively indicates that,

$$\lim_{k\to\infty} q_k^j = r_j \ , \ j=1(1)n \ . \tag{7.4.9}$$

Furthermore when no two roots have the same absolute value, and provided that no division by zero occurs in (7.4.8) during table construction, (7.4.9) implies,

$$\frac{d_{k+1}^j}{d_k^j} = \frac{q_k^{j+1}}{q_{k+1}^j} \to \frac{r_{j+1}}{r_j} < 1 \ , \tag{7.4.10}$$

and the d_k^j converge (geometrically) to zero. When d_k^j values do not converge p(x) has some roots with equal absolute values and a more complicated root finding procedure is required. Thus q_k^j approximates r_j, and d_k^j defines a measure indicating how close q_k^j is to the root. However, if d_k^j is not convergent and d_k^{j+1} and d_k^{j-1} do converge, the presence of a complex conjugate root is indicated. In this special circumstance the roots can be extracted by solving the quadratic,

$$p_j = x^2 - A_j x + B_j \ , \tag{7.4.11}$$

where,

$$A_j = \lim_{k\to\infty} (q_{k+1}^j + q_k^j), \quad B_j = \lim_{k\to\infty} (q_k^j q_k^{j+1})$$

REMARK: Choosing $x_{-n+1}=\ldots x_{-1}=0$, $x_0=1$ guarantees $c_1 \neq 0$ in (7.4.5). The

proof of this and the above results is beyond the scope of the thesis but the reader is referred to Rutishauser [51].

	q_0^1		q_{-1}^2		q_{-2}^3		q_{-3}^4
0		d_0^1		d_{-1}^2		d_{-2}^3	
	q_1^1		q_0^2		q_{-1}^3		q_{-2}^4
0		d_1^1		d_0^2		d_{-1}^3	
	q_2^1		q_1^2		q_0^3		q_{-1}^4
0		d_2^1		d_1^2		d_0^3	
	q_3^1		q_2^2		q_1^3		q_0^4
0		d_3^1		d_2^2		d_1^3	
	q_4^1		q_3^2		q_2^3		q_1^4
0		d_4^1		d_3^2		d_2^3	
	q_5^1	⋮	q_4^2	⋮	q_3^3	⋮	q_2^4
	⋮		⋮		⋮		⋮

TABLE 7.4.1: Generalised quotient difference table

Next observe that the type of QD table depends on the choice of k.

(i) The 'column-by-column' (c-by-c) Table: results when $k \geq 0$ for all j and produces an open trapezium structure, with the top boundary enforcing the use of (7.4.7a) and (7.4.8a).

(ii) The 'row-by-row' (r-by-r) Table: results when $k < 0$ for some j and causes the table to form an open ended rectangular structure by the use of (7.4.7b) and (7.4.8b).

The c-by-c method is extremely sensitive to rounding error and requires the construction of the x_k terms as starting values. The r-by-r method controls the error by the use of fictitious entries chosen to force the correct behaviour onto the upper boundary of the c-by-c table. The fictitious entries are introduced by filling the top two rows of the table with the values,

row 1 $\quad -a_1/a_0 \qquad 0 \qquad 0 \qquad 0 \qquad 0 \qquad \cdots \qquad 0$

row 2 $\quad 0 \qquad a_2/a_1 \quad a_3/a_2 \quad a_4/a_3 \qquad\qquad a_n/a_{n-1} \qquad 0$

Both methods are illustrated in Tables (7.4.2) and (7.4.3) for the polynomial x^2-x-1 associated with the Fibonacci sequence x_k+x_{k+1} for $k>0$. Notice the greater stability in the second root and the fact that the x_k values are not required by the r-by-r method which is a clear advantage.

k	x_k	d_k^0	q_k^1	d_k^1	q_k^2	d_k^2
0	1	0				
			1.0000			
1	1	0		1.0000		
			2.0000	-.5000	-1.0000	-.0001
2	2	0				
			1.5000	.1667	-.5001	-.0001
3	3	0				
			1.6667	-.0667	-.6669	.0005
4	5	0				
			1.6000	.0250	-.5997	.0007
5	8	0				
			1.6250	-.0096	-.6240	-.0082
6	13	0				
			1.6154	.0037	-.6226	
7	21	0				
			1.6190			
8	34	0				

TABLE 7.4.2: Column by column method for x^2-x-1

k	d	q	d	q	d
		1		0	
1	0		1		0
		2		-1	
2	0		-.5000		0
		1.5000		-.5000	
3	0		.1667		0
		1.6667		-.6667	
4	0		-.0667		0
		1.6000		-.6000	
5	0		.0250		0
		1.6250		-.6250	
6	0		-.0096		0
		1.6154		-.6154	
7	0		.0037		0
		1.6191		-.6191	
8	0				0

TABLE 7.4.3: Row by row method for x^2-x-1

Table (7.4.4) illustrates a suitable ordering of elements for the systolic construction of the QD-table. The pattern is similar to those used previously indicating that the same techniques are applicable, and suggest a linear systolic array of 2n cells for a QD-table with n roots (but 2n columns). In this intuitive design we define two types of cells corresponding to the two rhombus rules, and allocate Q-cells to odd numbered columns and D-cells to even columns of the table. Next we observe that a D-cell requires only multiply and divides, and the Q-cell only adds and subtracts. Thus applying the principle of unification we merge pairs of adjacent odd and even cells to create a single unified QD-cell with hardware equivalent to two inner product cells, producing a linear array with n QD-cells. This new array utilises the observation that the QD table can be partitioned by odd and even columns to yield two distinct but related tables of size n, one for root approximations, the second for indicating root convergence. Hence the QD array is essentially a unified array which interleaves the construction of both tables on the same linear array.

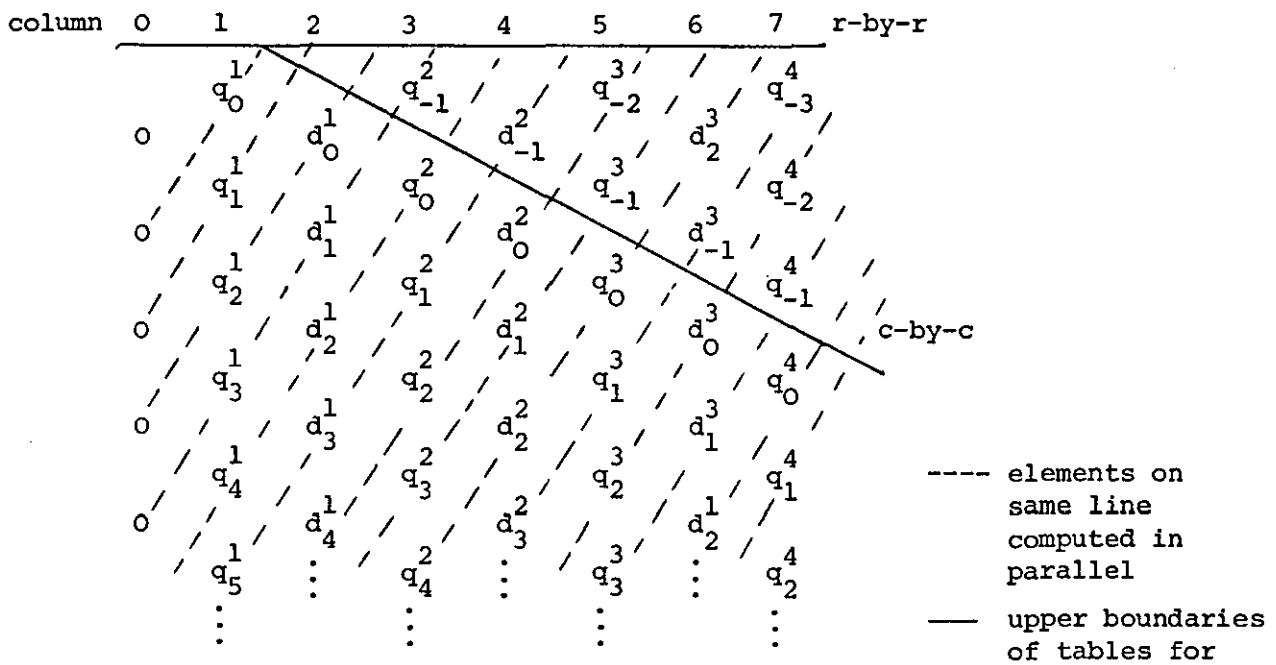
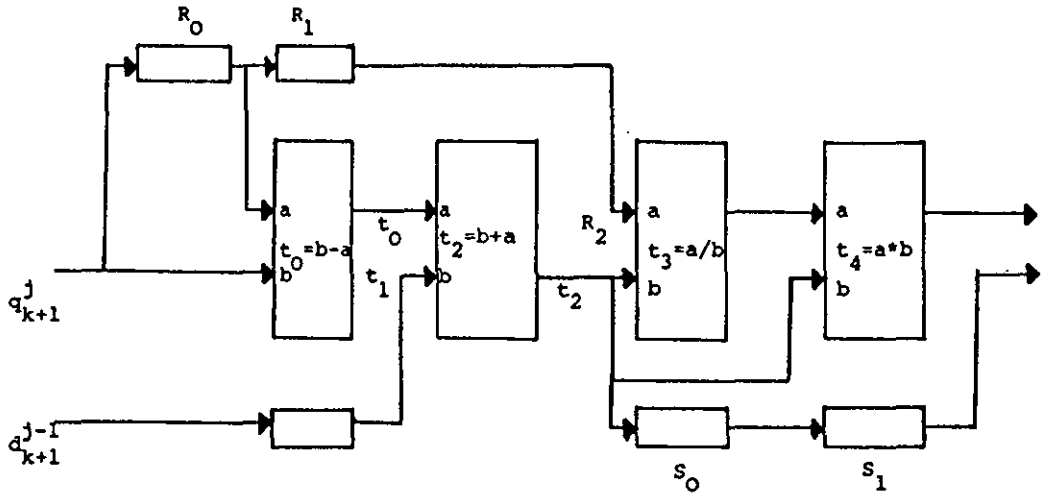
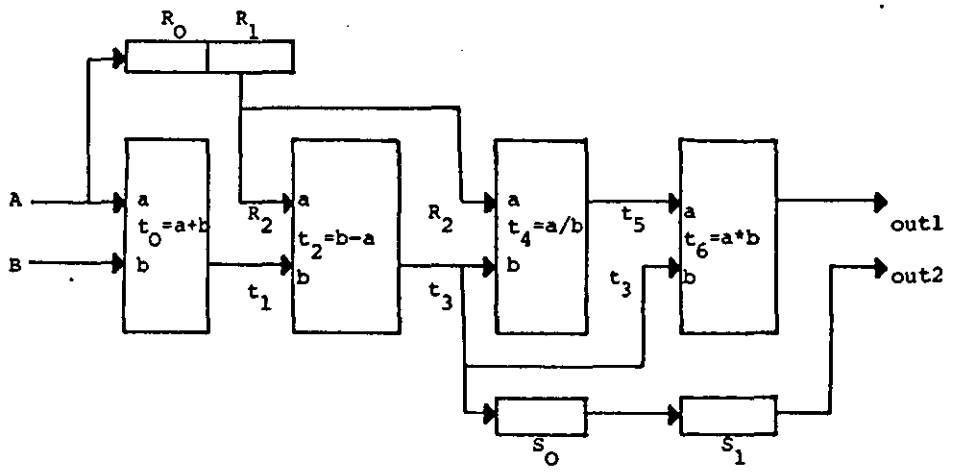


TABLE 7.4.4: Parallel evaluation of QD table entries

The QD cell computation is determined by (7.4.7) and (7.4.8), notice that (7.4.7) must be computed before (7.4.8) and demands a two level pipelined approach to the cell design. Choosing (7.4.7a) and (7.4.8a) produces the cell in Fig.(7.4.1a) suitable for the c-by-c method, and selecting (7.4.7b), (7.4.8b) the cell in Fig.(7.4.1b) is defined for the r-by-r method. Tracing the data flow in the respective cells reveals interesting relations regarding array data flow and the order of table generation. The c-by-c QD-cell creates a stationary array in the sense that every input q_k^j, d_{k+1}^j to cell j produces an output q_k^{j+1}, d_k^j tying columns 2j-1 and 2j, j=1(1)n of the QD table to cell j. Consequently the successive root approximations q_k^j, k=1,2,..., to r_j remain fixed in the same cell, and the array must contain n cells to evaluate all the roots. Now in order to judge root convergence we must obtain the values of d_k^j from the array, assuming a host machine supplies values to and from the array, collection of the d_k^j implies a fanin type network with an output associated with each cell. Once convergence of roots is achieved we encounter the additional problem of stopping the array and unloading the results. In contrast the r-by-r QD-cell produces a non-stationary array, with the inputs q_k^j, d_{k+1}^{j-1} of cell j producing the output q_{k+1}^j, d_{k+1}^j. It follows that successive approximations to the roots r_j, j=1(1)n are pumped systolically from left to right along the array. Furthermore, this non-stationary array requires z (>0) cells where z is the number of approximations to each root on a single pass through the array, with results emerging from the right hand end of the array without any special effort. Thus, the fact that the r-by-r is more stable than the c-by-c scheme together with the non-stationary QD-cell arrangement makes row-by-row table



a) Stationary c-by-c cell



b) Non-stationary r-by-r cell

FIGURE 7.4.1: QD Cells

construction preferable. A sequence of snapshots for the non-stationary QD-cell are shown in Table (7.4.5b) for the row-by-row orientated solution to,

$$x^4 - 10x^3 + 35x^2 - 50x + 24 = 0 \ , \qquad (7.4.12)$$

in Table (7.4.5a). Careful observation of the snapshot data reveals

k	d	q	d	q	d	q	d	q	d
		10		0		0		0	
1	0		-3.5000		-1.4286		-.4800		0
		6.5000		2.0714		.9486		.4800	
2	0		-1.1154		-.6542		-.2329		0
		5.3846		2.5326		1.3599		.7229	
3	0		-.5246		-.3513		-.1291		0
		4.8600		2.7059		1.5821		.8520	
4	0		-.2921		-.2054		-.0695		0
		4.5679		2.7926		1.7180		.9215	
5	0		-.1786		-.1264		-.0373		0
		4.3893		2.8448		1.8071		.9588	
6	0		-.1158		-.0803		-.0198		0
		4.2735		2.8803		1.8676		.9786	
7	0		-.0780		-.0521		-.0104		0
		4.1955		2.9062		1.9093		.9890	
8	0		-.0540		-.0342		-.0054		0
		4.1415		2.9260		1.9381		.9944	

TABLE 7.4.5a: Row-by-row QD table for $p(x)=x^4-10x^3+35x^2-50x+24=0$ (Exact solutions x=1,2,3,4).

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	
R[0]	0.0	-3.5	-1.4286	-0.48	0.0	0.0	0.0	
R[1]	0.0	0.0	-3.5	-1.4286	-0.48	0.0	0.0	
R[2]	0.0	0.0	0.0	-3.5	-1.4286	-0.48	0.0	
t[0]	0.0	6.5	-1.4286	-0.48	0.0	0.0	0.0	
t[1]	0.0	0.0	6.5	-1.4286	-0.48	0.0	0.0	
t[2]	0.0	0.0	6.5	2.0714	0.9486	0.48	0.0	
t[3]	0.0	0.0	0.0	6.5	2.0714	0.9486	0.48	
t[4]	0.0	0.0	0.0	-0.53846	-0.6897	-0.5060	0.0	
t[5]	0.0	0.0	0.0	0.0	-0.53846	-0.6897	-0.5060	
t[6]	0.0	0.0	0.0	0.0	-1.1154	-0.6542	-0.2429	
s[0]	0.0	0.0	0.0	6.5	2.0714	0.9486	0.48	
s[1]	0.0	0.0	0.0	0.0	6.5	2.0714	0.9486	
A	-3.5	-1.4286	-0.48	0.0	0.0	0.0	0.0	
B	10	0.0	0.0	0.0	0.0	0.0	0.0	etc.
out 1	0	0	0	0.0	0.0	-1.1154	-0.6542	
out 2	0	0	0	0.0	0.0	6.5	2.0714	

TABLE 7.4.5b: Snapshots of r-by-r cell operation for above problem

further improvement to the r-by-r array. First the cell latency is c=4. Thus for z=n, after n cycles the $\lceil n/c \rceil = \lceil n/4 \rceil$ QD-cell has performed one complete computation, and the last a_n/a_{n-1} value of the top two table rows has been input to cell 1. It follows that wrapping the output of cell $\lceil n/4 \rceil$ around to cell 1 forms a systolic ring with $\lceil n/4 \rceil$ QD-cells. This ring will generate an infinite table of root approximations, with $\lceil n/4 \rceil$ rows of the table on each cycle of the ring. Allowing a trade-off of two inner product cells for each QD-cell implies that all the roots of p(x) in (7.4.1) can be found with only $\frac{1}{2}n$ inner product cells.

Timing of the r-by-r arrays is complicated by the fact that convergence depends on the input polynomial. If we let ℓ be the total number of QD table rows required for convergence of all roots a linear array of z cells requires $\lceil \ell/z \rceil$ passes of the root approximations through the array. The approximation to r_1 on the first pass is output after cz=4z cycles, and has to wait n-4z cycles for the remaining roots of the pass to enter the array before the second pass can start. Thus, the total computation time is

$$T = \lceil \ell/z \rceil n + 4z \ , \tag{7.4.13}$$

and for a systolic ring where z=n/4

$$T = \lceil 4\ell/n \rceil n + n < 2n + 4\ell \ . \tag{7.4.14}$$

Finally, the generation of the first two starting rows in the r-by-r table can be pipelined with array operation by the addition of a simple boundary cell consisting of a divider, negater and some switching logic, as shown in Fig.(7.4.2).

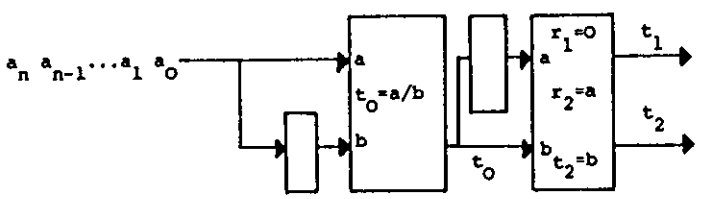


FIGURE 7.4.2: Boundary cell

A control bit c_1 tagged to the a_i is used to select the correct negator output according to,

$$t_1 = \begin{cases} r_1 & c_1=0 \\ r_2 & c_1=1 \end{cases}$$

and can also be used (with suitable delays) to control the switching of the input data from host machine to the $\lceil n/4 \rceil$ th QD-cell in the systolic ring.

We conclude that the non-stationary arrays incorporate better systolic data flow and can be considered superior to the stationary arrays as the computations are numerically more stable. Each non-stationary array with the addition of a boundary cell requires as input the polynomial coefficients $a_0, a_1, \ldots, a_n$ and produces an output sequence of root approximations (q_j^k) and a sequence of error indicators (d_k^j). These sequences by convergence or divergence of the d_k^j values can determine either:

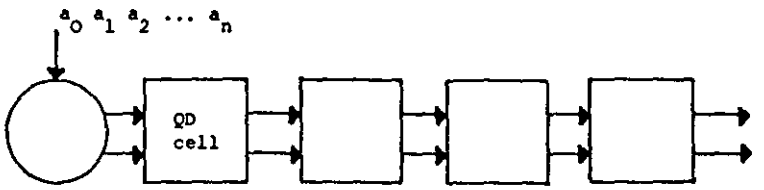
(i) all the roots of the polynomial (where they are distinct)

(ii) the existance of multiple roots

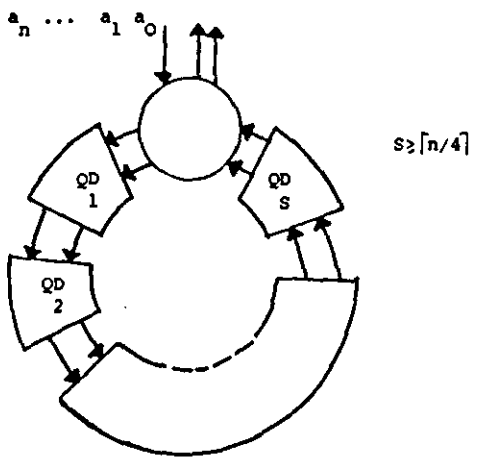
(iii) the existance of complex conjugate roots,

and in the latter case the q_k^j sequence provides enough data for a quadratic from which the complex roots can be extracted, (by standard techniques), to be constructed.

Clearly, the utility of the array is limited as a general root finder (by (i)), however, we suggest that it could be used as an inexpensive 'add-on' extra to an existing machine to provide quick root approximations and/or provide data to influence the choice or prime more sophisticated root finding procedures. In this sense the array is a system resource which could be called as a procedure, and a soft-systolic version of the algorithm is given in the Appendix.



a) linear array



b) Systolic ring (with modified boundary cell)

FIGURE 7.4.3: Systolic arrays for the QD algorithm

7.5 A SYSTOLIC SIMPLEX ALGORITHM

Next we consider more complex table generating algorithms for linear programming and in particular the Simplex algorithm. Linear programming techniques themselves are extremely useful in many applications such as:

1. Agricultural applications - National and regional scale
2. Procurement of contract awards
3. Economic aids (Leontief inter-industry model)
4. Industrial applications (chemical, coal, airline, etc.)
5. Military applications (strategic and logistic)
6. Personnel assignment
7. Production scheduling and inventory control
8. Structural design
9. Traffic analysis
10. Transportation problems and network theory
11. Travelling/salesman problem
12. Statistics, combinatorial analysis and graph theory
13. Design of optical filters.

Thus a fast and efficient systolic design is well justified. A linear programming (LP) problem consists of a linear function,

$$H = c_1x_1 + \ldots + c_nx_n \ , \tag{7.5.1}$$

which is to be minimised or maximised subject to certain constraints

$$a_{i1}x_1+\ldots+a_{in}x_n \lesssim b_i, \ 0\lesssim x_j, \ i=1(1)m, \ j=1(1)n. \tag{7.5.2}$$

The problem can be written in matrix vector notation as,

$$H(x) = C^Tx = \text{minimum}, \ Ax\lesssim b, \ 0\lesssim x \ , \tag{7.5.3}$$

and from linear programming theory it is known that the minimum (or maximum) occurs at an extreme feasible point. A point $(x_1,\ldots,x_n)$ is

feasible if its coordinates satisfy all (n+m) constraints, whereas an extreme feasible point forces at least n of the constraints to become equalities. By introducing slack variables $x_{n+1},\ldots,x_{n+m}$ the constraints are converted to the form,

$$a_{i1}x_1 + a_{i2}x_2 + \ldots + a_{in}x_n + x_{n+i} = b_i,\ i=1(1)m \qquad (7.5.4)$$

permitting extreme feasible points to be located by having n or more variables (including the slack variables) zero. A solution point is a minimum point of H, if there is more than one solution point, there is more than one extreme feasible point and any such point can be used as a solution.

To date systolic arrays for the LP problem have been limited to least squares approximation for linear systems, where,

$$Ax = b\ , \qquad (7.5.5)$$

is the overdetermined m×n system (for m>n) in (7.5.3) and which satisfies the equations approximately in some 'best' sense. Essentially we calculate the residual,

$$r = b - Ax\ , \qquad (7.5.6)$$

and consider the function

$$\phi(x) = r^T r\ , \qquad (7.5.7)$$

and choose x to minimise $\phi(x)$ (i.e. the sum of the squares). It follows that,

$$\phi(x) = (b-Ax)^T(b-Ax) = x^TA^TAx-(bA^Tx+x^TA^Tb)+b^Tb$$
$$= x^TAAx-2x^TA^Tb+b^Tb \qquad (7.5.8)$$

and (7.5.7) is minimised when the gradient vector $\text{grad}(\phi(x)) = 2A^TAx-2A^Tb=0$ hence,

$$A^TAx = A^Tb\ . \qquad (7.5.9)$$

Thus, forming A^TA and A^Tb produces an n×n matrix problem which can be

solved by the arrays developed in the previous chapters. However the solution does not solve (7.5.5) exactly and we consider the more flexible simplex algorithm which solves the original system by table manipulation.

The simplex algorithm is a method which starts at some extreme feasible point and by a sequence of exchanges proceeds systematically by steadily reducing H to other extreme points until a solution point is found. The use of slack variables (which must be non-negative like the other x_i) allow the identification of extreme feasible points. Since the inequality in $Ax \leq b$ implies a slack variable being zero, an extreme point is one where at least n of the variables $x_1,\ldots,x_{n+m}$ are zero. Alternatively, an extreme feasible point is one where at most m variables are non-zero. The matrix coefficients of (7.5.4) can be expressed as,

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} & 1 & 0 & \cdots & 0 \\ a_{21} & a_{22} & & & 0 & 1 & \ddots & 0 \\ \vdots & \vdots & & & \vdots & \ddots & \ddots & \vdots \\ a_{m1} & \cdots & & a_{mn} & 0 & \cdots & 0 & 1 \end{bmatrix} \tag{7.5.10}$$

with the last m columns corresponding to slack variables. Thus, the (n+m) columns of the matrix can be written as $v_1,v_2,\ldots,v_{n+m}$ and (7.5.4) written as,

$$x_1v_1+x_2v_2 + \ldots + x_{n+m}v_{n+m} = b \ , \tag{7.5.11}$$

and if an extreme feasible point (say for simplicity) $x_{m+1}=\ldots=x_{m+n}=0$ is known there are at most m non-zero variables, hence,

$$x_1v_1+x_2v_2+ \ldots + x_mv_m = b \ , \tag{7.5.12}$$

and

$$H = x_1c_1+x_2c_2+ \ldots + x_mc_m \ . \tag{7.5.13}$$

If the vectors $v_1,\ldots,v_m$ are linearly independent, all (n+m) vectors can be expressed in terms of this basis, viz.

$$v_j = v_{1j}v_1 + \ldots v_{mj}v_m \ , \ j=1(1)n+m \tag{7.5.14}$$

also let,

$$h_j = v_{1j}c_1 + \ldots + v_{mj}c_m - c_j \ , \ j=1(1)n+m. \tag{7.5.15}$$

The Simplex method tries to reduce H by including some amount px_k for k>m and p positive. Thus, in order to preserve the constraints we multiply (7.5.14) with j=k by p and subtract (7.5.12) to get,

$$(x_1-pv_{1k})v_1 + (x_2-pv_{2k})v_2 + \ldots + (x_m-pv_{mk})v_m + pv_k = b, \tag{7.5.16}$$

and from (7.5.13) and (7.5.15) the new H is,

$$(x_1-pv_{1k})c_1+(x_2-pv_{2k})c_2+\ldots+(x_m-pv_{mk})c_m+pc_k = H_1-ph_k \tag{7.5.17}$$

Clearly to reduce H_1 $h_k>0$, and p must be as large as possible without making (x_i-pv_{ik}) negative hence,

$$x_\ell/v_{\ell k} = \min_i (x_i/v_{ik}) = p \ , \tag{7.5.18}$$

with the minimum taken over only the positive v_{ik} terms. Clearly with this choice of p the c_ℓ coefficient must become zero, and as the remaining points are non-negative we have created a new extreme feasible point with a better result $\bar{H}_1=H_1-ph_k$. The basis also needs to be updated by exchanging v_ℓ for v_k, which is performed as follows,

$$v_k = v_{1k}v_1 + \ldots + v_{mk}v_m \ . \tag{7.5.19}$$

Solving for v_ℓ and substituting into (7.5.14) yields,

$$v_j = \bar{v}_{1j}v_1 + \ldots \bar{v}_{\ell-1,j}\, v_{\ell-1}+\bar{v}_{kj}v_k+v_{\ell+1,j}\, \bar{v}_{\ell+1}+\ldots\bar{v}_{mj}v_m,$$

where,

$$\bar{v}_{ij} = \begin{cases} v_{ij}-(v_{\ell j}/v_{\ell k})v_{ik}, & i\neq\ell \\ v_{ij}/v_{\ell k} \ , & i=\ell \end{cases} \tag{7.5.20}$$

Substituting for v_ℓ in (7.5.11) gives,

$$\bar{x}_1v_1+\ldots+\bar{x}_{\ell-1}v_{\ell-1} + \bar{x}_kv_k+\bar{x}_{\ell+1}v_{\ell+1}+\ldots+\bar{x}_mv_m = b \ ,$$

with

$$\bar{x}_i = \begin{cases} x_i - (x_\ell / v_{\ell k}) v_{ik}, & i \neq \ell \\ x_i / v_{\ell k}, & i = \ell. \end{cases} \tag{7.5.21}$$

Also, $\bar{h}_j = \bar{v}_{1j} c_1 + \ldots + \bar{v}_{mj} c_m - c_j = h_j - (v_{\ell j}/v_{\ell k}) h_k$

with,

$$\bar{H}_1 = H_1 - (x_\ell / v_{\ell k}) h_k .$$

The method is then iterated until either all the h_j are negative, or until for some $h_k > 0$ no v_{ik} is positive. In the first case, the current point is as good as any adjacent extreme point. In the second case, p can be arbitrarily large and there is no minimum for H. For further reading on LP problems, their applications and the Simplex method see Chvatal [80], Wu & Coppins [81], Llewellyn [64], Gass [69].

The Simplex procedure can be represented compactly by the tabular form,

$$\begin{bmatrix} x_1 & v_{11} & v_{12} & \cdots & v_{1,n+m} \\ x_2 & v_{21} & v_{22} & \cdots & v_{2,n+m} \\ \vdots & \vdots & \vdots & & \vdots \\ x_m & v_{m1} & v_{m2} & \cdots & v_{m,n+m} \\ H_1 & h_1 & h_2 & \cdots & h_{n+m} \end{bmatrix} \tag{7.5.22}$$

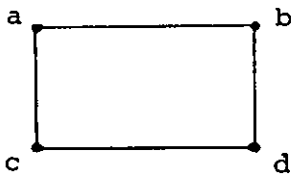
and summarized by the following six steps:

(i) Call $v_{\ell k}$ the pivot (i.e. $p = x_\ell / v_{\ell k}$) the part to be added.

(ii) Divide the entries in the pivot row by the pivot.

(iii) The pivot column becomes zero except for 1 in the pivot position.

(iv) All other entries are modified by the rectangle rule



$$d=d-(\frac{b}{a})c$$

with $a=v_{\ell k}$ and $c=v_{ik}$.

(v) Find the largest new h_j, j=1(1)n+m which is positive (terminate if there are none).

(vi) Find the new pivot according to (7.5.18) with the column indexed j. If none are positive then stop.

The global structure of a wavefront orientated architecture is shown in Fig.(7.5.1), and basically consists of an (m+1)*(n+m+1) orthogonally

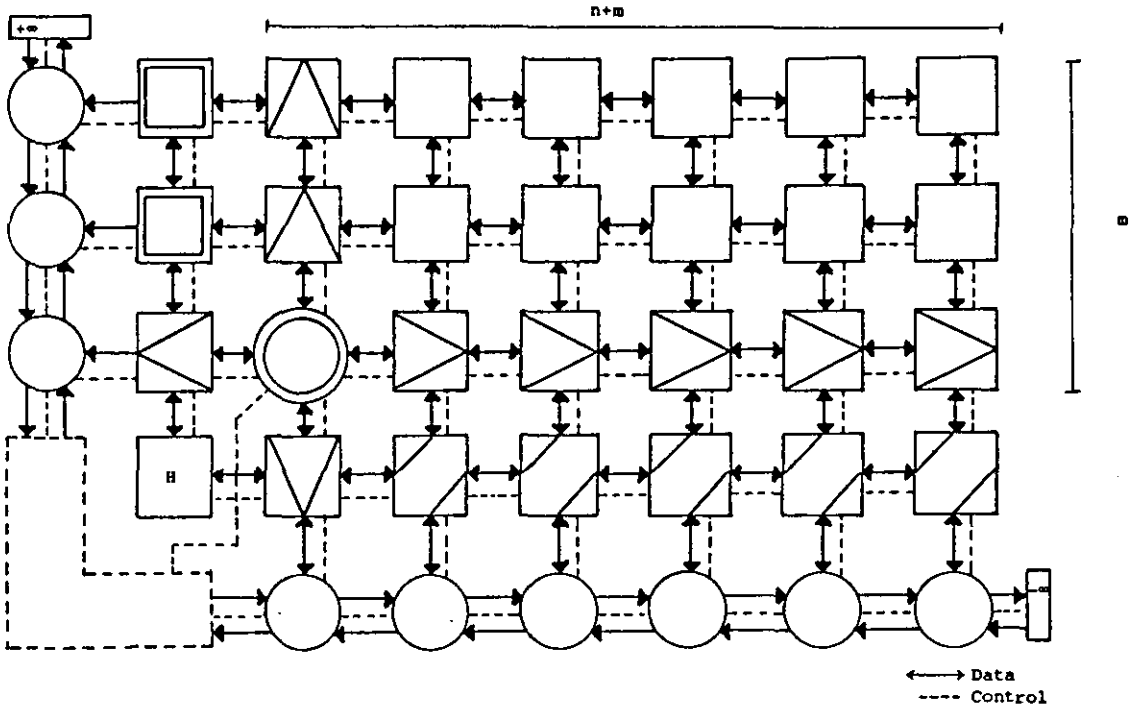


FIGURE 7.5.1: Systolic array for the Simplex algorithm (m=3, n=3)

connected array representing Table (7.5.22) and two boundary arrays which are used for sorting rows and columns of the table. All the connections are bi-directional except for the control lines which consist of two 2-bit one-way connections. These two control lines per link allow the specification of two superimposed and disjoint control networks on the array of processing elements. One network is used for row and column sorting, the other for the application of the rectangle rule and pivot formation. The essential idea behind the array structure is to keep different operations in fixed positions in order to simplify cell definitions. This means positioning the pivot element by a sequence of systolic operations so that it always resides in the (m,2) position of the table, thus making row m the pivot row and column 2 the pivot column. The pivot is chosen to maximise the H reduction so the first step is to find the maximum h_k which in turn selects the v_k (to be swapped with v_ℓ) automatically. Clearly sorting the h_j, j=1(1)n+m into descending order from left to right places h_k and v_k in column 2. Likewise we must sort rows to find the minimised p (after neglecting negative and zero values) by sorting the quotients from (7.5.18) into ascending order from top to bottom with zero and negative values pushed to the area above the largest p. In addition the row and column sorters maintain indexes for rows and columns to keep track of vectors moved in and out of the basis and the m variables in the current solution, so that the final solution is easily recovered after termination of the array. For any swaps generated by the sorters the corresponding table elements must be realigned and this is achieved by control values generated by the sorters and pumped south to north and west to east for column and row sorting respectively. Hence after the two sorts the values

$v_{\ell k}$ and x_{ℓ} must be in positions (m,2) and (m,1) making all the required data for improving the extreme feasible point locally placed. Finally, the column and sorters place the data necessary for the termination tests in the vicinity of the controller which requires only four states to control the whole iteration and,

State (i) Exchange

State (ii) Column sort

State (iii) Form pivot contenders

State (iv) Row sort,

to define a loose sequential structure on the Simplex iteration. Individual cells cycle from (i)-(iv), while from a global viewpoint the array can be in many different states simultaneously. The computation is essentially performed by overlapping the computational wavefronts generated by each state in a manner that prevents interference of individual states and admits widespread parallelism. Alternatively we can consider the whole Simplex iteration as a single wavefront which undergoes a series of reflections and refractions at array boundaries. The practical value of this is that the controller needs only to prompt the pivot cell to change state, using triggers from the row and column sorters. Furthermore, the cost of each cell is bounded by the time of a single inner product step. A row or column swap (i.e., the time to swap elements in adjacent horizontal or vertically aligned cells) is also an inner product step the first half cycle we send the cell element and on the second half cycle receive its replacement - which simplifies communication when a processor switches from row or column state to exchange or pivot forming states.

The global computation of the array can be easily understood by

tracing the wavefronts associated with each state as demonstrated in Fig.(7.5.2). For simplicity t=1 (in Fig.(7.5.2)) represents the first cycle after the starting table has been loaded into the array, and t=25 depicts the start of the next cycle. It follows that wavefronts of different states never interfere and an estimate for the time of a Simplex iteration can be identified.

Theorem 7.5.1: A single change of an extreme feasible point in the standard Simplex algorithm with n unknowns and m constraints using an orthogonally connected array of O((m+2)(n+m)) cells requires

$$T = (2n+4m+6) \text{ cycles.}$$

Proof: (by observation of the dataflow in Fig.(7.5.2)).

(i) The exchange state requires (n+m) cycles to reach the right hand array boundary, and an extra cycle before the last element can be loaded into the column sorter (i.e. n+m+1 cycles).

(ii) If this last element is the largest h_j it will take (n+m) cycles to reach the h_k position in the pivot column, by a sequence of interchanges.

(iii) On the next cycle the last swap enters the (m+1)st row of the table going south to north and on the second cycle reaches the pivot cell.

(iv) Thus, on the third cycle after the end of column sorting, the pivots drop into 'form-pivot' state as the correct v_k from the pivot downwards have been formed.

(v) After a further m cycles the last pivot contenders enter the row sorter, if this value is the smallest pivot (i.e. p) it requires a further m cycles to reach the pivot row.

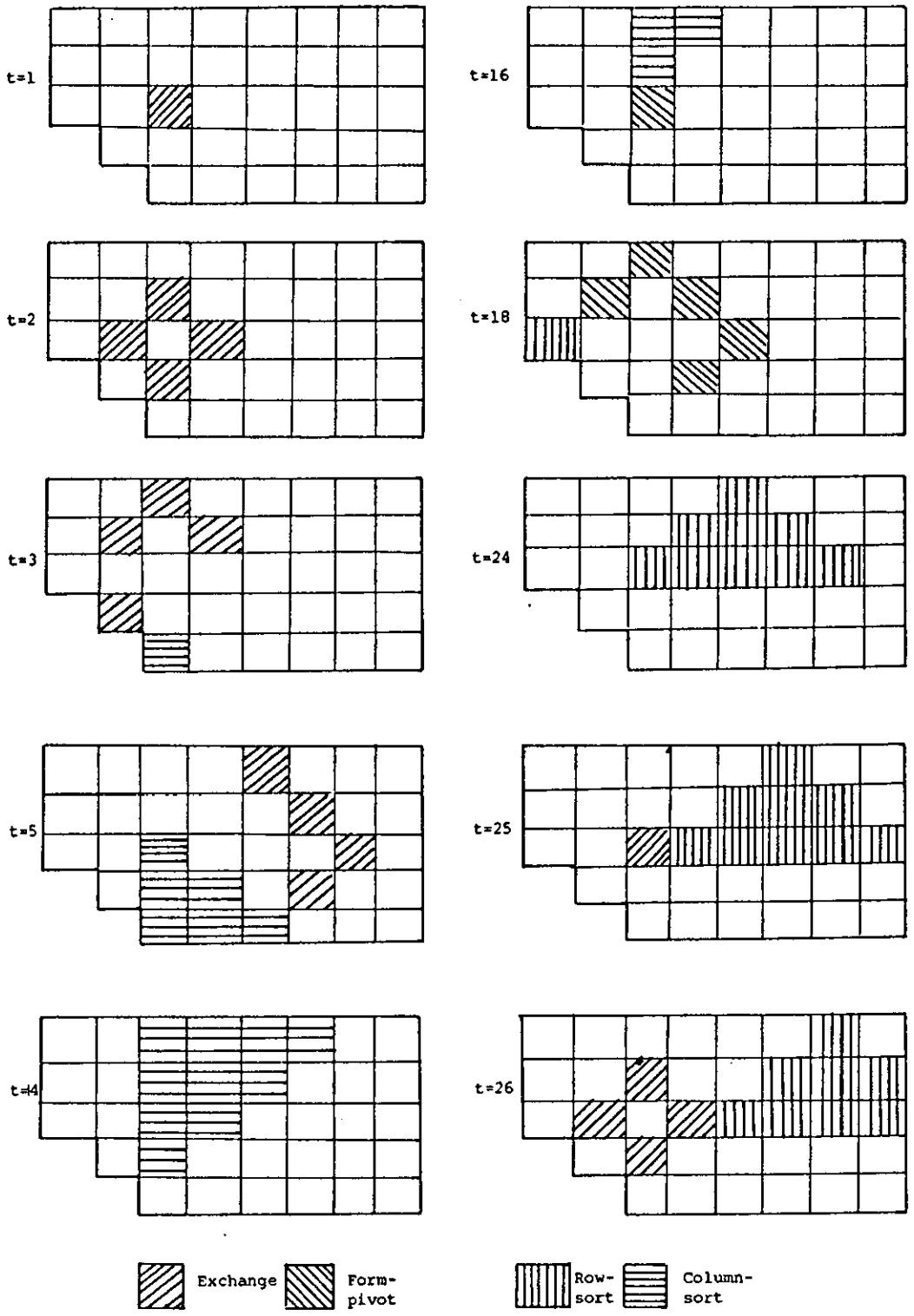


FIGURE 7.5.2: Snapshots of the computational wavefronts for simplex iteration

(vi) An additional 2 cycles sees the last row swap operations pass the pivot column. Thus all the columns to the left of and including the pivot are correct column and row sorted and the next exchange can start.

The H-cell also contains the improved minimum.

Summing these timings ensure the bound T=(2n+4m+6) for a single Simplex update. The area bound is given by:

(i) The table elements require (m+1)(m+n) ips cells

(ii) Column sorting at most (m+n) ips cell equivalents

(iii) Row sorting m ips cell equivalents

(iv) The unknown x_i values and H-cell

a) 2 ips for H and pivot row cell

b) 2(m-1) for remaining unknowns

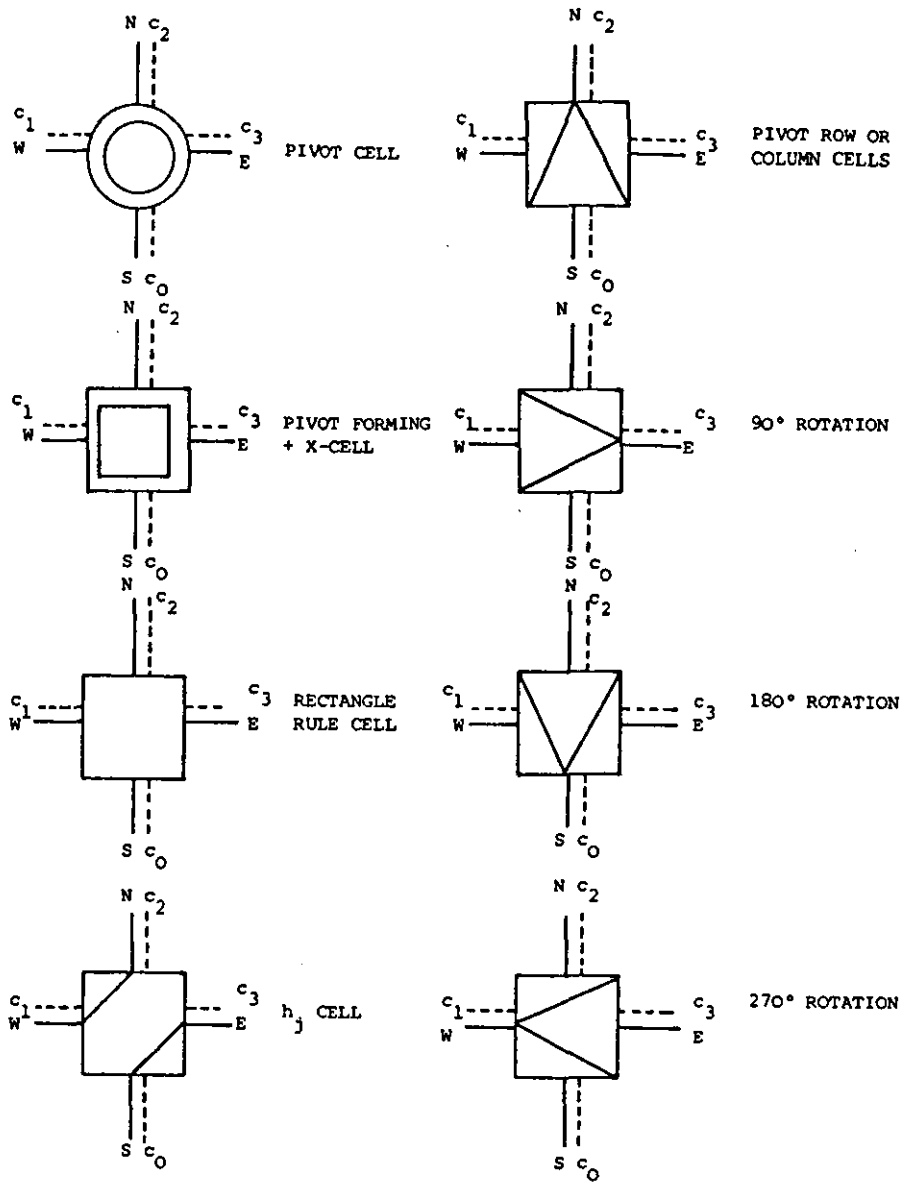
yielding (m+2)(m+n) + 3m ips cell equivalents.

Corollary (7.5.1): A search of z extreme points requires T=z(2n+4m+6)+2m cycles using the Simplex method.

Proof:

The loading and unloading of the starting and final tableaux requires at most an additional time of 2m cycles, then repeating the argument in Theorem (7.5.1) yields the timing immediately.

The Simplex array was simulated using OCCAM and the listing appears in the Appendix, from which the cell definitions can be found using the following key.

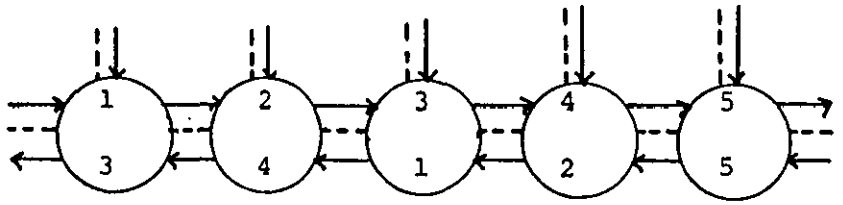


CELL DEFINITIONS: (see code Appendix for internal working)

Snapshots of the array operation on a test example are given below, and the following pertinant remarks concerning array implementation may be of use.

The idea of column sorting is to move the column associated with the largest h_j to the pivot column position, and as stated above the problem reduces to sorting the h_j into descending order from left to right and performing the same swaps on the v_j column vectors. A linear

systolic array for sorting is shown in Fig.(7.5.3a) and implements the well known odd-even transposition or parallel bubblesort and consists of (n+m) cells. The sorting network is slightly different from the standard array because our starting values must be loaded sequentially and the sorting started systolically. The standard array assumed cells were already loaded and that array cells started simultaneously. Our sorting network also generates an (n+m) control bit vector on every cycle (pumped south to north through the table) to control column swaps. Finally the cells must keep track of the index j of v_j so that the row sorter can be informed which column is the new pivot column and hence determine the variable introduced to the solution. Fig.(7.5.3b) illustrates the sorter operation on the worst case list for an array of 5 cells. The key values to watch are the c_1 and $\bar{c}_1$ control values because these determine the array operation time. On the trip left to right c_1 loads the h_j values from the H-cells of the table portion of the array and starts the cells into odd and even operation mode. Where two cells decide to swap elements the control bit c_2 in each cell is set pumped up into the table to swap column elements. On reaching the rightmost sorting cell c_1 loads the last value and falls off the array, and a neutral element '$-\infty$' is used to prevent erroneous swaps at the array boundary. Next the $\bar{c}_1$ signal enters and moves right to left pushing the final maximum of h_j in front of it (by two cycles) and closing down the sorting cells. Thus after 2(m+n)=10 (in this case) the maximum h_k resides in the leftmost cell along with its index k and the last column swap data is about to enter the tableau. After a further two cycles the $\bar{c}_1$ filter bit completes the startup-closedown control cycle and the last column swap has reached the pivot row (verifying the timing 2(n+m+1) for sorting above). It follows that the filter bit falling off the



a) Odd-even sorter

t ARRAY CELLS

t															
1	◿	0	1	◿	0		◿	0		◿	0		◿	0	
		1			-			-			-			-	
		3			4			1			2			5	
2	0	0	0	◿	0	1	◿	0		◿	0		◿	0	
	1	1			2			-			-			-	
	3	3			4			1			2			5	
3	0	0	0	0	0	0	◿	0	1	◿	0		◿	0	
		1	2	1	2			3			-			-	
		3	4	3	4			1			2			5	
4	0	1	0	0	2	0	0	0	0	◿	0	1	◿	0	
	2	2			1	3	1	3			4			-	
	4	4			3	1	3	1			2			5	
5	0	0	0	0	1	0	0	2	0	0	0	0	0	0	1 →
		2	3	2	3			1	4	1	4			5	-∞
		4	1	4	1			3	2	3	2			5	◿
6	0	1	0	0	2	0	0	1	0	0	2	0	0	0	0
	3	3			2	4	2	4			1	5	1	5	
	1	1			4	2	4	2			3	5	3	5	
7	0	0	0	0	1	0	0	2	0	0	1	0	0	2	0 ←
		3	4	3	4			2	5	2	5			1	-∞
		1	2	1	2			4	5	4	5			3	◿
8	0	1	0	0	2	0	0	1	0	0	2	0	3	0	0
	4	4			3	5	3	5			2	1	2	1	
	2	2			1	5	1	5			4	3	4	3	
9	0	0	0	0	1	0	0	2	0	3	0	0	0	0	0
		4	5	4	5			3	2	3	2			1	-∞
		2	5	2	5			1	4	1	4			3	◿
10	0	1	0	0	2	0	3	0	0	0	0	0	0	0	0
		5			4	3	4	3			2	1	2	1	
		5			2	1	2	1			4	3	4	3	

C_1	C_2	C_1
W_1	a	E_1
W_2	j	E_2

b) Snapshots

FIGURE 7.5.3: Modified odd-even transposition sort

array can be used to prompt the controller into 'form-pivot' state.

The row sorting mechanism works in an identical manner to the column sorter and requires 2m cycles to complete sorting and an additional 2 cycles to closedown all the cells (verifying the timing 2(m+1) in Theorem (7.5.1) for sorting). At the end of row sorting the pivot resides in cell (m,2) and the row label (i.e. the x_ℓ index) has been placed in the bottom row sorting cell along with the minimum p. Hence the filter bit which has moved from top to bottom row closing down cells can be used to load the index k from the column sorter leftmost cell (via the controller) to overwrite the row label and set the controllers Exchange state simultaneously. Further complications arise when we consider sorting with negative p values, because under normal sorting conditions these will bubble to the bottom of the sorter. A forced swap for negative and zero p values implemented by a status flag set by the comparator determining the swap solves the problem simply, causing the undesirable values to bubble to the top of the array. If the bottom row sorting cell still contains a zero or negative value at the end of sorting, all the values in the sorter must be non-positive. Hence the status bits of the bottom cell also flag a termination condition as no improvement to H is possible, and can inhibit the overwriting of the row label. Similarly a status flag in column sorter cells can be adopted to flag $h_k<0$ and trap the second termination condition. Finally some general remarks about table element swapping. Fig.(7.5.4) illustrates the pipelining of both column and row swapping it should be clear that the control vectors output by the sorters consist of the pairs (1,2) punctuated by pairs of zeroes where no swapping occurs. The special null vector therefore corresponds to no swaps, which can only be produced when a list is fully sorted or a sorter is switched off. It follows that even if some portion of the array drops into a sorting state before the sorter begins row and column data will remain undamaged. This preservation of data is

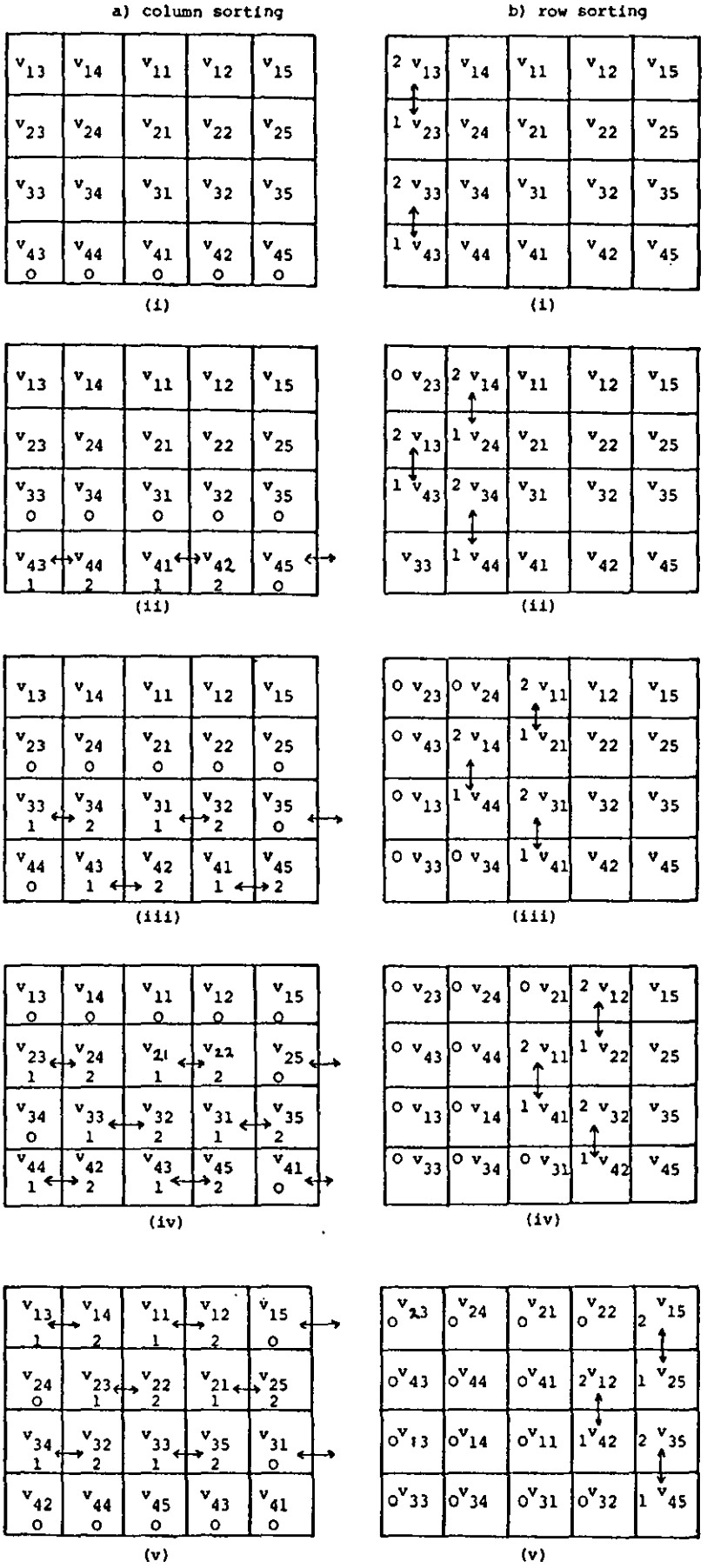


FIGURE 7.5.4: Control flow in sorting

important because it allows flexible state transitions, and the use of column and row states as idling states between wavefronts. In particular it allows the overlapping of row and column table modifications.

Next, we consider the exchange process - which rewrites the table in terms of the new basis - essentially exchanging the vectors entering and leaving the basis. We can define three basic types of operation and by virtue of the static positioning of the pivot row and column three basic cell types. The basic operations are a) find reciprocal in pivot cell, b) divide pivot row by pivot element, c) zero out the pivot column.

Using prompts from the controller the pivot cell in Fig.(7.5.1) orchestrates the whole computation (including sorting) and is the source of the starting wavefronts. The pivot cell controls operations using the second control network by triggering cell states by pumping control signals through the network. Fig.(7.5.5) illustrates the exchange control flow from which the following actions are defined:

(i) Whenever a cell receives two true controls on the same cycle it performs the rectangle rule.

(ii) If a single control value which is true arrives from the north or south, output the table value and zero the register.

(iii) If a single value arrives from east or west perform a division by the pivot.

(iv) If the cell is the pivot cell and the state is 'row-sort', a control input sets state = 'exchange' and:

  a) The reciprocal of the pivot is found, the result sent east and west.

  b) Overwrite the pivot with 1, and set all control outputs true,

which characterise the dataflow.

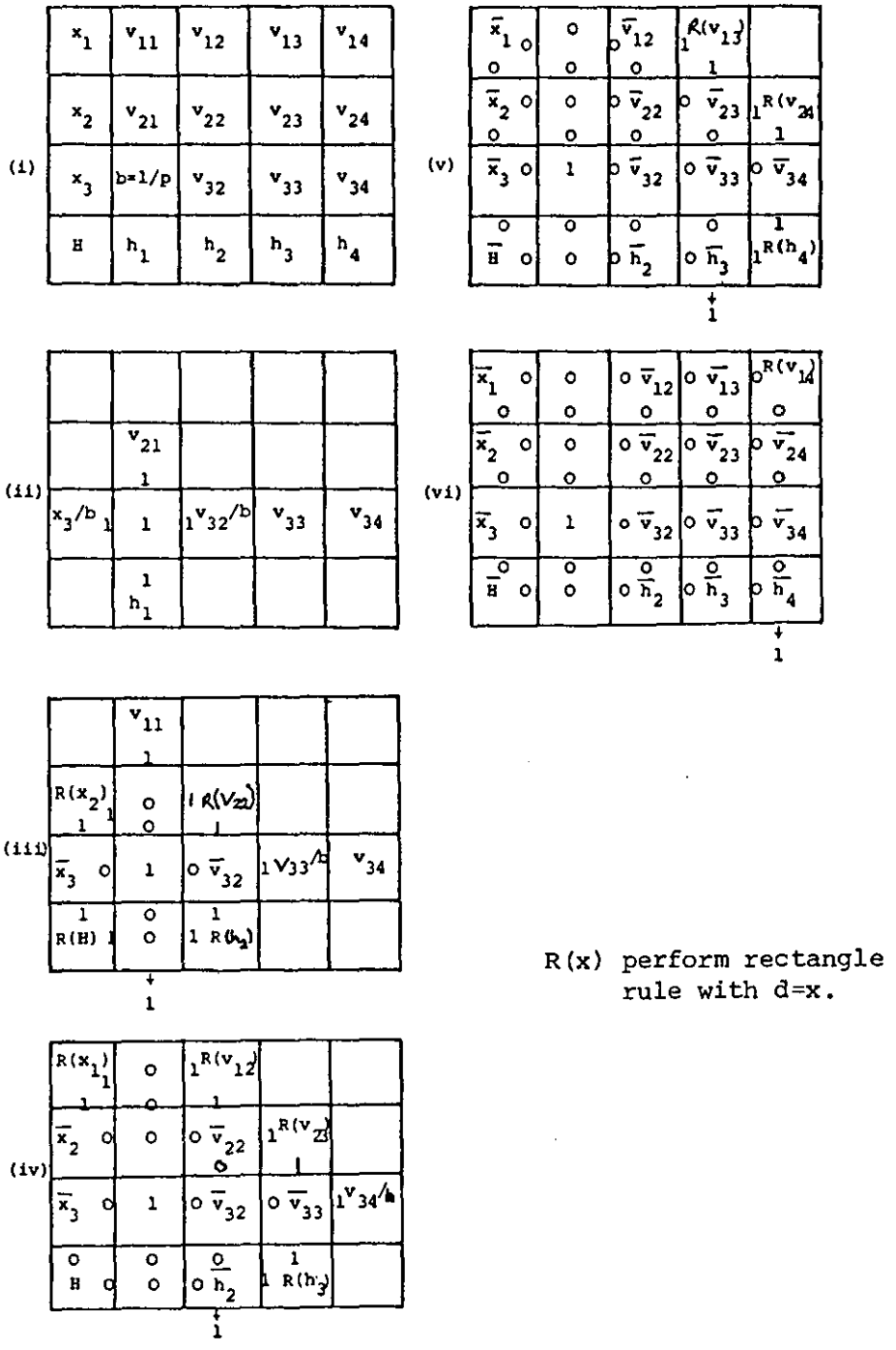


FIGURE 7.5.5: Control wavefront for exchange

(i) Control values falling off the southern boundary correspond to the startup procedure for the column sorter.

(ii) A control value travelling horizontally or vertically continues to do so until it falls off the array.

(iii) A control signal arriving in a pivot column cell is also refracted east and west.

(iv) A control signal arriving in a pivot row cell is also refracted north and south.

(v) Refracted signals continue in motion as for (ii).

Notice that this implies that exchange and column sorting can be overlapped.

At the end of column sorting the pivot cell gets pushed into 'form pivot' state, and the pivot column and column immediately to the left must form all the contenders for (7.5.18) which must then be loaded into the row sorter. The control actions are shown in Fig.(7.5.6).

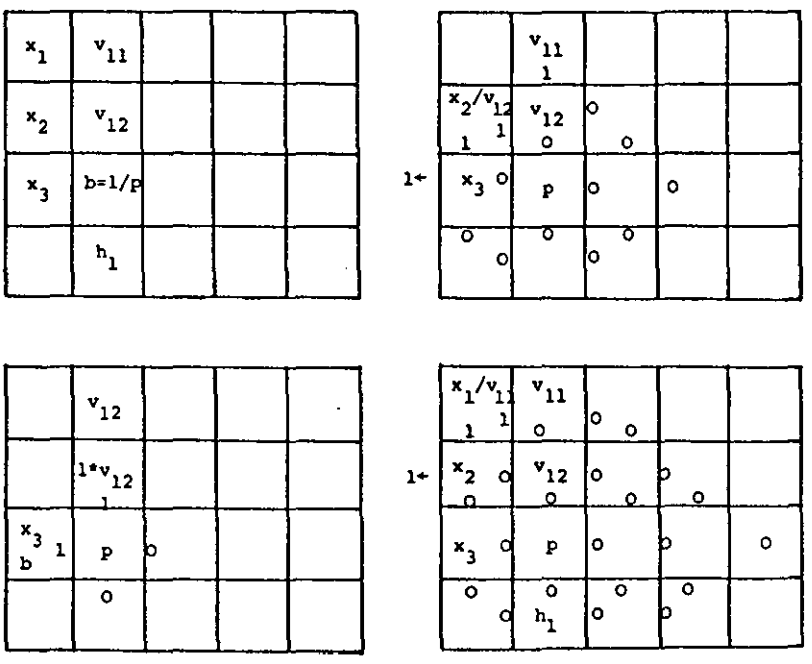


FIGURE 7.5.6: Pivot forming dataflow

The control values falling off the left boundary are in the correct form for loading and starting the row sorter. Clearly the control for the cells left of the pivot column become more complex and indicates the different cells of Fig.(7.5.1) described by the soft-systolic code in the Appendix.

To conclude this section we consider some theoretical and computational issues regarding the standard Simplex method and the array presented.

The problem of ties:

During the column sort phase we select the vector v_j corresponding to $\max_j(h_j)=h_k$ to achieve the greatest immediate decrease in the objective function. A tie occurs when more than one j occurs with maximum h_j. The problem is resolved arbitrarily in the standard Simplex theory by choosing the lowest (or highest) index j - which proves to be a good choice. Although the current array description is adequate for breaking ties, we can incorporate this strategy by performing a swap according to

$$(h_j<h_i) \text{ or } ((h_j=h_i) \text{ and } (i>j))$$

in a column cell. This modification requires at most an additional comparator in each sorter cell, and is justified by the fact that tie breaking with this rule requires approximately m changes of basis to find the minimum. Thus with z*m Corollary (7.5.1) gives a loose bound for the full Simplex calculation.

Degeneracy:

A non-degenerate feasible solution is a feasible solution with exactly m positive x_i, if there are less than m positive x_i the solution is degenerate. If the above condition occurs at least one x_i is zero and it would be possible to choose p=0 in (7.5.18), producing no

reduction in H. If this lack of improvement continued for a number of Simplex iterations it is possible to repeat a basis and the solution process breaks down (the array would become stuck in an infinite loop). Degeneracy is indicated by less than m x_i values being positive, or (7.5.18) producing ties (and implying that more than one variable leaves the solution on a single iteration). Fortunately, to date degeneracy has only been exhibited by artifically constructed problems, and the normal course of action is to use p=0 when it occurs and break ties in a similar manner to the column sorter solution.

Artificial Basis Techniques:

Throughout the array description we have assumed that a basis (hence extreme feasible point) was known. When this is not the case an artificial basis must be constructed which will produce a feasible basis for the original problem. The details of artificial basis can be found in standard texts on linear programming and are not discussed here, but the array of Fig.(7.5.1) is easily upgraded to deal with them. Essentially, we add an additional row of cells in the $(m+2)^{nd}$ position which contain their own h_j type elements. The algorithm is then controlled by two phases. In the first phase the h_j values are sorted and the table updated until all the elements are non-positive. If all $h_j=0$ the resulting basis is feasible for the original problem, and if all $h_j<0$ the original problem was not feasible. In the former case we can continue by applying the sorting to the $(m+1)^{th}$ row or true h_j values to obtain a minimum. For the latter case the table is abandoned. Again, the extra hardware is justified by the fact that a full artificial basis of m columns requires approximately z=2m iterations to find the minimum feasible solution which otherwise would not be solvable.

TEST EXAMPLE

Minimise $H = -2x_1 - x_2$

subject to $0 \leq x_1, \quad 0 \leq x_2,$

$-x_1 + 2x_2 \leq 2, \quad x_1 + x_2 \leq 4$

Introducing slack variables we produce the following tables:

3	2	-1	2	1	0	0
4	4	1	1	0	1	0
5	3	1	0	0	0	1
	0	2	1	0	0	0
		1	2	3	4	5

3	3	0	0	1	-2	3
2	1	0	1	0	1	-1
1	3	1	0	0	0	1
	-7	0	0	0	-1	-1
		1	2	3	4	5

As the first action of the array is a modification, and the correct pivot is in the correct place variable x_1 is swapped with x_5 so we load the tableau with,

3	2	-1	2	1	0	0
4	4	1	1	0	1	0
1	3	1	0	0	0	1
	0	2	1	0	0	0
		1	2	3	4	5

After a few iterations (2) we get the following result from the OCCAM program.

a) With trace=on

1	3	1	0	0	0	1
3	3	0	0	1	-2	3
2	1	0	1	0	1	-1
	-7	0	0	0	-1	-1
		1	2	3	4	5

b) with trace=off

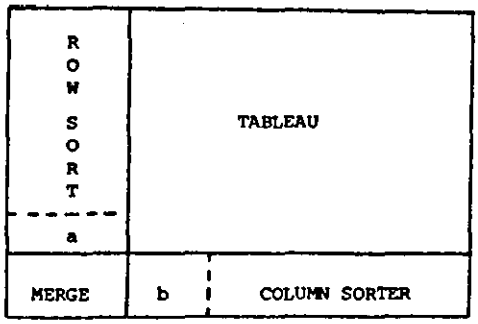
```
Results

    [1] = 3.000000
    [3] = 3.000000
    [2] = 1.000000

    [H] = -7.000000

[i] = variable i
```

which is a row and column permuted form of the correct final tableau.

SNAPSHOTS OF CONTROL WAVEFRONT FOR THE TEST EXAMPLE

Output format:

(c_0,c_1,c_2,c_3) - In Tableau
(c_0,c_1,c_2,j) - Column sorter
(c_0,c_1,c_2,j) - Row sorter
(c_4,c_0,c_2,j_1) - dummy.a
(c_0,c_1,c_3,j) - dummy.b
$(c_0,c_1,c_2,0)$ - merge



a=dummy.a
b=dummy.b

Pivot cell is marked on first snapshot.

```
0003 | 0000  0000  0000  0000  0000  0000
0004 | 0000  0000  0000  0000  0000  0000
0001 | 0000 [0000] 0000  0000  0000  0000
0000 | 0000  0000  0000  0000  0000  0000
---------------------------------------------
0010 | 0000| 0001  0002  0003  0004  0005

start cycle
 0003  0000  0000  0000  0000  0000  0000
 0004  0000  0000  0000  0000  0000  0000
 0001  0000  0000  0000  0000  0000  0000
 0000  0000  0000  0000  0000  0000  0000
 0000  0000  0001  0002  0003  0004  0005

start cycle
 0003  0000  0000  0000  0000  0000  0000
 0004  0000  1000  0000  0000  0000  0000
 0001  0001  0000  0100  0000  0000  0000
 0000  0000  0010  0000  0000  0000  0000
 0000  0000  0001  0002  0003  0004  0005

start cycle
 0003  0000  1000  0000  0000  0000  0000
 0004  1001  0000  1100  0000  0000  0000
 0001  0000  0000  0000  0100  0000  0000
 0000  0011  0000  0110  0000  0000  0000
 0000  0000  1001  0002  0003  0004  0005

start cycle
 0003  1001  0000  1100  0000  0000  0000
 0004  0000  0000  0000  1100  0000  0000
 0001  0000  0000  0000  0000  0100  0000
 1000  0000  0000  0000  0110  0000  0000
 0000  0000  0001  1012  0003  0004  0005

start cycle
 0003  0000  0000  0000  1100  0000  0000
 0004  0000  0000  0000  0000  1100  0000
 0001  0000  0000  0000  0000  0000  0100
 0000  0000  0000  0000  0000  0110  0000
 0000  0000  0002  0001  1013  0004  0005
```

```
start cycle
 0003  0000  0000  0000  0000  1100  0000
 0004  0000  0000  0000  0000  0000  1100
 0001  0000  0000  0000  0000  0000  0000
 0000  0000  1000  2000  0000  0000  0110
 0000  0000  0002  0001  0003  1014  0005

start cycle
 0003  0000  0000  0000  0000  0000  1100
 0004  0000  0000  0000  0000  0000  0000
 0001  0000  1000  2000  0000  0000  0000
 0000  0000  0000  0000  0000  0000  0000
 0000  0000  0002  0001  0003  0004  1015

start cycle
 0003  0000  0000  0000  0000  0000  0000
 0004  0000  1000  2000  0000  0000  0000
 0001  0000  0000  0000  0000  0000  0000
 0000  0000  0000  0000  0000  0000  0000
 0000  0000  0002  0001  0003  0004  0005

start cycle
 0003  0000  1000  2000  0000  0000  0000
 0004  0000  0000  0000  0000  0000  0000
 0001  0000  0000  0000  0000  0000  0000
 0000  0000  0000  0000  0000  0000  0000
 0000  0000  0002  0001  0003  0004  0105

start cycle
 0003  0000  0000  0000  0000  0000  0000
 0004  0000  0000  0000  0000  0000  0000
 0001  0000  0000  0000  0000  0000  0000
 0000  0000  0000  0000  0000  0000  3000
 0000  0000  0002  0001  0003  0104  0005

start cycle
 0003  0000  0000  0000  0000  0000  0000
 0004  0000  0000  0000  0000  0000  0000
 0001  0000  0000  0000  0000  0000  3000
 0000  0000  0000  0000  0000  3000  0000
 0000  0000  0002  0001  0103  0004  0005
```

```
start cycle
 0003  0000  0000  0000  0000  0000  0000
 0004  0000  0000  0000  0000  0000  3000
 0001  0000  0000  0000  0000  3000  0000
 0000  0000  0000  0000  3000  0000  0000
 0000  0000  0002  0101  0003  0004  0005

start cycle
 0003  0000  0000  0000  0000  0000  3000
 0004  0000  0000  0000  0000  3000  0000
 0001  0000  0000  0000  3000  0000  0000
 0000  0000  0000  3000  0000  0000  0000
 0000  0000  0102  0001  0003  0004  0005

start cycle
 0003  0000  0000  0000  0000  3000  0000
 0004  0000  0000  0000  3000  0000  0000
 0001  0000  0000  3000  0000  0000  0000
 0000  0000  3000  0000  0000  0000  0000
 0000  1010  0002  0001  0003  0004  0005

start cycle
 0003  0000  0000  0000  3000  0000  0000
 0004  0000  0000  3000  0000  0000  0000
 0001  0000  3000  0000  0000  0000  0000
 0002  0000  0000  0000  0000  0000  0000
 0110  0000  0002  0001  0003  0004  0005

start cycle
 0003  0000  0000  3000  0000  0000  0000
 0004  0000  3000  0000  0000  0000  0000
 0001  0000  0000  0000  0000  0000  0000
 0002  0000  0000  0000  0000  0000  0000
 0000  0000  0002  0001  0003  0004  0005

start cycle
 0003  0000  3000  0000  0000  0000  0000
 0004  0000  1000  0000  0000  0000  0000
 0001  0001  0000  0000  0000  0000  0000
 0002  0000  0000  0000  0000  0000  0000
 0000  0000  0002  0001  0003  0004  0005
```

```
start  cycle
 0003  0000  1000  0000  0000  0000  0000
 0004  0001  0000  0000  0000  0000  0000
 1001  0000  0000  0000  0000  0000  0000
 0002  0000  0000  0000  0000  0000  0000
 0000  0000  0002  0001  0003  0004  0005

start  cycle
 0003  0001  0000  0000  0000  0000  0000
 1014  0000  0000  0000  0000  0000  0000
 0001  0000  0000  0000  0000  0000  0000
 0002  0000  0000  0000  0000  0000  0000
 0000  0000  0002  0001  0003  0004  0005

start  cycle
 1013  0000  0000  0000  0000  0000  0000
 0001  0000  0000  0000  0000  0000  0000
 0004  0000  0000  0000  0000  0000  0000
 0002  0000  0000  0000  0000  0000  0000
 0000  0000  0002  0001  0003  0004  0005

start  cycle
 0001  0000  0000  0000  0000  0000  0000
 0003  0200  0000  0000  0000  0000  0000
 0004  0100  0000  0000  0000  0000  0000
 0002  0000  0000  0000  0000  0000  0000
 0000  0000  0002  0001  0003  0004  0005

start  cycle
 0101  0200  0000  0000  0000  0000  0000
 0003  0100  0200  0000  0000  0000  0000
 0004  0000  0100  0000  0000  0000  0000
 0002  0000  0000  0000  0000  0000  0000
 0000  0000  0002  0001  0003  0004  0005

start  cycle
 0001  0300  0200  0000  0000  0000  0000
 0103  0000  0100  0200  0000  0000  0000
 0004  0000  0000  0100  0000  0000  0000
 0002  0000  0000  0000  0000  0000  0000
 0000  0000  0002  0001  0003  0004  0005
```

```
start  cycle
 0001  0000  0300  0200  0000  0000  0000
 0003  0300  0000  0100  0200  0000  0000
 0104  0000  0000  0000  0100  0000  0000
 0002  0000  0000  0000  0000  0000  0000
 0000  0000  0002  0001  0003  0004  0005

start  cycle
 0001  0000  0000  0300  0200  0000  0000
 0003  0000  0300  0000  0100  0200  0000
 0004  0300  0000  0000  0000  0100  0000
 0112  0000  0000  0000  0000  0000  0000
 0000  0000  0002  0001  0003  0004  0005

start  cycle
 0001  0000  0000  0000  0300  0200  0000
 0003  0000  0000  0300  0000  0100  0200
 0012  0000  0300  0000  0000  0000  0100
 0000  0000  0000  0000  0000  0000  0000
 1010  0000  0002  0001  0003  0004  0005

start  cycle
 0001  0000  0000  0000  0000  0300  0200
 0003  0000  0000  0000  0300  0000  0100
 0002  0000  0000  0300  0000  0000  0000
 0000  0000  0000  0000  0000  0000  0000
 0000  0000  0002  0001  0003  0004  0005

start  cycle
 0001  0000  0000  0000  0000  0000  0300
 0003  0000  1000  0000  0000  0300  0000
 0002  0001  0000  0100  0300  0000  0000
 0000  0000  0010  0000  0000  0000  0000
 0000  0000  0002  0001  0003  0004  0005

start  cycle
 0001  0000  1000  0000  0000  0000  0000
 0003  1001  0000  1100  0000  0000  0300
 0002  0000  0000  0000  0100  0300  0000
 0000  0011  0000  0110  0000  0000  0000
 0000  0000  1002  0001  0003  0004  0005
```

```
start  cycle
 0001  1001  0000  1100  0000  0000  0000
 0003  0000  0000  0000  1100  0000  0000
 0002  0000  0000  0000  0000  0100  0300
 1000  0000  0000  0000  0110  0000  0000
 0000  0000  0002  1011  0003  0004  0005

start  cycle
 0001  0000  0000  0000  1100  0300  0000
 0003  0000  0000  0000  0000  1100  0000
 0002  0000  0000  0000  0000  0000  0100
 0000  0000  0300  0000  0000  0110  0000
 0000  0000  0001  0002  1013  0004  0005

start  cycle
 0001  0000  0000  0000  0000  1100  0000
 0003  0300  0000  0000  0000  0000  1100
 0002  1000  0000  0000  0000  0000  0000
 0000  0000  1000  2000  0000  0000  0110
 0000  0000  0001  0002  0003  1014  0005

start  cycle
 0001  0000  0000  0000  0000  0000  1100
 0003  0000  0000  0000  0000  0000  0000
 0002  0000  1000  2000  0000  0000  0000
 0000  0000  0000  0000  0000  0000  0000
 0000  0000  0001  0002  0003  0004  1015

start  cycle
 0001  0000  0000  0000  0000  0000  0000
 0003  0000  1000  2000  0000  0000  0000
 0002  0000  0000  0000  0000  0000  0000
 0000  0000  0000  0000  0000  0000  0000
 0000  0000  0001  0002  0003  0004  0005

start  cycle
 0001  0000  1000  2000  0000  0000  0000
 0003  0000  0000  0000  0000  0000  0000
 0002  0000  0000  0000  0000  0000  0000
 0000  0000  0000  0000  0000  0000  0000
 0000  0000  0001  0002  0003  0004  0105
```

```
start  cycle
 0001  0000  0000  0000  0000  0000  0000
 0003  0000  0000  0000  0000  0000  0000
 0002  0000  0000  0000  0000  0000  0000
 0000  0000  0300  0000  0000  0000  3000
 0000  0000  0001  0002  0003  0104  0005

start  cycle
 0001  0000  0000  0000  0000  0000  0000
 0003  0000  0000  0000  0000  0000  0000
 0002  0300  0000  0000  0000  0000  3000
 0000  0000  1000  0000  0000  3000  0000
 0000  0000  0001  0002  0103  0004  0005

start  cycle
 0001  0000  0000  0000  0000  0300  0000
 0003  0000  0000  0000  0000  0000  3000
 0002  0000  0000  0000  0000  3000  0000
 0000  0000  0000  0000  3000  0000  0000
 0000  0000  0001  0102  0003  0004  0005

start  cycle
 0001  0000  0000  0000  0000  0000  3000
 0003  0000  0000  0000  0000  3000  0000
 0002  0000  0000  0000  3000  0000  0000
 0000  0000  0000  3000  0000  0000  0000
 0000  0000  0101  0002  0003  0004  0005

start  cycle
 0001  0000  0000  0000  0000  3000  0000
 0003  0000  0000  0000  3000  0000  0000
 0002  0000  0000  3000  0000  0000  0000
 0000  0000  3000  0000  0000  0000  0000
 0000  7000  0001  0002  0003  0004  0005

correct termination
```

SNAPSHOTS FOR ARRAY CLOSEDOWN (6666=Dead cell)

start cycle
CCC5 C0CG 0000 0000 0C00 0000 3000
0CC3 G000 0030 0000 00C0 3000 0000
0002 C000 0000 3000 3C00 0000 C000
C000 0000 0000 3000 0CC0 0000 00C0
CCC0 C000 0101 3002 0CC3 0004 C005

start cycle
0005 0000 0000 0000 0000 3000 0000
0003 0000 0000 0000 3000 0000 0000
0002 0000 0030 3000 0000 0000 0000
C000 C000 3000 0000 0C00 0300 0000
7C00 666C 0001 0002 0C03 0004 0005

start cycle
0005 C000 0000 0000 3000 0000 0000
0003 C000 0000 3000 0000 0000 C000
0002 C000 3C00 0000 0CC0 0000 0000
0060 6000 0000 0000 C000 0000 C000
0660 0000 0061 3002 3003 0004 3005

start cycle
0005 0000 0000 3000 0000 0000 0000
C003 0000 3030 0000 0000 0000 0000
0062 6000 0000 3030 0000 0000 0000
C000 6660 6000 0000 C000 0000 0000
C660 0000 0061 0062 0003 0004 0005

start cycle
0065 0000 3000 0000 0000 0000 0000
0063 6000 0000 0C00 C000 0000 0000
0062 6660 6000 0000 0C00 0000 0000
C000 6666 6660 6000 0C00 0300 0000
0660 0000 0061 0062 0063 0004 C005

start cycle
0065 6000 0000 0000 0000 0000 0000
0063 6660 6000 0000 0000 0000 0000
0062 6666 6660 6000 0000 0000 0000
0C00 6666 6666 6660 6000 0000 0000
0660 CC00 0061 0062 0063 0064 0005

start cycle
CC65 666C 6000 0000 0C0C 0000 0C00
0063 6666 6660 6000 0C00 0000 0000
0062 6666 6666 6660 60C0 0000 0C00
C000 6666 6666 6666 6660 6300 C000
C660 0000 0061 0062 0063 0064 CC65

start cycle
CC65 6666 6660 6000 0000 0000 CCCC
0063 6666 6666 6660 6000 C000 0000
CC62 6666 6666 6666 6660 6000 0000
C0C0 6666 6666 6666 6666 6660 C0C0
0660 C000 0061 0062 0063 0064 0065

start cycle
0065 6666 6666 6660 6000 0000 000C
C063 6666 6666 6666 6660 6000 C00C
0062 6666 6666 6666 6666 6660 6C00
0000 6666 6666 6666 6666 6666 6660
0660 0000 0061 0062 0C63 0064 0065

start cycle
0065 6666 6666 6666 6660 6000 0000
0063 6666 6666 6666 6666 6660 600C
0062 6666 6666 6666 6666 6666 6660
C000 6666 6666 6666 6666 6666 6666
C660 0000 0061 0062 0063 0064 0065

start cycle
CC65 6666 6666 6666 6666 6660 60C0
0063 6666 6666 6666 6666 6666 6660
CC62 6666 6666 6666 6666 6666 6666
C0C0 6666 6666 6666 6666 6666 6666
C660 C000 0061 0062 0063 0064 CC65

start cycle
0065 6666 6666 6666 6666 6666 6660
0063 6666 6666 6666 6666 6666 6666
C062 6666 6666 6666 6666 6666 6666
C000 6666 6666 6666 6666 6666 6666
C66C 000G 0061 0062 0C63 0064 CC65

7.6 A SYSTOLIC CYLINDER FOR THE REVISED SIMPLEX ALGORITHM

The above standard scheme is not the one usually chosen for computer implementation, instead a revised form of the Simplex algorithm is used. This new algorithm can be implemented in two ways:

(a) The general form of the inverse

(b) The product form of the inverse

The second technique is often used in practice because it minimises the amount of information to be recorded using the products of elemental matrices. Both techniques however reduce the amount of computation required to update the basis and Simplex tableau, and size of table recorded in the machine's main memory. This latter point of data compaction is important for large LP problems and we examine the possibilities of transferring these characteristics to systolic arrays.

At the start of the standard algorithm extra vectors (at most m) are added to the table to form a basis. The basis consists of m linearly independent vectors and it follows that,

$$B = (v_1, v_2, \ldots, v_m) \ , \tag{7.6.1}$$

and that any other vector v_j is a linear combination of vectors from B, i.e.,

$$v_j = \alpha_{1j}v_1 + \alpha_{2j}v_2 + \ldots + \alpha_{mj}v_m$$

then,
$$\alpha_j = B^{-1}v_j \ , \tag{7.6.2}$$

where, $\alpha_j = (\alpha_{1j}, \alpha_{2j}, \ldots, \alpha_{mj})$.

From (7.5.3) putting B as the first m vectors of A such that,

$$Bx_0 = b, \quad x_0 \geq 0 \ , \tag{7.6.3}$$

with $x_0 = (x_{10}, x_{20}, \ldots, x_{m0})$ gives the first basic feasible solution

$$x_0 = B^{-1}b \ , \tag{7.6.4}$$

and from (7.6.2) all the remaining vectors of A can be determined from

B. The pieces used to determine the vector to be moved into the basis are given by, h_j, j=1(1)n where,

$$\left.\begin{aligned} &\text{a)}\quad h_j = z_j - c_j \\ &\text{with b)}\quad z_j = c_1\alpha_{1j} + c_2\alpha_{2j} + \ldots + c_m\alpha_{mj} \end{aligned}\right\} \tag{7.6.5}$$

Thus,

$$z_j = c_o\alpha_j = c_o B^{-1} v_j \ , \ j=1(1)n \tag{7.6.6}$$

with $c_o=(c_1,\ldots,c_m)$ so with the feasible basis B we compute the corresponding z_j, and a pricing vector π_i can be defined as,

$$\pi = c_o B^{-1}, \ \pi = (\pi_1,\pi_2,\ldots,\pi_m) \tag{7.6.7}$$

hence for a vector not in the basis,

$$h_j = \pi v_j - c_j \ . \tag{7.6.8}$$

It follows that we have all the information to move from feasible solution to feasible solution, using only the original A and C values. The main idea is that rather than transferring all the elements of the Simplex tableau we need only to transform the elements of B^{-1}. The explicit form of B at each iteration can be constructed as follows. Let $B=(v_1,v_2,\ldots,v_\ell,\ldots,v_m)$ be the old basis differing from the new basis $\bar{B}$ by a single vector,

$$\bar{B} = (v_1,v_2,\ldots,v_k,\ldots,v_m), \ v_\ell \neq v_k \ .$$

Then,

$$B^{-1}B = B^{-1}(v_1,v_2,\ldots,v_\ell,\ldots,v_m) = I \ , \tag{7.6.9}$$

and,

$$B^{-1}\bar{B} = B^{-1}(v_1,v_2,\ldots,v_k,\ldots,v_m) = \begin{bmatrix} 1 & 0 & \cdots & \alpha_{1k} & \cdots & 0 \\ 0 & 1 & & \alpha_{2k} & & \\ & & & \vdots & & \\ & & & \alpha_{\ell k} & & 0 \\ & & & \vdots & & \\ 0 & 0 & \cdots & \alpha_{mk} & \cdots & 1 \end{bmatrix} \tag{7.6.10}$$

Thus, an element b_{ij} of B^{-1} can be transformed to $\bar{b}_{ij}$ an element of $\bar{B}^{-1}$ in the corresponding position, by,

$$\left.\begin{aligned} \bar{b}_{\ell j} &= \frac{b_{\ell j}}{\alpha_{\ell k}} \\ \bar{b}_{ij} &= b_{ij} - \bar{b}_{\ell j}\alpha_{ik},\ i\neq\ell \end{aligned}\right\} \tag{7.6.11}$$

The revised Simplex method is then constructed as follows,

(i) introduce the additional variable $x_{n+m+1}=-H(x)$ from (7.5.3)

(ii) allow for artificial vectors (using artificial basis techniques) by the redundant equation,

$$a_{m+2,1}x_1 + a_{m+2,2}x_2 + \ldots + a_{m+2,n}x_n + x_{m+n+2} = b_{m+2},$$

where,

$$\left.\begin{aligned} a_{m+2,j} &= -\sum_{i=1}^{m} a_{ij},\ j=1(1)n \\ b_{m+2} &= -\sum_{i=1}^{m} b_i \end{aligned}\right\} \tag{7.6.12}$$

and with $a_{m+1,j}=c_j$ we have the matrix problem,

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} & 1 & & \\ a_{21} & & & & & \ddots & \\ \vdots & & & & & & 1 \\ a_{m1} & & & & & & \\ a_{m+1,1} & & & & & & \\ a_{m+2,1} & \cdots & & a_{m+2,n} & & & \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \\ x_{n+1} \\ \vdots \\ x_{n+m+2} \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \\ 0 \\ b_{m+2} \end{bmatrix} \tag{7.6.13}$$

$$\bar{A} \qquad\qquad U \qquad\qquad = \quad b$$

with $x_i \geq 0$, $i=1(1)n+m+2$.

The revised Simplex procedure can now be completely defined. In the algorithm definition we denote $\bar{A}_j$ as the columns of $\bar{A}$ and u_i the rows of the matrix U. As the procedure progresses the u_i represent the most recent update of the corresponding row. Row m+2 in $\bar{A}$ is used to

evaluate h_j, while artificial variables are still in the solution, row m+1 when artificial variables have been removed.

```
/* revised simplex algorithm */
PHASE I (Artificial variables in the solution, and all positive)
    WHILE x_{n+m+2}<0 DO
        {FOR j=1 TO n {δ_j=u_{m+2}Ā_j};
         IF ALL δ_j≥0 THEN {x_{n+m+2} MAX NO FEASIBLE SOLUTION EXISTS}
         ELSE {δ_k=min(δ_j)};
         FOR i=1 TO m+2 {x_ik=u_iĀ_k};
         P= min_{1≤i≤m}(x_iO/x_ik) = x_ℓO/x_ℓk ;
         FOR i=1 TO n+m+2
             {x̄_kO=x_ℓO/x_ℓk; x̄_iO=x_iO-x̄_kO x_ik  i≠k
              FOR j=1 TO m+2
                  {ū_ℓj=u_ℓj/x_ℓk; ū_ij=u_ij-ū_ℓj x_ik  i≠ℓ}
             };
       };
PHASE II: (No positive artificial variables in solution)
    {FOR j=1 TO n {γ_j=u_{m+2} Ā_j};
     WHILE γ_j<0 DO
         {γ_k=min(γ_j);
          FOR i=1 TO m+2 {x_ik=u_iĀ_k};
          P = min_{1≤i≤m} (x_iO/x_ik) = (x_ℓO/x_ℓk)
          IF all x_ik≤0 THEN {solution can be made arbitrarily large}
          FOR i=1 TO n+m+2
              {x̄_kO=x_ℓO/x_ℓk; x̄_iO=x_iO-x̄_kO x_ik  i≠k;
               FOR j=1 TO m+2
                   {ū_ℓj=u_ℓj/x_ℓk; ū_ij=u_ij-ū_ℓj x_ik  i≠ℓ};
              };
       };
PHASE III: STOP; x_{n+m+1} is at its max value - optimal stop.
N.B.: to simplify the algorithm x_i=x_iO.
```

We now proceed to explain two systolic arrays for the general form of the inverse, a method suitable for any m and n and a specialised version for m>n. The more general algorithm has a regular connection network when embedded in a cylindrical space, and leads to a volume efficient design by folding the cylinder. The second design is orthogonally connected, reduces the number of cells significantly and can be represented in a plane. In addition to the improved efficiency

of the revised Simplex method these new algorithms recognize that the pivot row and column can be located and moved within the array without a full sort (or total ordering). A partial ordering to locate max or min elements is sufficient and can be implemented with simplified cells.

The global view of the systolic cylinder is shown in Fig.(7.6.1) and can be considered as three individual sections, PART A, PART B and PART C, with dataflow around the cylinder interpreted as wavefronts across these sections.

PART A: is an n*(m+2) matrix of cells, with an additional column of n boundary cells to the left. The array contains the elements of $\bar{A}$ stored in the order of $\bar{A}_j$ in the jth row j=1(1)n. The boundary cells are initially empty except for the column index.

PART B: This is an (m+1)*(m+3) matrix of cells with a column of m boundary cells to the right. The array contains the (m+2)*(m+2) basis matrix initially U, and can be hardwired to start up with U=I. The (m+1)st row contains two rows (m+1) and (m+2) for smooth dataflow, while the (m+3)rd column contains the starting solution vector. The column boundary cells on the right containing the indexes of the solution variables.

PART C: This is a row of (m+5) cells which wrap around the top row of Part B to the bottom row of Part A, forming the cylinder.

Notice that the two phases of the revised Simplex algorithm are almost identical, except that we use u_{m+1} instead of u_{m+2} and allow additional termination conditions. Thus placing both u_{m+1} and u_{m+2} in the same row of part B reduces dataflow problems to only a single phase, with switching between phases controlled by the (m+1)st row of part B. The boundary cells will be used to detect the remaining termination conditions.

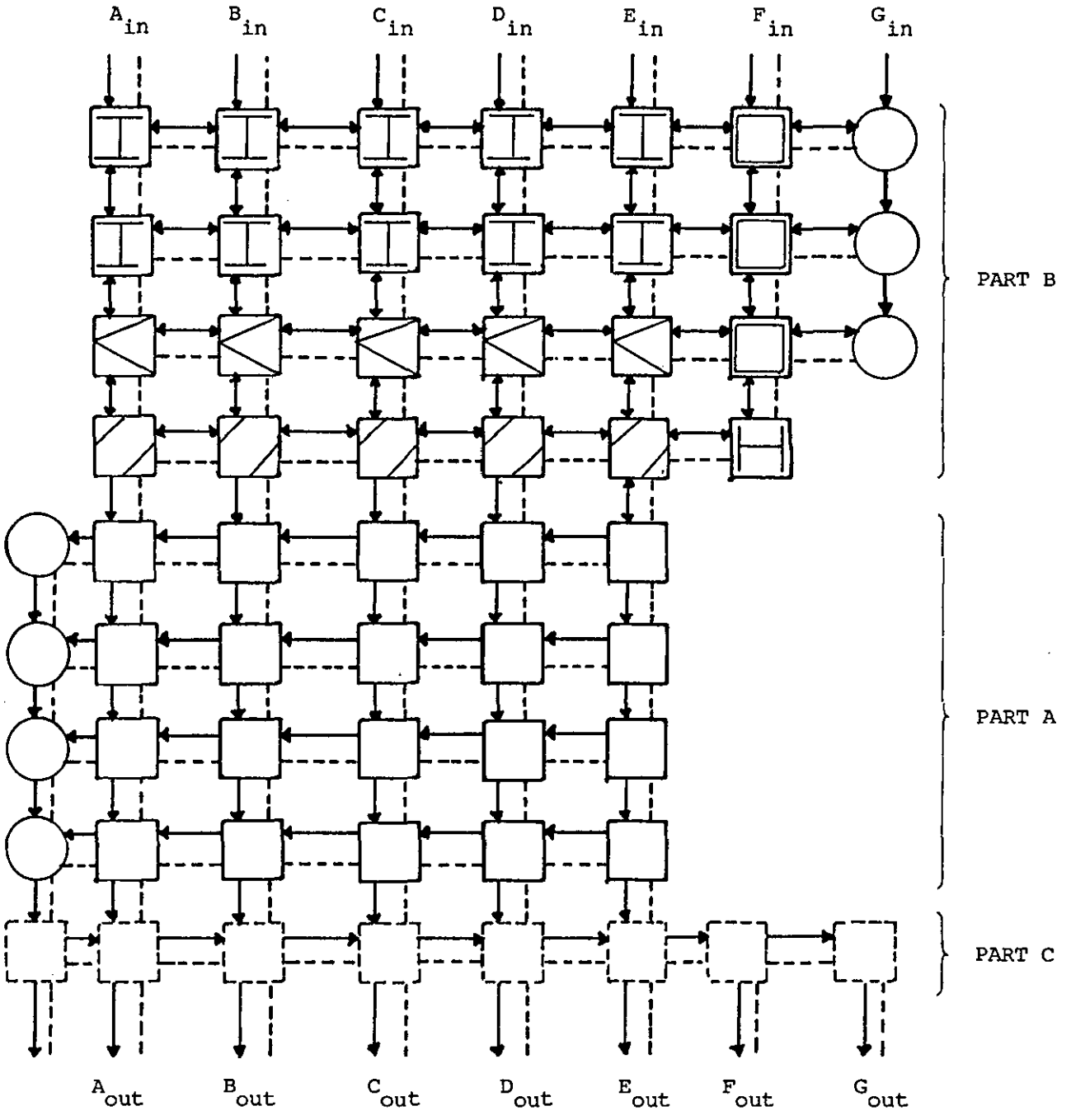


FIGURE 7.6.1: Cylindrical systolic array for revised simplex method (General m and n)

The wavefronts for the cylinder computation are shown in Figs. (7.6.2) and (7.6.3) and explained by the following commentary. At the start of computation the cell H in Fig.(7.6.1) performs a check on x_{n+m+2} and x_{n+m+1} to determine which row m+1 or m+2 is to be used and hence selecting phase I or phase II of the algorithm. On the check result a control signal is shifted left informing row m+1 cells whether to use u_{m+1} or u_{m+2}. As the control moves left it generates a sequence of control signals moving down the columns of the part A array, together with the associated value of the selected row (u_{m+1} or u_{m+2}) elements. A wavefront (w_1 in Fig.(7.6.2)) spreads out from the top right of the part A section generating δ_j (γ_j) depending on the phase, by accumulating partial products from right to left. On reaching the left boundary the δ_j (γ_j) values are loaded into the rows' boundary cell where it picks up its associated index j. w_1 is now reflected to form w_2 a wavefront

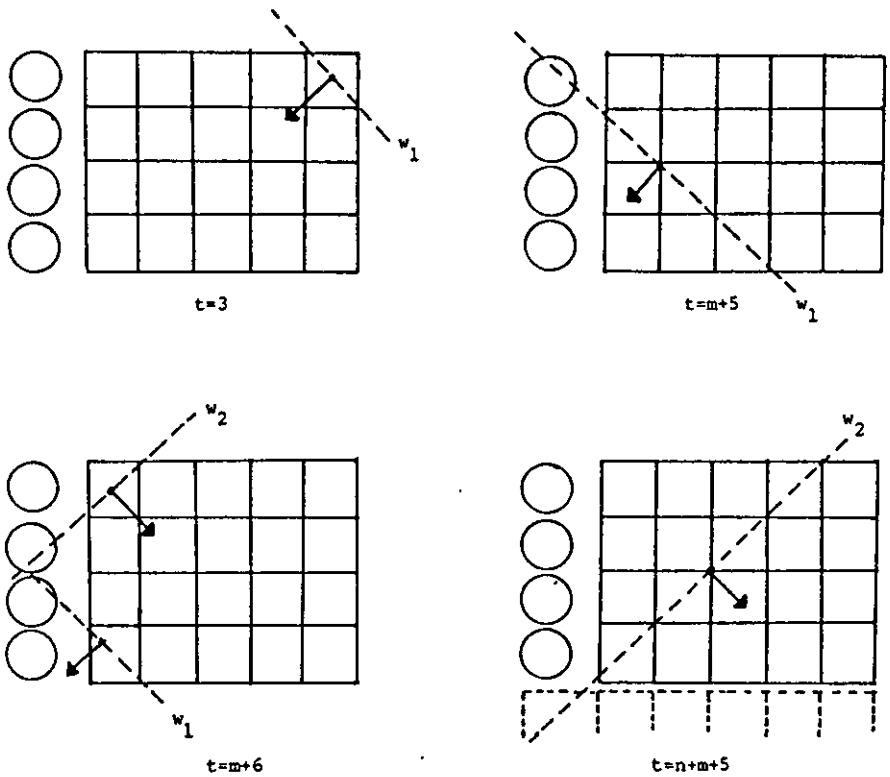


FIGURE 7.6.2: Wavefront progression Part A

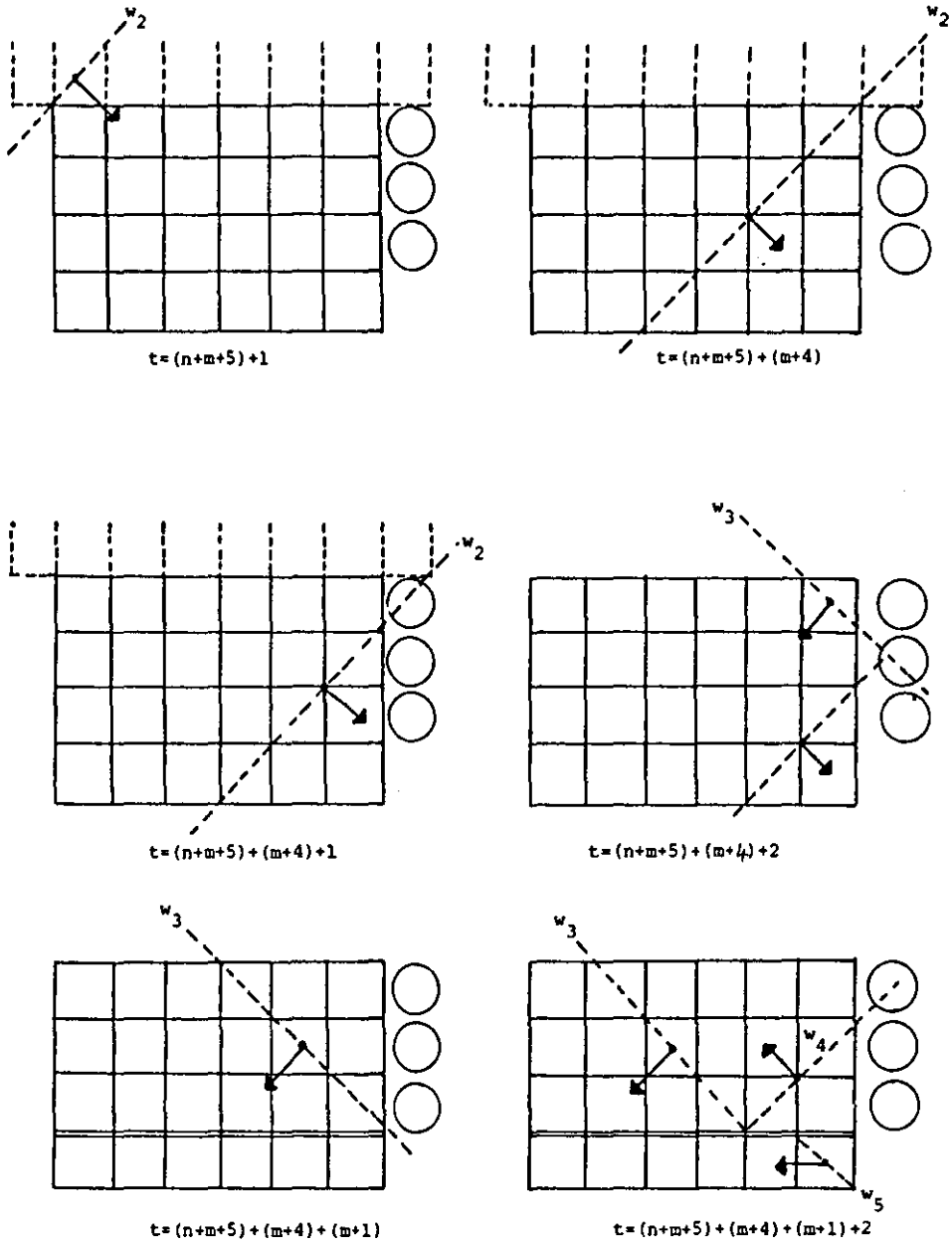


FIGURE 7.6.3: Wavefront progression for Part B

moving from the top left to bottom right corner of A. w_2 computes the partial ordering of δ_j (γ_j) values pushing the minimum to the bottom of the boundary cell column, and issuing a sequence of controls left to right along each row of part A transferring a copy of the corresponding A_j column towards the part C array. It follows that on the (n+m+5)th cycle the best δ_j (γ_j) and its associated index j are in the leftmost cell of the part C array. The next (m+2) cycles see w_2 load the part C cells with the column A_j. Thus, by the (n+m+5)th cycle we have identified

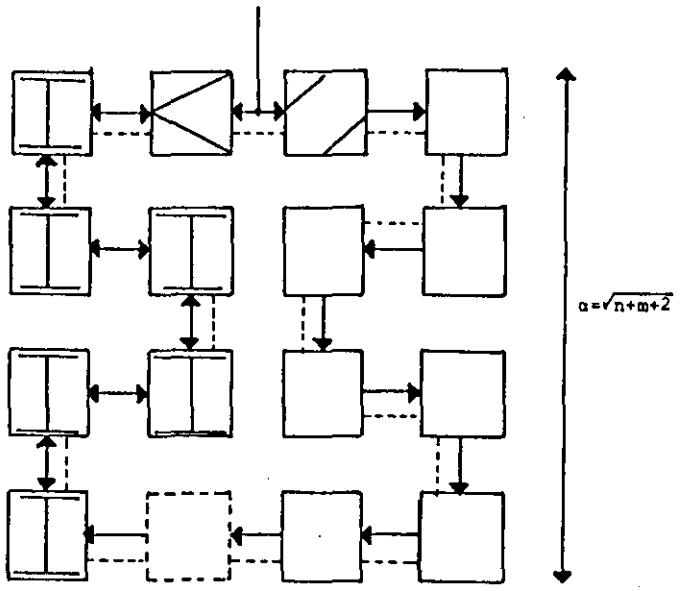
k and have A_k moving systolically in the array. Next we must insert x_k into the solution vector and insert v_k into the basis ejecting x_ℓ and v_ℓ from the solution and basis respectively. This is achieved simply by using the cylinder arrangement to propagate w_2 from part A to part B using part C. w_2 continues into part B forming a top left to bottom right wavefront (see Fig.(7.6.3)) which computes the x_{ik} values by accumulating partial products from left to right. At the same time the index k filters along the part C cells towards the righthand boundary column of part B. At time t=(n+m+5)+(m+4) cycles the first x_{ik} associated with the first row of part B cells is delivered to the first cell in column (m+3) and the value x_{io}/x_{ik} is computed, just as k reaches the rightmost part C cell. On the next cycle both x_{io}/x_{ik} and k are loaded into the top right boundary cell. Successive cycles sees the remaining results loaded into boundary cells as the value k is shifted down to the bottom cell. w_2 is reflected on reaching the right-hand boundary cells to form w_3 (moving top right bottom left) and computes the partial ordering to locate the index ℓ and the variable to be eliminated. Accompanying w_3 is a sequence of control bits issued by the boundary cells which cause row interchanges moving the pivot row to the mth cell row. Hence at time t=(n+m+5)+(m+4)+(m+1) the values, k,ℓ and p are known and reside in the bottom right boundary cell, with $x_{\ell k}$ and $x_{\ell o}$ set in the cell immediately left, and the basis update can start. The vector v_k is introduced to the basis by reflecting w_3 at the (m,m+3) position to form two wavefronts w_4 and w_5.

w_5 enters the H-cell at t=(n+m+5)+(m+4)+(m+1)+2 modifying its values allowing the test of the modified x_{n+m+2} and x_{n+m+1} values to decide the course of the next iteration, and triggering the overwrite of

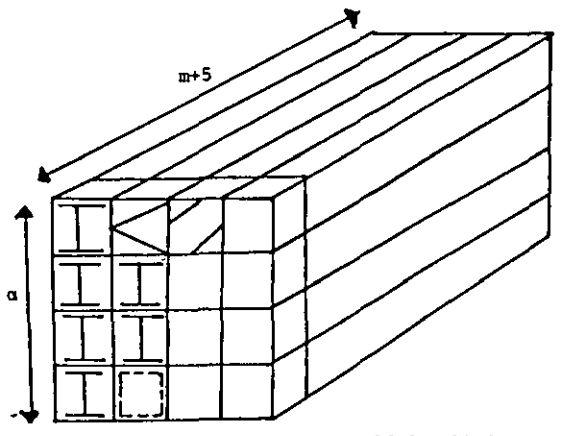
index ℓ by k in the boundary cell. w_5 then becomes w_1 on the next iteration. This implies that modification of the basis can be overlapped with the calculation of the next iteration. Clearly w_4 (which updates the basis) must leave the part B section before w_5 propagates through part A to enter part B, and demands that $n \geq m$ to yield an iteration time of $T=3m+n+12$ cycles. When $n<m$, w_5 and w_4 interfere causing w_5 to compute with the wrong basis elements. The problem is easily solved by adding m-n dummy (delay) part A cells to yield a timing $T=4m+12$ cycles per iteration.

The cell definitions are easily constructed from the revised algorithm and the wavefront patterns. Clearly the boundary cells are simpler than the previously designed sorting cells and use only unidirectional dataflow to construct the partial ordering. Part A cells are simply inner product cells augmented with control triggers and extra switching for transferring A_j data. Part B cells are also inner products with addition row swapping and are closer to the cell definitions for Fig.(7.5.1). Finally Fig.(7.6.1) is a point-to-point connected array and we assume that wavefronts encroaching on other parts of the array are cleaned up by the part C section or (m+1)st row of part B to preserve computation on subsequent iterations. A more efficient layout of the cylinder is achieved in 3-D by considering each column through the array to be a systolic ring and using a variation of Fig.(7.1.5) as shown in Fig.(7.6.4).

Although the cylinder provides an alternative and slightly faster array than the standard Simplex method of Theorem (7.5.1) we require O(mn) inner product type cells to store the A matrix which the revised Simplex algorithm was designed to avoid. The compacted array in Fig.



a) Ring segment of systolic cylinder (n=7, m=7)

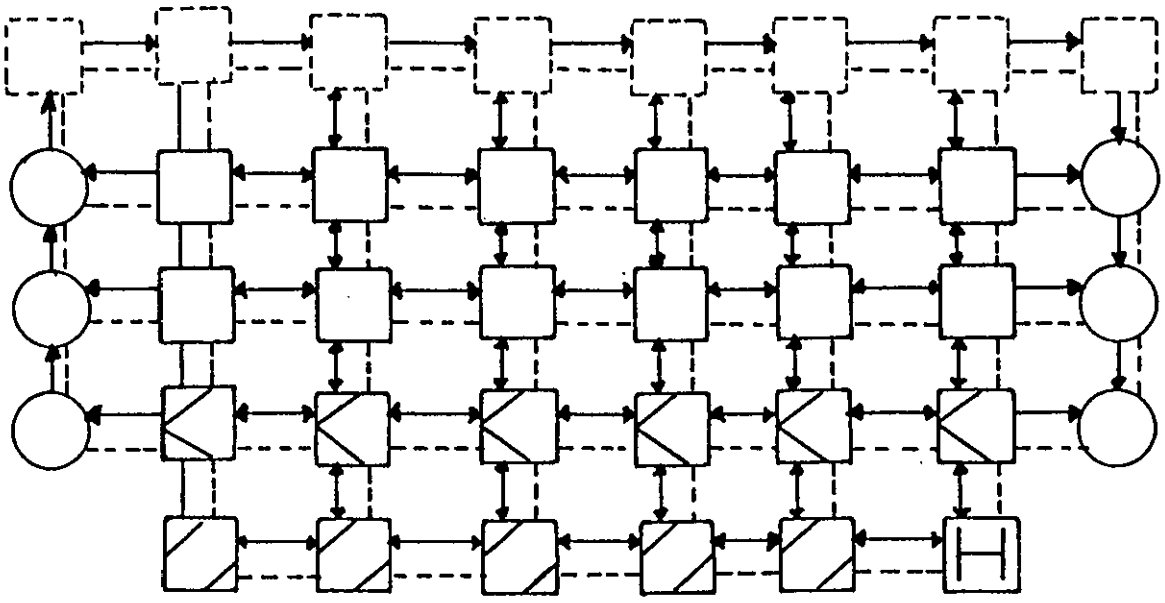


b) Cross-section of folded cylinder

FIGURE 7.6.4

(7.6.5a) remedies this problem for m>n by folding Fig.(7.6.1) along the part A, part B partition and mapping cells containing A_{ij} elements into cells containing u_{ji} elements of the basis inverse (see Fig.(7.6.5b)). The basic idea is to save O(nm) ips cells by adding extra control registers and switching to the $O(m^2)$ cells already required for recording the basis.

The start of an iteration, as before, begins in the H-cell, where



a) Array structure (n<m)

u_{11} / a_{11}	u_{12} / a_{21}	u_{13} / a_{31}	u_{14} / a_{41}	u_{15} / a_{51}	x_1
u_{21} / a_{12}	u_{22} / a_{22}	u_{23} / a_{32}	u_{24} / a_{42}	u_{25} / a_{52}	x_2
u_{31} / a_{13}	u_{32} / a_{23}	u_{33} / a_{33}	u_{34} / a_{43}	u_{35} / a_{53}	x_3
u_{41} / a_{14}	u_{42} / a_{24}	u_{43} / a_{34}	u_{44} / a_{44}	u_{45} / a_{54}	x_4

b) Initial loading of compacted array (m=3, n=3)

FIGURE 7.6.5: Compacted array for revised Simplex

we decide whether Phase I or Phase II is applicable. A control signal is propagated left along the (m+1)st row to select u_{m+1} or u_{m+2} sending it upwards with additional controls to generate a wavefront w_1 moving from the bottom right to top left corner of the array (see Fig.7.6.6). As w_1 partial products of δ_i (γ_i) are accumulated from right to left loading the values into their respective boundary cells on the left where a label j identifying column $\bar{A}_j$ also resides. w_1 is reflected by the boundary cells to become w_2 which propagates the value δ_k (γ_k) to the top of the boundary column as it moves to the right top corner of the array transferring column $\bar{A}_j$ to the top boundary (formerly part C) cells. On reaching the top left corner of the array w_2 deposits the value δ_k (γ_k) and the index k into the top boundary cells, before being reflected to form w_3 (a wavefront headed for the bottom right corner). w_3 pushes k along the top boundary cells to the right column of boundary cells, and produces control values associated with the $\bar{A}_j$ elements reflected by the top boundary back into the array to form the partial products of x_{ik} being accumulated left to right. On reaching the rightmost cell w_3 is reflected forming w_4 moving towards the bottom left corner which shifts k to the bottom right boundary cell, while forming the partial ordering $x_{\ell 0}/x_{\ell k}$, and producing control signals to move u_ℓ to the mth cell row. As w_3 leaves the array, k,ℓ and p are known and the basis update can be overlapped with w_4 replacing ℓ by k. The modification is performed by a wavefront w_5 propagated from bottom right to top left while w_6 modifies row m+2 of the array. Once the H-cell is modified the next iteration can start. Clearly the compacted array dataflow is simply a folded version of the systolic cylinder, which improves cell efficiency by interleaving wavefronts. The basic

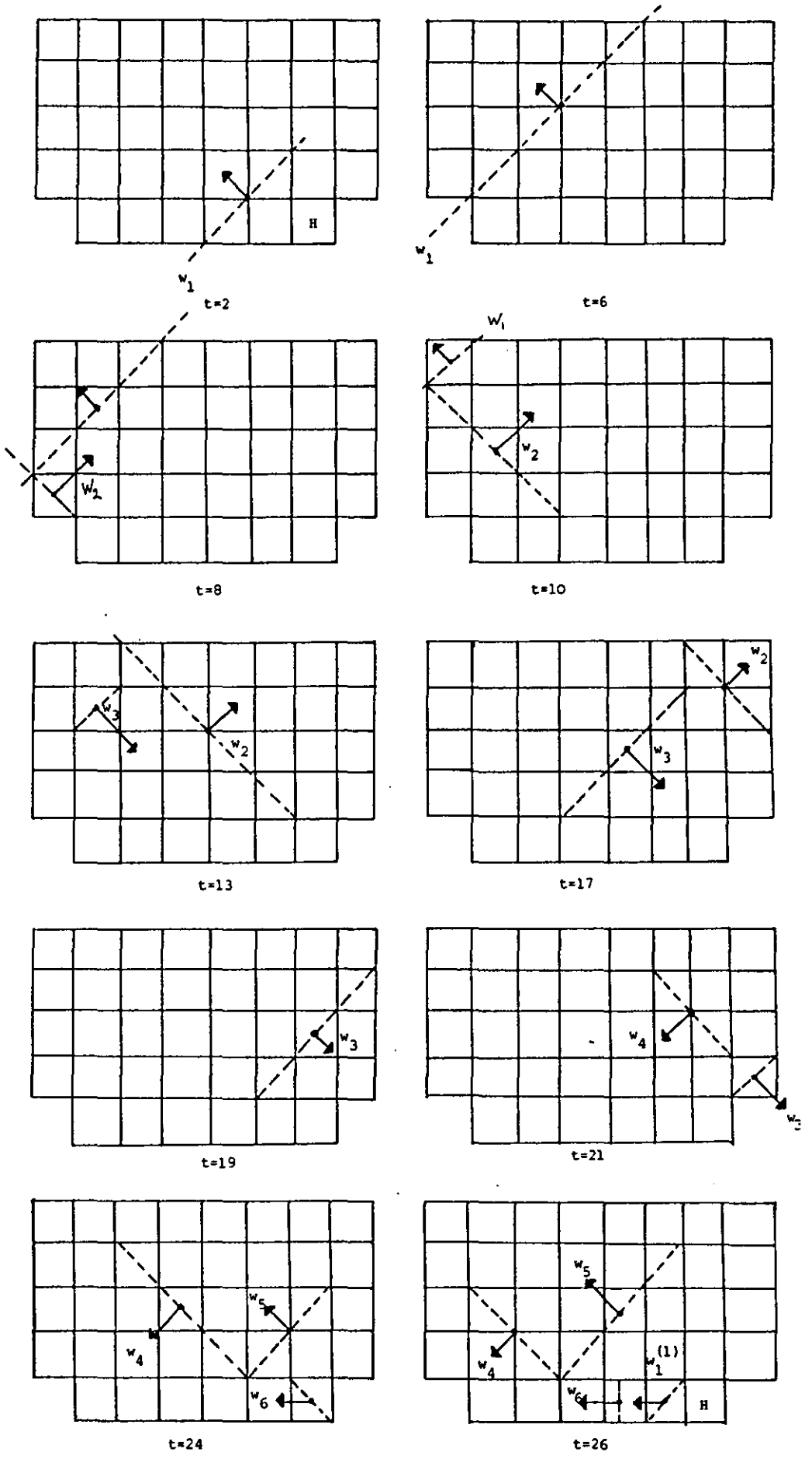


FIGURE 7.6.6: Snapshots of systolic wavefronts for the compacted revised simplex array

timings of the algorithm can be summarised as follows:

(i) (m+2) cycles for w_1 to reach the left boundary (i.e. compute the first δ_i (γ_i).

(ii) (m+2) cycles for $\delta_k = \min_i(\delta_i)$ to reach the top boundary.

(iii) (m+3) cycles for the k value to reach the right boundary.

(iv) (m+1) cycles for k to move to the bottom right boundary cell and produce p and ℓ

(v) 4 cycles for w_5 to enter the H cell and update its contents so that the next iteration can begin.

Thus one iteration requires T=4m+12 cycles as in the cylinder arrangement.

The cell requirements are:

(i) (m+2)*(m+2) for basis cells

(ii) 2m for left and right boundary cells

(iii) m+4 for upper boundary cells,

giving a total of $(m+2)^2+3m+4$ cells.

7.7 AN ORTHOGONAL DESIGN FOR THE ASSIGNMENT PROBLEM

Finally we present an array to implement the assignment problem which is stated simply as follows.

Let there be n tasks which must be performed by n individuals, the cost of individual i performing task j is denoted by c_{ij}. The problem is to assign people to the tasks in a way that minimises the cost of completing all the tasks. More formally,

let,

$$x_{ij} = \begin{cases} 1 \text{ if person i does task j} \\ 0 \text{ otherwise }, \quad i=1(1)n,\ j=1(1)n. \end{cases} \tag{7.7.1}$$

We minimise the total cost, according to the constraints that one person is assigned one task, and each task is assigned to 1 person.

That is minimise,

$$f = \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij} \qquad (7.7.2)$$

subject to

$$\left.\begin{aligned} &\sum_{j=1}^{n} x_{ij} = 1 \quad i=1(1)n \\ &\sum_{i=1}^{n} x_{ij} = 1 \quad j=1(1)n \\ &x_{ij} = 0 \text{ or } 1 \quad i=1(1)n,\ j=1(1)n \end{aligned}\right\} \qquad (7.7.3)$$

The problem can be represented by an n×n table or matrix $C=c_{ij}$ called the cost matrix, and solved by manipulating the table entries. An efficient method of solution is the Hungarian algorithm (see Wu & Coppins [81]) which can be stated simply as follows:

STEP 1: [FORM A REDUCED COST MATRIX]

a) For each row in the cost matrix locate the smallest number in the row and subtract it from each number in that row.

b) For each column in the resulting matrix locate the smallest number in the column and subtract from each number in that column.

STEP 2: [LINE DRAWING]

Find the minimum number of lines through rows and columns of the reduced cost matrix, such that every zero has a line through it.

IF the number of lines is n THEN STOP (optimal solution found)

ELSE proceed to STEP 3.

STEP 3: [FORM A NEW REDUCED COST MATRIX]

(a) Locate the smallest number in the matrix without a line through it.

(b) Subtract this number from all uncovered numbers.

(c) Add the number to all numbers on the intersection of two lines. (i.e. twice covered).

GOTO STEP 2.

The final solution is then constructed by assigning a worker to a job so that the reduced cost is zero. This is performed by first checking rows and then columns for rows and columns with only a single zero, the assignment is the (i,j) ordered pair locating the zero. This solution technique exhibits a number of convenient items for systolic solution. For instance, it involves only add/subtract operations implying simple and compact basic cells, and produces a square array rather than rectangular in the above Simplex algorithms producing tighter and simpler control.

EXAMPLE: From Wu & Coppins [81]

Consider the following assignment problem cost matrix

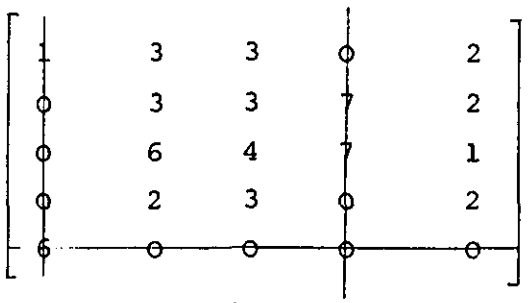
		Job				
		1	2	3	4	5
	1	2	4	5	1	4
	2	4	7	8	11	7
Worker	3	3	9	8	10	5
	4	1	3	5	1	4
	5	7	1	2	1	2

STEP 1: Preprocessing for initial reduced cost matrix

$$\begin{bmatrix} 1 & 3 & 4 & 0 & 3 \\ 0 & 3 & 4 & 7 & 3 \\ 0 & 6 & 5 & 7 & 2 \\ 0 & 2 & 4 & 0 & 3 \\ 6 & 0 & 1 & 0 & 1 \end{bmatrix}$$ row pass

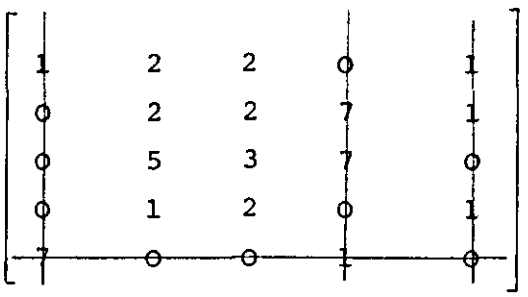
$$\begin{bmatrix} 1 & 3 & 3 & 0 & 2 \\ 0 & 3 & 3 & 7 & 2 \\ 0 & 6 & 4 & 7 & 1 \\ 0 & 2 & 3 & 0 & 2 \\ 6 & 0 & 0 & 0 & 0 \end{bmatrix}$$ column pass

STEP 2: Drawing minimum number of lines



STEP 3: Smallest uncovered number is 1

STEP 2: (Repeat)



STEP 3: Smallest uncovered number is 1 giving

$$\begin{bmatrix} 1 & 1 & 1 & 0^* & 1 \\ 0^* & 1 & 1 & 7 & 1 \\ 0 & 4 & 2 & 7 & 0^* \\ 0 & 0^* & 1 & 0 & 1 \\ 8 & 0 & 0^* & 2 & 1 \end{bmatrix}$$

which requires n=5 lines, solution assignment indicated by asterisks.

The systolic design is partitioned into two systolic arrays. The first array is a linearly connected array of n cells computing the reduced cost matrix of Step 1, and essentially performing a preprocessing task. The second array is an (n+2)*(n+2) orthogonally connected mesh performing STEPS 2 and 3, the core of the algorithm and is termed the Assignment Problem Iteration (API) Array. We could consider a third array to recover the solution as post processing, but the task is trivial and not pursued here.

The pre-processing array is shown in Fig.(7.7.1) and requires four passes through the array to produce the reduced cost matrix.

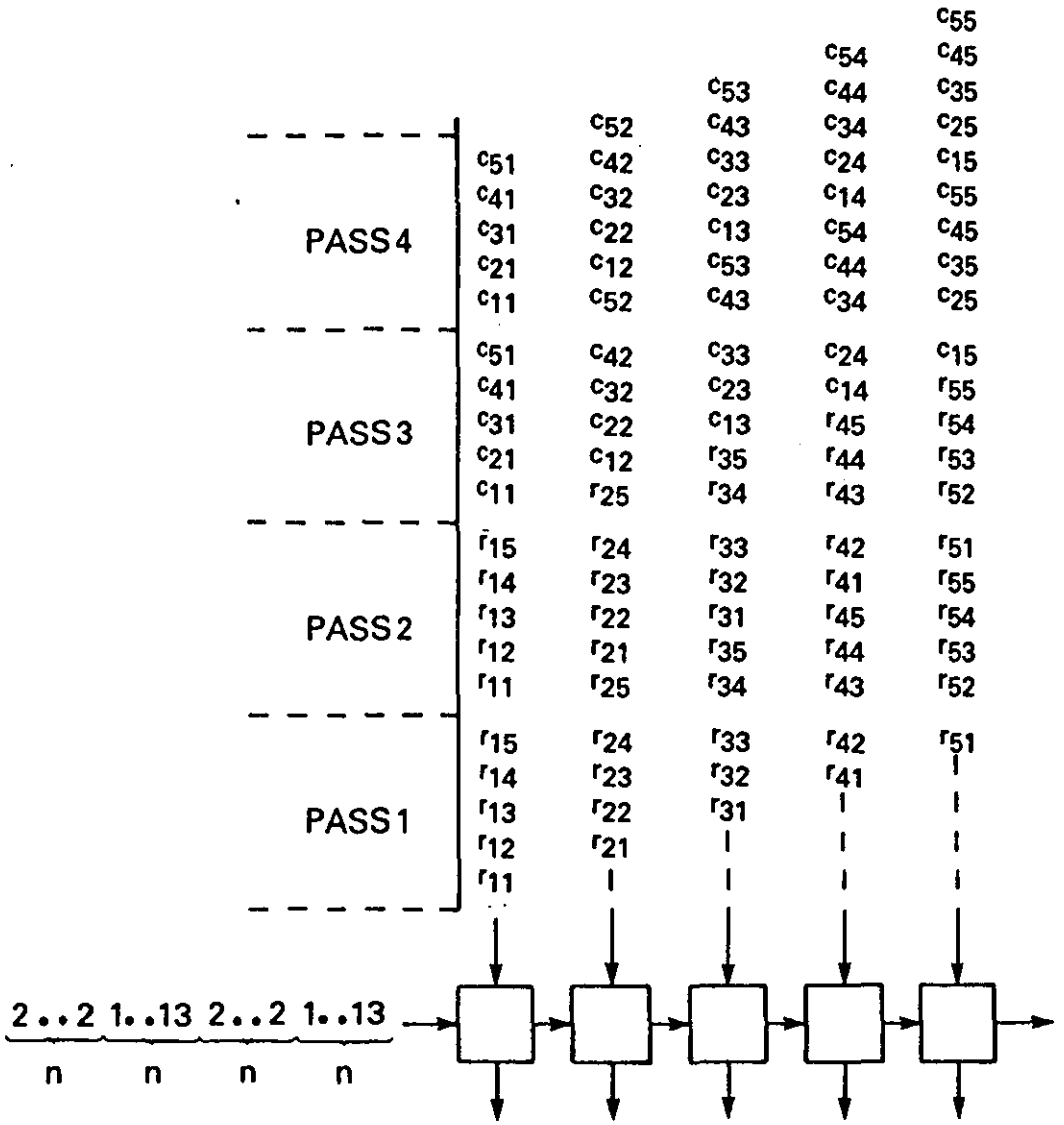


FIGURE 7.7.1: Preprocessing array (n=5)

PASS 1: Find minimum element of each row_i storing it in $cell_i$.

PASS 2: Subtract the stored value from all the elements in the row.

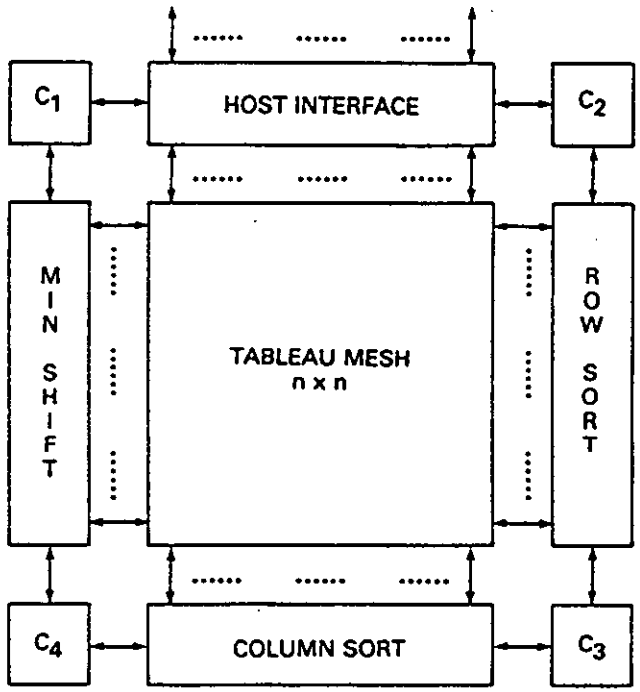
PASS 3: Find minimum element in each $column_i$ storing it in $cell_i$.

PASS 4: Subtract the stored value from all elements in the column.

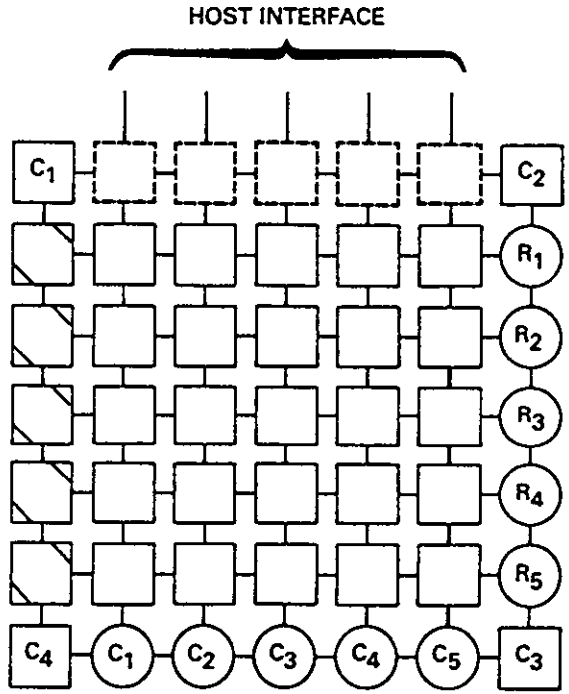
This implies that the basic cell requires a subtracter and a comparator, but if we include a status bit set by the subtractor to indicate negative values the less than condition can be detected without the comparator. (Essentially subtract the stored and incoming values, check the status bit and switch to the correct output accordingly, the subtract result is ignored). The change from pass 2 to pass 3 requires the matrix input to be turned from row ordering to column ordering. As the matrix is square the last column element leaves the array in a row pass, as the first column element is required to enter the array, and there is time to reorganise the data 'on-the-fly' by the host machine or a buffer. The data output by the pre-processing array at each pass is looped back to the array input forming a ring, on the last pass the ring can be broken to form a suitable interface for loading the API. Hence the total time for STEP I is T=5n cycles (where a cycle is the cost of add/subtract) with the last n cycles overlapped with API timings.

The API is shown in Fig.(7.7.2) and has the same global structure as the mesh used for rank annihilation Fig.(4.3.1.1) incorporating a systolic control ring (SCR) to generate any ordering of wavefronts across the grid. The boundary arrays and internal cell definitions are of course different, and the Hungarian algorithm can be computed by a number of phases as illustrated by Fig.(7.7.3).

PHASE 0: Loading of the reduced cost matrix, the SCR is not required and the loading of square meshes of processors with data is well understood.



a) Assignment procedure iteration (API) array



b) API systolic array (n=5)

FIGURE 7.7.2: Structure of assignment problem mesh

PHASE I: (start of the iteration algorithm)

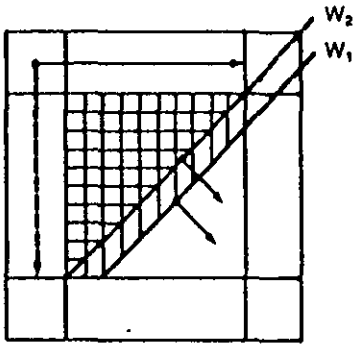
Two wavefronts w_1 and w_2 are generated as follows:-

a) c_1 generates controls moving systolically along c_1-c_2 and c_1-c_4 through the host interface and min shift arrays, producing w_1. At the start, the min shift cell immediately below c_1 contains the smallest uncovered element in the current reduced cost matrix (zero for the starting matrix of Step I). As w_1 moves across the tableau it performs the reduced cost modification (of Step 3) according to covered line positions.

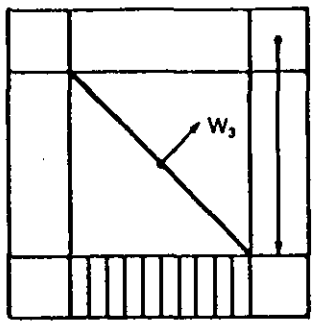
b) On the next cycle after c_1 generated controls for w_1 a second control c_1-c_2 and c_1-c_4 produces a wavefront w_2 parallel to w_1 which counts the number of uncovered zeros in each column.

PHASE II: On reaching c_2 and c_4 the controls associated with w_2 are relayed along c_2-c_3, c_4-c_3. At this time w_2 and w_1 are half-way across the tableau and the first column has completed its zero count or column zero weight (CZW). Thus, as the control moves along c_4-c_3 the CZW's are loaded into the column sorter (in the same manner as the Simplex tableau see Fig.(7.5.3)) and starts an ODD-EVEN transposition sort bubbling the max CZW right and the min CZW left. As weights are swapped a wavefront w_3 of swap controls propagates across the tableau re-aligning column elements.

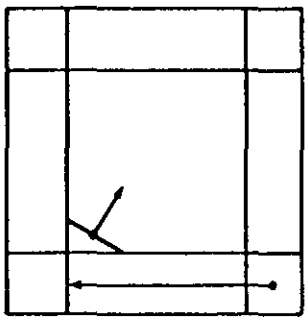
PHASE III: When controls reach c_3, w_1 and w_2 have left the mesh, and the max CZW is in the sort cell immediately left of c_3 and w_3 is halfway across the grid. Controls now travel along c_3c_4 and c_3c_2, the former signals closing down the column sorter. Hence when controls reach c_4 and c_2 the CZW's are completely sorted and the last column swap instruction has entered the mesh.



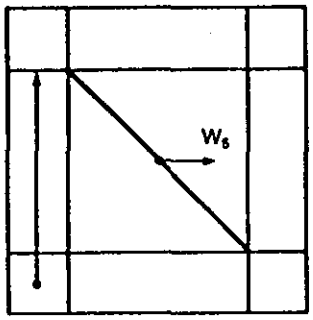
PHASE I



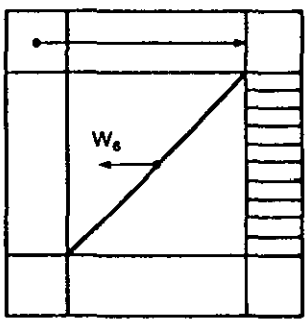
PHASE II



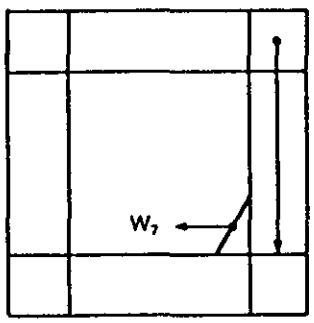
PHASE III



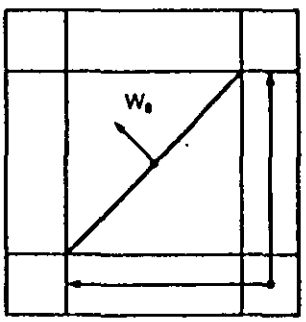
PHASE IV



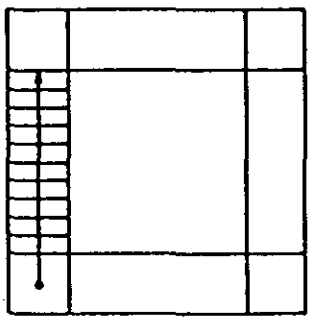
PHASE V



PHASE VI



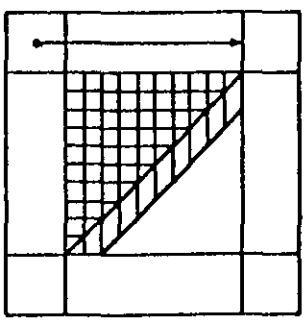
PHASE VIII



PHASE IX

FIGURE 7.7.3: API WAVEFRONTS

The computational phase for API phase VII not shown, is the line drawing section.



PHASE X (loop)

PHASE IV: To complete the control cycle signals move along c_4-c_1 and c_2-c_1 and a wavefront w_5 performing the same task as w_2 but collecting the row zero weights (RZWs) by counting uncovered zeros in each row. When the controls reach c_1 the bottom row has completed its count.

PHASE V: This phase is analogous to phase II, and c_1 generates a new signal c_1-c_2. As w_5 continues to move right its controls (in the absence of SCR values) are used to load the RZW weights from bottom to top and start the sorter. Thus when the new SCR control reaches c_2, all the row weights have entered the row sorter and the minimum RZW is in the cell immediately below c_2. A wavefront w_6 generated by the sort is propagated left to perform relevant row swaps, hence at the end of the phase w_6 is half way across the tableau.

PHASE VI: The control now moves c_2-c_3 and c_4-c_3 and closes down the row sorter, while the remaining row swaps are carried out on the tableau. At the end of the phase the row weights are fully sorted and the last row swap has entered the table mesh. All sorting cells are off and the max RZW and CZW reside in cells adjacent to c_3.

PHASE VII: c_3 now takes control of the algorithm and initiates line drawing. The basic technique for drawing the minimum number of lines is given below and is only outlined here.

a) c_3 collects both maximum RZW and CZW values to select the largest.

b) IF both values are zero THEN no uncovered zeros exist GOTO PHASE VIII.

IF n lines have been drawn THEN optimal solution STOP

ELSE draw a line.

A line is drawn by zeroing the selected RZW (or CZW) and issuing a

control value along the last row (or column) of the table mesh setting the cell elements line state (covered, twice covered, uncovered). Where a cell element is zero and uncovered the associated RZW (CZW) in the adjacent sorter cell must be decremented making the element disappear from future line drawing. After a short delay to allow the marker wavefront sufficient head start and to avoid interference, c_3 issues controls along c_3-c_4 and c_3-c_2 to re-activate the sorters, which re-sort the modified RZW's and CZW's, while bubbling the marked row or column away from the line marking area of the array. On reaching c_4 and c_2 the signals are returned to c_3 (along the reverse paths) closing down the sorters. On reaching c_3 the row and column lists are re-sorted and we return to a) above to draw another line.

PHASE VIII: If we reach this phase both the CZW and RZW lists have been reduced to zero in less than n lines and a modification of the cost matrix is required. c_3 releases control propagating signals along c_3-c_4 and c_3-c_2 producing wavefront w_8 which moves the minimum uncovered element of each row left. When the control on c_3-c_4 reaches c_4 the minimum of the last table row is available.

PHASE IX: Signals are relayed by c_4 and c_2 to travel c_2-c_1 and c_4-c_1 and complete the second circuit of the SCR. As the signal for c_4 moves to c_1 it loads the minimum row values (MRV) into the min. shift array, which constructs a partial ordering on the MRV filtering towards c_1. When signals reach c_1 the minimum uncovered table resides in the cell immediately below c_1. We now GOTO PHASE I.

STOP: If stop is reached in Phase VII the API is closed down and the final tableau output.

REMARK: We assume that like the simplex array, sorting cells contain an

index for row and columns so that the final result is easily recovered.

The line drawing method uses a simple heuristic technique to decide where to draw lines - i.e. try to cover as many zeros as possible with each line. But can we be sure that we always draw the minimum number of lines?

Theorem 7.7.1: The systolic API always draws the minimum number of lines for a given reduced cost matrix.

Proof:

(i) If we do not cover all the zeros, on resorting the RZW, and CZW lists c_3 will receive a non-zero value from a list and draw another line.

(ii) Let k_{min} be the minimum number of lines, and put $k>k_{min}$ such that without loss of generality $k=k_{min}+1$. From (i) and the heuristic above we must draw a redundant line, whose zeros are all covered at the time of drawing. This implies the elements of the CZW and RZW must be zero hence c_3 cannot draw a line. This is a contradiction, thus proving the theorem.

A program implementing the API mesh as described above is given in the appendix and also defines the cell operations. The use of the SCR control flow simplifies the array timing which is given as follows. The control values travel around the SCR exactly twice to complete steps 2 and 3 of the algorithm. On the second SCR cycle we perform line drawing which has a variable time. If we denote the time spent line drawing as T_{ld} a single API iteration costs,

$$T_i = 8n + \theta_i T_{ld} + 8 \ , \ 0<\theta_i<1 \ , \qquad (7.7.4)$$

as the cost of a complete SCR cycle is 4(n+1) cell cycles. If we perform z iterations the total time is,

$$\sum_{i=1}^{z} T_i = \sum_{i=1}^{z} 8n + \sum_{i=1}^{z} \theta_i T_{\ell d} + 8z$$

$$= 8nz + 8z + \sum_{i=1}^{z} \theta_i T_{\ell d} \quad . \tag{7.7.5}$$

The time to draw a single line is bounded by the cost of resorting the CZW and RZW lists, and is equivalent to the time of traversing a side of the API twice i.e.,

$$T_1 = 2(n+1) + k \ , \tag{7.7.6}$$

with k>0 is a constant delay required for separating individual wavefronts. Now if we assume that once a line is drawn it is never removed,

$$\sum_{i=1}^{z} \theta_i T_{\ell d} = nT_1 = 2n^2 + n + kn \ , \tag{7.7.7}$$

as we can draw at most n lines, consequently, if we draw all n lines on a single iteration we get the lower bound,

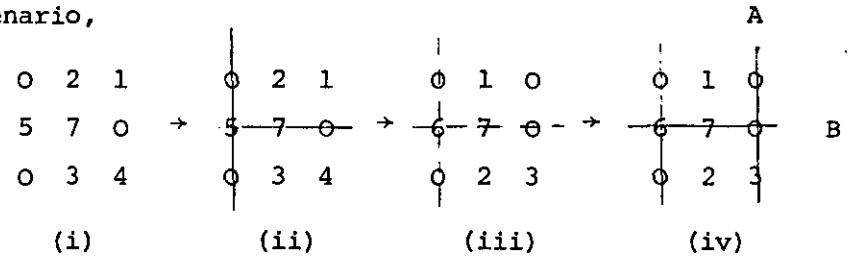
z=1 $$T_\ell = 2n^2 + (k+9)n + 8 \ , \tag{7.7.8}$$

and if we draw one line every iteration the upper bound

z=n $$T_u = 10n^2 + (k+9)n \ . \tag{7.7.9}$$

(7.7.5) is verified by the test example given below with accompanying program generated snapshots of control flow and table images.

In general, however, some lines will have to be removed, and could increase the computation time. For example, consider the following 3*3 scenario,



corresponding to the mesh operation as explained. In step (iv) the algorithm uses three lines when just two will suffice, contradicting

Theorem (7.7.1). Clearly the line B is redundant and should be removed after A is drawn. The problem occurs because lines are retained between successive iterations on the API. Modifying w_1 in PHASE I to uncover elements after performing the table update solves the problem because c_3 always starts line drawing afresh on each iteration. Unfortunately this means that not only redundant lines disappear hence,

$$\sum_{i=1}^{z} \theta_i T_{\ell d} = \sum_{i=1}^{z-1} (n-1)T_1 + nT_1 = \sum_{i=1}^{z} (n-1)T_1 + T_1$$
$$= T_1(z(n-1)+1) \qquad (7.7.10)$$

as at most (n-1) lines can be drawn on the first z-1 iterations and n on the last, yielding the revised upper bound,

$$T_u = z(2n^2+n(8+k)+6-k)+2(n+1)+k \qquad (7.7.11)$$
$$= 2n^3+n^2(8+k)+(8-k)n+(2+k) \ .$$

REMARK: To implement this procedure, the program code requires the line state to be cleared after an update in t.cell, and n=0 at the end of a line drawing phase in controller.3.

A more flexible approach is to incorporate erasure of redundant lines in the line drawing phase to yield,

$$T = 8z(n+1) + \sum_{i=1}^{z} \theta_i T_{\ell d} + \sum_{i=1}^{z} \phi_i T_E \ , \qquad (7.7.12)$$

where $0<\phi_i\leq 1$ and T_E is the time spent erasing lines, and thus avoiding redrawing the same lines on successive updates of the reduced cost matrix. Removing lines, however, presents a number of problems.

(i) Identification: Clearly, a line is redundant if all its zeros lie on the intersections with other lines, as on the next cost matrix update they become nonzero.

(ii) Location: Redundant lines must be removed before we attempt to find the table minimum otherwise an incorrectly updated table will result.

(iii) Implementation:

a) two wavefronts must pass over the mesh to identify redundant lines one for rows and one for columns.

b) a further two wavefronts are required to erase the lines.

c) the count of lines must be modified to avoid premature termination of the algorithm.

Line erasure can be implemented effectively as follows. Notice that c_3 selects the MAX(CZW,RZW) and hence only one list can be re-sorted, and generate swap data. For instance, suppose the CZW is selected by c_3, the CZW list must be resorted, but the RZW is unchanged and already sorted. Consequently, the column line which is to be drawn can only product redundant row lines. It follows that the sorter start up signals travelling c_3-c_4 and in particular c_3-c_2 can be used to generate a redundancy status bit vector R such that,

$$R(i) = \begin{cases} 1 \text{ for a redundant line occupying table row } i \\ 0 \text{ otherwise.} \end{cases}$$

The row sorter sets R(i)=1, i=1(1)n making all lines redundant initially. The R(i) travel on the leading wavefront of swaps generated by the column sorter along with the new line column being bubbled away from c_3, thus ensuring that each R(i) meets all the elements of row i. We reset R(i)=0 whenever an uncovered or once covered zero element is encountered Thus, when the column sort signal c_3-c_4 reaches c_4 the last row is identified as required or redundant. Next, as we closedown the sorters with c_4-c_3 and c_2-c_3 return signals, an additional signal along c_4-c_1 can be used to reflect the R(i) incident on the minshift array tagging them to the last wavefront of column swaps. As each R(i) is now set to indicate redundant lines, the return trip can be used to erase lines, by

modifying each elements line state. It follows that when the sorter stop signal returns to c_3 the c_4-c_1 redundancy control reaches c_1 and the last table row is completely erased, and the first row about to start. Consequently, drawing of the next line can be overlapped with erasure. Furthermore if we include an adder in the minshift cells, the arrival of the R(i) bits can be used to count the number of redundant row lines (say r). Thus moving the line count from c_3 to c_1 means that the arrival of the redundancy control at c_1 can set the number of lines as n_d-r+1 where n_d is the total number of lines drawn (we remove r and add 1 column line). A similar argument holds for resorting the RZW list with the column redundancy vector C(i), i=1(1)n, except that c_2 uses a signal c_2-c_1 and the host interface accumulates the column redundancy count c, with n_d-c+1 the updated line count. The practical point is that erasure is overlapped with line drawing and sorter close down hence,

$$\sum_{i=1}^{z} \phi_i T_E = 0 \ , \qquad (7.7.13)$$

reducing (7.7.12) to (7.7.5) and yielding the timing bounds (7.7.8) and (7.7.9).

Finally, we consider the complexity of the cells. Fig.(7.7.4) indicates a loose structure for the table cell the most complex cell in the mesh. Lines drawn are represented by a line.state variable (LS) which can take three states:

(i) uncovered - no line

(ii) covered - by a single line

(iii) twice covered - by two lines at an intersection.

In the program we have used general variables, but in hardware terms line states require only three bits, where,

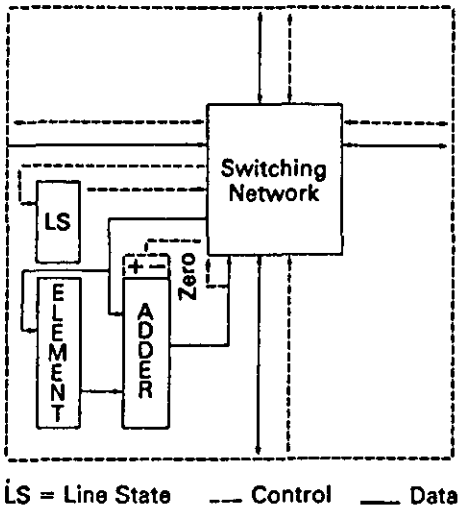


FIGURE 7.7.4: Tableau cell arrangement

100 no line

001 single line

010 intersection

and line drawing or erasure is implemented simply by circulating shifts left or right respectively. For the purposes of column and row swapping we can consider these 3-bits tagged to the actual table element. By the mechanics of the array the row and column of table cells next to the column and row sorters are the only ones to receive line drawing commands. As lines are marked on the tableau cells, the cells containing previously unmarked zeros must generate a signal to modify the associated RZW/CZW value in the adjacent sorter cell. The tableau must therefore be able to detect a zero, which can be achieved by taking the NAND of the element bits (or maintaining a zero-status flag also tagged to the element). Additional hardware is also required to perform the cost matrix update and calculation of the RZW's and CZW's. Using the line.state bits and the zero flag the calculations can be controlled as follows,

Line state	Zero	Action
OO	X	Subtract minimum element from cell element
Ol	X	Add zero to Tableau element
lO	X	Add minimum to cell element

(a)

Line state	Zero	Action
OO	1	Add 1 to row or column index
Ol	X	Null (add zero) "
lO	X	Null (add zero) "

X=don't care.

(b)

TABLE 7.7.1

The last problem is the location of the minimum element, requiring the comparison of the incoming minimum from the right with the cell element, and the resulting minimum to be placed on the left output. The comparison can be performed by subtracting the cell element from the incoming element, the new minimum is then the cell element if the result is positive and the input element if negative (recall that all table elements must be positive). The result can be detected by the sign bit of the adder/subtracter arrangement. We conclude that with an adder or subtracter, and 4 bits (line.state and zero flag) together with combinational logic for the control commands, the tableau cells have a simple structure. By similar arguments we conclude that the sorter cells and min.shift arrays can also be represented by adder/subtracter or equivalent cell structures with SCR control logic. The controllers themselves are simple state machines with c_3 the most complex requiring a counter for line counting (with line erasure this

is moved to c_1 and requires an adder/subtracter). Consequently the API area is bounded by the cost of (n+2)*(n+2) tableau cell arrangements. Including the time (5n) for the preprocessing and 2n cycles to unload the table produces $O(n^2)$ adder/subtracter cells and the time bounds,

$$2n^2 + (16+k)n+8 \leq T \leq 10n^2 + (16+k)n \ , \qquad (7.7.14)$$

with line erasure, and,

$$2n^2+(16+k)n+8 \leq T \leq 2n^3+n^2(8+k)+(15-k)n+(2+k) \qquad (7.7.15)$$

without line removal, giving the erasure procedure a distinct computational advantage.

7.8 SUMMARY

In this chapter we have considered the application of systolic arrays to table manipulating and generating algorithms. First a systolic array for improving numerical approximations to integrals using Richardson's extrapolation procedure in the form of Romberg integration was considered. Two designs for generating a table of size n were presented, the first a linear systolic array of n cells and the second a systolic ring using only 1/3 of the cells. Both designs required 3n ips cycles to construct the table, a significant improvement over the $O(n^2)$ steps required sequentially.

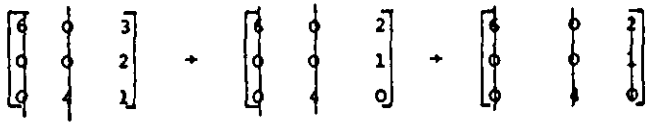
Next the construction of extrapolation tables used in the solution of Ordinary Differential Equations (ODE's) associated with initial value type problems was examined first for a low order formula i.e. Eulers method which is combined with extrapolation to improve estimates of solution. The technique was extended to the Bulirsch and Stoer algorithm and a generic systolic array form given to extrapolation table construction. A generic timing T=c(n-1) was

DESIGN TESTING

Example: Consider the starting API matrix (n=3)

$$\begin{bmatrix} 6 & 0 & 3 \\ 0 & 0 & 2 \\ 0 & 4 & 1 \end{bmatrix}$$

Then following sequence produces



line drawing modification termination

Below are snapshots of the API-array simulated on a VAX 11/750 BSD 4-2 using Loughborough OCCAM [6]. The above test requires (Z=)2 iterations of the method, fully testing the array. Two snapshots are present.

(i) Control wavefront: indicating systolic control flow + generated wavefronts. Each entry is a quadruple of the form

a) (c_0,c_1,c_2,c_3) Tableau cell

b) (i,c_1,c_2,c_0) Row sort cell

c) (j,c_1,c_2,c_3) Col sort cell

d) (min,c_0,c_2,c_3) Min shift cell

e) relevant signals for controllers.

(ii) Table output: Indicates modifications, minimum uncovered elements, computed row/column weights and sorter generate sweeps. Each entry is a single value

a) Telement - Tableau cell

b) Row weight - row sort

c) Col weight - col sort

d) Minimum row - min shift

e) Zero except for controller 1 - collects minimum element for modification.

REMARK: The above example is artificial in that a genuine reduced cost matrix would have a zero in each row and column but it serves its purpose as a test matrix.

Remarks on Tests

1. The total number of cycles from algorithm start to the beginning of the OCCAM closedown is 96. This bound on k is essentially correct, and verified by the above timing elements.

2. Control signals key for snapshots

c_0	c_1	c_2	c_3	ACTION	
1				Swap with right cell	Tableau Cell
2				Swap with left cell	
	1			Construct row weight	
		1		Construct col weight	
	3	3		Modify element	
4				Shift row minimum left	
			3	Draw row line	
3				Draw col line	
	1			Load row weight *	Row sort Cell
	2			Modify row weight *	
	4			Null *	
		2		Stop sort cell	
		3		"	
		4		"	
		1		Load column weight *	Col sort Cell
		2		Modify column weight *	
		4		Null	
	1			Stop sort cell	
	3			"	
	4			"	
2				Load row minimum and test with row below	min shift
← 6 →				Close this I/O port	OCCAM Termination

N.B. other signals and controller codes pass through cells unchanged, c_1 and c_3 modify control signals.

Control Wavefront Snapshots

```
Mar 21 16:46 1986  result Page 1


start cycle                            start cycle
 0000  0000  0000  0000  0000           0000  0000  0000  0000  0001
 0000  0000  0000  0000  1000           0000  0000  0000  0000  1000
 0000  0000  0000  0000  2000           0000  0000  0000  0000  2010
 0000  0000  0000  0000  3000           0000  0000  0000  0010  3300
 0000  1000  2000  3000  0000           0000  1000  2010  3030  3003

start cycle                            start cycle
 0000  0000  0000  0000  0000           0000  0000  0000  0000  0001
 0000  0000  0000  0000  1000           0000  0000  0000  0000  1000
 0000  0000  0000  0000  2000           0000  0000  0000  0000  2000
 0000  0000  0000  0000  3000           0000  0000  0000  0000  3010
 0000  1000  2000  3000  0000           0000  1000  2000  3010  3003

start cycle                            start cycle
 0000  0300  0000  0000  0001           0000  0000  0000  0000  0001
 0030  0000  0000  0000  1000           0000  0000  0000  0000  1000
 0000  0000  0000  0000  2000           0000  0000  0000  0000  2000
 0000  0000  0000  0000  3000           0000  0000  0000  0000  3000
 0000  1000  2000  3000  3003           0000  1000  3000  2000  3013

start cycle                            start cycle
 0000  0100  0300  0000  0001           0000  0000  0000  0000  0001
 0000  0330  0000  0000  1000           0000  0000  0000  0000  1000
 0030  0000  0000  0000  2000           0000  0000  0000  0000  2000
 0000  0000  0000  0000  3000           0000  0000  1000  2000  3000
 0000  1000  2000  3000  3003           0000  3000  1000  2001  2003

start cycle                            start cycle
 0000  0000  0100  0300  0001           0000  0000  0000  0000  0001
 0000  0010  0330  0000  1000           0000  0000  0000  0000  1000
 0000  0330  0000  0000  2000           0000  0000  1000  2000  2000
 0030  0000  0000  0000  3030           0000  1000  2000  0000  3000
 0000  1000  2000  3000  3003           0000  3000  1001  2000  2003

start cycle                            start cycle
 0000  0000  0000  0100  0301           0000  0000  0000  0000  0001
 0000  0000  0010  0330  1000           0000  0000  1000  2000  1000
 0000  0010  0330  0000  2000           0000  1000  2000  0000  2000
 0000  0330  0000  0000  3000           0000  0000  0000  0000  3000
 0030  1000  2000  3000  3003           0000  3001  1000  2000  2003

start cycle                            start cycle
 0000  0000  0000  0000  0101           0000  0000  1000  2000  0001
 0000  0000  0000  0010  1300           0000  1000  2000  0000  1000
 0000  0000  0010  0330  2000           0000  0000  0000  0000  2000
 0000  0010  0330  0000  3000           0000  0000  0000  0000  3000
 0000  1030  2000  3000  3003           0001  3000  1000  2000  2003

start cycle                            start cycle
 0000  0000  0000  0000  0001           0000  1000  2000  0000  0001
 0000  0000  0000  0000  1010           0000  0000  0000  0000  1000
 0000  0000  0000  0010  2300           0000  0000  0000  0000  2000
 0000  0000  0010  0330  3000           0100  0000  0000  0000  3000
 0000  1010  2030  3000  3003           0000  3000  1000  2000  2003
```

```
Mar 21 16:46 1986  result Page 2


start cycle                            start cycle
 0000  0000  0000  0000  0001           0000  0000  0000  0000  0011
 0000  0000  0000  0000  1000           0000  0000  0000  0000  1000
 0100  0000  0000  0000  2000           0000  0002  0000  0000  3020
 0000  0100  0000  0000  3000           0000  0001  0000  0000  2000
 0000  3000  1000  2000  2003           0000  3000  1000  2000  2002

start cycle                            start cycle
 0000  0000  0000  0000  0001           0000  0000  0000  0000  0011
 0100  0000  0000  0000  1000           0000  0000  0000  0000  1000
 0000  0100  0000  0000  2000           0002  0000  0000  0000  3000
 0000  0000  0100  0000  3000           0001  0000  0000  0000  2020
 0000  3000  1000  2000  2003           0000  3000  1000  2000  2002

start cycle                            start cycle
 1000  0000  0000  0000  0001           0000  0000  0000  0000  0011
 0000  0100  0000  0000  1000           0000  0000  0000  0000  1000
 0000  0000  0100  0000  2000           0000  0000  0000  0000  3000
 0000  0000  0000  0100  3000           0000  0000  0000  0000  2000
 0000  3000  1000  2000  2003           0000  3000  1000  2000  2022

start cycle                            start cycle
 0000  0200  0000  0000  0001           0000  0000  0000  0000  0011
 0000  0000  0100  0000  1000           0000  0000  0000  0000  1000
 0000  0000  0000  0100  2000           0000  0000  0000  0000  3000
 0000  0000  0000  0000  3100           0000  0000  0000  0000  2000
 0000  3000  1000  2000  2003           0000  3000  1000  2000  2002

start cycle                            start cycle
 0000  0000  0200  0000  0001           0000  0000  0000  0000  0011
 0000  0000  0000  0100  1000           0000  0000  0000  0000  1000
 0000  0000  0000  0000  2100           0000  0000  0000  0000  3000
 0000  0000  0000  0000  3000           0000  0000  0000  0000  2000
 0000  3000  1000  2000  2003           0000  3000  1000  2003  2002

start cycle                            start cycle
 0000  0000  0000  0200  0001           0000  0000  0000  0000  0011
 0000  0000  0000  0000  1100           0000  0000  0000  0000  1000
 0000  0000  0000  0000  3000           0000  0000  0000  0000  3000
 0000  0000  0000  0000  2000           0000  0000  0000  3000  2000
 0000  3000  1000  2000  2003           0000  3000  1003  2000  2002

start cycle                            start cycle
 0000  0000  0000  0000  0211           0000  0000  0000  0000  0011
 0000  0000  0000  0000  1000           0000  0000  0000  0000  1000
 0000  0000  0000  0002  3000           0000  0000  0000  3000  3000
 0000  0000  0000  0001  2000           0000  0000  0000  0000  2204
 0000  3000  1000  2000  2002           0000  3003  2000  1000  2002

start cycle                            start cycle
 0000  0000  0000  0000  0011           0000  0000  0000  0000  0011
 0000  0000  0000  0000  1020           0000  0000  0000  3000  1000
 0000  0000  0002  0000  3000           0000  0000  0000  0000  3404
 0000  0000  0001  0000  2000           0000  0000  1000  2000  2000
 0000  3000  1000  2000  2002           0000  3000  2000  1000  1002
```

start cycle
0000 0000 0000 3000 3011
0000 0000 0000 0000 1204
0000 0000 1000 2000 3000
0000 0000 0000 0000 2000
0000 3300 2000 1000 1002

start cycle
0000 0000 0000 0000 4001
0000 0000 1000 2000 1000
0000 0000 0000 0000 3000
0000 0000 0000 0000 2000
0000 3000 2300 1000 1002

start cycle
0000 0000 1000 2000 0001
0000 0000 0000 0000 1040
0000 0000 0000 0000 3000
0000 0000 0000 0000 2000
0000 3000 2000 1300 1002

start cycle
0000 0000 0000 0000 0001
0000 0000 0000 0000 1000
0000 0000 0000 0000 3040
0000 0000 0000 0000 2000
0000 3000 2000 1000 1302

start cycle
0000 0000 0000 0000 0001
0000 0000 0000 0000 1000
0000 0000 0000 0000 3000
0000 0000 0000 0000 2040
0000 3000 2000 1000 1002

start cycle
0000 0000 0000 0000 0001
0000 0000 0000 0000 1000
0000 0000 0000 3000 3000
0000 0000 0000 0000 2000
0000 3000 2000 1000 1042

start cycle
0000 0000 0000 0000 0001
0000 0000 0000 3000 1000
0000 0000 0000 0000 3000
0000 0000 0000 0000 2000
0000 3000 2000 1000 1002

start cycle
0000 0000 0000 0000 3001
0000 0000 0000 0000 1000
0000 0000 0000 3000 3000
0000 0000 0000 0000 2000
0000 3000 2000 1003 1002

start cycle
0000 0000 0000 0000 0001
0000 0000 0000 0000 1000
0000 0000 0000 0000 3000
0000 0000 0000 3C00 2000
0000 3000 2003 1000 1002

start cycle
0000 0000 0000 3000 0001
0000 0000 0000 0000 1000
0000 0000 0000 3000 3000
0000 0000 0000 0000 2204
0000 3003 2000 1000 1002

start cycle
0000 0000 0000 0000 0001
0000 0000 0000 3000 1000
0000 0000 0000 0000 3204
0000 0000 0000 0000 2000
0000 3000 2000 1200 1002

start cycle
0000 0000 0000 0000 0001
0000 0000 0000 0000 1406
0000 0000 0000 3000 3000
0000 0000 0000 0000 2000
0000 3300 2000 1000 1002

start cycle
0000 0000 0000 3000 4001
0000 0000 0000 0000 1000
0000 0000 0000 0000 3000
0000 0000 0000 0000 2000
0000 3000 2300 1000 1002

start cycle
0000 0000 0000 0000 0001
0000 0000 0000 0000 1040
0000 0000 0000 0000 3000
0000 0000 0000 0000 2000
0000 3000 2000 1300 1002

start cycle
0000 0000 6000 0000 0001
0000 0000 0000 0000 1000
0000 0000 0000 0000 3040
0000 0000 0000 0000 2000
0000 3000 2000 1000 1302

start cycle
0000 0000 0000 0000 0001
0000 0000 0000 0000 1000
0000 0000 0000 0000 3000
0000 0000 0000 0000 2040
0000 3000 2000 1000 1002

start cycle
0000 C000 0000 0000 0001
0000 0000 0000 3000 1000
0000 0000 0000 0000 3000
0000 0000 0000 3000 2000
0000 3000 2000 1000 1042

start cycle
0000 0000 0000 0000 0001
0000 0000 0000 0000 1000
0000 0000 0000 0000 3000
0000 0000 0000 0000 2000
0000 3000 2000 1000 1002

start cycle
0000 0000 0000 9000 0001
0000 0000 0000 0000 1000
0000 0000 0000 3000 3000
0000 0000 0000 0000 2030
0000 3000 2000 1002 1002

start cycle
0000 0000 0000 0000 0001
0000 0000 0000 0000 1000
0000 0000 0000 0000 3000
0000 0000 0000 4000 2000
0000 3000 2002 1000 1002

start cycle
0000 0000 0000 0000 0001
0000 0000 0000 0000 1060
0000 0000 0000 4000 3000
0000 0000 4000 0000 2000
0000 3002 2000 1000 1002

start cycle
0000 0000 0000 0000 0001
0000 0000 0000 4000 1000
0000 0000 4000 3000 3000
0000 4000 0000 0000 2000
0002 3000 2000 1000 1002

start cycle
0000 0000 0000 4000 0001
0000 0000 4000 0000 1000
0000 4000 0000 3000 3000
2200 0000 0000 0000 2000
0000 3000 2000 1000 1002

start cycle
0000 0000 4000 0000 0001
0000 4000 0000 0000 1000
1200 0000 0000 3000 3000
2000 0000 0000 0000 2000
0000 3000 2000 1000 1002

start cycle
2000 0000 0000 0000 0001
1000 0000 0000 0000 1000
1000 0000 0000 0000 3000
2000 0000 0000 0000 2000
0000 3000 2000 1000 1002

start cycle
0000 0300 0000 0000 0001
1030 0000 0000 0000 1000
1000 0000 0000 0000 3000
2000 0000 0000 0000 2000
0000 3000 2000 1000 1002

start cycle
0000 0100 0300 0000 3001
1000 0330 0000 0000 1000
1030 0000 0000 0000 3000
2000 0000 0000 0000 2000
0000 3000 2000 1000 1002

start cycle
0000 0000 0100 0300 0001
1000 0010 0330 0000 1000
1000 0330 0000 0000 3000
2030 0000 0000 0000 2000
0000 3000 2000 1000 1002

start cycle
0000 0000 0000 0100 0301
1000 0000 0010 0330 1000
1000 0010 0330 0000 3000
2000 0330 0000 0000 2000
0030 3000 2000 1000 1002

start cycle
0000 0000 0000 0000 0101
1000 0000 0000 0010 1300
1000 0000 0010 0330 3000
2000 0010 0330 0000 2000
0000 3030 2000 1000 1002

start cycle
0000 0000 0000 0000 0001
1000 0000 0000 0000 1010
1000 0000 0000 0010 3300
2000 0000 0010 0330 2000
0000 3010 2030 1000 1002

start cycle
0000 0000 0000 0000 0001
1000 0000 0000 0000 1000
1000 0000 0000 0000 3010
2000 0000 0000 0010 2300
0000 3000 2010 1030 1002

start cycle
0000 0000 0000 0000 0001
1000 0000 0000 0000 1000
1000 0000 0000 0000 3000
2000 0000 0000 0000 2010
0000 2000 3000 1010 1002

start cycle
0000 0000 0000 0000 0001
1000 0000 0000 0000 1000
1000 0000 0000 0000 3000
2000 1000 2000 0000 2000
0000 2000 1000 3000 1012

start cycle
0000 0000 0000 0000 0001
1000 0000 0000 0000 1000
1000 1000 2000 0000 3000
2000 0000 1000 2000 2000
0000 2000 1000 3001 3002

start cycle
0000 0000 0000 0000 0001
1000 1000 2000 0000 1000
1000 0000 1000 2000 3000
2000 0000 0000 0000 2000
0000 2000 1001 3000 3002

start cycle
0000 1000 2000 0000 0001
1000 0000 1000 2000 1000
1000 0000 0000 0000 3000
2000 0000 0000 0000 2000
0000 2001 1000 3000 3002

start cycle
0000 0000 1000 2000 0001
1000 0000 0000 0000 1000
1000 0000 0000 0000 3000
2000 0000 0000 0000 2000
0001 2000 1000 3000 3002

start cycle
0000 0000 0000 0000 0001
1000 0000 0000 0000 1000
1000 0000 0000 0000 3000
2100 0000 0000 0000 2000
0000 2000 1000 3000 3002

start cycle
0000 0000 0000 0000 0001
1000 0000 0000 0000 1000
1100 0000 0000 0000 3000
2000 0100 0000 0000 2000
0000 2000 1000 3000 3002

start cycle
0000 0000 0000 0000 0001
1100 0000 0000 0000 1000
1000 0100 0000 0000 3000
2000 0000 0100 0000 2000
0000 2000 1000 3000 3002

start cycle
1000 0000 0000 0000 0001
1000 0100 0000 0000 1000
1000 0000 0100 0000 3000
2000 0000 0000 0100 2000
0000 2000 1000 3000 3002

start cycle
0000 0200 0000 0000 0001
1000 0000 0100 0000 1000
1000 0000 0000 0100 3000
2000 0000 0000 0000 2100
0000 2000 1000 3000 3002

start cycle
0000 0000 0200 0000 0001
1000 0000 0000 0100 1000
1000 0000 0000 0000 3100
2000 0000 0000 0000 2000
0000 2000 1000 3000 3002

start cycle
0000 0000 0000 0200 0001
1000 0000 0000 0000 1100
1000 0000 0000 0000 2000
2000 0000 0000 0100 3000
0000 2000 1000 3000 3002

start cycle
0000 0000 0000 0000 0201
1000 0000 0000 0000 1000
1000 0000 0000 0002 2000
2000 0000 0000 0001 3000
0000 2000 1000 3000 3003

start cycle
0000 0000 0000 0100 0001
1000 0000 0000 0000 1020
1000 0000 0002 0100 2000
2000 0000 0001 0000 3000
0000 2000 1000 3000 3003

start cycle
0000 0000 0000 0000 0001
1000 0000 0000 0000 1000
1000 0002 0000 0000 2020
2000 0001 0000 0000 3000
0000 2000 1000 3000 3003

start cycle
0000 0000 0000 0000 0001
1000 0000 0000 0000 1000
1002 0000 0000 0000 2000
2001 0000 0000 0000 3020
0000 2000 1000 3000 3003

start cycle
0000 0000 0000 0000 0001
1000 0000 0000 0000 1000
1000 0000 0000 0000 2000
2000 0000 0000 0000 3000
0000 2000 1000 3000 3023

start cycle
0000 0000 0000 0000 0001
1000 0000 0000 0000 1000
1000 0000 0000 0000 2000
2000 0000 0000 0000 3000
0000 2000 1000 3000 3003

start cycle
0000 0000 0000 0000 0001
1000 0000 0000 0000 1000
1000 0000 0000 0000 2000
2000 0000 0000 0000 3000
0000 2000 1000 3003 3003

start cycle
0000 0000 0000 0000 0001
1000 0000 0000 0000 1000
1000 0000 0000 0000 2000
2000 0000 0000 0000 3000
0000 2000 1003 3000 3003

start cycle
0000 0000 0000 0000 0001
1000 0000 0000 0000 1000
1000 0000 0000 0000 2000
2000 0000 0000 3000 3000
0000 2000 1000 3000 3003

start cycle
0000 0000 0000 0000 0001
1000 0000 0000 0000 1000
1000 0000 0000 3000 2000
2000 0000 0000 0000 3204
0000 2003 1000 3000 3003

start cycle
0000 0000 0000 0000 0001
1000 0000 0000 3000 1000
1000 0000 0000 0000 2404
2000 0000 0000 0000 3000
0000 2000 1000 3000 3003

start cycle
0000 0000 0000 3000 0001
1000 0000 0000 0000 1404
1000 0000 0000 0000 2000
2000 0000 0000 0000 3000
0000 2300 1000 3000 3003

start cycle
0000 0000 0000 0000 4001
1000 0000 0000 0000 1000
1000 0000 0000 0000 2000
2000 0000 0000 0100 3000
0000 2000 1300 3000 3003

start cycle
0000 0000 0000 0000 0001
1000 0000 0000 0000 1040
1000 0000 0000 0000 2000
2000 0000 0000 0000 3000
0000 2000 1000 3300 3003

start cycle
0000 0000 0000 0000 0001
1000 0000 0000 0000 1000
1000 0000 0000 0000 2040
2000 0000 0000 0100 3000
0000 2000 1000 3003 3303

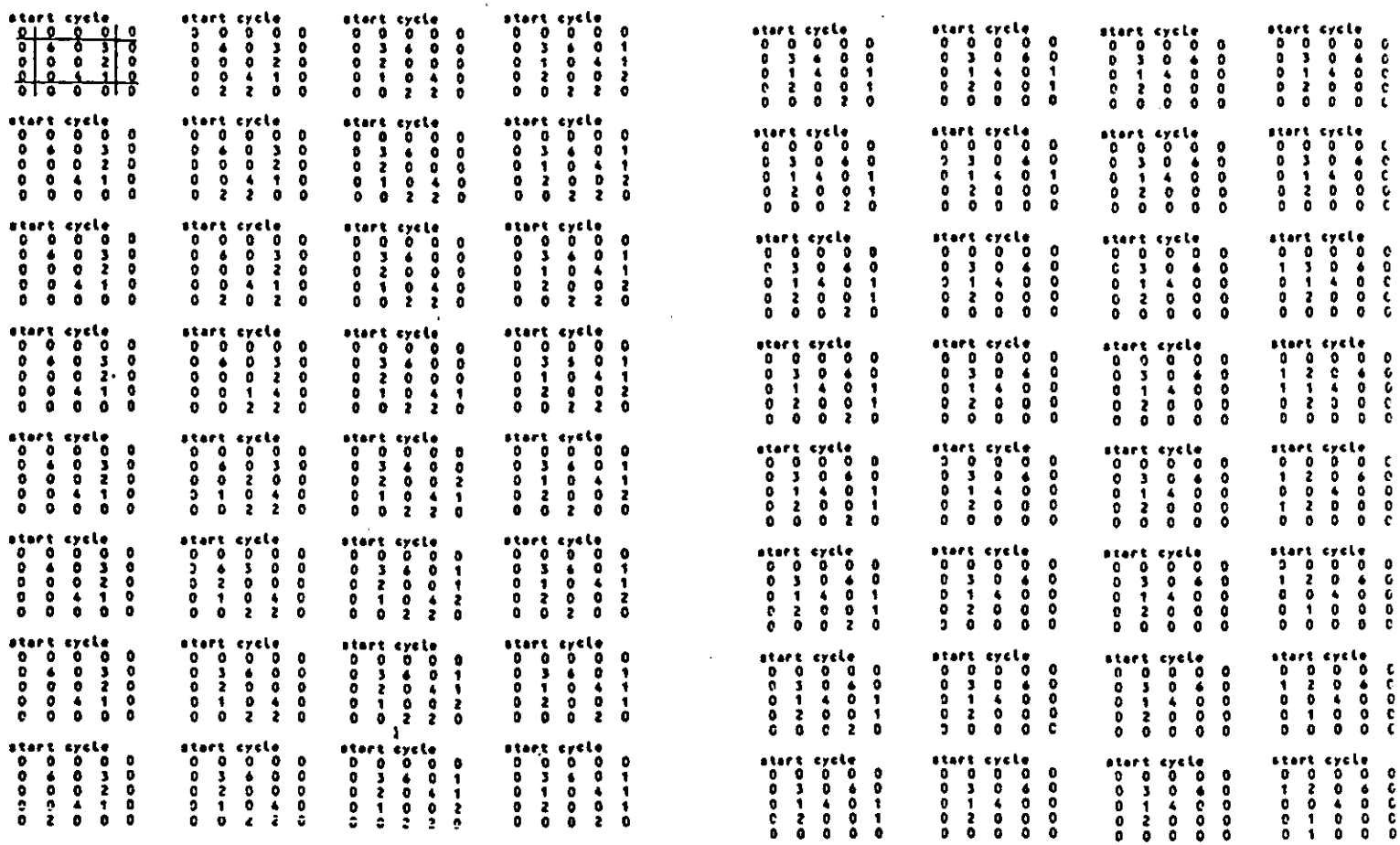
start cycle
0000 0000 0000 0000 0001
1000 0000 0000 0000 1300
1000 0000 0000 0000 2000
2000 0000 0000 0000 3040
0000 2000 1000 3000 3003

start cycle
0000 0000 0000 0000 0001
1000 0000 0000 0000 1000
1000 0000 0000 0000 2000
2000 0000 0000 0000 3043
0000 2000 1000 3000 3003

start cycle
0000 0000 0000 0100 0001
1000 0000 0000 0000 1000
1000 0000 0000 0000 2000
2000 0000 0000 0000 3000
0000 2000 1000 3000 3003

start cycle
0000 0000 0000 0100 1001
1000 0000 0000 0000 1000
1000 0000 0000 0000 2000
2000 0000 0000 0000 3006
0000 2000 1000 3006 3003

Table Output Snapshots



start cycle

0	0	0	0	0
1	2	0	6	0
0	0	4	0	0
0	1	0	0	0
0	1	0	0	0

start cycle

0	0	0	0	0
1	2	0	6	0
0	0	4	0	0
0	1	0	0	0
0	0	1	0	0

start cycle

0	0	0	0	0
1	2	0	6	0
0	0	4	0	0
0	0	1	0	0
0	0	0	1	0

start cycle

0	0	0	0	0
1	2	0	6	0
0	6	0	0	0
0	0	0	1	0
0	0	0	1	0

start cycle

0	0	0	0	0
1	0	2	6	0
0	4	0	0	0
0	0	0	1	0
0	0	0	1	0

start cycle

0	0	0	0	0
1	0	6	2	0
0	4	0	0	0
0	0	0	1	0
0	0	0	1	0

start cycle

0	0	0	0	0
1	0	6	2	0
0	4	0	0	0
0	0	0	1	0
0	0	0	1	0

start cycle

0	0	0	0	0
1	0	6	2	0
0	4	0	0	0
0	0	0	1	0
0	0	0	1	0

start cycle

0	0	0	0	0
1	0	6	2	0
0	4	0	0	0
0	0	0	1	0
0	0	0	1	0

start cycle

0	0	0	0	0
1	0	6	2	0
0	4	0	0	0
0	0	0	1	0
0	0	0	1	0

start cycle

0	0	0	0	0
1	0	6	2	0
0	4	0	0	0
0	0	0	1	0
0	0	0	1	0

start cycle

0	0	0	0	0
1	0	6	2	0
0	4	0	0	0
0	0	0	1	0
0	0	0	1	0

start cycle

0	0	0	0	0
1	0	6	2	0
0	4	0	0	1
0	0	0	1	0
0	0	0	1	0

start cycle

0	0	0	0	0
1	0	6	2	0
0	4	0	0	0
0	0	0	1	1
0	0	0	1	0

start cycle

0	0	0	0	0
1	0	6	2	0
0	4	0	1	0
0	0	0	0	1
0	0	0	1	0

start cycle

0	0	0	0	0
1	0	6	2	0
0	4	0	1	0
0	0	0	0	1
0	0	0	1	0

start cycle

0	0	0	0	0
1	0	6	2	0
0	0	0	1	0
0	4	0	0	1
0	0	0	1	0

start cycle

0	0	0	0	0
1	0	6	2	0
0	0	0	1	0
0	4	0	0	1
0	0	0	1	0

start cycle

0	0	0	0	0
1	0	6	2	0
0	0	0	1	0
0	4	0	0	1
0	0	0	1	0

start cycle

0	0	0	0	0
1	0	6	2	0
0	0	0	1	0
0	4	0	0	1
0	0	0	1	0

start cycle

0	0	0	0	0
1	0	6	2	0
0	0	0	1	0
0	4	0	0	1
0	0	0	0	0

start cycle

0	0	0	0	0
1	0	6	2	0
0	0	0	1	0
0	4	0	0	1
0	0	0	0	0

start cycle

0	0	0	0	0
1	0	6	2	0
0	0	0	1	0
0	4	0	0	0
0	0	0	0	0

start cycle

0	0	0	0	0
1	0	6	2	0
0	0	0	1	0
0	4	0	0	0
0	0	0	0	0

start cycle

0	0	0	0	0
1	0	6	2	0
0	0	0	1	0
0	4	0	0	0
0	0	0	0	0

start cycle

0	0	0	0	0
1	0	6	2	0
0	0	0	1	0
0	4	0	0	0
0	0	0	0	0

start cycle

0	0	0	0	0
1	0	6	2	0
0	0	0	1	0
0	4	0	0	0
0	0	0	0	0

start cycle

0	0	0	0	0
1	0	6	2	0
0	0	0	1	0
0	4	0	0	0
0	0	0	0	0

start cycle

0	0	0	0	0
1	0	6	2	0
0	0	0	1	0
0	4	0	0	0
0	0	0	0	0

start cycle

0	0	0	0	0
1	0	6	2	0
0	0	0	1	0
0	4	0	0	0
0	0	0	0	0

start cycle

0	0	0	0	0
1	0	6	2	0
0	0	0	1	0
0	4	0	0	0
0	0	0	0	0

start cycle

0	0	0	0	0
1	0	6	2	0
0	0	0	1	0
0	4	0	0	0
0	0	0	0	0

given with c the cell latency, and an area efficient systolic ring constructed with $\lceil n/c \rceil$ basic cells implementing the extrapolation procedure. Various simplifications on the generic structure were discussed indicating that arrays could be optimized depending upon the special structure of particular problems and the amount of table data to be generated (i.e. full table, diagonal entries only or final improved result). Finally we introduced the idea of the adaptive table generating array which could predict the convergence rate, and hence minimise the number of starting values evaluated to achieve convergence. This led to the problem of generating a table larger than the array could accommodate in a single pass, and multipass table generation was briefly considered to produce a fixed size array more suited to VLSI construction.

From these experiments it was clear that extrapolation table generation with its recurrent form of table construction using simple rules for relating table elements fitted many other table algorithms. We investigated a method of array templating for the fast derivation of systolic arrays for computationally related table algorithms (i.e. differencing techniques). The concept of array unification was introduced to combine similar systolic arrays by combining cell functions to produce a minimal hardware arrangement. Two types of templates were discovered indicating that the earlier work was a special case of a general template. In particular, we showed that equally spaced arguments produced tables which allowed the arguments to be inferred by the cell function. Whereas, unequally spaced arguments require arguments to be explicitly represented in the cell, restricting the method of computation (and preventing systolic rings). Equally

spaced arguments produced a column-wise table generation, while unequally spaced arguments generated a table row wise. Although the problems considered were computationally simple the implications are far-reaching. A number of algorithms implemented on the same cell architecture will produce a very cost effective VLSI design. Indeed recent trends in systolic array development and particularly the CMU WARP processor (H.T. Kung [84a]) are aimed at more flexible array desing. The Unified Systolic Array for Differencing (USAD) offers an interesting alternative for fast computation of approximating functions.

Next we turned our attention to the problem of generating open ended or potentially infinite tables, and examined the effect on table construction by rules based on triangular and rectangular tables. As a vehicle for discussion the Quotient-Difference (QD) algorithm for producing all the roots of a polynomial was used. Two designs were produced, one where polynomial roots remained fixed in cells (or stationary) and a second where root approximations moved systolically (i.e. nonstationary). The former scheme requiring n QD-rule cells for a polynomial of degree n. The latter with cells proportional to the number of root approximations, forming a natural multipass array, and by extension a systolic ring for generating an infinite sequence of approximations.

Finally, systolic arrays for the more complex table based Simplex and assignment problems were developed. This time the emphasis was on table manipulation rather than construction resulting in larger and more complex arrays. The resulting arrays used a combination of wave-front and systolic control movements. For the standard Simplex method the time of the array was bounded by,

$$T = z(2n+4m+k)+2m$$

ips cycles for n unknowns, m constraints, and the examination of z feasible points, where,

$$z = \begin{cases} \text{approx. } m \text{ for ordinary Simplex} \\ \text{approx. } 2m \text{ for Simplex with artificial basis} \end{cases}$$

and k=6 a small constant. The number of cells was given by,

$$A = \begin{cases} (m+2)(m+n)+3m & \text{ordinary} \\ (m+2)(m+n)+3m+(m+1) & \text{artificial basis.} \end{cases}$$

The arrays reduce the computation by an order of magnitude when compared with the sequential algorithm requiring $O(zm(n+m))$. In contrast to the above technique a second array computing the revised Simplex method was considered using the general form of the inverse technique. For $m<n$ a systolic cylinder with a volume efficient layout resulted and for $m\geq n$ a compacted planar layout. These revised arrays required,

$$T = z(4m+12) + O(2m)$$

ips cycles and the compacted array required only $(m+2)^2+3m+4$ cells. These results should be compared with the least squares array of Gentleman & H.T. Kung [81] which requires $T=O(6n)$ and $O(n^2)$ cells and is applicable for $m>n$, but requires the construction of normal equations (omitted from the time) and the existence of $(A^TA)^{-1}$. Clearly the new arrays require more area and time but are more general extending easily to solve the Chebyshev (min-max) problem for over-determined systems.

Last but not least we considered the assignment problem, a special case of the more general transportation (LP) problem, which reduces to an integer programming problem and requires a square ($n\times n$) matrix

representation. In particular, we implemented the Hungarian algorithm which allowed a reduction of cell complexity by using only integer add/subtract operations. An orthogonally connected (n+2)*(n+2) wave-front mesh incorporating a Systolic Control Ring (SCR) for generating wavefronts and restricting special cells to the periphery of the array was produced which required,

$$2n^2+(16+k)n + 8 \leq T \leq 10n^2 + (16+k)n$$

cycles, where a cycle is the cost of add/subtract and control switching time, rather than an inner product step.

We conclude that for table generating methods the number of cells is proportional to the tables smallest dimension, and that often designs can be optimised (depending on the table construction rule) to produce multipass architectures with size independent of the problem size. Thus together, with the ideas of templating and unification table generating systolic arrays provide the possibility of cheap add-on devices to bring parallelism to a sequential machine or off-load computation of a parallel host in the form of chip-table generators, akin to the standard mathematical tables used heavily before widespread numeric computation.

For the more complex table manipulating problems, the size of arrays is proportional to the number of table elements which are held or stored throughout the computation. It is acknowledged that the Simplex problems can be extremely large making the usefulness of a hybrid or hard systolic realisation of these schemes questionable. For Simplex problems the idea of decoupling can be applied to produce a series of smaller problems which would limit the size of the array, and permit a type of iterative/multipass solution to the whole problem.

Unfortunately decoupling occurs only in limited cases and the decoupled subproblems may still be large. The assignment problem by virtue of its simple cell structure and control indicates some possibilities in Wafer Scale Integration (WSI) schemes, where a wafer API (WAPI) as shown in Fig.(7.8.1) might produce a large array.

Thus table generating arrays are suited to a hybrid, hard, systolic approach, while table manipulating schemes remain (for the present) purely soft-systolic in nature.

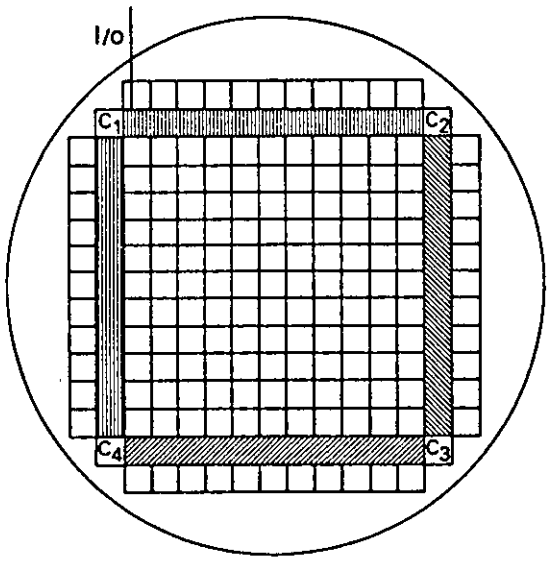


FIGURE 7.8.1: Mapping of API onto wafer

CHAPTER 8

THE SOLUTION OF CERTAIN PARTIAL DIFFERENTIAL EQUATIONS (PDE'S) BY SYSTOLIC MARCHING TECHNIQUES

"Holism or reductionism for parallelism?"

Definition: Hol-ism (philosophy)
Tendency in nature to form wholes that are more than the sum of the parts by ordered groupings.

Definition: Reductionism
Analysis of complex things into simple constituents; the view that a system can be fully understood in terms of its isolated parts.

In this chapter we extend the ideas of table generation, manipulation and array unification principles to derive geometric rather than algorithmic interpretations of finite difference solutions to P.D.E.'s.

In particular, we consider the solution of 1-D (heat conduction) and 2-D (unsteady diffusion) parabolic P.D.E.'s by the asymmetric approximations of Saul'ev [64] and the Group Explicit (GE) techniques of D.J. Evans and Abdullah [83a,b] to derive a Linear Asymmetric Marching Processor (LAMP) array and a Unified Group Explicit Parabolic Solver (UGEPS).

Array compaction techniques are employed in the form of Hopscotch methods (Gourlay [70]) to provide area-efficient alternatives when portions of the solution 'table' or region can be omitted, and fast arrays where computational rules used to derive basic cells are optimised.

Finally, we briefly consider the extension of the schemes to a group explicit technique for the solution of a hyperbolic equation of first order (Sahimi [86]).

8.1 INTRODUCTION TO ASYMMETRIC AND GROUP EXPLICIT METHODS

Before discussing the construction of systolic arrays implementing the numerical solution of parabolic P.D.E.'s we briefly review the finite difference techniques involved with particular attention focussed on the following two problems. Firstly, the 1-D parabolic equation for the simple heat conduction problem,

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} \quad , \quad 0 \leq x \leq 1, \; t \geq 0 \quad , \tag{8.1.1}$$

and secondly, the 2-D unsteady diffusion equation of the form,

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \quad , \quad 0 \leq y \leq 1, \quad 0 \leq x \leq 1 \quad . \tag{8.1.2}$$

The first solution technique considered is the use of stable asymmetric explicit finite difference equations of Saul'ev [64]. It is well known that generally an explicit type of method results in the simplest computational procedures. Since simplicity is usually related to computational cost and from a systolic point of view, to simple basic cells and low cycle time it is desirable to retain an explicit type of approximation procedure. The Saul'ev formulae are chosen in preference to others (like the classical implicit and explicit schemes) because of their attractive pipelining features and stability characteristics.

Equation (8.1.1) is one of the most frequently occurring parabolic equations and can be approximated by the two-time level finite difference approximation below. The range of the variable x is divided into equal mesh points $0=x_0<x_1<\ldots<x_{m-1}<x_m$ with step size $h=1/m$ and $x_i=ih$, $i=0(1)m$. Similarly the range $[0,t_z]$ of the variable t is divided as $0=t_0<t_1<\ldots t_{k-1}<t_k=t_z$ in z equal parts, giving $\ell=t_z/z$ and $t_k=k\ell$, $k=0(1)z$. Finally we introduce a set of Dirichlet boundary conditions,

$$\left.\begin{aligned} &\text{a)}\quad u(0,t) = u(1,t) = 0 \ , \quad 0\leq t\leq t_z \\ &\text{and initial values,} \\ &\text{b)}\quad u(x,0) = f(x) \qquad , \quad 0<x<1 \ . \end{aligned}\right\} \tag{8.1.3}$$

Next, the approximations for the partial derivatives are chosen as follows,

$$\frac{\partial u_{i,k}}{\partial t} \simeq \frac{u_{i,k+1}-u_{ik}}{\ell} + O(h) \ , \tag{8.1.4}$$

and,

$$\alpha \frac{\partial^2 u_{i,k}}{\partial x^2} \simeq \frac{\alpha}{h}\left(\frac{\partial u_{i+\frac{1}{2},k}}{\partial x} - \frac{\partial u_{i-\frac{1}{2},k}}{\partial x}\right) + O(h^2) \tag{8.1.5}$$

with,

$$\frac{\partial u_{i-\frac{1}{2},k}}{\partial x} \simeq \frac{\partial u_{i-\frac{1}{2},k+1}}{\partial x} - \ell \frac{\partial^2 u(x_{i-\frac{1}{2}}, t_k+\theta\ell)}{\partial x \partial t} \ , \quad 0\leq\theta\leq 1$$

using the Mean Value Theorem.

By substitution into (8.1.1) yields,

$$\frac{u_{i,k+1}-u_{i,k}}{\ell} \approx \frac{\alpha}{h}\left(\frac{\partial u_{i+\frac{1}{2},k}}{\partial k} - \frac{\partial u_{i-\frac{1}{2},k+1}}{\partial x}\right) + \frac{(1-\alpha)}{h}\left(\frac{\partial u_{i+\frac{1}{2},k}}{\partial x} - \frac{\partial u_{i-\frac{1}{2},k}}{\partial x}\right) + \frac{\alpha\ell}{h}\frac{\partial^2(x_{i-\frac{1}{2}}, t_k+\theta\ell)}{\partial x \partial t} + O(\ell+h^2) \quad . \tag{8.1.6}$$

Thus, we can write (8.1.6) as,

$$\frac{u_{i,k+1}-u_{i,k}}{\ell} = \frac{\alpha}{h^2}(u_{i-1,k+1}-u_{i,k+1}-u_{i,k}+u_{i+1,k}) + \frac{(1-\alpha)}{h^2}(u_{i-1,k}-2u_{i,k}+u_{i+1,k})+R_{i,k} \quad , \tag{8.1.7}$$

where $R_{i,k}=O(\alpha/h + h^2+\ell)$ is the error of approximation.

We obtain the final equation with multiplication by h^2 and neglecting the error term $h^2R_{i,k}$, if h is chosen to be small, as,

$$u_{i,k+1} = \frac{1}{w+\alpha}[\alpha u_{i-1,k+1}+(1-\alpha)u_{i-1,k}+u_{i+1,k}-(2-w-\alpha)u_{i,k}] \tag{8.1.8}$$

with $w=\frac{h^2}{\ell}$, $0\leq\alpha\leq1$.

This formula leads to a number of asymmetric formulae attributed to Saul'ev [64]. We are particularly interested in the case when $\alpha=1$, which produces the equation,

$$u_{i,k+1} = \frac{1}{w+1}[u_{i-1,k+1}+u_{i+1,k}-(1-w)u_{i,k}] \tag{8.1.9a}$$

and a related equation yields,

$$u_{i,k+1} = \frac{1}{1+w}[u_{i+1,k+1}+u_{i-1,k}-(1-w)u_{i,k}] \tag{8.1.9b}$$

denoting $r=\ell/h^2$ and $w=1/r$ we obtain,

$$u_{i,k+1} = \frac{1}{1+r}[ru_{i-1,k+1}+ru_{i+1,k}+(1-r)u_{i,k}] \tag{8.1.10a}$$

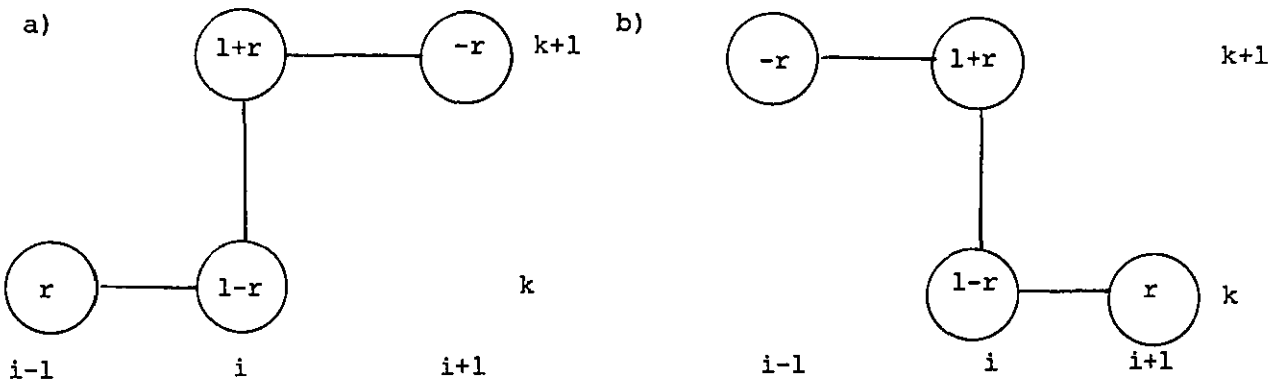
$$u_{i,k+1} = \frac{1}{1+r}[ru_{i+1,k+1}+ru_{i-1,k}+(1-r)u_{i,k}] \tag{8.1.10b}$$

and when r=1, the simple equations,

$$u_{i,k+1} = \tfrac{1}{2}[u_{i-1,k+1}+u_{i+1,k}] \qquad (8.1.11a)$$

$$u_{i,k+1} = \tfrac{1}{2}[u_{i+1,k+1}+u_{i-1,k}] \ . \qquad (8.1.11b)$$

Alternatively, (8.1.10) in their implicit form represent the computational molecules.



(8.1.12)

and the solution to (8.1.1) is obtained by applying these molecules to an infinite rectangular grid similar to Fig.(2.1a) with boundary initial conditions described by (8.1.3). Solutions can be obtained in a number of ways:

(i) Use (8.1.10a) (or (8.1.12b)) only proceeding level by level using a Left to Right (LR) movement on each level along the x-direction in which case the formula becomes explicit.

(ii) Use (8.1.10b) (or (8.1.12a)) in the same way as (i) but move from Right to Left (RL) on each level and again the formula is explicit.

(iii) Alternate between LR and RL on successive levels.

(iv) Perform LR and RL (or RL then LR) on the same time level producing two results for each mesh point on the level. Take the average of the estimates to produce a more accurate result due to the cancellation of truncation error terms emanating from opposite signs.

These alternative strategies illustrate the concept of marching, where the computational molecules representing the formulae on the grid can be considered as marching from point to point along the x-direction gradually moving up the time levels. From a numerical viewpoint the marching can be formulated as a triangular linear system by numbering the points on level k+1 from left to right, or vice versa to yield,

$$Au^{(k+1)} = (A+C)u^{(k)} , \tag{8.1.13a}$$

and,

$$A^T u^{(k+1)} = (A^T+C)u^{(k)} , \tag{8.1.13b}$$

where,

$$A = \begin{bmatrix} w+\alpha & & & \\ -\alpha & w+\alpha & & \\ & \ddots & \ddots & \\ & & -\alpha & w+\alpha \end{bmatrix} , \quad C = \begin{bmatrix} -2 & 1 & & \\ 1 & -2 & & \\ & \ddots & \ddots & 1 \\ & & 1 & -2 \end{bmatrix}$$

Next consider the unsteady diffusion equation (or 2-D heat conduction problem) (8.1.2), the region to be considered is a rectangular domain like Fig. (2.1b) with boundary and initial conditions,

$$u(0,y,t) = f_1(y,t), \quad u(x,0,t) = f_3(x,t)$$

$$u(1,y,t) = f_2(y,t), \quad u(x,1,t) = f_4(x,t)$$

and $$u(x,y,0) = f_5(x,y) . \tag{8.1.14}$$

Asymmetric equations can be described in a similar way to the 1-D case. Let $u_{i,j,k} \equiv u(ih,jh,k\ell)$ so that the region is square for simplicity, the analogous equation to (8.1.7) is,

$$\begin{aligned} \frac{u_{i,j,k+1}-u_{i,j,k}}{\ell} &= \frac{\alpha}{h^2}(u_{i-1,j,k+1}-u_{i,j,k+1}-u_{i,j,k}+u_{i+1,j,k}) \\ &+ \frac{(1-\alpha)}{h^2}(u_{i-1,j,k}-2u_{i,j,k}+u_{i+1,j,k}) \\ &+ \frac{\beta}{h^2}(u_{i,j+1,k+1}-u_{i,j,k+1}-u_{i,j,k}+u_{i,j-1,k}) \end{aligned}$$

$$+\frac{(1-\beta)}{h^2}(u_{i,j+1,k}-2u_{i,j,k}+u_{i,j-1,k})+R_{i,j,k}\,, \qquad (8.1.15)$$

where $R_{i,j,k}=O(\alpha h+\beta h+\ell+h^2)$, $0\leq\alpha,\beta\leq 1$.

By neglecting the $R_{i,j,k}$ term with some manipulation the formula becomes,

$$u_{i,j,k+1} = \frac{1}{w+\alpha+\beta}\,[\alpha u_{i-1,j,k+1}+\beta u_{i,j+1,k+1}+(1-\alpha)u_{i-1,j,k}+$$

$$(1-\beta)u_{i,j+1,k}-(4-w-\alpha-\beta)u_{i,j,k}+u_{i+1,j,k}+u_{i,j-1,k} \qquad (8.1.16)$$

and with $\alpha=\beta=1$

$$u_{i,j,k+1} = \frac{1}{w+2}[u_{i-1,j,k+1}+u_{i,j+1,k+1}-(2-w)u_{i,j,k}+u_{i+1,j,k}+$$

$$u_{i,j-1,k}] \qquad (8.1.17)$$

and with $\frac{1}{w}=r$, we have,

$$u_{i,j,k+1} = \frac{1}{1+2r}[ru_{i-1,j,k+1}+ru_{i,j+1,k+1}+(1-2r)u_{i,j,k}+ru_{i+1,j,k}$$

$$+ru_{i,j-1,k}] \qquad (8.1.18a)$$

Likewise, the following similar equations can be deduced,

$$u_{i,j,k+1} = \frac{1}{1+2r}[ru_{i+1,j,k+1}+ru_{i,j+1,k+1}+(1-2r)u_{i,j,k}+ru_{i-1,j,k}$$

$$+ru_{i,j-1,k}] \qquad (8.1.18b)$$

$$u_{i,j,k+1} = \frac{1}{1+2r}[ru_{i-1,j,k+1}+ru_{i,j-1,k+1}+(1-2r)u_{i,j,k}+ru_{i+1,j,k}$$

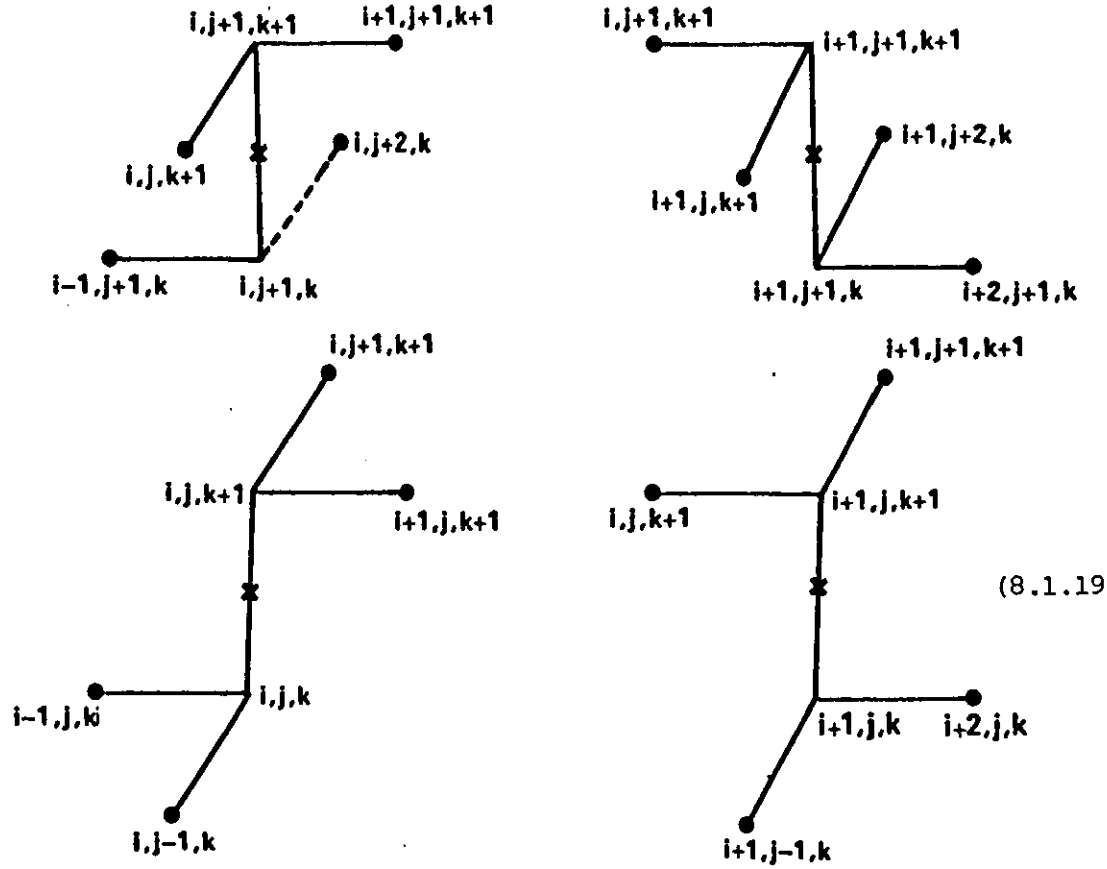
$$+ru_{i,j+1,k}] \qquad (8.1.18c)$$

and,

$$u_{i,j,k+1} = \frac{1}{1+2r}[ru_{i+1,j,k+1}+ru_{i,j-1,k+1}+(1-2r)u_{i,j,k}+ru_{i-1,j,k}$$

$$+ru_{i,j+1,k}] \qquad (8.1.18d)$$

Like the 1-D problem the implicit forms of (8.1.18) can be represented by computational molecules, viz.



(8.1.19)

and the solution to equation (8.1.2) under the constraints (8.1.14) can be achieved by a variety of techniques:

(i) Use equation (8.1.18a) starting with (i=1,j=m-1,k=0) moving from left to right (LR).

(ii) Use equation (8.1.18b) starting at (i=m-1,j=m-1,k=0) moving right to left (RL).

(iii) Use equation (8.1.18c) starting at (i=1, j=1,k=0) moving left to right (LR).

(iv) Use equation (8.1.18d) starting at (i=m-1,j=1, k=0) moving right to left (RL).

(v) Alternating schemes are given by:

(a) Using (i)-(iv) on successive cycles starting again every four cycles.

(b) Select two formulas e.g. (ii) and (iii) and alternate cycling every two time levels.

(vi) Averaging schemes: Use two schemes on the same level and average the results for each grid point to obtain improved results.

These are also marching type algorithms, and for a suitable ordering of mesh points we obtain the following linear systems,

$$Au^{(k+1)} = Bu^{(k)} , \tag{8.1.20a}$$

for (8.1.18a,b) and,

$$A^T u^{(k+1)} = B^T u^{(k)} , \tag{8.1.20b}$$

for (8.1.18c,d) where,

$$A = \begin{bmatrix} a & & & \\ -b & a^T & & O \\ & \ddots & \ddots & \\ O & & -b & a^T \end{bmatrix} , \quad a = \begin{bmatrix} \gamma & & & \\ -\alpha & \gamma & & O \\ & \ddots & \ddots & \\ O & & -\alpha & \gamma \end{bmatrix}$$

$$b = \begin{bmatrix} \beta & & \\ & O & \\ O & & \beta \end{bmatrix}$$

and

$$B = \begin{bmatrix} c & e & & \\ (e-b) & c^T & e & O \\ & \ddots & \ddots & \\ O & & (e-b) & c \end{bmatrix} , \quad c = \begin{bmatrix} (\gamma-4) & 1 & & \\ (1-\alpha) & (\gamma-4) & 1 & O \\ & \ddots & \ddots & \\ O & & (1-\alpha) & (\gamma-4) \end{bmatrix}$$

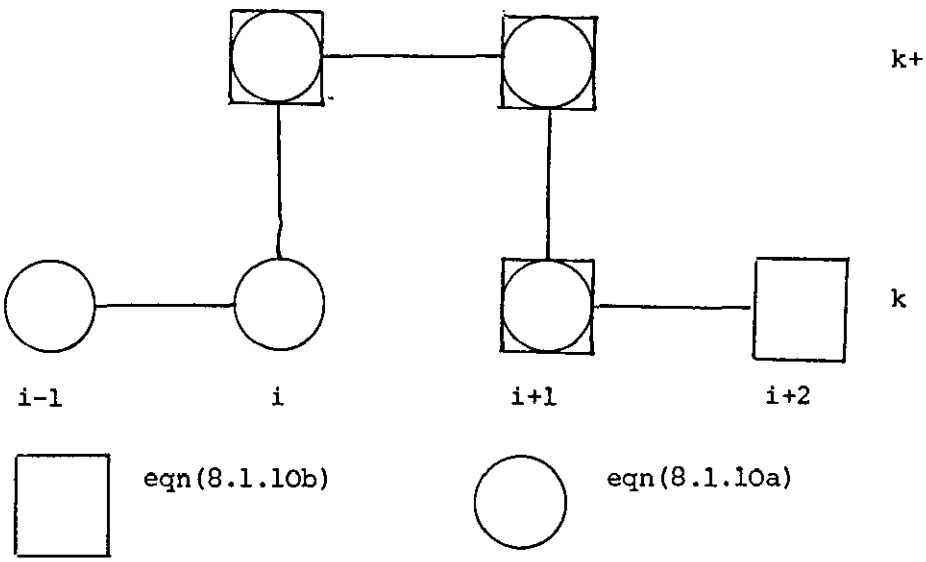
$$e = \begin{bmatrix} \overline{1} & & \\ & 1 & \\ & & \ddots \\ & & & \underline{1} \end{bmatrix} \qquad \gamma = w+\alpha+\beta$$

such that A and B are square matrices of order $(m-1)^2$ and a,c of order (m-1). Thus, the bandwidth of A and B are m and 2m-1 respectively for m internal mesh points over the region.

Now there are many formulations and molecules which can be adopted for the solution of P.D.E.'s which are divided into explicit and implicit techniques. Explicit methods tend to restrict the time step to small values in order to retain numerical stability, while implicit methods involve the solution of large linear systems of equations but allow greater stepsizes. A small stepsize $\Delta t=k$ requires a large number of time steps implying an enormous amount of computational work to reach level t_z and previously has produced a bias towards implicit rather than explicit methods. The asymmetric approximations of Saul'ev are semi-explicit because they produce an implicit method whose associated linear systems (8.1.13) are easily solvable (A is lower triangular). Recently Evans & Abdullah [83a,b] produced a new variation on the use of asymmetric equations and introduced the Group Explicit (GE) methods allowing a simple explicit method with no upper limit on the stepsize. Their work indicated that the method compared favourably with other numerical methods. We consider GE methods for our two sample problems (8.1.1) and (8.1.2).

For the 1-D heat conduction problem we combine the molecules of (8.1.12) by coupling them in groups of 2 adjacent points along the x-direction of the grid to produce implicit equations which are easily

converted to explicit form. The essential idea being to retain the accuracy and large stepsize of the implicit method with the computational simplicity of the explicit technique by utilising truncation error cancellations and alternating strategies on the gridpoints to produce unconditional stability. To achieve this goal we produce a hybrid or unified computational molecule of the form,



(8.1.21)

which can be represented by a 2x2 linear system as,

$$\begin{bmatrix} 1+r & -r \\ -r & 1+r \end{bmatrix} \begin{bmatrix} u_{i,k+1} \\ u_{i+1,k+1} \end{bmatrix} = \begin{bmatrix} 1-r & 0 \\ 0 & 1-r \end{bmatrix} \begin{bmatrix} u_{i,k} \\ u_{i+1,k} \end{bmatrix} + \begin{bmatrix} ru_{i-1,k} \\ ru_{i+2,k} \end{bmatrix} \quad (8.1.22)$$

or in explicit form,

$$\begin{bmatrix} u_{i,k+1} \\ u_{i-1,k+1} \end{bmatrix} = \frac{1}{(1+2r)} \begin{bmatrix} 1+r & r \\ r & 1+r \end{bmatrix} \left\{ \begin{bmatrix} 1-r & 0 \\ 0 & 1-r \end{bmatrix} \begin{bmatrix} u_{ik} \\ u_{i+1,k} \end{bmatrix} + \begin{bmatrix} ru_{i-1,k} \\ ru_{i+2,k} \end{bmatrix} \right\} \quad (8.1.23)$$

For the ungrouped (or single) points that could occur near the left or right boundary we use the asymmetric formula,

$$u_{m-1,k+1} = \frac{1}{(1+r)} [ru_{m,k+1}+ru_{m-2,k}+(1-r)u_{m-1,k}] \qquad (8.1.24)$$

for the right boundary and,

$$u_{1,k+1} = \frac{1}{(1+r)} [ru_{0,k+1}+ru_{2,k}+(1-r)u_{1,k}] \qquad (8.1.25)$$

for the left boundary, where we assume m equal intervals. When m is even the number of internal points is odd (i.e. (m-1)odd) and produces a single ungrouped point, and a variety of GE schemes result. First define,

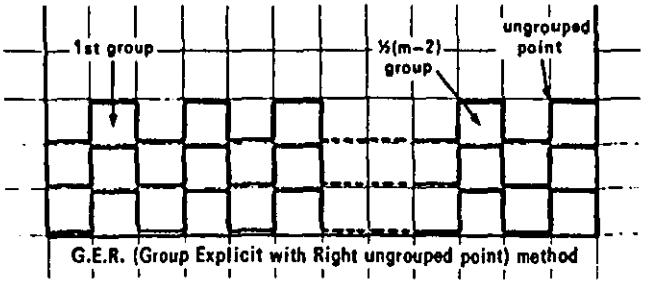
$$G^{(i)} = \begin{bmatrix} 1 & & -1 \\ & & \\ -1 & & 1 \end{bmatrix} , \quad i=1(1)\tfrac{1}{2}(m-2), \quad j=0,1,... \qquad (8.1.26)$$

and put,

$$G_1 = \begin{bmatrix} G^{(1)} & & & \\ & G^{(2)} & & \\ & & \ddots & \\ & & & G^{\frac{1}{2}(m-2)} \\ & & & & 1 \end{bmatrix} , \quad G_2 = \begin{bmatrix} 1 & & & \\ & G^{(1)} & & \\ & & \ddots & \\ & & & G^{\frac{1}{2}(m-2)} \end{bmatrix}$$

then,

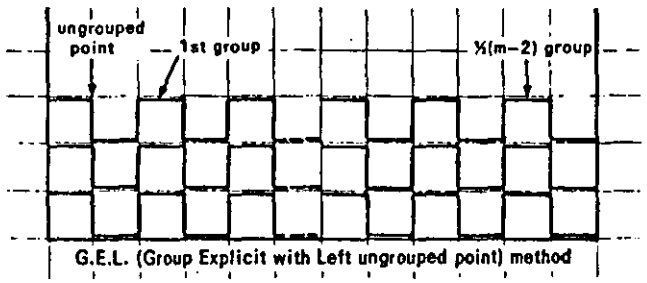
Even Number of Intervals



G.E.R. (Group Explicit with Right ungrouped point) method

$$(I+rG_1)u_{k+1} = (I-rG_2)u_k+b_1$$

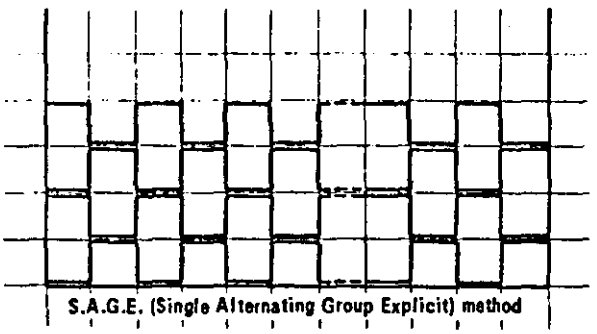
$$b_1^T = (ru_{0,k},0,\ldots,0,ru_{m,k+1}) \qquad (8.1.27)$$



G.E.L. (Group Explicit with Left ungrouped point) method

$$(I+rG_2)u_{k+1} = (I-rG_1)u_k+b_2$$

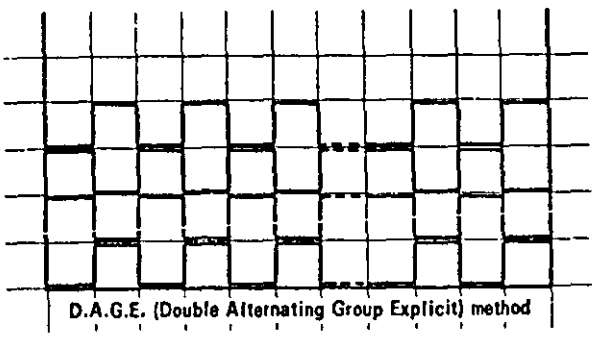
$$b_2^T = (ru_{0,k+1},0,\ldots,0,ru_{m,k}) \qquad (8.1.28)$$



S.A.G.E. (Single Alternating Group Explicit) method

$$(I+rG_1)u_{k+1} = (I-rG_2)u_k+b_1$$

$$(I+rG_2)u_{k+2} = (I-rG_1)u_{k+1}+b_2 \qquad (8.1.29)$$



D.A.G.E. (Double Alternating Group Explicit) method

$$(I+rG_1)u_{k+1} = (I-rG_2)u_k+b_1$$

$$(I+rG_2)u_{k+2} = (I-rG_1)u_{k+1}+b_2$$

$$(I+rG_2)u_{k+3} = (I-rG_1)u_{k+2}+b_2$$

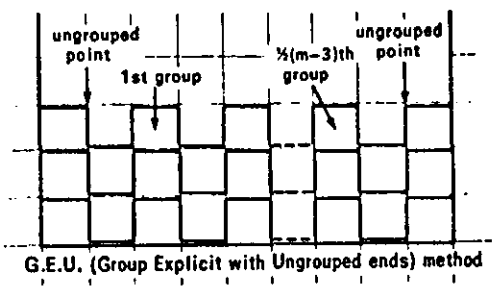
$$(I+rG_1)u_{k+4} = (I-rG_2)u_{k+3}+b_1 \qquad (8.1.30)$$

and for an odd number of intervals the number of internal points (i.e. m-1) is even so at each time level we have either $\frac{1}{2}(m-1)$ complete groups or $\frac{1}{2}(m-3)$ groups and two ungrouped points, one adjacent to each boundary. We then define,

$$\hat{G}_1 = \begin{bmatrix} 1 & & & & \\ & G^{(1)} & & & \\ & & G^{(2)} & & \\ & & & \ddots & \\ & & & & G^{\frac{1}{2}(m-3)} \\ & & & & & 1 \end{bmatrix}, \quad \hat{G}_2 = \begin{bmatrix} G^{(1)} & & & \\ & G^{(2)} & & \\ & & \ddots & \\ & & & G^{\frac{1}{2}(m-1)} \end{bmatrix}$$

and produce the following group explicit methods:

Odd Number of Intervals

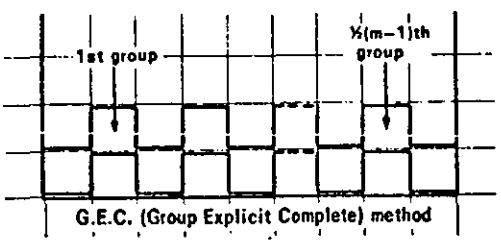


G.E.U. (Group Explicit with Ungrouped ends) method

$$(I+r\hat{G}_1)u_{k+1} = (I-r\hat{G}_2)u_k + b_3$$

$$b_3^T = (ru_{0,k+1}, 0, \ldots, 0, ru_{m,k+1})$$

(8.1.31)

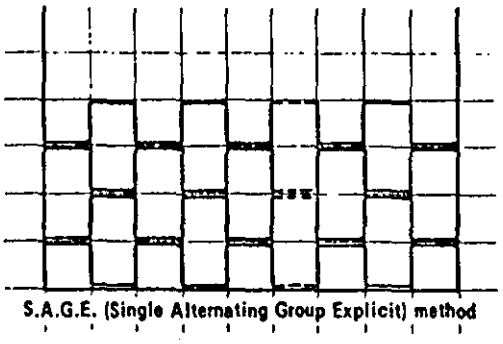


G.E.C. (Group Explicit Complete) method

$$(I+r\hat{G}_2)u_{k+1} = (I-r\hat{G}_1)u_k + b_4$$

$$b_4^T = (ru_{0,k}, 0, \ldots, 0, ru_{m,k})$$

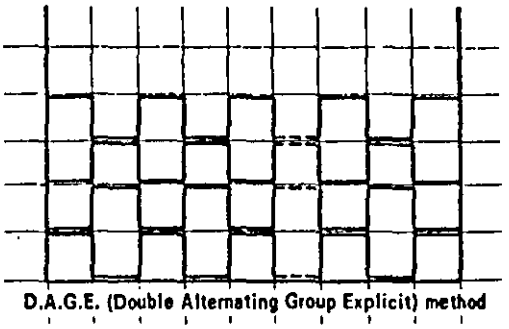
(8.1.32)



S.A.G.E. (Single Alternating Group Explicit) method

$$(I+r\hat{G}_1)u_{k+1} = (I-r\hat{G}_2)u_k + b_3$$

$$(I+r\hat{G}_2)u_{k+2} = (I-r\hat{G}_1)u_{k+1} + b_4 \qquad (8.1.33)$$



D.A.G.E. (Double Alternating Group Explicit) method

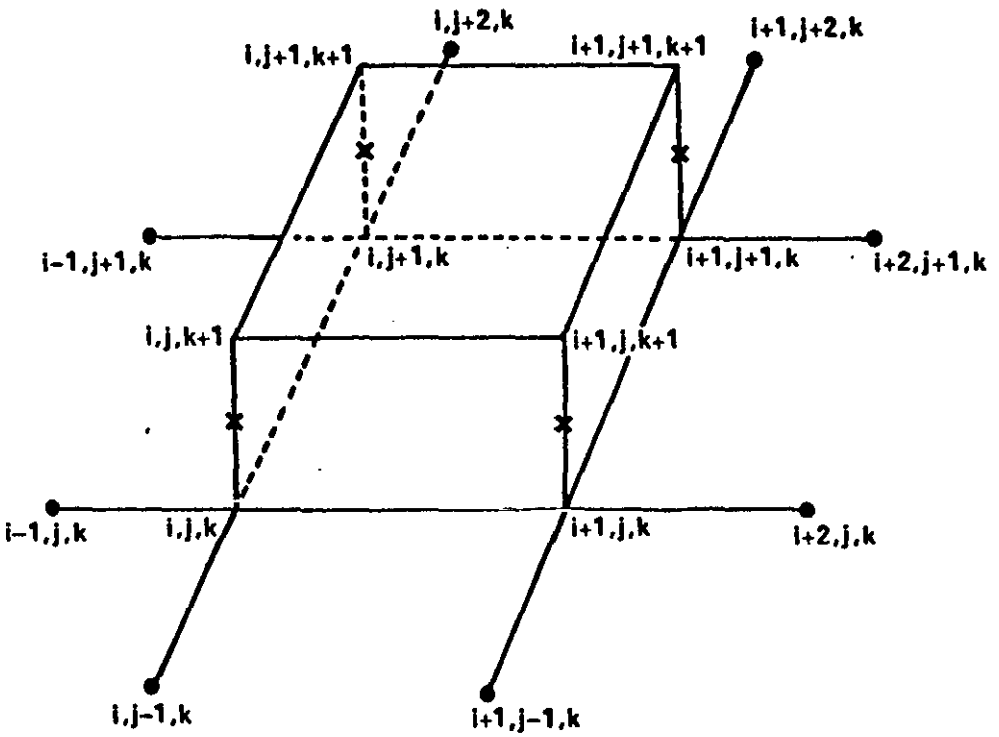
$$(I+r\hat{G}_1)u_{k+1} = (I-r\hat{G}_2)u_k + b_3$$

$$(I+r\hat{G}_2)u_{k+2} = (I-r\hat{G}_1)u_{k+1} + b_4$$

$$(I+r\hat{G}_2)u_{k+3} = (I-r\hat{G}_1)u_{k+2} + b_4$$

$$(I+r\hat{G}_1)u_{k+4} = (I-r\hat{G}_2)u_{k+3} + b_3 \qquad (8.1.34)$$

The group explicit technique is extended to the 2-D problem (8.1.2) by unifying the molecules (8.1.19) to produce a group of 4-points (i,j,k), (i+1,j,k), (i,j+1,k), and (i+1,j+1,k) of the form,



(8.1.35)

approximating (8.1.2) at these points using a finite difference formulation produces after some manipulation,

$$au_{i,j,k+1} = cu_{i,j,k}+bu_{i-1,j,k}+gu_{i+1,j,k}+du_{i+2,j,k}+bu_{i,j-1,k}$$
$$+du_{i+1,j-1,k}+eu_{i-1,j+1,k}+fu_{i,j+1,k}+gu_{i+1,j+1,k}$$
$$+du_{i+2,j+1,k}+eu_{i,j+2,k}+du_{i+1,j+2,k}\ , \qquad (8.1.36)$$

where the coefficients a-g (defined in Fig.8.1.1) are small degree polynomials in r. Similar relations hold for $u_{i,j+1,k+1}$, $u_{i+1,j,k+1}$ and $u_{i+1,j+1,k+1}$, producing a fully explicit result. Like the 1-D case, additional explicit relations can be derived for partial groups on the edges of the x-y region and corners yielding the equations below.

TYPE 1: SOUTH BOUNDARY:

$$a_1u_{i,1,k+1} = c_1u_{i,0,k+1}+b_1u_{i+1,0,k+1}+c_1u_{i-1,1,k}+b_1u_{i+2,1,k}+d_1u_{i,1,k}$$
$$+e_1u_{i+1,1,k}+c_1u_{i,2,k}+b_1u_{i+1,2,k}$$

$$a_1u_{i+1,1,k+1} = b_1u_{i,0,k+1}+c_1u_{i+1,0,k+1}+b_1u_{i-1,1,k}+c_1u_{i+2,1,k}+e_1u_{i,1,k}$$
$$+d_1u_{i+1,1,k}+b_1u_{i,2,k}+c_1u_{i+1,2,k} \qquad (8.1.37)$$

TYPE 2: EAST BOUNDARY:

$$a_1u_{m-1,j,k+1} = c_1u_{m-2,j,k}+b_1u_{m-2,j+1,k}+d_1u_{m-1,j,k}+e_1u_{m-1,j+1,k}$$
$$+c_1u_{m-1,j-1,k}+b_1u_{m-1,j+2,k}+c_1u_{m,j,k+1}+b_1u_{m,j+1,k+1}$$

$$a_1u_{m-1,j+1,k+1} = b_1u_{m-2,j,k}+c_1u_{m-2,j+1,k}+e_1u_{m-1,j,k}+d_1u_{m-1,j+1,k}$$
$$+b_1u_{m-1,j-1,k}+c_1u_{m-1,j+2,k}+b_1u_{m,j,k+1}+c_1u_{m,j+1,k+1} \qquad (8.1.38)$$

TYPE 3: NORTH BOUNDARY:

$$a_1u_{i,m-1,k+1} = c_1u_{i,m-2,k}+b_1u_{i+1,m-2,k}+d_1u_{i,m-1,k}+e_1u_{i+1,m-1,k}$$
$$+c_1u_{i-1,m-1,k}+b_1u_{i+2,m-1,k}+c_1u_{i,m,k+1}+b_1u_{i+1,m,k+1}$$

$$a_1u_{i+1,m-1,k+1} = b_1u_{i,m-2,k}+c_1u_{i+1,m-2,k}+e_1u_{i,m-1,k}+d_1u_{i+1,m-1,k}$$
$$+bu_{i-1,m-1,k}+c_1u_{i+2,m-1,k}+b_1u_{i,m,k+1}+cu_{i+1,m,k+1} \qquad (8.1.39)$$

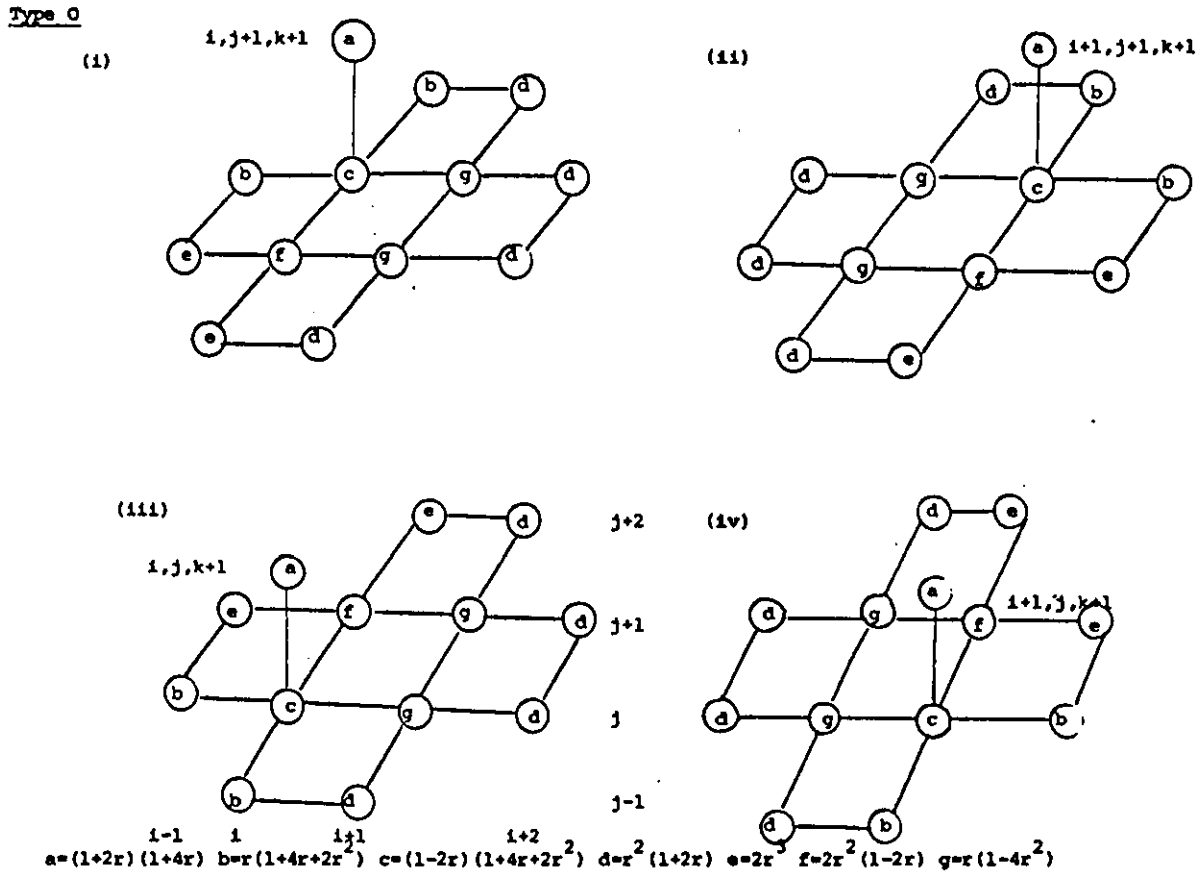


FIGURE 8.1.1: Molecule for a 4 point group (Type O)

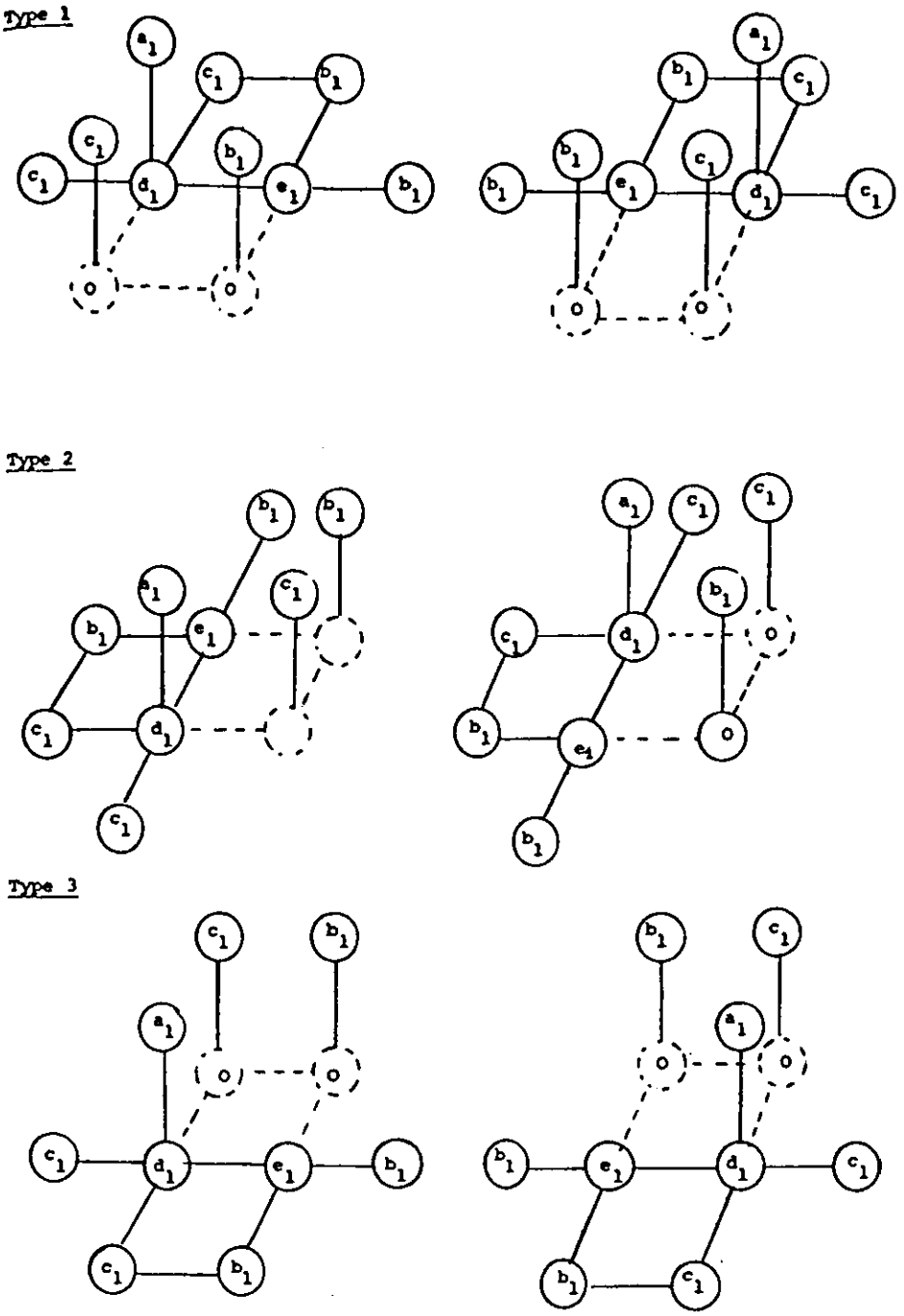


FIGURE 8.1.1: (cont.) Boundary 2-point molecules

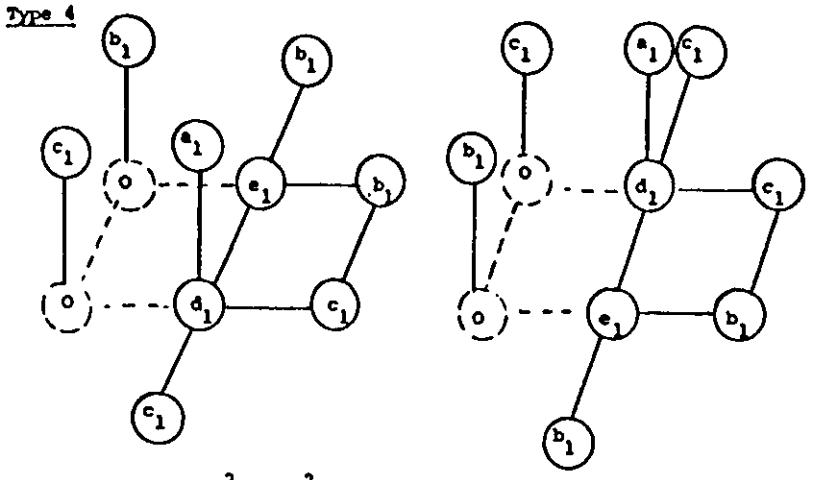


FIGURE 8.1.1: Two-point boundary molecules (cont.)

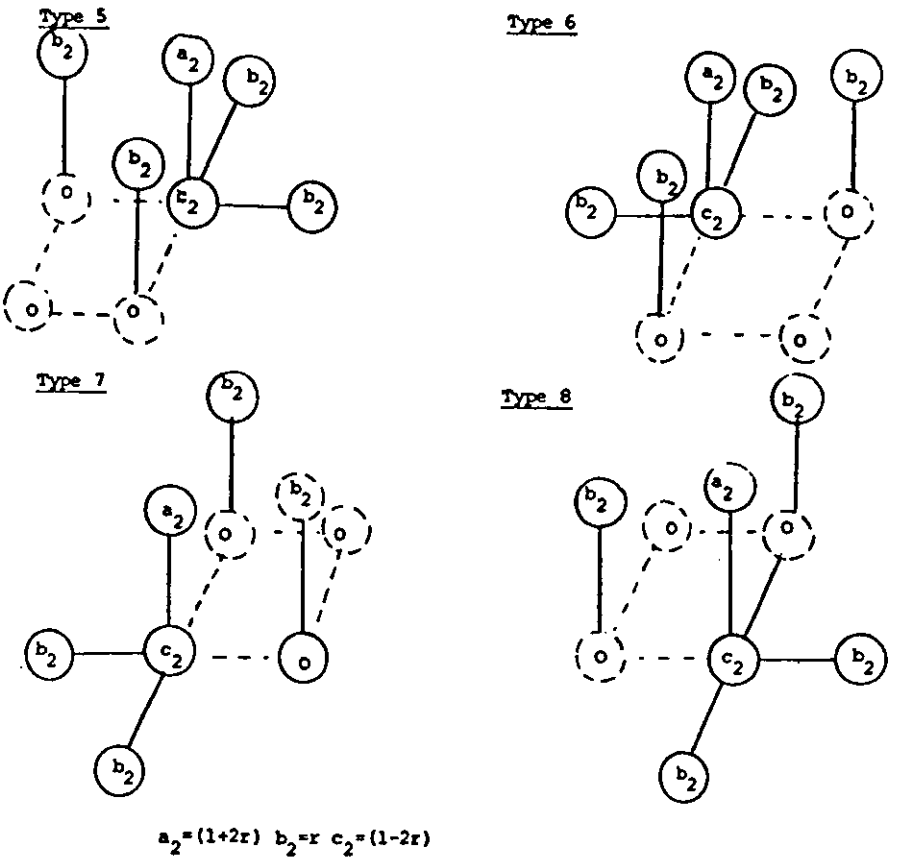


FIGURE 8.1.1: (cont.) One-point corner molecules

TYPE 4: WEST BOUNDARY

$$a_1 u_{1,j,k+1} = c_1 u_{0,j,k+1} + b_1 u_{0,j+1,k+1} + c_1 u_{1,j-1,k} + b_1 u_{1,j+2,k} + d_1 u_{1,j,k} + e_1 u_{1,j+1,k} + c_1 u_{2,j,k} + b_1 u_{2,j+1,k}$$

$$a_1 u_{1,j+1,k+1} = b_1 u_{0,j,k+1} + c_1 u_{0,j+1,k+1} + b_1 u_{1,j-1,k} + c_1 u_{1,j+2,k} + e_1 u_{1,j,k} + d_1 u_{1,j+1,k} + b_1 u_{2,j,k} + c_1 u_{2,j+1,k} \quad (8.1.40)$$

TYPE 5: BOTTOM LEFT CORNER

$$a_2 u_{1,1,k+1} = b_2 u_{0,1,k+1} + b_2 u_{1,0,k+1} + b_2 u_{1,2,k} + c_2 u_{1,1,k} + b_2 u_{2,1,k} \quad (8.1.41)$$

TYPE 6: BOTTOM RIGHT CORNER

$$a_2 u_{m-1,1,k+1} = b_2 u_{m-1,0,k+1} - b_2 u_{m,1,k+1} + b_2 u_{m-1,2,k} + c_2 u_{m-1,1,k} + b_2 u_{m-2,1,k} \quad (8.1.42)$$

TYPE 7: TOP RIGHT CORNER

$$a_2 u_{m-1,m-1,k+1} = b_2 u_{m-2,m-1,k} + c_2 u_{m-1,m-1,k} + b_2 u_{m-1,m-2,k} + b_2 u_{m,m-1,k+1} + b_2 u_{m-1,m,k+1} \quad (8.1.43)$$

TYPE 8: TOP LEFT CORNER

$$a_2 u_{1,m-1,k} = b_2 u_{2,m-1,k} + c_2 u_{1,m-1,k} + b_2 u_{1,m-2,k} + b_2 u_{0,m-1,k+1} + b_2 u_{1,m,k+1} \quad (8.1.44)$$

The computational molecules and coefficients for the above cases are given in Fig.(8.1.1). The GE methods follow directly from these molecules depending on whether the x-y region is divided into an odd or even number of parts in each direction. Fig.(8.1.2) indicates the various schemes that can be derived. If we use molecules of type 0 for the first $[\frac{1}{2}(m-2)]^2$ groups of 4 points anchored at (1,1,k) and molecules of type 2 at (m-1,j,k) j=1(1)m-2, molecule type 7 at (m-1,m-1,k) and type 3 for (i,m-1,k), i=1(1)m-2, the GER(x) or Group Explicit with ungrouped right points in

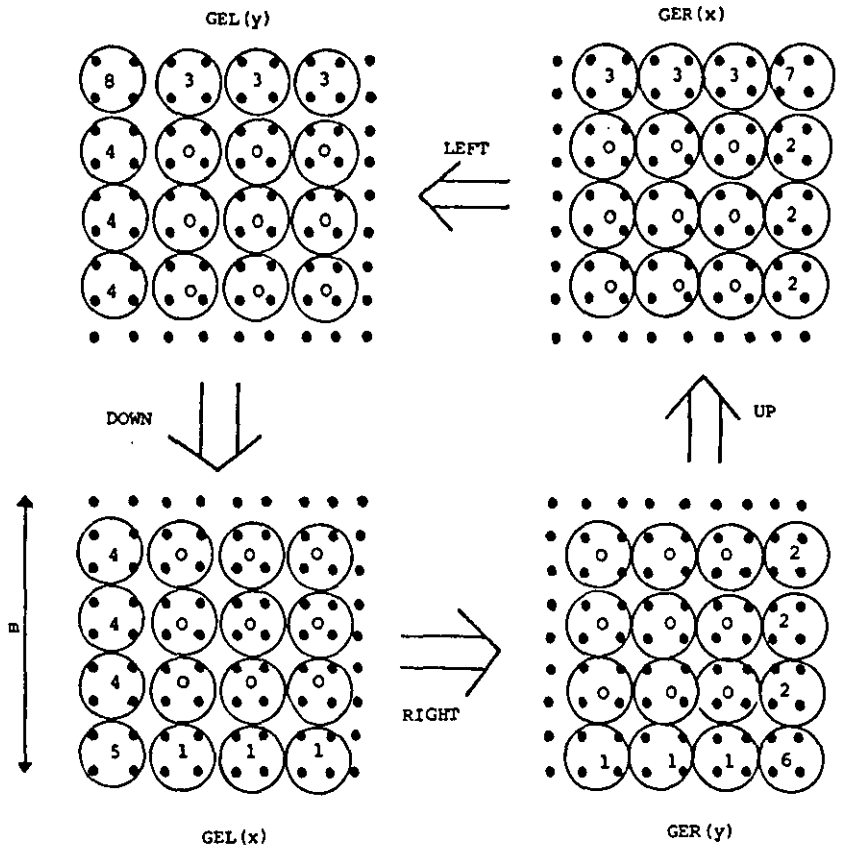


FIGURE 8.1.2: Shift cycle for four basic GE schemes indicating molecule types and groups (m=9).

the x-direction results. Similarly Group Explicit with left ungrouped point in the x-direction GEL(x) and GER(y), GEL(y) are derived as shown in Fig.(8.1.2). The GER and GEL schemes are conditionally stable for $r \leq 1$ while the introduction of an alternating strategy produces unconditionally stable procedures. Once again from Fig.(8.1.2) a Single Alternating Group Explicit (SAGE) scheme can be constructed by alternating GER(x) and GEL(x) on successive cycles. While a Double Alternating Group Explicit (DAGE) scheme is produced by GER(x), GEL(x), GEL(x), GER(x). Similar methods involving y and x,y combinations are easily derived. Notice that SAGE schemes correspond to a (Anti-) clockwise shift of two places, while a DAGE (x or y) is achieved by two clockwise shifts followed by two anti-clockwise shifts around Fig.(8.1.2). Alternation is achieved by a simple cyclic rotation.

8.2 ALGORITHMIC VS GEOMETRIC SOLUTION OF P.D.E.'S

The use of finite difference methods in which a grid of points is superimposed over a region to be solved leads naturally to the solution of approximations to P.D.E.'s by linear systems. Indeed, a matrix representation exhibits (as we have seen), for certain orderings of the points, useful Linear Algebra properties such as diagonal dominance, positive definiteness, and sparse banded matrix structures. As the solution process for a series of time levels is essentially iterative, with an application of the explicit or implicit solution procedure applied on each iteration, the theory of convergence of iterative matrix methods can be applied in an analogous manner to the stability problems of finite difference methods.

A brief survey of systolic arrays indicates that the pre-occupation with Linear Algebra and specifically matrix techniques is also prevalent. This attitude is understandable in both applications. In the former case stability is an important aspect of the applied technique, without a matrix notation the theory would be difficult to analyse. The latter is justified by the recurrence relations exhibited in matrix formulations and the reduction in most cases to a few primitive arithmetic operations, permitting parallelism on locally connected basic cells of simple structure. Systolic arrays have further captured the imagination by creating designs which have area proportional to the bandwidth of the matrix considered. However, for problems where the bandwidth is related to problem size, the band itself may also be sparse and a source of redundant cells. Such cells under special circumstances can be replaced by simple delays to compact the design.

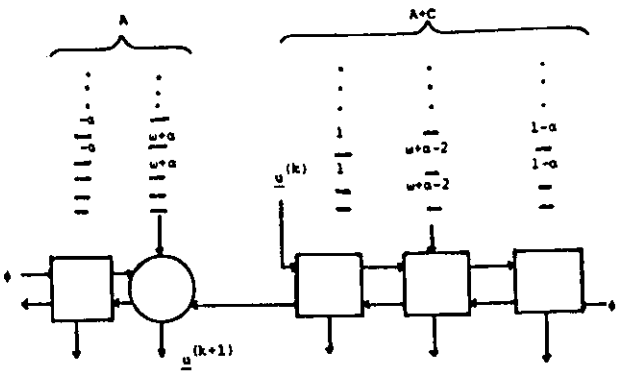
We shall consider systolic arrays for the sample problems under two

circumstances:

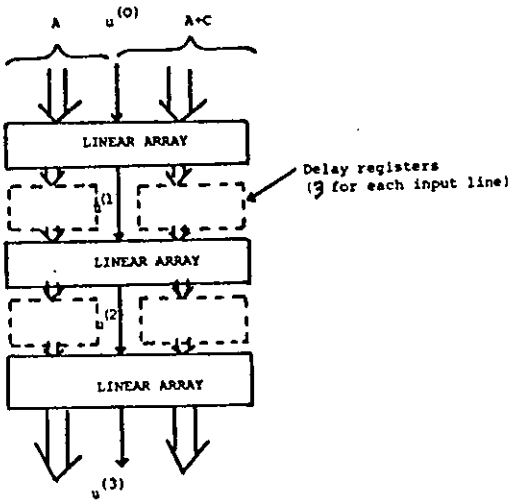
(i) when only the final time level t_z is required

(ii) when each level t_k, k=1(1)t_z is required,

and compare the designs with the intuitive cascaded iteration array (CIA). For instance, a single iteration of the 1-D problem in (8.1.13) requires a tridiagonal array of 3 cells computing $(A+C)u^{(k)}$, and a back substitution array of two cells to produce $u^{(k+1)}$ as shown in Fig.(8.2.1a). Using Theorem (3.2.3.1) with p=2 and allowing an extra cycle in the back-substitutor gives $T=2m+4t_z$ cycles for computing t_z levels (see Fig.(8.2.1b)).



a) Cascaded linear iteration array (1-D) case



b) Cascaded Scheme for three levels

FIGURE 8.2.1

Each iteration requires 5 IPS equivalent cells and sandwiched between each level are 3 delay registers per ips for synchronising data on different levels. It follows that we require a total of 5 t_z IPS cells and $15(t_z-1)$ registers to reach level t_z. For the 2-D problem the bandwidth of A and B are m and 2m-1 respectively. Hence each linear array requires 3m-1 cells, and has latency 2m cycles, yielding $T=2m^2+2mt_z$ cycles, $(3m-1)t_z$ IPS cell equivalents and $(t_z-1)(3m-1)(2m-1)$ synchronizing delay registers between iterations. Notice that this latter problem has internal band sparsity which can allow reductions in hardware. Furthermore if we allow multipass computations with only $\bar{t}_z<t_z$ linear arrays, hardware can be minimised with,

$$T = \begin{cases} \left\lceil \frac{t_z}{\bar{t}_z} \right\rceil \{2m+4\bar{t}_z\} & \text{1-D case} \\ \left\lceil \frac{t_z}{\bar{t}_z} \right\rceil \{2m^2+2m\bar{t}_z\} & \text{2-D case} \end{cases} \qquad (8.2.1)$$

These designs based on the matrix or algorithmic expression of the asymmetric approximations are intuitive, but we intend to show that they are not necessarily the best. For instance, from a theoretical viewpoint the parameters w and r indicate that ℓ is restricted by h. This restriction often means a large number of levels to achieve good accuracy and the above intuitive computational approach although suited to sequential machines translates to a systolic design with cells proportional to t_z. When $t_z>m$ we may be better with a design where cells are proportional to m, permitting fast computation of many levels. To facilitate this work we discard the explicit use of a matrix representation, and base the systolic design on the use of computational molecules and templates describing primitive operations for constructing successive time-levels.

Essentially, we question the algorithmic matrix notation for the derivation of systolic arrays in preference to a direct mapping (or geometric) approach to the problem. Naturally the designs produce systolic marching arrays. Finally to keep our techniques in context we remark that the matrix representation is still indispensible for theoretical studies of stability but can be neglected for actual computation by the systolic array.

8.3 LINEAR ASYMMETRIC MARCHING PROCESSOR (LAMP) ARRAYS

The marching processors we develop avoid the 'Linear Algebra' trap by considering the mesh points of the solution as elements of tables. For the 1-D case an open ended table is applicable implying table generating techniques, whereas the 2-D problem has a fixed sized table indicating a table manipulation approach to design.

Consider again the 1-D equation (8.1.1), its initial and boundary conditions (8.1.3), and recall the table construction techniques from Chapter 7. The order of computation as shown in Fig.(8.3.1) is similar

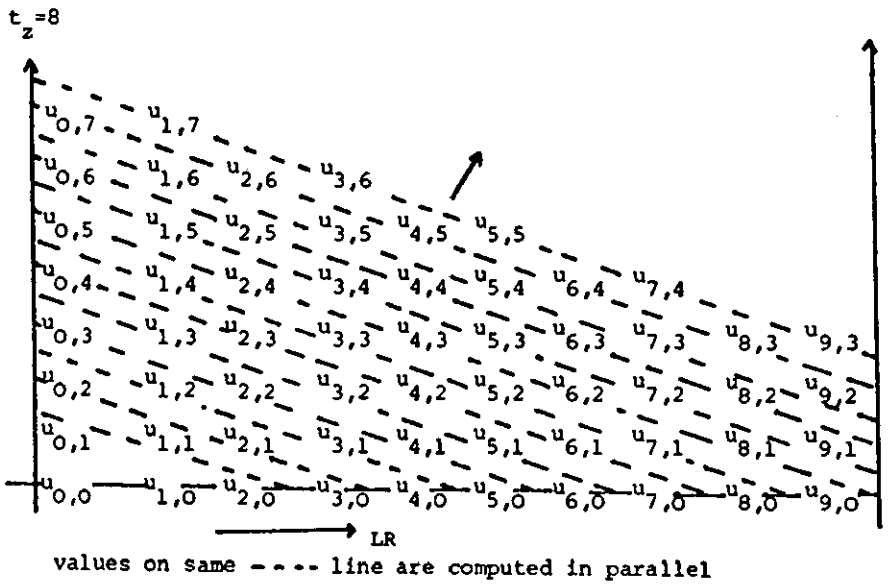
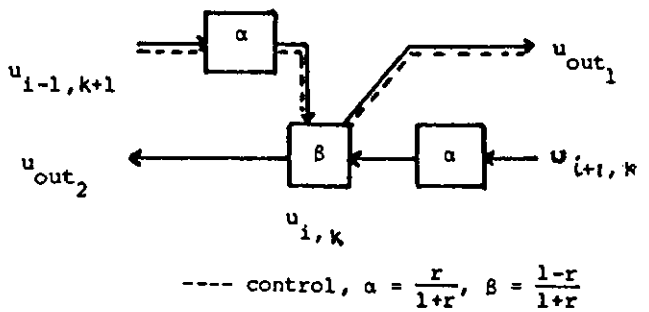


FIGURE 8.3.1: Computation order for multiple level LR scheme

to that in the extrapolation table generation, implying an allocation of cells to columns such that cell i computes column i of the regions mesh points. Thus a linear array of (m-1) cells will compute all the levels t_o to t_z. The LR version of the marching array with operation snapshots is shown in Fig.(8.3.2) with the basic cell derived from the molecule rule (8.1.12b) resulting from (8.1.10a).




```
IF
   control THEN
   [(i)  t1=αu(i-1,k+1)
         t2=αu(i+1,k)                    } α cells
         t3=βu(i,k)
    (ii) u(i,k)=t1+t2+t3
         u(out1)=u(out2)=t1+t2+t3        } β cells
   ]
ELSE
   [u(out1)=u(i,k)
   u(out2)=u(i,k)
   ]
```

(8.3.1)

the array operation proceeds as follows:

(i) At startup the cells are loaded with the initial values of (8.1.3b) for level t_o.

(ii) On the first cycle of operation the leftmost cell receives an input from the host corresponding to the left boundary value of (8.1.3b) for level t_1, and an associated control tag bit. The cell performs the molecule rule producing the value $u_{1,1}$ and overwriting the old result.

(iii) On the next cycle, the control tag shifts right with the new level value triggering cell 2, which accepts the cell 3 output to execute the molecule rule and overwrite its stored value. Meanwhile cell 1 is idle, waiting for the cell 2 result.

(iv) Thus, dataflow of the LR array forms an eddy type wavefront pattern similar to the odd-even sorter, in Chapter 7, in which

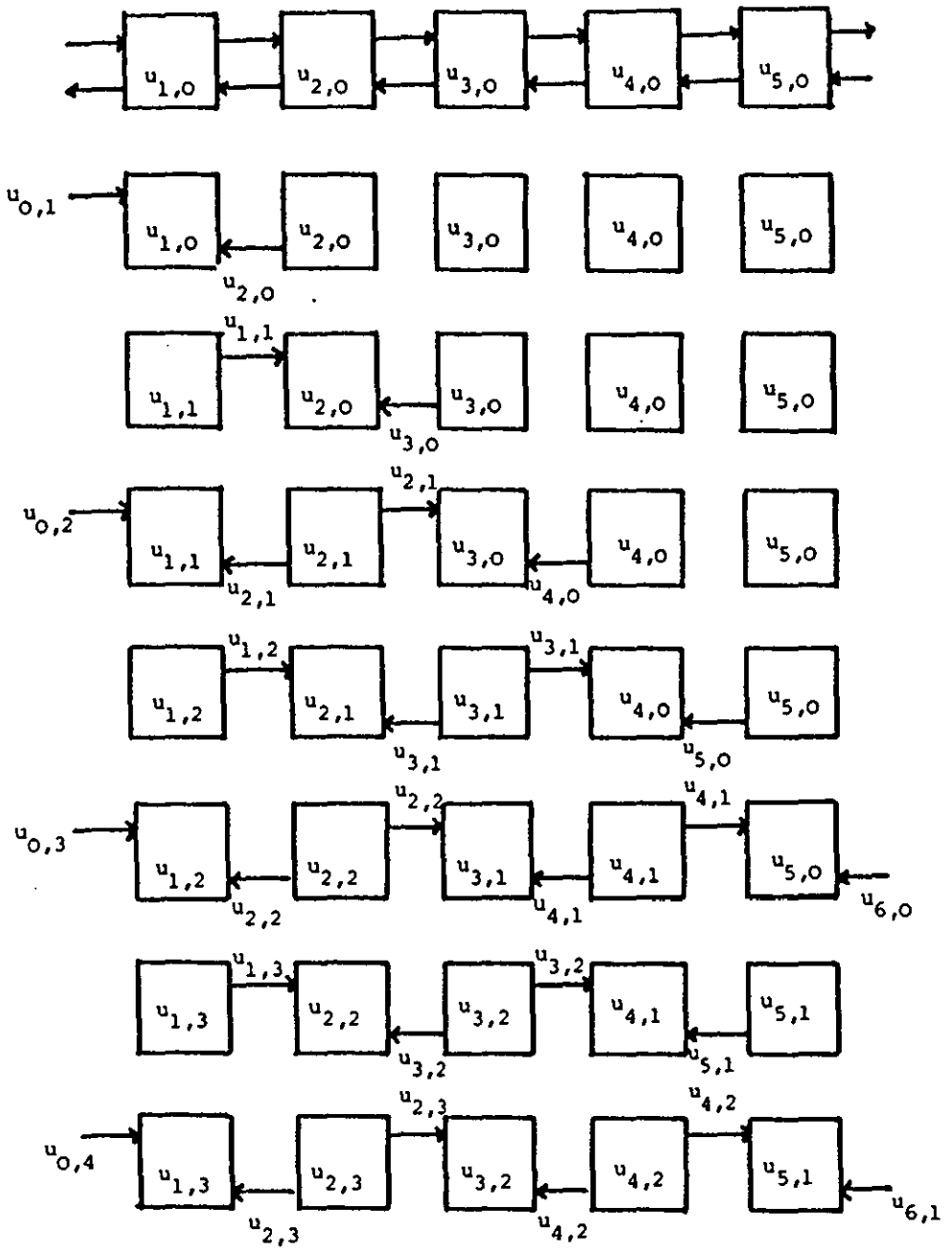


FIGURE 8.3.2: Systolic computation of the LR solution

odd and even cells are active on alternate cycles. In general cell i collects $u_{i-1,k+1}$ from its left, $u_{i+1,k}$ from its right, contains $u_{i,k}$ and produces $u_{i,k+1}$.

Also notice that the first right boundary value is required when the tag bit reaches the rightmost cell. Consequently the complete array is

controlled by the left boundary input with the form,

$$\begin{array}{lcccccccc} \text{DATA} & u_{0},t_{z} & \delta & \cdots & \delta\ u_{0,3} & \delta & u_{0,2} & \delta & u_{0,1} \\ \text{TAG BIT} & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \end{array} \longrightarrow \qquad (8.3.2)$$

with cells started and controlled by the tag bit. The cell (8.3.1) is a simple intuitive mapping of the computational rule into a cell structure requiring 3 multipliers and 2 adders. By combining the α cells into a single inner product cell we require just two ips equivalents and the basic cell cycle time is bounded by 1 ips + 1 add or 1.5 ips cycles. It follows that after $3t_z$ ips cycles the leftmost LR cell has received its last boundary value and computed its t_z level value. Allowing the last control value to march across the array gives the upper bound,

$$T_c = 1.5m + 3t_z \text{ ips cycles} , \qquad (8.3.3)$$

for array computation time, neglecting loading and unloading. If only the level t_z is required, cells can be loaded and unloaded from the left and right edges of the array. We need at most 1.5*(m+2) cycles to pipeline α,β and $u_{i,0}$ into the cells for loading, and 1.5m cycles to unload level t_z. Thus,

$$T_s = 1.5m+1.5(m+2) + 1.5m + 3t_z = 4.5m + 3t_z + 3 \qquad (8.3.4)$$

is the total computation time including input/output. When all the levels are required (like the extrapolation tables) an output for each cell is added, and loading and unloading is performed in parallel adding only 4(1.5) cycles to (8.3.3) yielding,

$$T_p = 1.5m + 3(t_z+2) . \qquad (8.3.5)$$

Comparing these timings with the cascaded scheme of (8.2.1) yields the speed-up relations,

$$S_p = \frac{\left\lceil \frac{t_z}{\bar{t}_z} \right\rceil \{2m+4\bar{t}_z\}}{4.5m+3t_z+3} ,$$

hence for $S_p>1$ and t_z divisible by $\bar{t}_z$

$$t_z > \left\{ \frac{4.5m+3}{2m+\bar{t}_z} \right\} \bar{t}_z \qquad (8.3.6)$$

for the sequential loading scheme to compute the t_z level faster than the cascaded scheme with $\bar{t}_z$ linear arrays. Likewise for the parallel loading scheme,

$$S_p = \frac{\left\lceil \frac{t_z}{\bar{t}_z} \right\rceil \{2m+4\bar{t}_z\}}{1.5m+3t_z+6} \Rightarrow t_z > \frac{(1.5m+6)}{(2m+\bar{t}_z)} \bar{t}_z . \qquad (8.3.7)$$

In terms of cell count the LR array requires 2(m-1) ips equivalents while the cascaded form requires $5\bar{t}_z$, hence for a cell saving,

$$5\bar{t}_z > 2(m-1) \Rightarrow \bar{t}_z > \frac{2}{5}(m-1) . \qquad (8.3.8)$$

Substituting in (8.3.6) and (8.3.7) for (8.3.8) yields,

$$t_z > \begin{cases} \frac{(4.5m+3)(m-1)}{(6m-1)} & \text{for (8.3.6)} \\ \frac{(1.5m+6)(m-1)}{(6m-1)} & \text{for (8.3.7)} \end{cases} \qquad (8.3.9)$$

to produce both cell savings and speed-up using the LR marching array. For example, if m=10

$$t_z > \begin{cases} \frac{48\times9}{59} = 7.32 \\ \frac{21\times9}{59} = 3.20 \end{cases} \qquad (8.3.10)$$

which is easily satisfied, provided we solve long narrow regions. Where the arrays are used repeatedly for different stepsizes h,h/2,h/4, etc. the cascased iterative scheme has a clear advantage as its array size is independent of m the number of mesh point columns. The LR array

must be redefined with more cells to accommodate the extra points.

Next, observe, that although the molecule rules are asymmetric the molecules themselves are simple reflections of each other, hence the above array is a unified array for the LR and RL computations. To produce an RL scheme simply load the initial values in reverse order and interchange the left and right boundary inputs. The control tag is still input on the left.

Finally for the 1-D case, consider the problems of alternating and averaging LR (RL) sweeps. An array to perform an alternating scheme is easy to construct as indicated by Fig.(8.3.3a). The array consists of two tiers, each of m-1 cells which are modified versions of (8.3.1). Operation of the array is simple with each tier representing an LR or RL sweep. Initially the top tier is loaded with starting values and the bottom tier cleared, the order of loading depending on the type of alternation, LR/RL or RL/LR. In the case of LR/RL the top tier starts computation like an LR scheme, except that there is only one control tag bit set (corresponding to $u_{0,1}$ in (8.3.2)), and molecule rule results are stored in the cell immediately below the active cell, on the second tier. On leaving the rightmost top tier cell the control bit is fed back into the bottom tier which is by now loaded with first tier results, and initiates an RL sweep. The bottom tier results are loaded back into the top tier and on emerging from the leftmost bottom tier cell, the control completes an alternation cycle and can be piped back into the top tier, forming a primitive Systolic Control Ring (SCR).

Notice that only a single cell in the whole array is active on a particular cycle, consequently the array can be compacted to a single tier of m-1 LR-cells modified to accept a triggering tag bit signal

from either direction and perform the correct molecule rule. The circulating tag bit picks up the correct boundary values as they enter and leave the ends of the array. It follows that the total time to compute and output t_z levels is,

$$T_{alt} = 1.5(m-1)t_z + 4(1.5) \qquad (8.3.11)$$

Comparing this to the cascaded form we observe that the LR/RL switch requires the reversal of the vector u between successive iterations, preventing cascading, and forcing $\bar{t}_z$=1, and a time,

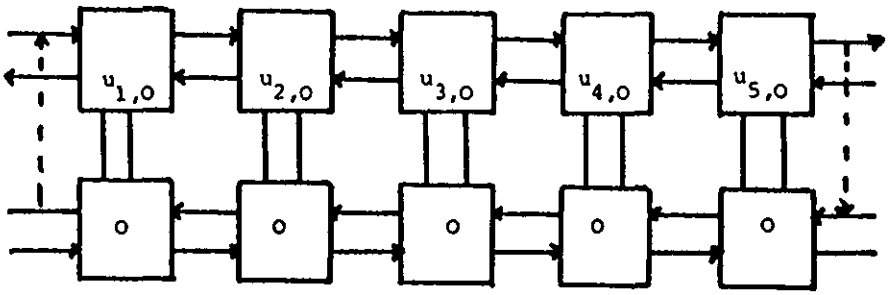
$$T = (2m+4)t_z \qquad (8.3.12)$$

from which a speedup is immediately observed. However, the cascaded form requires only a single array of 5 ips cells with greater efficiency than the m-1 LR- cells required by the LR/RL version. Consequently the speed-up does not compensate for the hardware increase.

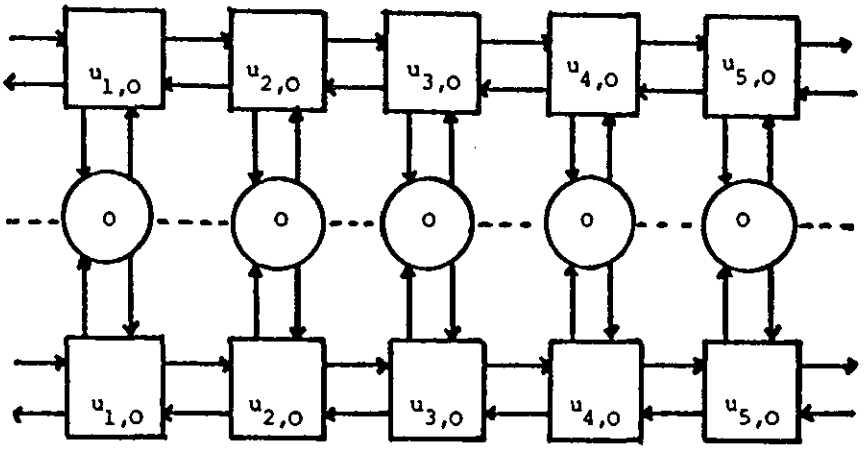
An array for averaging LR and RL sweeps on the same time level is shown in Fig.(8.3.3b). This time we have three tiers, the top tier acts like an LR sweep and the bottom tier like an RL sweep, and are operated in parallel. The central tier is a row of independent cells which are initially empty, but collect a result from each tier, finds the average and returns results to the adjacent top and bottom tier cells. As results are calculated left to right in the top tier and right to left in the bottom tier, it follows that the central averaging cell when m-1 is odd and the two central cells when m-1 is even compute the first average. Thus, the time for an averaging sweep with parallel loading is,

$$T_{avg} = 1.5t_z m + 4(1.5) \ , \qquad (8.3.13)$$

allowing an extra cycle for the final averages in the left and rightmost cells of the top and bottom tiers to be loaded. Again the low cell



a) Alternating LR/RL or RL/LR array



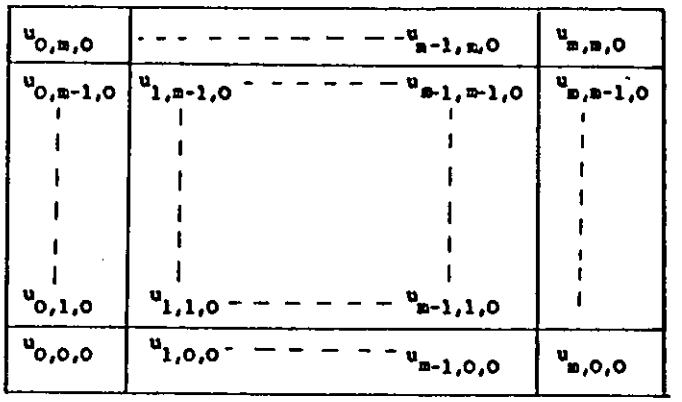
b) Averaging LR/RL array

FIGURE 8.3.3: Alternating and averaging schemes

efficiency implies that tiers can be merged, as long as the LR and RL controls do not trigger the same cell on the same cycle and the stored level values are only overwritten by the averaging cell. A compacted two tier design with m-1 modified LR cells and m-1 averaging cells is easily formed. The algorithmic version of the array cannot be cascaded yielding $\bar{t}_z$=1 again, but by filling the synchronising neutral elements of the matrix and vector inputs in Fig.(8.2.1) with A^T,A^T+C and reversing the ordering of the u_k vector, calculations of LR and RL sweeps can be interleaved. The real problem, however, is the formation of the average, and makes the method impractical, because it must be performed after all the RL and LR values are computed.

Next, consider the 2-D unsteady diffusion equation (8.1.2) with boundary and initial conditions (8.1.14). Again the comparison with the cascaded iterative array will be considered, only this time each of the $\bar{t}_z$ linear array requires 3m-1 cells due to the form of (8.1.20). Although the sparse structure of B in the 2-D case can be used to limited effect by replacing full processors with delay cells the bandwidth, hence linear iteration arrays, are still proportional to the size of the solution region. For this reason we expect better area/time trade-offs with marching processors for the 2-D problem.

Now, the addition of an extra space dimension introduces a greater number of permutations to the variety of starting positions and alternating strategies employed. For simplicity, we examine computations on a square grid employing the equations (8.1.18). The first step is to abandon the matrix organisation of the problem treating the discretized x-y plane as a table of values to be modified. The immediate consequence of such a tabular approach is a dense table of $(m+1)^2$ elements



The outer rows and columns corresponding to the boundary values, and the embedded (m-1)*(m-1) elements the initial conditions. In an analogous manner to the derivation of LR(RL) 1-D arrays, a 3-D table updating wavefront is apparent (Fig.(8.3.4)), which maps the computation onto a wavefront array processor.

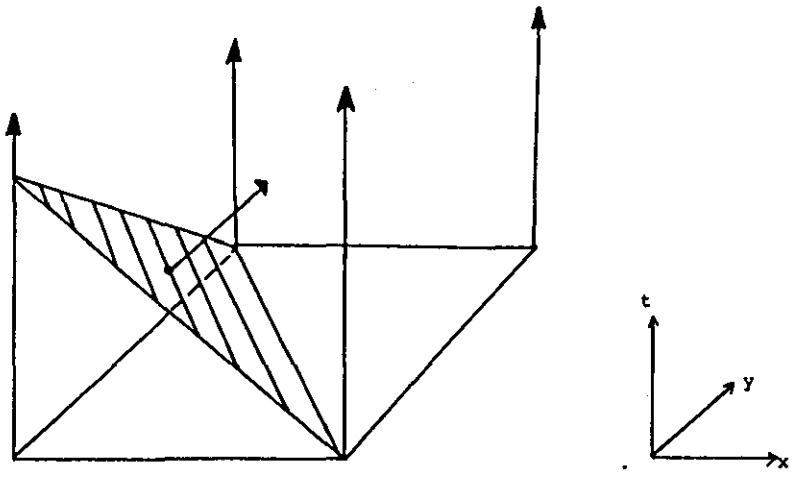
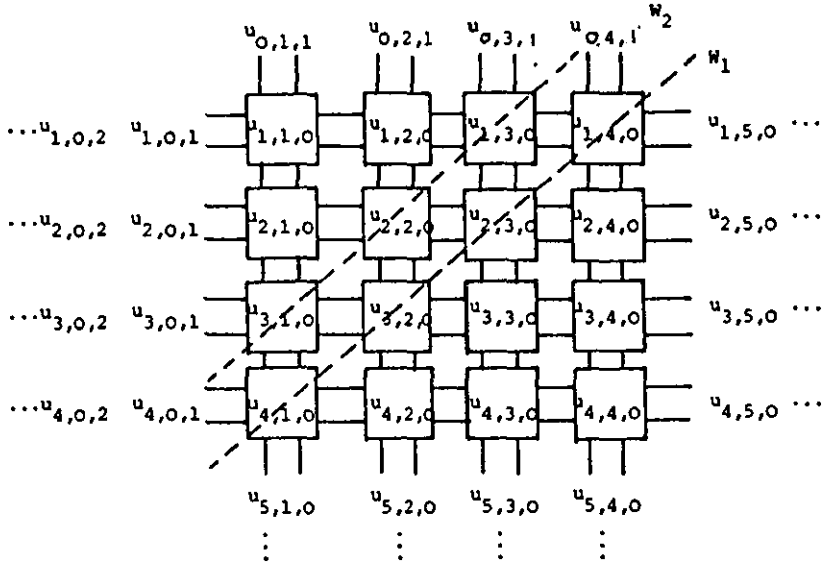


FIGURE 8.3.4: 3-D Wavefront

The processor arrangement for (8.1.18c) is shown in Fig.(8.3.5) together with the basic processor cell using 3 ips and an adder, with a cycle time of 1 ips + 2 adds. The processor requires $(m-1)^2$ basic cells or $3(m-1)^2$ ips + $(m-1)^2$ adders, and boundary values are input on each boundary of the array in a skewed fashion to match wavefront progression. It follows that each wavefront requires $2(m-1)$ cell cycles or $4(m-1)$ ips cycles to complete a single pass over the array, and that successive wavefronts are separated by a single cell cycle. In Fig.(8.3.5a) W_1 is a computation wave, W_2 a dummy wave and W_3 (the next wave) starts computation for the next level. Hence, a close connection between the odd-even operation of the 1-D implementation and the 2-D extension is apparent. Thus, to compute t_z with no intermediate level output,

$$T = 2t_z + 2(m-1) \text{ cell cycles or } 4t_z + 4(m-1) \text{ ips cycles} \quad (8.3.14)$$

Compared to $O(m^2)$ time in (8.2.1) required by the cascaded scheme. Recall, however, that the cascaded form uses only $\bar{t}_z(3m-1)$ ips cells, ignoring the cost of additional delay registers. Hence approximately $\bar{t}_z > \frac{2}{3}m$ arrays must be used before the wavefront model competes on hardware terms.



a) Wavefront processor arrangement of 2-D problem (n=6)

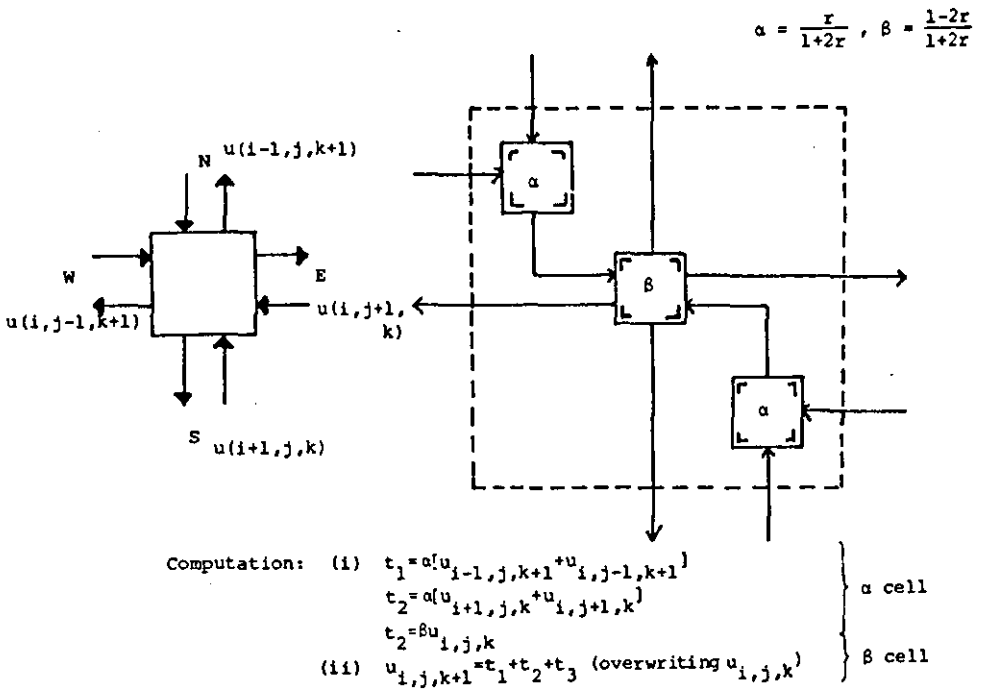
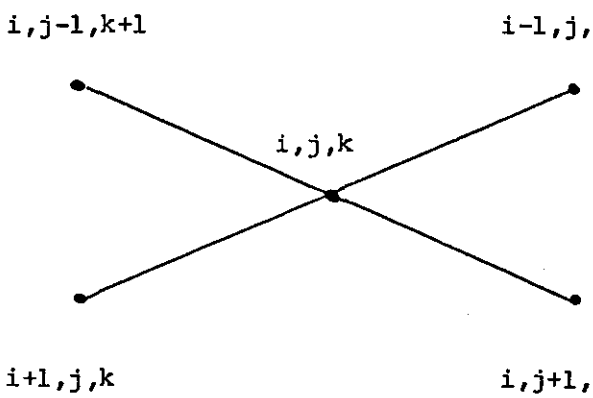


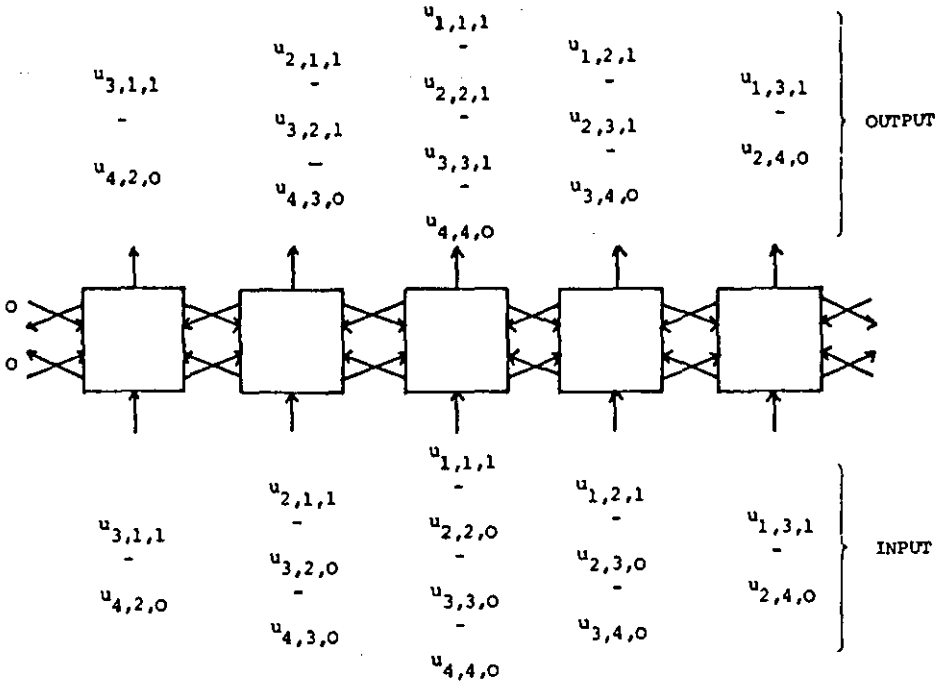
FIGURE 8.3.5: 2-D Wavefront computation

REMARK: The timings above omit the O(m) time required to load α,β and the initial values but as the cascaded array also uses O(m) time to synchronise the first iteration, this is not significant.

Next we reduce the number of processors in the wavefront model while also coping with the problem of outputting all the intermediate time level results. First, notice that like the LR(RL) schemes (8.1.18) all have the same molecule structure with different degrees of rotation, and so rotating the table data allows any molecule to be computed, (e.g. for (8.1.18c) an implicit rotation of 90° was used). Second, generating output for each level converts the problem from compute bound to input/output bound. Whereas the cascaded scheme requires only a single output line for each t_z and no modification to computation time. The wavefront processor demands O(m) cycles for output between each wavefront and at least m-1 outputs. It follows that only a single wavefront can be active on the processor at any instant in time, and at most (m-1) cells can be active giving a very poor efficiency. Fortunately, a single wavefront progression can be simulated by a linear array of cells (see Yang & Lee [86]), and such an array of 2m-3 cells is shown in Fig.(8.3.6). The basic cell implements a universal molecule template,



(8.3.15)



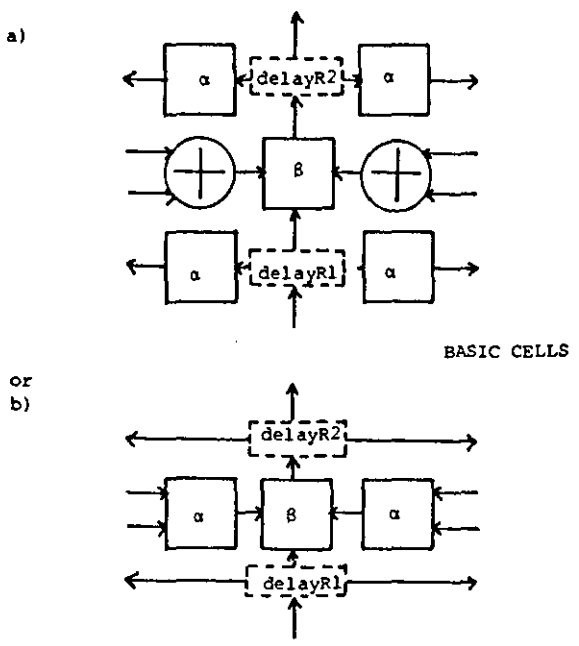
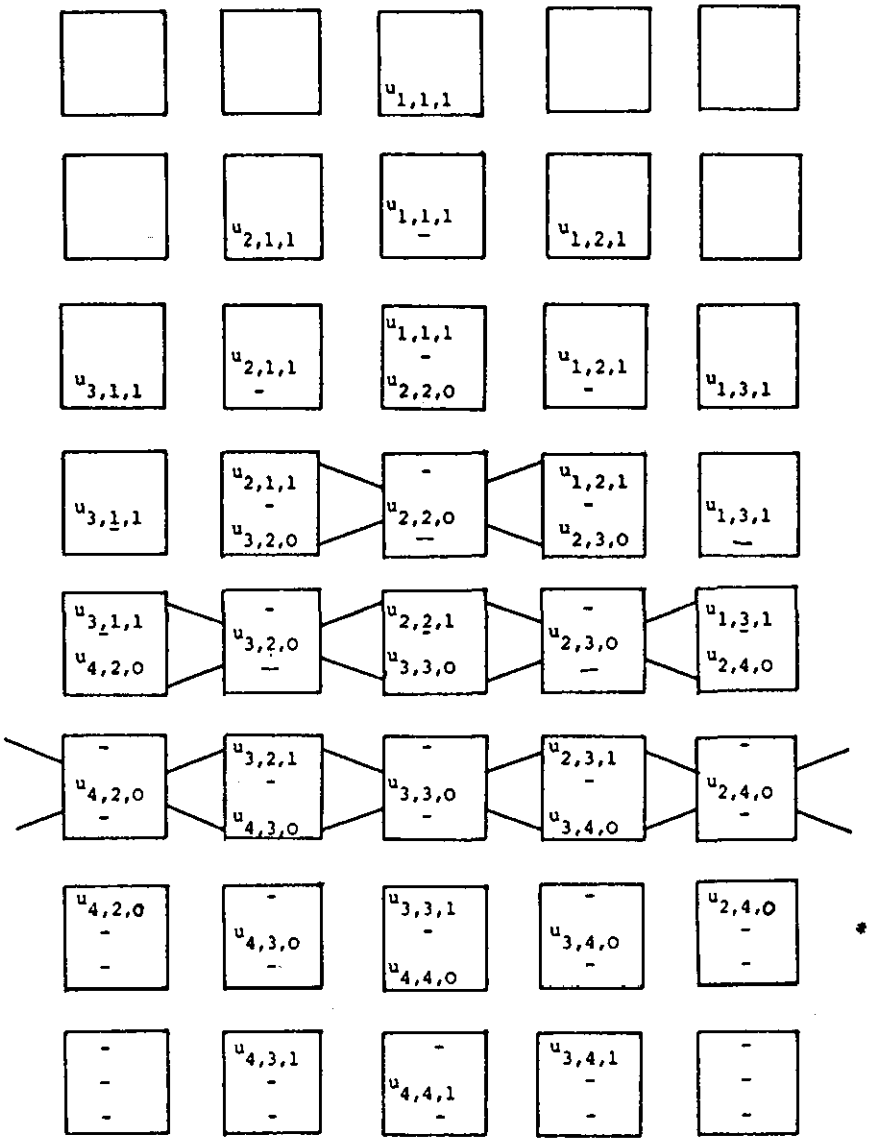


FIGURE 8.3.6: Linear asymmetric marching processor (LAMP) array n=4

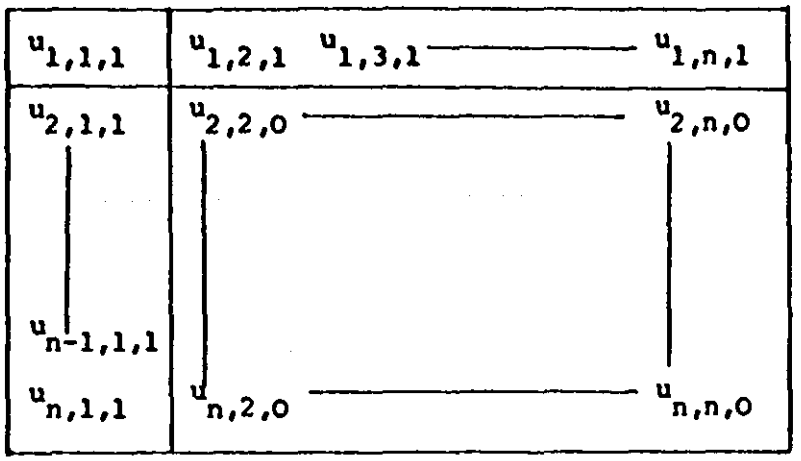


c) Computation on LAMP array (n+1=4)

* Disable the array here as last value is computed on previous cycle

FIGURE 8.3.6: (cont.)

for the formulas (8.1.18), the first cell Fig.(8.3.6a) is an intuitive mapping, the second Fig.(8.3.6b) an optimised version. For convenience and ease of explanation the table element indices are re-numbered to give a standard table form,



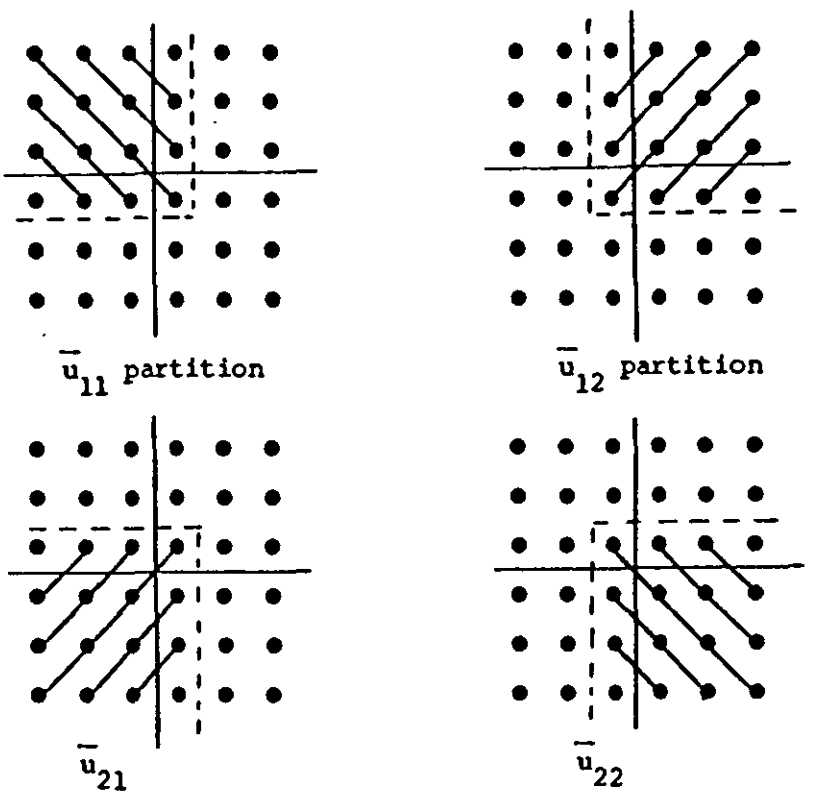
Now the operation of the LAMP array is clear. Computation starts in the central cell after three synchronising cell cycles (6 ips cycles). On the fourth cell cycle the centre cell and its two adjacent neighbours contain values which line up to form a molecule. Thus, the central cell modifies its value overwriting the central molecule element with $u_{i,i,k+1}$. The next cycle sees the two adjacent cells become the active centres of molecules and perform a table update. It follows that cells work on alternate cycles like the 1-D case with start-up controls coming from the centre cell. In general, when cell w contains a value in the β-cell the neighbouring cells w-1 and w+1 contain kth level values in R1 and (k+1)th level values in R2. After n cycles all the cells must be activated and the LAMP array computes with efficiency $e=\frac{1}{2}$ (or 50%) the same as the cascaded scheme. Furthermore, we observe that only α,β need to be loaded, and that the length of input data is 2n. Allowing 3 cycles to load α,β and 3 cycles for start-up on each wavefront pass. The LAMP array simulates the wavefront processor in

$$T = 2t_z(2n+3)+3 \text{ ips cycles} , \qquad (8.3.16)$$

and requires at most 2(n-1)-1 LAMP cells or 6(n-1)-3 ips cells, (i.e. the bandwidth of the table neglecting the two corner elements never modified). Comparing this to the cascaded timing (8.2.1) the LAMP

array is significantly faster with $O(nt_z)$ rather than $O(n^2t_z)$ ips cycles, while in terms of cells uses $O(n)$ rather than $O(t_z n)$ ips equivalents.

The LAMP array can be improved further by using a mixture of molecules for the same level. As an illustration we adopt (8.1.18a-d) computing them in parallel to reduce computation time. All the formulas have the same truncation error terms so no errors due to differences in approximation accuracy should occur. Consider the case when there is an even number of internal points in the x-y plane. The tabular representation is partitioned into 4 $\bar{u}_{ij}$ blocks of size $\frac{1}{2}m$, e.g. for m=6

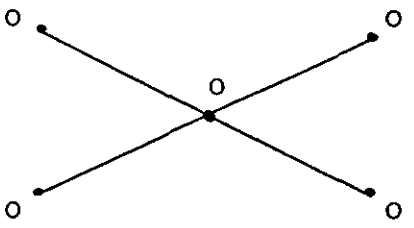


Each block requires the sharing of points around its boundaries. It follows that by sharing (or duplicating) a little data all the blocks can be computed in parallel. A single block requires $2(\frac{1}{2}m)-1=m-1$ LAMP cells or $3(m-1)$ ips cells, and pipelining the 4 blocks through the array

gives a time,

$$T = 8t_z\{2(\tfrac{1}{2}m)+3\}+3 = 8t_z(m+3)+3 \text{ ips cycles,} \qquad (8.3.17)$$

which is still a significant improvement over the cascaded scheme but uses half the area of the original LAMP array. Computation time can be further compressed by observing that a cells idle cycle uses the neutral molecule,



(8.3.18)

producing a zero result (which incidentally explains why no explicit control structure is discussed). This neutral molecule can be brought into play by interleaving two partitions as shown in Fig.(8.3.7) to produce a time,

$$T = 4t_z(m+3)+3 \text{ ips cycles ,} \qquad (8.3.19)$$

comparable to the full LAMP array time (8.3.16), with $n=\frac{1}{2}m$.

REMARK: When the tables' internal points are odd, partitioning requires some points to be calculated more than once from different directions. We then have the problem of deciding which estimate is best, and implies some kind of averaging scheme.

Now more accurate solutions are obtained if different molecules are alternated on different time levels (due to truncation term cancellations). For the 1-D case alternation introduced a sequential bias to computation making the marching arrays grossly inefficient. In the 2-D case problems with alternation disappear. The LAMP array in Fig.(8.3.6) is operated in a multipass mode with t_z passes for t_z levels. Consequently the only problem is rotating the data to fit the correct

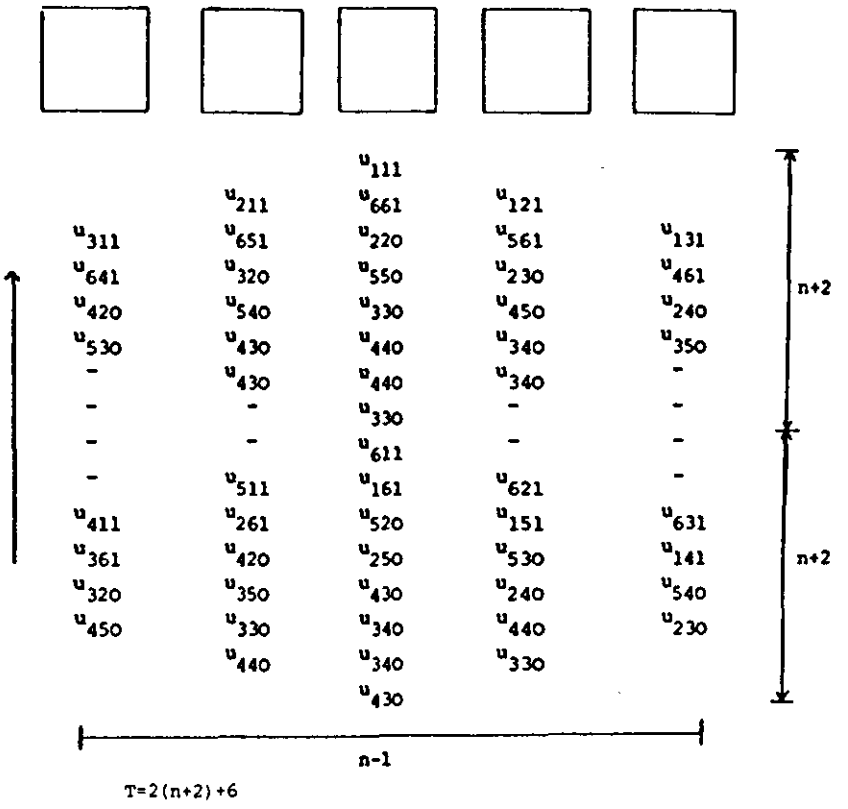


FIGURE 8.3.7: Compressed LAMP array input (n=6)

molecule on subsequent passes. Suppose the molecules are chosen in a strictly clockwise (or anticlockwise) sequence the data re-ordering is simplified as the points needed to start the next pass begin to output after only (m+4) cell cycles. Further analysis reveals that the current level has at most m cycles left to run and so data can be interleaved. At the end of the current level the interleaved next level will have computed (m-3) cycles. We conclude that every new level can be started after a delay of only (m+1) cycles reducing the array operation time to

$$T = (t_z+1)(m+1)+3+3 \text{ cell cycles} \qquad (8.3.20)$$

or $$T = 2(t_z+1)(m+1)+12 \text{ ips cycles.}$$

The data pipelining is shown in Fig.(8.3.8). In contrast the cascaded scheme is again restricted to $\tilde{t}_{z+1}$ producing a good area comparison but requires a re-ordering of the solution vector 'on-the-fly' and is not

possible. Even if interleaving was possible $T=O(t_z m^2)$ making the LAMP scheme superior anyway.

Finally we remark that like the 1-D scheme averaging cannot be implemented by either 2-D arrays, because we must wait for the results of two passes over the same level. Although the time for passes can be reduced by interleaving, output elements of the passes to be averaged are physically separated and must be re-ordered by the host machine to produce nearest neighbour relationships. We conclude that marching and alternating techniques can be adequately implemented by arrays, averaging is not such a good scheme from a systolic point of view.

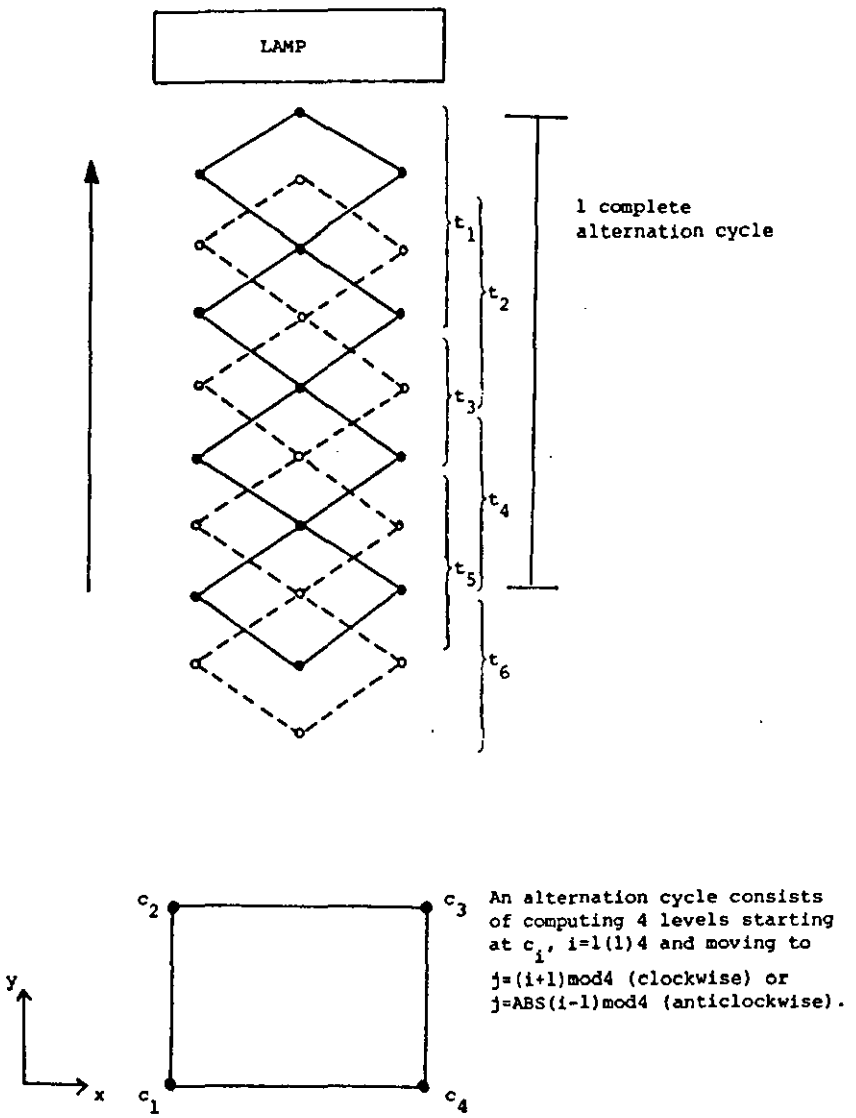


FIGURE 8.3.8: Alternating of successive levels using the LAMP array

8.4 A GENERIC 1-D GROUP EXPLICIT ARRAY

The idea behind the application of systolic arrays to asymmetric approximations is that they are:

(i) Computationally simple

(ii) The rapid evaluation of levels can be used to offset the improved accuracy of implicit schemes by reducing Δt and evaluating more intermediate levels.

Essentially we improve on more accurate slower formulas by balancing accuracy and computation speed of the systolic array.

The above arguments assumed that m, the division along the x-direction remains constant while t_z varies, making the marching arrays attractive. But we can also vary the parameters r and h too. Suppose r is fixed, halving the step size h gives,

$$r = \frac{4\tau}{h^2} \Rightarrow \tau = \frac{\ell}{4} \quad . \tag{8.4.1}$$

That is, introducing twice as many x-points produces four times as many time levels. Thus, when $t_z=n$ the number of initial points m is doubled to 2m producing $t_z=4n$ levels, requiring a doubling in marching cells from m-1 to 2(m-1) in the 1-D case, and 2m-3 to 4m-6 in the 2-D case. The computation time of the marching processors is doubled, but if we assume fixed hardware for the cascaded scheme the time is quadrupled. Furthermore h=1/m, ℓ=1/z, by definition, hence,

$$r = \frac{(1/z)}{(1/m^2)} = m^2/z \text{ for } r \lesssim 2 \Rightarrow z = \frac{m^2}{2} \quad . \tag{8.4.2}$$

Hence for practical purposes where most explicit formulas remain stable the number of levels $t_z=O(m^2)$ making the marching processors extremely attractive compared with the cascaded scheme with fixed array size $\bar{t}_z$.

However if r is varied (which occurs for unconditionally stable

formulas) with either the number of levels or number of points increased it follows:-

that with $r = \frac{\ell}{h^2}$ reducing r yields $\frac{1}{\alpha}r = \frac{1}{\alpha}\frac{\ell}{h^2}$ for $\alpha>1$.

Thus fixing h implies that ℓ is reduced creating more intermediate levels, if ℓ is fixed $\bar{h}=\sqrt{\alpha}h$ is increased, requiring smaller arrays hence faster times. This implies that the cascaded scheme loses the tradeoff again. Alternatively when r is increased $\alpha r=\alpha\frac{\ell}{h^2}$ for $\alpha>1$ producing $\bar{\ell}=\alpha\ell$ (reducing the levels) or $\bar{h} = \frac{h}{\sqrt{\alpha}}$ (increasing the points) which favours the cascaded scheme.

It is clear that to produce further benefits from a geometric approach to P.D.E. solution we must select new methods (and hence new arrays) which solve at least some of the following problems.

(i) Modify computation time:

a) increasing r to reduce the number of levels calculated

b) admit more parallelism to speed-up the arrays

c) simplify basic cells

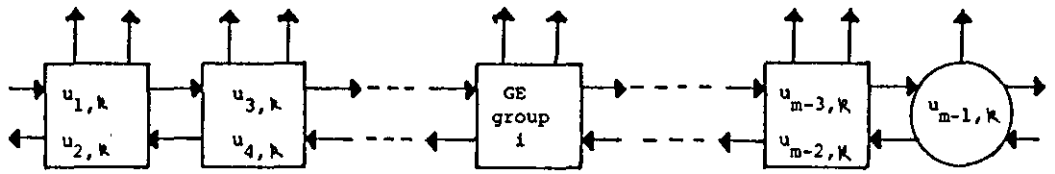
(ii) Control the output of results easily

(iii) Control array size: by attempting to divorce size from both t_z and m.

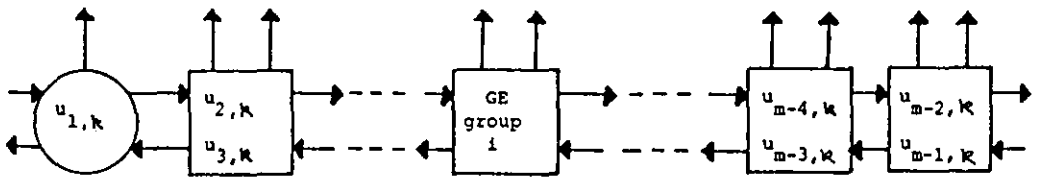
To develop improvements along the lines of (i) and (ii) we consider the Group Explicit (GE) techniques. The GE method has a certain appeal because it allows unconditional stability in the form of SAGE and DAGE schemes, and provides 2×2 decoupled linear systems with weakened computational relationships which admit parallelism. From a geometric viewpoint the methods provide molecules which can be applied simultaneously on the grid, simplifying alternation and averaging schemes.

From the calculations in (8.1.27), (8.1.28), (8.1.31) and (8.1.32) a simple and intuitive basic GE-cell solving the 2×2 system (81.23)

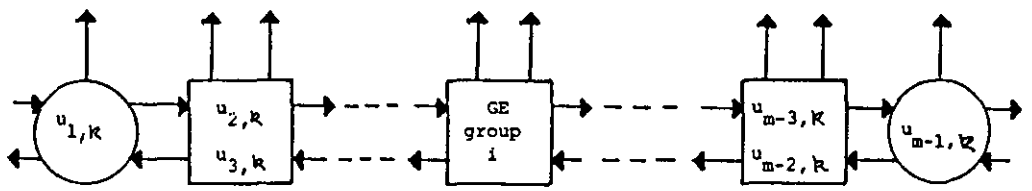
GER ARRAY:



GEL ARRAY:



GEU ARRAY:



GEC ARRAY:

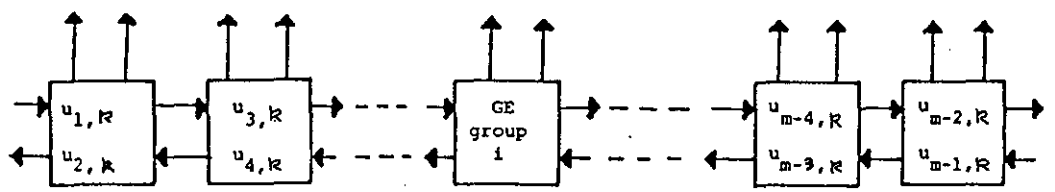
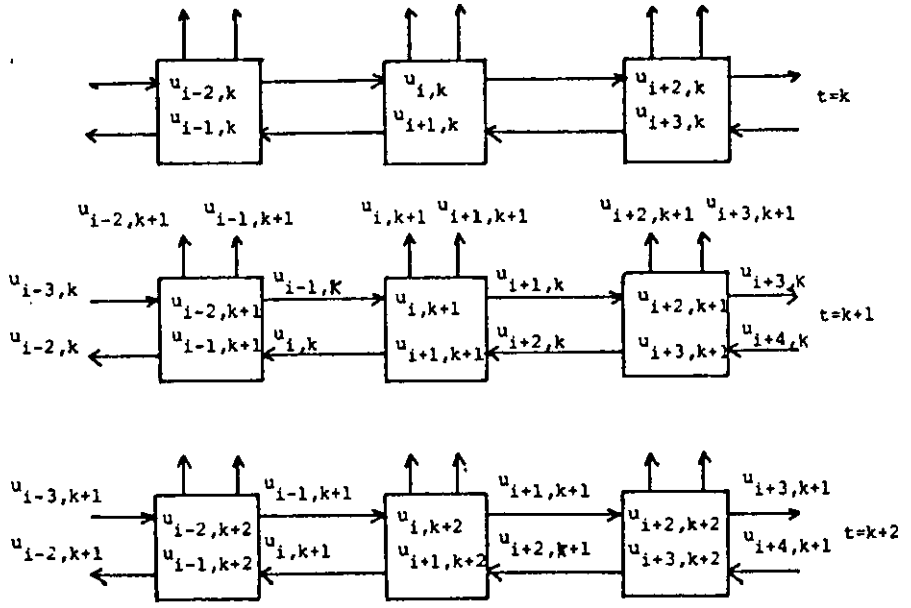


FIGURE 8.4.1: Basic GE array formats

using the unified molecule (8.1.21) suggests itself. Ungrouped points naturally represent a second type of boundary cell, using (8.1.12a) on the right and (8.1.12b) on the left. Using both cells linear arrays for GEL, GER, GEU and GEC as shown in Fig.(8.4.1) are obvious, with the general dataflow of the form,



DATAFLOW FOR EDDY WAVEFRONT

which produces a "one time level, one cycle" approach to generating the solution region.

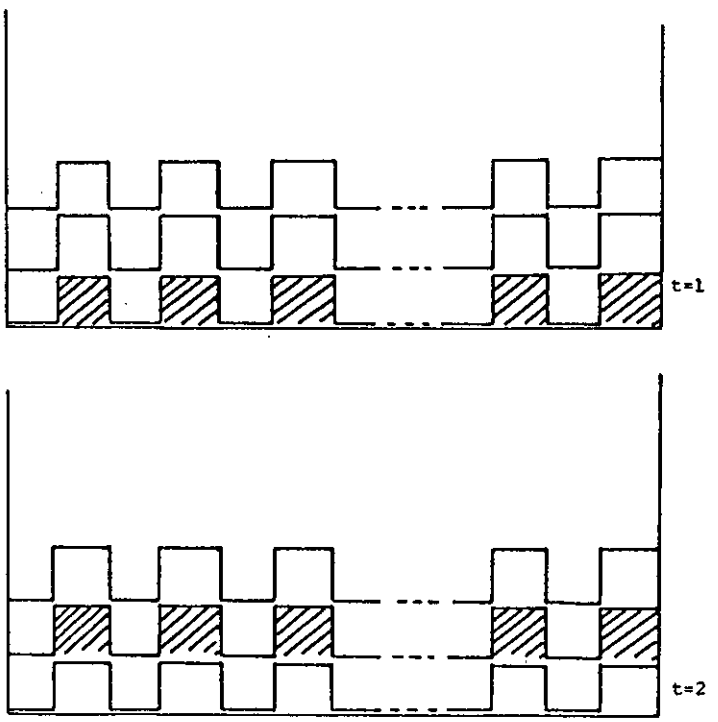
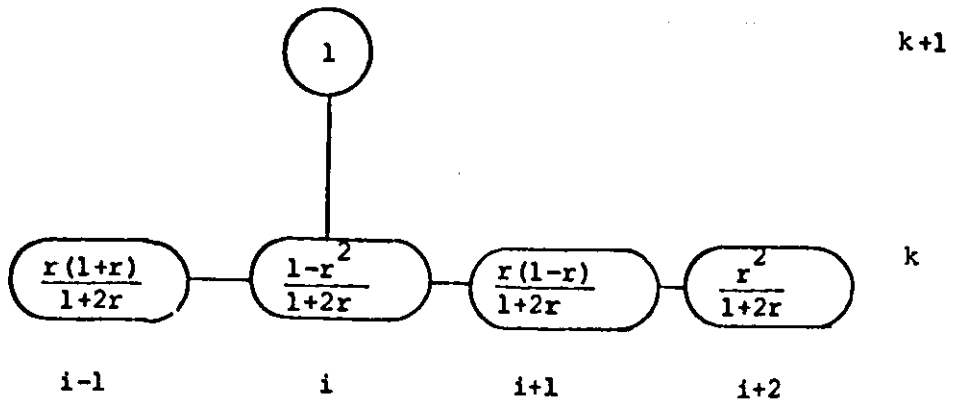
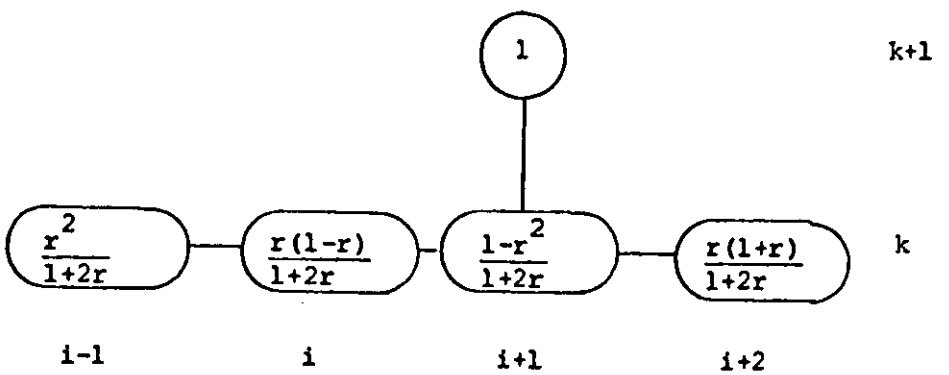


FIGURE 8.4.2: One time level one cycle table generation

From the dataflow and GE equations (8.1.23)-(8.1.25) and the partitioning of the unified molecule (8.1.21) into the explicit form,



(8.4.3)



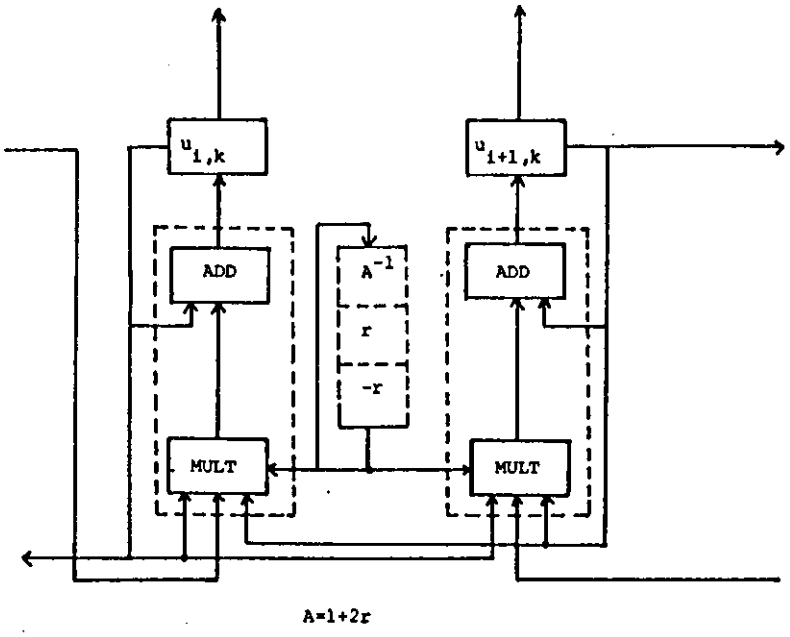
the GE-cell computation

$$
\begin{array}{l|l}
t_1 = u_{ik} - ru_{ik} & p_1 = u_{i+1,k} - ru_{i+1,k} \\
t_2 = t_1 + ru_{i-1,k} & p_2 = p_1 + ru_{i+2,k} \\
t_3 = t_2 + rt_2 & p_3 = p_2 + rp_2 \\
t_4 = t_3 + rp_2 & p_4 = p_3 + rt_2 \\
t_5 = t_4/|A| \equiv u_{i,k+1} & p_5 = p_4/|A| \equiv u_{i+1,k+1}
\end{array}
\qquad (8.4.4)
$$

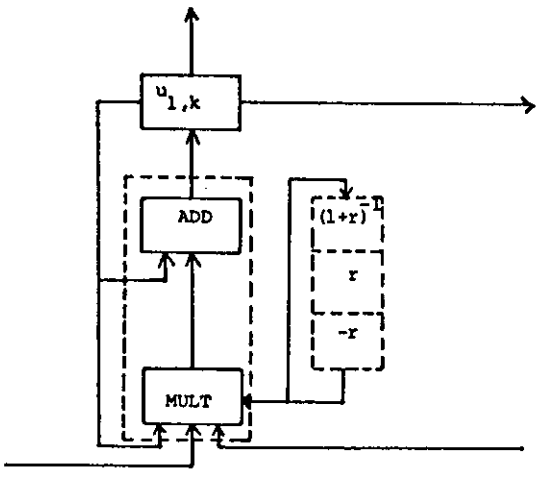
where $|A| = 1+2r$, is defined, and the symmetrical cell structure of Fig. (8.4.3a) is apparent. Likewise from (8.1.2) boundary cell computation is arranged as

RIGHT BOUNDARY:

$$
\begin{aligned}
t_1 &= u_{m-1,k} - ru_{m-1,k} \\
t_2 &= t_1 + ru_{m-2,k} \\
t_3 &= t_2 + ru_{m,k+1} \\
u_{m-1,k+1} &= \frac{1}{1+r}\, t_3
\end{aligned}
\qquad (8.4.5)
$$



a) GE-cell



b) Boundary cell

FIGURE 8.4.3: GE cells structure

LEFT BOUNDARY:

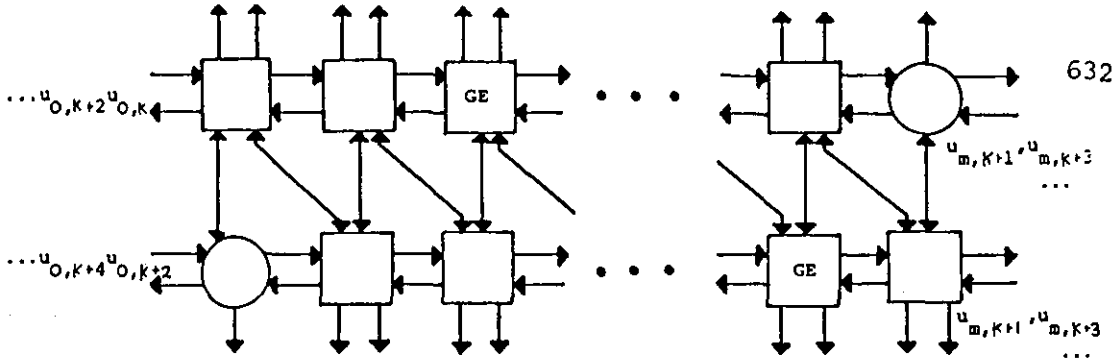
$$\begin{aligned} t_1 &= u_{1,k} - ru_{1,k} \\ t_2 &= t_1 + ru_{2,k} \\ t_3 &= t_2 + ru_{0,k-1} \\ u_{1,k+1} &= \frac{1}{1+r}\, t_3 \end{aligned} \tag{8.4.6}$$

yielding the cell structure Fig.(8.4.3b). Observe that only the parameters $-r$, r, and A^{-1} must be loaded into GE cells and $-r, r, (1+r)^{-1}$ into boundary cells, along with the associated initial values before computation begins. Cell operation is trivial when we assume a five state control program to select the operands for the accumulating ips cell at each step and to circulate the parameter list correctly in each cell. Each level is then computed after 5 ips cycles which denotes a single GE cycle, and an array requires at most

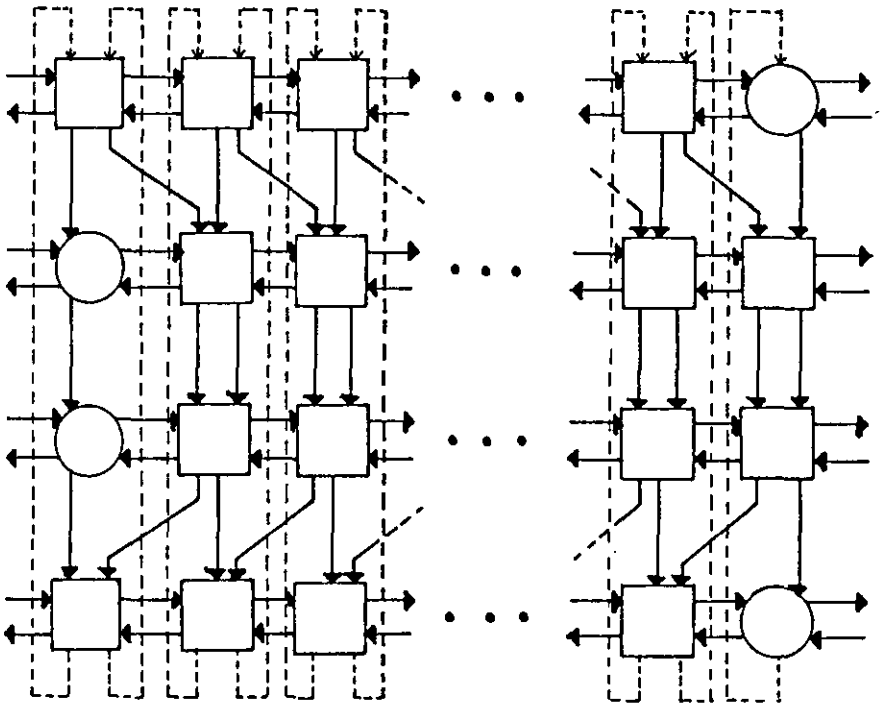
$$T = 5t_z + (m-1) + 3 \ , \tag{8.4.7}$$

ips cycles to load the parameters, initial conditions (in pipelined fashion), and compute t_z levels. The timing is reduced to $5t_z+4$ cycles if parallel loading is adopted. A typical array requires $\lfloor (m-1)/2 \rfloor$ GE cells and a boundary cell, except for the GEU which uses two boundary cells and $\lfloor (m-1)/2 \rfloor - 1$ GE cells, and the GEC which requires no boundary cells at all. Consequently, as a GE cell requires 2 ips cells and a trivial switching program a full GE array uses at most m ips cell equivalents, giving good area savings over the asymmetric marching processor.

In order to incorporate unconditional stability (with $r \geq 1$) we must use the Alternating Group Explicit (AGE) form of calculation. Using the basic GER and GEL components in Fig.(8.4.1) the SAGE and DAGE methods of (8.1.29) and (8.1.30) can be implemented as shown by Fig.(8.4.4)



a) SAGE systolic array



b) DAGE systolic array

FIGURE 8.4.4: Alternating Group Explicit schemes

(similar arrays exist with (8.1.33) and (8.1.34) using GEU and GEC arrays). The SAGE scheme uses two tiers, the first a GER array, the second a GEL array, which are connected in a straight forward manner to ensure data is shuffled right for the 2nd tier and left for the first tier (the lines are bi-directional). Alternating the use of the arrays then creates ungrouped points at the right and left positions respectively on alternate GE cell cycles. Thus, tiers accept the boundary values associated with even k on the left side, and k odd on the rightside. Consequently every alternate boundary value is not required. Finally,

the two tiers (like the LR/RL scheme) operate in mutually exclusive fashion such that tier 1 produces only the odd time levels, tier 2 only the even numbered levels. If we compare this with the asymmetric alternating scheme Fig.(8.3.3a) we have improved efficiency because in the SAGE scheme all the cells of a single tier operate every alternate cycle. However, further improvements are still possible by unifying the GER and GEL schemes. This is achieved by merging both arrays and using a logical toggle in each cell which selects the type of array used. For example, if we set

$$\text{Toggle} = \begin{cases} 0 \text{ select tier 1 = GER} \\ 1 \text{ select tier 2 = GEL} \end{cases}$$

alternating is controlled by Toggle=NOT Toggle performed at the end of every cycle. Analysis of the dataflow in the SAGE schemes also indicates that different tier calculations can be performed by the same GE-cell structure using simple shifts of data right or left. For instance, the operation of a 4 cell merged array is,

$u_{1,k+1}$	$u_{3,k+1}$	$u_{5,k+1}$	$u_{7,k+1}$	
$u_{2,k+1}$	$u_{4,k+1}$	$u_{6,k+1}$	δ	GER
δ	$u_{2,k+1}$	$u_{4,k+1}$	$u_{6,k+1}$	SHIFT RIGHT 1
$u_{1,k+1}$	$u_{3,k+1}$	$u_{5,k+1}$	$u_{7,k+1}$	
δ	$u_{2,k+2}$	$u_{4,k+2}$	$u_{6,k+2}$	GEL
$u_{1,k+2}$	$u_{3,k+2}$	$u_{5,k+2}$	$u_{7,k+2}$	
$u_{1,k+2}$	$u_{3,k+2}$	$u_{5,k+2}$	$u_{7,k+2}$	SHIFT LEFT 1
$u_{2,k+2}$	$u_{4,k+2}$	$u_{6,k+2}$	δ	

where the shuffling of data is achieved by the calculations

$$\left.\begin{array}{l|l|l} t_6 = 0+(1*u_{i-1,k+1}) & p_6 = 0+(1*t_5) & \text{shift right} \\ t_6 = 0+(1*p_5) & p_6 = 0+(1*u_{i+2,k+1}) & \text{shift left} \end{array}\right\} \quad (8.4.8)$$

which add an extra ips cycle to each GE-cell and are implemented by adding the constant 1 to the parameter list, and using

$$\text{Toggle} = \begin{cases} 0 \text{ shift left} \\ 1 \text{ shift right} \end{cases}$$

to control the multiplier and operand switching. A significant problem is encountered at the array edges where boundary cells must be unified with GE-cells, as the boundary computation requires only half the hardware. The problem is resolved by using the right portion of the GE-cell for the left boundary, and the left half of the cell for a right boundary. Inserting a dummy calculation $t_3=0+1*t_3$ in (8.4.5) and (8.4.6) ensures that the boundary cell uses as many cycles as an ordinary GE-cell. The unused portion of the GE-cell is then loaded with appropriate boundary values, and the computation preserved by modifying the cell control. For instance, a boundary cell on the left is simulated by the GE-computation,

$$\begin{array}{l|l} t_1 = u_{0,k+1} & p_1 = u_{1,k} - ru_{1,k} \\ t_2 = u_{0,k+1} & p_2 = p_1 + ru_{2,k} \\ t_3 = u_{0,k+1} & p_3 = 0+1*p_2 \\ t_4 = u_{0,k+1} & p_4 = p_3 + rt_2 \\ t_5 = u_{0,k+1} & p_5 = \frac{1}{1+r} p_4 \end{array} \qquad (8.4.9)$$

(a similar formula is available for the right boundary). Observe that we require the parameters $-r, r, A, (1+r)^{-1}$, and 1 in the boundary cells with control switched by the toggle from (8.4.8) to (8.4.4) to achieve alternation.

The same approach for SAGE is applied to the DAGE scheme producing the intuitive array of Fig.(8.4.4b). This time we have a four tier array arranged as GER, GEL, GEL, GER and the array becomes non-planar by

incorporating a feedback loop to wrap tier 4 to tier 1. In addition, it is no longer possible to output all result levels directly. Clearly the property of mutual exclusive tier operation still applies and some compaction is possible. From the DAGE array structure it follows immediately that the middle two tiers can be combined by performing two GEL steps sequentially, yielding,

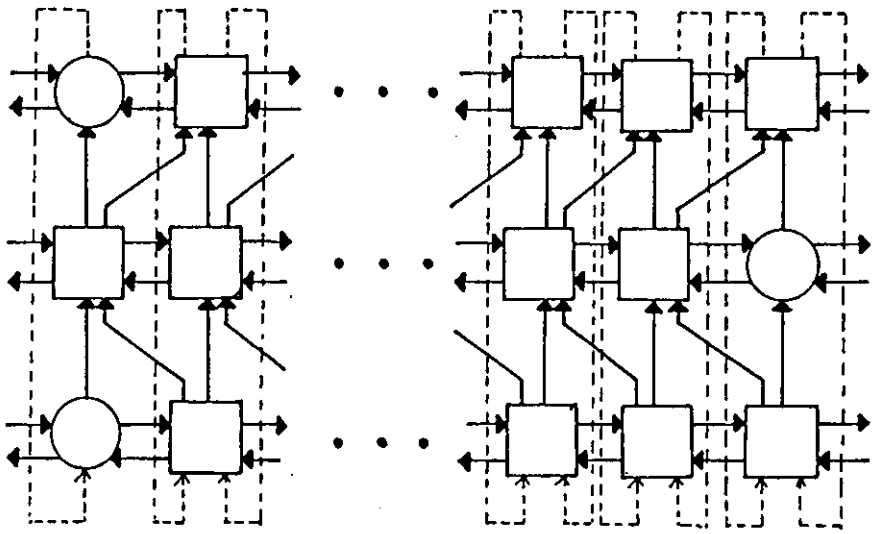


FIGURE 8.4.5: Merge of DAGE array

The new tiers 1 and 2 correspond to a SAGE array which can be merged by the preceding SAGE discussion. A single unified array is derived by observing that a DAGE scheme is a cycle of the form,

(i) compute GER for two GE-cycles

(ii) shift right

(iii) compute GEL for two GE-cycles

(iv) shift left

where the array is started on the second GE-cycle of (i) or (iii), by using a two-bit toggle in the merged SAGE array such that,

$$\text{toggle} = \begin{cases} 0 \quad 0 & \text{GER} \\ 0 \quad 1 & \text{GER and shift right} \\ 1 \quad 0 & \text{GEL} \\ 1 \quad 1 & \text{GEL and shift left} \end{cases}$$

with the shift commands implemented by (8.4.8) and no shift to the left side of (8.4.9).

In a soft-systolic frame these designs are simple to simulate and results from OCCAM programs given in the Appendix (where a GE-cell cycle is constructed from sequential execution of the control programs) are shown in Fig.(8.4.8). However the designs are simple enough to indicate a hard/hybrid systolic frame implementation. So far we have considered only the simple parabolic form of (8.1.1), a more general is given by,

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + g(x,t) \ , \ 0 \leq x \leq 1, \ t \geq 0 \ , \tag{8.4.10}$$

where $g(x,t)$ is a given function which must be calculated by the host for all levels before array operation. By analogous reasoning to the derivation of (8.1.23), a similar formula incorporating a modified function $\bar{g}(x,t)$ suitably multiplied by terms involving r is produced. Modifying the GE-cell (8.4.4) by replacing t_5 and p_5 by,

$$t_5 = t_4 * |A|^{-1} + \bar{g}(x,t), \quad p_5 = p_4 * |A|^{-1} + \bar{g}(x+h,t) \tag{8.4.11}$$

generalises the unified GE-array (with boundary cells ungraded similarly). It follows that the unified array with control programs of six instructions (incorporating shifting for a cell can simulate any of the 1-D GE methods (8.1.27)-(8.1.34) with an array of at most $\frac{1}{2}m$ GE-cells. For a chip based implementation we also have to consider the problem of the large host-array interface - requiring m inputs

and outputs (2 in and 2 out for each cell) to supply the $\bar{g}(x,t)$ and read the resulting level t values at each GE step. This is achieved by buffering data in a similar manner to the adaptive extrapolation table generator in Fig.(7.2.9), viz.

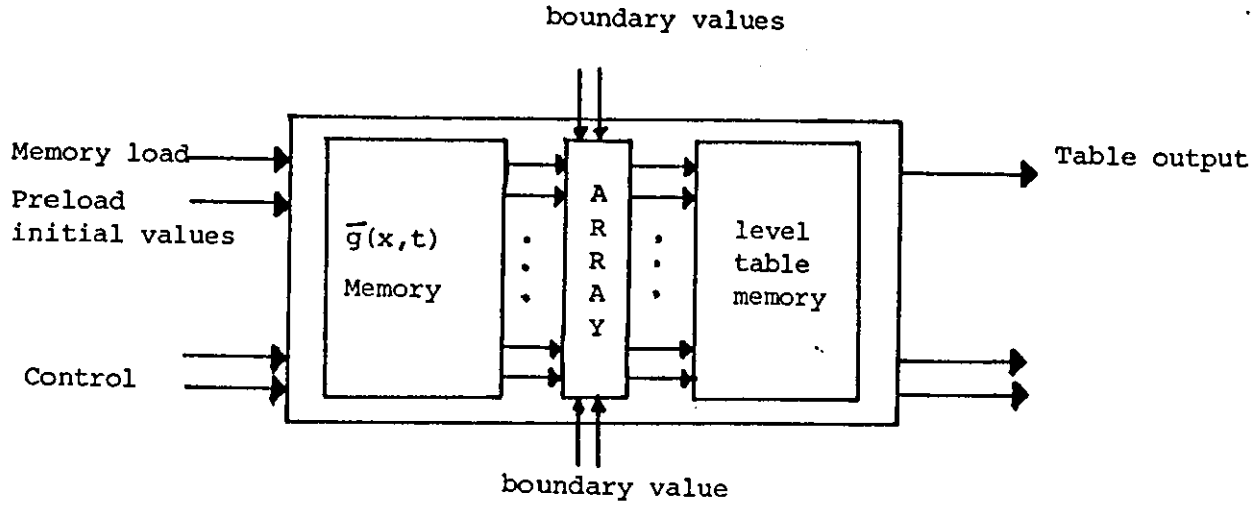


FIGURE 8.4.6: GE-array chip scheme

The 'freeze' command used in the former table generator to evaluate more starting values, can be used simply to unload the computed levels and refill the $\bar{g}(x,t)$ memory allowing the computation of any number of levels. The time of the generic GE-array is then,

$$T = 6t_z + (\bar{t}_z + 1)(\text{freeze-time}) + m + 3 , \qquad (8.4.12)$$

ips cycles, where the freeze-time is the cost of loading/unloading memory buffers, m+3 is the cost of loading starting parameters and initial values, and $\bar{t}_z = t_z/(\text{buffer size})$.

Finally, we remark that the above arrays are applicable only for Dirichlet boundary conditions, where periodic boundary conditions prevail the coefficients matrices for (8.1.27)-(8.1.34) will be different. For example, if the number of intervals along x is even, the number of unknown points is also even, and defining,

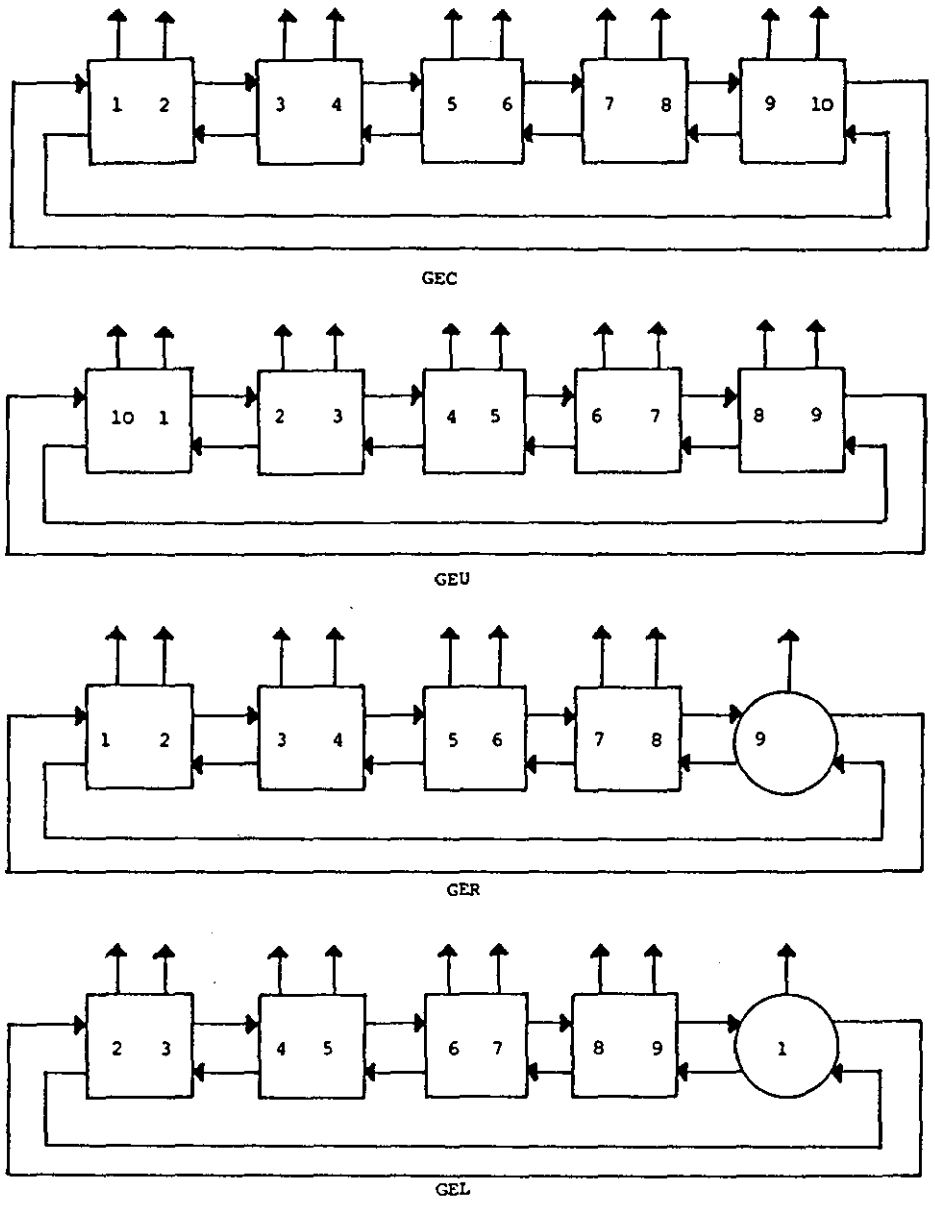


FIGURE 8.4.7: Various arrays for periodic GE methods

$$\hat{\hat{G}}_1 = \begin{bmatrix} 1 & & & & -1 \\ & G^{(1)} & & & \\ & & G^{(2)} & & \\ & & & \ddots & \\ & & & & G^{\frac{1}{2}(m-2)} \\ -1 & & & & 1 \end{bmatrix}$$

using G in (8.1.26) the GEC and GEU schemes are given by,

$$(I+r\hat{G}_2)u_{k+1} = (I-r\hat{\hat{G}}_1)u_k \tag{8.4.13}$$

and

$$(I+r\hat{\hat{G}}_1)u_{k+1} = (I-r\hat{G}_2)u_k \tag{8.4.14}$$

respectively. Thus periodic boundary conditions imply a systolic ring type structure as shown in Fig.(8.4.7), and by analogous reasoning to that above a generic periodic GE-array is easily derived. Finally

Test Example:

Initial conditions

$$u(x,0) = 4x(1-x),\ 0 \leq x \leq 1$$

boundary conditions

$$u(0,t) = u(1,t) = 0,\quad t \geq 0.$$

The exact solution is,

$$u(x,t) = \frac{32}{h^3} \sum_{k=1,3,5\ldots}^{\infty} \frac{1}{k^3} e^{-k^2\pi^2 t} \sin(k\pi x)$$

Results of the OCCAM program are given below in Fig.(8.4.8).

```
ger  r = 0.100000

0.352000  0.632000  0.832000  0.952000  0.992000  0.952000  0.832000  0.632000  0.352727  0.000000
0.344733  0.624067  0.824000  0.944000  0.984000  0.944000  0.824006  0.624067  0.346050  0.000000
0.338077  0.616243  0.816004  0.936001  0.976000  0.936000  0.816027  0.616243  0.339865  0.000000
0.331932  0.608557  0.808027  0.928003  0.968000  0.928003  0.808073  0.608557  0.334093  0.000000

0.326219  0.601023  0.800074  0.920009  0.960001  0.920009  0.800153  0.601023  0.328672  0.000000
0.320875  0.593651  0.792155  0.912022  0.952004  0.912022  0.792276  0.593651  0.323552  0.000000
0.315847  0.586442  0.784281  0.904044  0.944009  0.904043  0.784450  0.586442  0.318693  0.000000
0.311092  0.579395  0.776459  0.896078  0.936018  0.896078  0.776680  0.579395  0.314062  0.000000

0.306576  0.572509  0.768696  0.888129  0.928034  0.888129  0.768974  0.572509  0.309632  0.000000
0.302269  0.565777  0.760997  0.880199  0.920057  0.880199  0.761334  0.565777  0.305381  0.000000
0.298147  0.559194  0.753368  0.872293  0.912092  0.872292  0.753763  0.559194  0.301292  0.000000
0.294189  0.552755  0.745811  0.864412  0.904139  0.864412  0.746269  0.552755  0.297347  0.000000

0.290378  0.546453  0.738328  0.856562  0.896202  0.856561  0.738841  0.546453  0.293534  0.000000
0.286698  0.540282  0.730923  0.848744  0.888284  0.848743  0.731492  0.540282  0.289842  0.000000
0.283138  0.534236  0.723595  0.840961  0.880387  0.840961  0.724219  0.534236  0.286260  0.000000
0.279687  0.528310  0.716346  0.833217  0.872515  0.833217  0.717022  0.528310  0.282780  0.000000

0.276334  0.522497  0.709176  0.825513  0.864669  0.825513  0.709901  0.522497  0.279393  0.000000
0.273073  0.516793  0.702085  0.817852  0.856853  0.817852  0.702856  0.516793  0.276094  0.000000
0.269895  0.511192  0.695072  0.810236  0.849070  0.810235  0.695886  0.511193  0.272877  0.000000
0.266795  0.505691  0.688137  0.802666  0.841321  0.802666  0.688991  0.505691  0.269735  0.000000

0.263767  0.500284  0.681279  0.795145  0.833609  0.795145  0.682170  0.500284  0.266664  0.000000
0.260807  0.494967  0.674497  0.787674  0.825936  0.787674  0.675422  0.494967  0.263660  0.000000
0.257909  0.489737  0.667790  0.780254  0.818305  0.780254  0.668746  0.489737  0.260719  0.000000
0.255070  0.484591  0.661158  0.772886  0.810717  0.772886  0.662142  0.484591  0.257837  0.000000

0.252287  0.479524  0.654598  0.765572  0.803173  0.765572  0.655608  0.479524  0.255011  0.000000
0.249556  0.474533  0.648111  0.758312  0.795677  0.758312  0.649143  0.474533  0.252239  0.000000
0.246874  0.469617  0.641694  0.751107  0.788228  0.751107  0.642747  0.469617  0.249516  0.000000
0.244240  0.464771  0.635348  0.743958  0.780829  0.743958  0.636418  0.464771  0.246842  0.000000
```

All results for Δx=0.1, zeros indicate inactive cells of the array.

```
ger  r = 0.100000

.000000  0.352727  0.632000  0.832000  0.952000  0.992000  0.952000  0.832000  0.632000  0.352000
.000000  0.346050  0.624067  0.824006  0.944000  0.984000  0.944000  0.824000  0.624067  0.344733
.000000  0.339865  0.616243  0.816027  0.936000  0.976000  0.936001  0.816004  0.616243  0.338077
.000000  0.334093  0.608557  0.808073  0.928003  0.968000  0.928003  0.808027  0.608557  0.331931

000000  0.328672  0.601023  0.800153  0.920009  0.960001  0.920009  0.800074  0.601023  0.326219
000000  0.323552  0.593651  0.792276  0.912022  0.952004  0.912022  0.792155  0.593651  0.320875
000000  0.318693  0.586442  0.784450  0.904043  0.944009  0.904044  0.784281  0.586442  0.315847
000000  0.314062  0.579395  0.776680  0.896078  0.936018  0.896078  0.776459  0.579395  0.311092

000000  0.309632  0.572509  0.768974  0.888129  0.928034  0.888129  0.768696  0.572508  0.306576
000000  0.305381  0.565777  0.761334  0.880199  0.920057  0.880199  0.760997  0.565777  0.302269
000000  0.301292  0.559194  0.753763  0.872292  0.912092  0.872293  0.753368  0.559194  0.298147
000000  0.297347  0.552755  0.746269  0.864412  0.904139  0.864412  0.745811  0.552755  0.294189

000000  0.293534  0.546453  0.738841  0.856561  0.896202  0.856562  0.738328  0.546453  0.290378
000000  0.289842  0.540282  0.731492  0.848743  0.888284  0.848744  0.730923  0.540282  0.286698
000000  0.286260  0.534236  0.724219  0.840961  0.880387  0.840961  0.723595  0.534236  0.283138
000000  0.282780  0.528310  0.717022  0.833217  0.872515  0.833217  0.716346  0.528310  0.279687

000000  0.279393  0.522497  0.709901  0.825513  0.864669  0.825513  0.709176  0.522497  0.276334
000000  0.276094  0.516793  0.702856  0.817852  0.856853  0.817852  0.702085  0.516793  0.273073
000000  0.272877  0.511193  0.695886  0.810235  0.849070  0.810236  0.695072  0.511192  0.269895
000000  0.269735  0.505691  0.688991  0.802666  0.841321  0.802666  0.688137  0.505691  0.266795

000000  0.266664  0.500284  0.682170  0.795145  0.833609  0.795145  0.681279  0.500284  0.263767
000000  0.263660  0.494967  0.675422  0.787674  0.825936  0.787674  0.674497  0.494967  0.260807
000000  0.260719  0.489737  0.668746  0.780254  0.818305  0.780254  0.667790  0.489737  0.257909
000000  0.257837  0.484591  0.662142  0.772886  0.810717  0.772886  0.661158  0.484591  0.255070

000000  0.255011  0.479524  0.655608  0.765572  0.803173  0.765572  0.654598  0.479524  0.252287
000000  0.252239  0.474533  0.649143  0.758312  0.795677  0.758312  0.648111  0.474533  0.249556
000000  0.249516  0.469617  0.642747  0.751107  0.788228  0.751107  0.641694  0.469617  0.246874
000000  0.246842  0.464771  0.636418  0.743958  0.780829  0.743958  0.635348  0.464771  0.244240
```

FIGURE 8.4.8: OCCAM program test results

ger r = 0.500000

0.320000	0.600000	0.800000	0.920000	0.960000	0.920000	0.800000	0.600000	0.333333	0.000000
0.295000	0.565000	0.760000	0.880000	0.920000	0.880000	0.761667	0.565000	0.311111	0.000000
0.276250	0.533750	0.721875	0.840625	0.880208	0.840625	0.723139	0.533750	0.292037	0.000000
0.260547	0.505391	0.685964	0.802266	0.841033	0.802266	0.690385	0.505391	0.275262	0.000000
0.246624	0.479326	0.652170	0.765156	0.802818	0.765156	0.657326	0.479326	0.260218	0.000000
0.233921	0.455139	0.620308	0.729428	0.765801	0.729428	0.625874	0.455139	0.246515	0.000000
0.222151	0.432933	0.590196	0.695142	0.730123	0.695142	0.595945	0.432933	0.233885	0.000000
0.211148	0.411292	0.561682	0.662316	0.695860	0.662316	0.567460	0.411292	0.222139	0.000000
0.200802	0.391259	0.534637	0.630938	0.663038	0.630938	0.540345	0.391259	0.211144	0.000000
0.191038	0.372311	0.508958	0.600978	0.631651	0.600978	0.514531	0.372311	0.200801	0.000000
0.181798	0.354356	0.484555	0.572395	0.601675	0.572395	0.489955	0.354356	0.191037	0.000000
0.173038	0.337316	0.461350	0.545140	0.573070	0.545140	0.466555	0.337316	0.181798	0.000000
0.164723	0.321130	0.439276	0.519162	0.545790	0.519162	0.444275	0.321130	0.173038	0.000000
0.156822	0.305742	0.418271	0.494408	0.519787	0.494408	0.423060	0.305742	0.164723	0.000000
0.149310	0.291108	0.398279	0.470825	0.495006	0.470825	0.402858	0.291108	0.156822	0.000000
0.142165	0.277184	0.379249	0.448360	0.471397	0.448360	0.383622	0.277184	0.149310	0.000000
0.135366	0.263933	0.361132	0.426963	0.448907	0.426963	0.365305	0.263933	0.142165	0.000000
0.128895	0.251320	0.343883	0.406584	0.427485	0.406584	0.347863	0.251320	0.135366	0.000000
0.122736	0.239313	0.327460	0.387176	0.407082	0.387176	0.331253	0.239313	0.128895	0.000000
0.116873	0.227882	0.311822	0.368694	0.387651	0.368693	0.315437	0.227882	0.122736	0.000000
0.111290	0.216998	0.296932	0.351092	0.369145	0.351092	0.300376	0.216998	0.116873	0.000000
0.105975	0.206635	0.282753	0.334330	0.351523	0.334330	0.286034	0.206635	0.111290	0.000000
0.100914	0.196768	0.269252	0.318368	0.334740	0.318368	0.272377	0.196768	0.105975	0.000000
0.096095	0.187372	0.256396	0.303168	0.318759	0.303168	0.259373	0.187372	0.100914	0.000000
0.091507	0.178425	0.244154	0.288694	0.303540	0.288694	0.246989	0.178425	0.096095	0.000000
0.087137	0.169905	0.232496	0.274910	0.289048	0.274910	0.235196	0.169905	0.091507	0.000000
0.082977	0.161793	0.221395	0.261785	0.275248	0.261785	0.223966	0.161793	0.087137	0.000000
0.079015	0.154068	0.210825	0.249286	0.262106	0.249286	0.213273	0.154068	0.082977	0.000000

ger r = 0.500000

0.000000	0.333333	0.600000	0.800000	0.920000	0.960000	0.920000	0.800000	0.600000	0.320000
0.000000	0.311111	0.565000	0.761667	0.880000	0.920000	0.880000	0.760000	0.565000	0.295000
0.000000	0.292037	0.533750	0.725139	0.840625	0.880208	0.840625	0.721875	0.533750	0.276250
0.000000	0.275262	0.505391	0.690385	0.802266	0.841033	0.802266	0.685964	0.505391	0.260547
0.000000	0.260218	0.479326	0.657326	0.765156	0.802818	0.765156	0.652170	0.479326	0.246624
0.000000	0.246515	0.455139	0.625874	0.729428	0.765801	0.729428	0.620308	0.455139	0.233921
0.000000	0.233885	0.432933	0.595945	0.695142	0.730123	0.695142	0.590196	0.432933	0.222151
0.000000	0.222139	0.411292	0.567460	0.662316	0.695860	0.662316	0.561682	0.411292	0.211148
0.000000	0.211144	0.391259	0.540345	0.630938	0.663038	0.630938	0.534637	0.391259	0.200802
0.000000	0.200801	0.372311	0.514531	0.600978	0.631651	0.600978	0.508958	0.372311	0.191038
0.000000	0.191037	0.354356	0.489955	0.572395	0.601675	0.572395	0.484555	0.354356	0.181798
0.000000	0.181798	0.337316	0.466555	0.545140	0.573070	0.545140	0.461350	0.337316	0.173038
0.000000	0.173038	0.321130	0.444275	0.519162	0.545790	0.519162	0.439276	0.321130	0.164723
0.000000	0.164723	0.305742	0.423060	0.494408	0.519787	0.494408	0.418271	0.305742	0.156822
0.000000	0.156822	0.291108	0.402858	0.470825	0.495006	0.470825	0.398279	0.291108	0.149310
0.000000	0.149310	0.277184	0.383622	0.448360	0.471397	0.448360	0.379249	0.277184	0.142165
0.000000	0.142165	0.263933	0.365305	0.426963	0.448907	0.426963	0.361132	0.263933	0.135366
0.000000	0.135366	0.251320	0.347863	0.406584	0.427485	0.406584	0.343883	0.251320	0.128895
0.000000	0.128895	0.239313	0.331253	0.387176	0.407082	0.387176	0.327460	0.239313	0.122736
0.000000	0.122736	0.227882	0.315437	0.368693	0.387651	0.368694	0.311822	0.227882	0.116873
0.000000	0.116873	0.216998	0.300376	0.351092	0.369145	0.351092	0.296932	0.216998	0.111290
0.000000	0.111290	0.206635	0.286034	0.334330	0.351523	0.334330	0.282753	0.206635	0.105975
0.000000	0.105975	0.196768	0.272377	0.318368	0.334740	0.318368	0.269252	0.196768	0.100914
0.000000	0.100914	0.187372	0.259373	0.303168	0.318759	0.303168	0.256396	0.187372	0.096095
0.000000	0.096095	0.178425	0.246989	0.288694	0.303540	0.288694	0.244154	0.178425	0.091507
0.000000	0.091507	0.169905	0.235196	0.274910	0.289048	0.274910	0.232496	0.169905	0.087137
0.000000	0.087137	0.161793	0.223966	0.261785	0.275248	0.261785	0.221395	0.161793	0.082977
0.000000	0.082977	0.154068	0.213273	0.249286	0.262106	0.249286	0.210825	0.154068	0.079015

FIGURE 8.4.8: (cont.)

stage 1 r = 0.100000

0.352000	0.632000	0.832000	0.952000	0.992000	0.952000	0.832000	0.632000	0.352727	0.000000
0.000000	0.345455	0.624000	0.824000	0.944000	0.984000	0.944000	0.824000	0.624121	0.345333
0.338667	0.616242	0.816000	0.936000	0.976000	0.936000	0.816020	0.616222	0.339284	0.000000
0.000000	0.333113	0.608444	0.808040	0.928000	0.968000	0.928003	0.808037	0.608631	0.332926
0.327185	0.601020	0.800074	0.920007	0.960000	0.920006	0.800119	0.600975	0.327724	0.000000
0.000000	0.322335	0.593306	0.792198	0.912012	0.952002	0.912022	0.792188	0.593728	0.322113
0.317041	0.586436	0.784302	0.904038	0.944006	0.904036	0.784371	0.586367	0.317522	0.000000
0.000000	0.312710	0.579228	0.776544	0.896059	0.936014	0.896078	0.776526	0.579468	0.312470
0.307899	0.572499	0.768749	0.888118	0.928027	0.888114	0.768838	0.572411	0.308334	0.000000
0.000000	0.303943	0.565593	0.761132	0.880169	0.920048	0.880196	0.761105	0.565841	0.303715
0.299531	0.559182	0.753460	0.872275	0.912079	0.872268	0.753565	0.559077	0.299934	0.000000
0.000000	0.295906	0.552961	0.745998	0.864369	0.904122	0.864404	0.745961	0.552810	0.295656
0.291781	0.546439	0.738462	0.856537	0.896181	0.856526	0.738579	0.546321	0.292156	0.000000
0.000000	0.288606	0.540081	0.731161	0.848686	0.888258	0.848730	0.731115	0.540329	0.288158
0.284534	0.534220	0.723769	0.840928	0.880357	0.840913	0.723896	0.534093	0.284986	0.000000
0.000000	0.281366	0.528104	0.716632	0.833145	0.872479	0.833195	0.716577	0.528350	0.281121
0.277707	0.522480	0.709387	0.825470	0.864628	0.825451	0.709521	0.522345	0.278040	0.000000
0.000000	0.274713	0.516985	0.702413	0.817764	0.856806	0.817822	0.702351	0.516927	0.274471
0.271235	0.511174	0.695317	0.810183	0.849016	0.810160	0.695454	0.511033	0.271552	0.000000
0.000000	0.268390	0.505482	0.688501	0.802566	0.841261	0.802626	0.688433	0.505719	0.268153
0.265071	0.500263	0.681553	0.795083	0.833543	0.795056	0.681695	0.500120	0.265372	0.000000
0.000000	0.262355	0.494759	0.674892	0.787560	0.825864	0.787626	0.674818	0.494990	0.262122
0.259174	0.489718	0.668088	0.780181	0.818226	0.780151	0.668232	0.489570	0.259463	0.000000
0.000000	0.256571	0.484383	0.661578	0.772759	0.810631	0.772825	0.661499	0.484609	0.256343
0.253513	0.479503	0.654916	0.765489	0.803081	0.765455	0.655061	0.479393	0.253790	0.000000
0.000000	0.251011	0.474326	0.648351	0.758172	0.795578	0.758241	0.648467	0.474548	0.250787
0.248063	0.469596	0.642027	0.751014	0.788123	0.750978	0.642172	0.469465	0.248330	0.000000
0.000000	0.245691	0.464566	0.635802	0.743806	0.780717	0.743876	0.635715	0.464783	0.245432

stage 1 r = 0.500000

0.320000	0.600000	0.800000	0.920000	0.960000	0.920000	0.800000	0.600000	0.333333	0.000000
0.000000	0.306667	0.560000	0.760000	0.880000	0.920000	0.880000	0.760000	0.566667	0.300000
0.280000	0.533333	0.720000	0.840000	0.880000	0.840000	0.723333	0.530000	0.268889	0.000000
0.000000	0.271111	0.500000	0.686667	0.800000	0.840000	0.801667	0.685000	0.506111	0.265000
0.250000	0.478889	0.650000	0.763333	0.800834	0.762500	0.653889	0.475000	0.257037	0.000000
0.000000	0.242963	0.450000	0.621111	0.725417	0.762917	0.727361	0.618750	0.455463	0.237500
0.225000	0.432037	0.587709	0.692014	0.726389	0.690834	0.591412	0.428125	0.230988	0.000000
0.000000	0.219012	0.406354	0.562026	0.657049	0.691424	0.658901	0.559479	0.411200	0.214063
0.203177	0.390519	0.531702	0.626725	0.657975	0.625452	0.535050	0.386771	0.206421	0.000000
0.000000	0.197899	0.367639	0.508622	0.594838	0.626088	0.596513	0.506111	0.371736	0.193385
0.183720	0.353260	0.481139	0.567355	0.595675	0.566100	0.484124	0.349748	0.188374	0.000000
0.000000	0.178993	0.332429	0.460308	0.538407	0.566728	0.539900	0.457924	0.336249	0.174874
0.166215	0.319651	0.435418	0.513518	0.539154	0.512326	0.438074	0.316399	0.170374	0.000000
0.000000	0.161955	0.300816	0.416384	0.487284	0.512922	0.488614	0.416363	0.304224	0.158200
0.150408	0.289270	0.394051	0.464753	0.487950	0.463642	0.396419	0.286281	0.154141	0.000000
0.000000	0.146539	0.272230	0.377011	0.441001	0.464198	0.442185	0.374962	0.275290	0.143141
0.136115	0.261705	0.356615	0.420604	0.441593	0.419580	0.358732	0.259051	0.139474	0.000000
0.000000	0.132633	0.246365	0.341195	0.399104	0.420092	0.400163	0.339315	0.249103	0.129526
0.123183	0.236914	0.322734	0.380644	0.399633	0.379704	0.324633	0.234420	0.126210	0.000000
0.000000	0.120032	0.222939	0.308779	0.361184	0.380174	0.362133	0.307062	0.225421	0.117210
0.111479	0.214406	0.292071	0.344476	0.361638	0.343618	0.293777	0.212136	0.114210	0.000000
0.000000	0.108628	0.201773	0.279441	0.326865	0.344047	0.327718	0.277877	0.203996	0.106068
0.100888	0.194035	0.264320	0.311744	0.327291	0.310962	0.265856	0.191973	0.103356	0.000000
0.000000	0.098307	0.182604	0.252889	0.295806	0.311353	0.296574	0.251467	0.184605	0.095986
0.091302	0.175598	0.239205	0.282121	0.296190	0.281410	0.240589	0.173727	0.093530	0.000000
0.000000	0.088967	0.165253	0.228860	0.267697	0.281766	0.268389	0.227549	0.167060	0.086863
0.082627	0.158913	0.216475	0.255313	0.268043	0.254667	0.217725	0.157216	0.084441	0.000000
0.000000	0.080513	0.149551	0.207113	0.242259	0.254990	0.242884	0.205942	0.151183	0.078608

FIGURE(8.4.8)cont.

dage 1 r = 0.100000 (a)

0.352000	0.632000	0.832000	0.952000	0.992000	0.952000	0.832000	0.632000	0.352727	0.000000
0.000000	0.345455	0.624000	0.824000	0.944000	0.984000	0.944000	0.824000	0.624121	0.345333
0.000000	0.339372	0.616133	0.816012	0.936000	0.976000	0.936001	0.816011	0.616333	0.338576
0.332992	0.608564	0.608022	0.928002	0.968000	0.928002	0.808056	0.608512	0.333047	0.000000
0.327094	0.601108	0.800070	0.920008	0.960001	0.920004	0.800126	0.600089	0.327812	0.000000
0.000000	0.322269	0.593370	0.792201	0.912013	0.952002	0.912022	0.792186	0.593664	0.322179
0.000000	0.317635	0.586252	0.784370	0.904029	0.944007	0.904046	0.784308	0.586354	0.316928
0.312554	0.579381	0.776514	0.896071	0.936014	0.896067	0.776558	0.579315	0.312625	0.000000
0.307782	0.572611	0.768739	0.888127	0.928028	0.888108	0.768853	0.572301	0.308449	0.000000
0.000000	0.303877	0.565674	0.781132	0.880175	0.920049	0.880192	0.761107	0.565760	0.303800
0.000000	0.300052	0.558956	0.753575	0.872254	0.912080	0.872293	0.753456	0.559305	0.299410
0.295744	0.552720	0.745956	0.864393	0.904124	0.864384	0.746004	0.552651	0.295817	0.000000
0.291659	0.546556	0.738444	0.856555	0.896183	0.856513	0.738603	0.546207	0.292273	0.000000
0.000000	0.288317	0.540144	0.731156	0.888700	0.888260	0.848720	0.731123	0.540247	0.288246
0.000000	0.285002	0.533974	0.723916	0.840992	0.880359	0.840954	0.723755	0.534341	0.284414
0.281207	0.528262	0.716582	0.833179	0.872481	0.833166	0.716631	0.528193	0.281279	0.000000
0.277587	0.522594	0.709363	0.825496	0.864631	0.825432	0.709553	0.522233	0.278155	0.000000
0.000000	0.274625	0.516666	0.702603	0.817785	0.856809	0.817808	0.702365	0.516746	0.274558
0.000000	0.271643	0.510919	0.695484	0.810133	0.849020	0.810216	0.695295	0.511291	0.271119
0.268234	0.505635	0.688444	0.802607	0.841264	0.802591	0.688494	0.505567	0.268306	0.000000
0.264955	0.500374	0.681522	0.795114	0.833547	0.795033	0.681732	0.500012	0.265484	0.000000
0.000000	0.262270	0.674837	0.674877	0.787583	0.825867	0.787607	0.674837	0.674914	0.262204
0.000000	0.259569	0.669460	0.660267	0.780123	0.818231	0.780217	0.660060	0.669829	0.259062
0.256423	0.664530	0.661316	0.772805	0.810635	0.772787	0.661566	0.664463	0.256490	0.000000
0.253402	0.679608	0.654881	0.765524	0.803086	0.765430	0.655103	0.679251	0.253897	0.000000
0.000000	0.250929	0.674401	0.648532	0.758198	0.795582	0.758222	0.648491	0.674673	0.250868
0.000000	0.248433	0.669361	0.642212	0.750949	0.788128	0.751053	0.641995	0.669702	0.247956
0.245510	0.664708	0.635737	0.743855	0.780722	0.743836	0.635786	0.664642	0.245573	0.000000

dage 1 r = 0.500000 (n)

0.320000	0.600000	0.800000	0.920000	0.960000	0.920000	0.800000	0.600000	0.333333	0.000000
0.000000	0.306667	0.560000	0.760000	0.880000	0.920000	0.880000	0.760000	0.566667	0.300000
0.000000	0.288889	0.530000	0.723333	0.840000	0.880000	0.840833	0.722500	0.535000	0.278333
0.269000	0.506111	0.485000	0.801667	0.840417	0.801250	0.687917	0.500417	0.271111	0.000000
0.248264	0.479792	0.451927	0.744670	0.801927	0.763698	0.654879	0.475469	0.257176	0.000000
0.000000	0.242689	0.450096	0.622231	0.726927	0.764184	0.728403	0.619583	0.456027	0.237734
0.000000	0.230927	0.428438	0.592532	0.692508	0.728365	0.694172	0.589927	0.433071	0.223602
0.214219	0.411730	0.560473	0.660449	0.693340	0.659146	0.963421	0.406769	0.218891	0.000000
0.201857	0.391353	0.533799	0.629196	0.660917	0.627761	0.534745	0.387467	0.208552	0.000000
0.000000	0.197737	0.367828	0.510275	0.597158	0.628479	0.598631	0.507614	0.372648	0.193733
0.000000	0.188522	0.350514	0.485983	0.568674	0.598595	0.570199	0.483487	0.354315	0.182683
0.175258	0.337252	0.459596	0.542289	0.569437	0.541041	0.462257	0.333085	0.178999	0.000000
0.165328	0.320725	0.437784	0.516503	0.542310	0.515203	0.440247	0.317444	0.170695	0.000000
0.000000	0.162018	0.301556	0.418614	0.490047	0.515853	0.491278	0.416323	0.305471	0.158722
0.000000	0.154525	0.287423	0.398695	0.466639	0.491257	0.467899	0.396564	0.290513	0.149745
0.143711	0.276610	0.377031	0.444976	0.467269	0.443911	0.379206	0.273155	0.146753	0.000000
0.135597	0.263079	0.359146	0.423797	0.444981	0.422699	0.361157	0.260355	0.139969	0.000000
0.000000	0.132892	0.247371	0.343438	0.402064	0.423248	0.403069	0.341527	0.250563	0.130177
0.000000	0.126754	0.235787	0.327096	0.382853	0.403057	0.383880	0.325324	0.238306	0.122828
0.117893	0.226925	0.309320	0.365076	0.383567	0.364190	0.311093	0.224076	0.120378	0.000000
0.111241	0.215829	0.294647	0.347697	0.365077	0.346787	0.296288	0.213581	0.114818	0.000000
0.000000	0.109023	0.202946	0.201763	0.329842	0.347242	0.330682	0.200184	0.205553	0.104790
0.000000	0.103989	0.193441	0.268355	0.314100	0.330675	0.314939	0.266892	0.193500	0.100763
0.096720	0.186172	0.253770	0.299515	0.314519	0.298783	0.255219	0.183028	0.098754	0.000000
0.091263	0.177068	0.241733	0.285256	0.299513	0.284503	0.243074	0.175219	0.094194	0.000000
0.000000	0.089444	0.166498	0.231162	0.270623	0.284880	0.271293	0.229862	0.168634	0.087609
0.000000	0.085314	0.158701	0.220162	0.257691	0.271289	0.258377	0.218958	0.160387	0.082665
0.079351	0.152738	0.208196	0.245725	0.258034	0.245123	0.209382	0.150812	0.081018	0.000000

FIGURE(8.4.8)cont.

8.5 A UNIFIED GROUP EXPLICIT PARABOLIC SOLVER (UGEPS)

Next consider the GE solution to the 2-D equation (8.1.2) defined by (8.1.35)-(8.1.44) and the molecules in Fig.(8.1.1). Fig.(8.1.2) represents an intuitive mapping of x-y points onto a mesh connected processor. Each mesh point representing a processor containing the current approximation to that grid point. Initially processing elements are loaded with $u_{i,j,0}$ values (with processor i,j receiving $u_{i,j,0}$) and after k cycles element i,j holds $u_{i,j,k}$. The mesh is orthogonally connected and for a single scheme like GER(x) the points remained fixed in processors throughout the computation. Each processor must be loaded with the coefficients of the molecule it computes, and on each GE-cycle executes a formula of at most size (8.1.36). Normalising (8.1.36) by dividing by a, and loading the resulting coefficients a single ips cell computes the most complex molecule in 12 ips, which is the cost of producing a complete time level. Thus t_z time levels of a scheme like GER(x), GER(y), GEL(x) or GEL(y) requires $T=12t_z$ cycles neglecting the O(m) time for loading coefficients and initial data. Considering the mesh in more detail reveals similar problems to the 2-D asymmetric wave-front mesh. Firstly, the array performs well only if the final time level is to be output as previous levels can be overwritten. Second a molecule of type O has twelve points associated with it, and it is possible for only four of them to be adjacent to the processor requiring them. The remaining points located in second nearest neighbour cells. This complicates systolic design as each cell in addition to its own calculations must route approximations to the correct processor. As a result the control of a cell is context sensitive with the (i,j) position determining the control sequence. Thus a non-alternating group explicit method requires

Reduced Instruction set processors which can be preloaded with a control program. Alternating Group Explicit (AGE) schemes are implemented by shifting the approximated points in one of four compass directions (N,E, S,W) and modifying the control program to execute different types of molecule in the required sequence. This requires loading each processor with coefficients of all relevant molecules increasing cell memory size, which must be offset against the unconditional stability of the AGE calculation with larger step sizes.

An alternative approach to single point, single processor is to allocate processors to groups. This immediately reduces the number of processors to $\lfloor m/2 \rfloor^2$ and alleviates the communication problem as each processor contains 4 grid-points with the remaining 8 of 12 points in nearest neighbour processors (two each). The processor is now context free except for calculations on the boundaries which will be dealt with shortly. First, consider a Type O processor: its job is to compute the four molecules associated with the four points it contains. These molecules have the same coefficients but distributed differently, thus by sharing values the storage associated with a group is reduced by 75%. However this is offset against the increased program size which for a single ips cell per group requires 48 ips cycles to compute the 4 type O molecules sequentially. It follows that the processor must contain at least 48 instructions for selecting operands and coefficients. The Type O cell is the most complex and serves as a cost bound for all molecules.

Alternating schemes are much simpler to implement with a group processor correspondence if we permit simple shifting of points internally and externally between processors. For instance, imagine each Type O cell to contain 4 registers holding the values $u_{i,j,k}$, $u_{i+1,j,k}$, $u_{i,j+1,k}$

and $u_{i+1,j+1,k}$. A cycle of shifts corresponding to Fig.(8.1.2) is achieved as an anticlockwise rotation of the form,

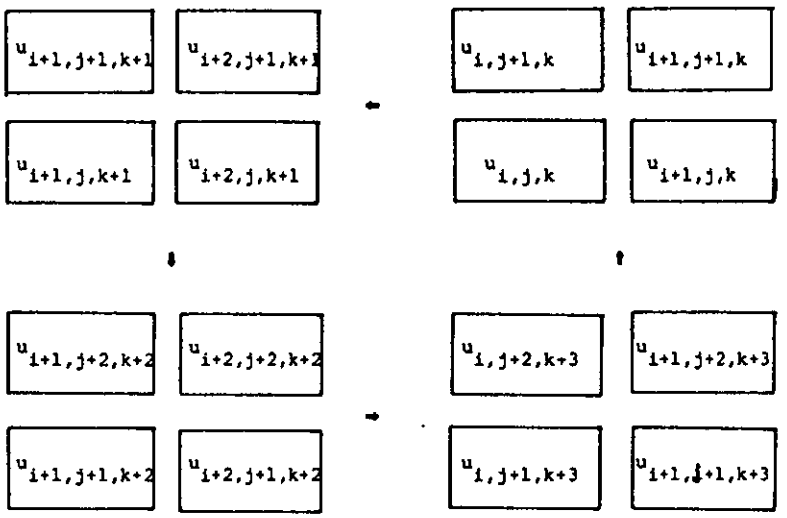


FIGURE 8.5.1: Rotation of parameters on a mesh

Now as the Type O molecule is the most complex, the above cycle is the longest possible for an AGE scheme. It follows that there can only be a total of four computing sequences to make up the 48 instruction program. Similarly, it follows that processors on the periphery of the design can only change into 4 types of molecule also demanding 48 instructions. Thus each processor requires a single ips cell, 15 co-efficient registers (see molecule diagrams), 4 registers for approximations and in addition enough memory for the program. This latter scheme saves processors and reduces the memory size but still requires a program store.

Now consider the templates of the computational molecules shown in Fig.(8.5.2), which are the 2-D representations of molecules in Fig. (8.1.1) with the following key:

co-efficient z, is multiplied with the value at the grid point it covers on level k.

as above but the result of the molecule computation produces the value on level k+1 at this grid point.

imaginary point having no effect on the computation.

the co-efficient z is multiplied by the approximate value on level k+1.

Key to 2-D Computational Molecule Templates

All the molecule types can be represented by a single unified molecule structure or template with coefficients determining the type. Further, a particular grid point can be computed from one molecule for a single level, and by rotation for AGE schemes only four possible template instances are used by a particular grid-point. Arranging the resulting four sets of coefficients in a cyclic queue implicitly defines the control program at each point. Thus, each grid point processor and hence each group processor is context free and contains four cyclic queues one for each grid-point. Combining the two ideas of a basic grid point processor and a group processor a macro-GE cell can be derived (see Fig.(8.5.3)). A macro-GE cell consists of four basic ips type cells one for each grid point, and has the same input/output organisation as a group cell. Each basic cell consists of 2 multipliers and 3 adders plus registers for the approximated grid point and cyclic queues for coefficient data (12 registers in each). Hence a macro-cell has 12 adders and 8 multipliers, 48 coefficient registers and 4 point approximation registers but no program store. The operation of the basic grid-point cell is easily derived from the templates by dividing them into quadrants. Each quadrant is represented by only two coefficients

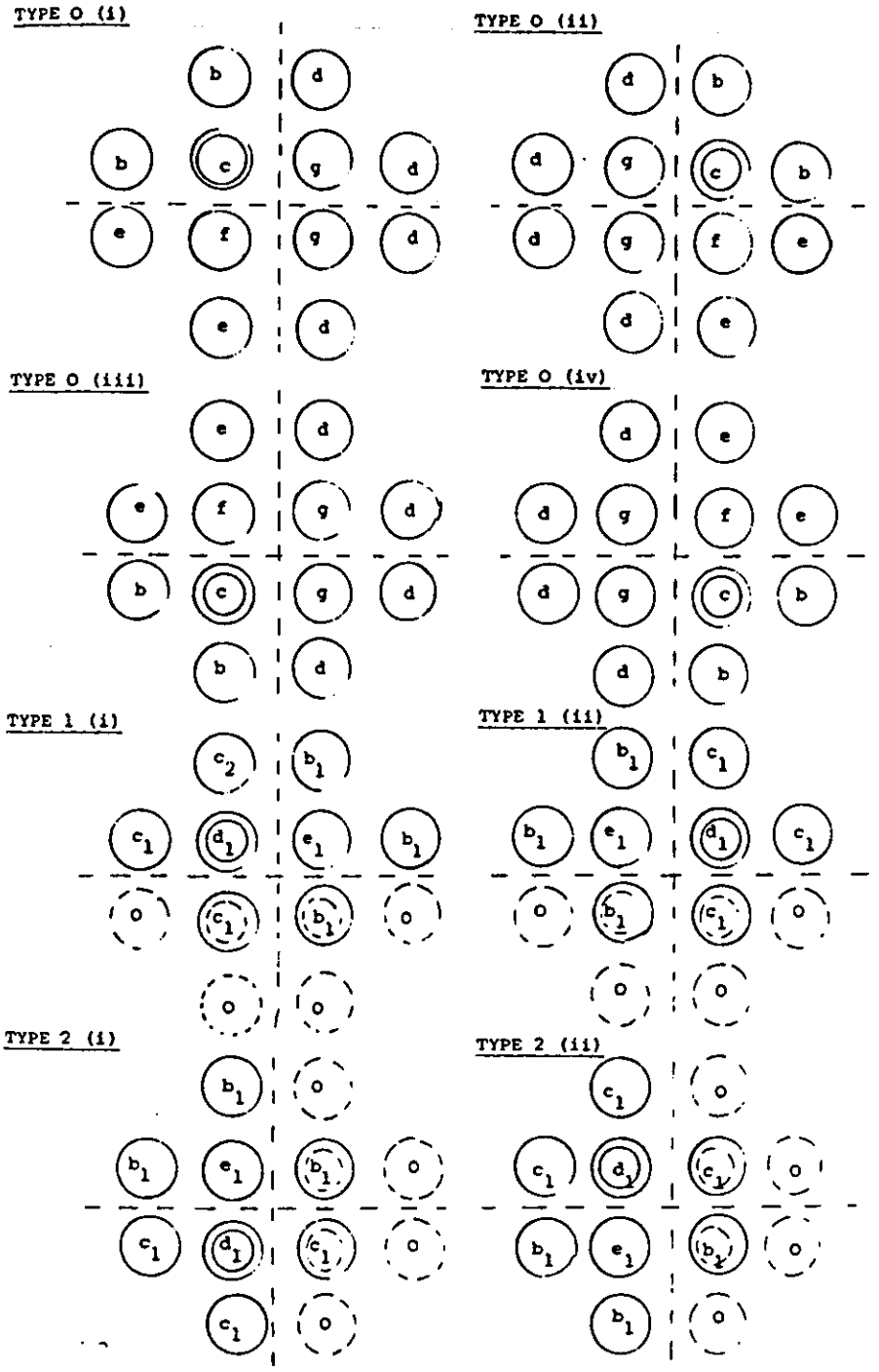


FIGURE 8.5.2: Molecule templates for macro cell register loading

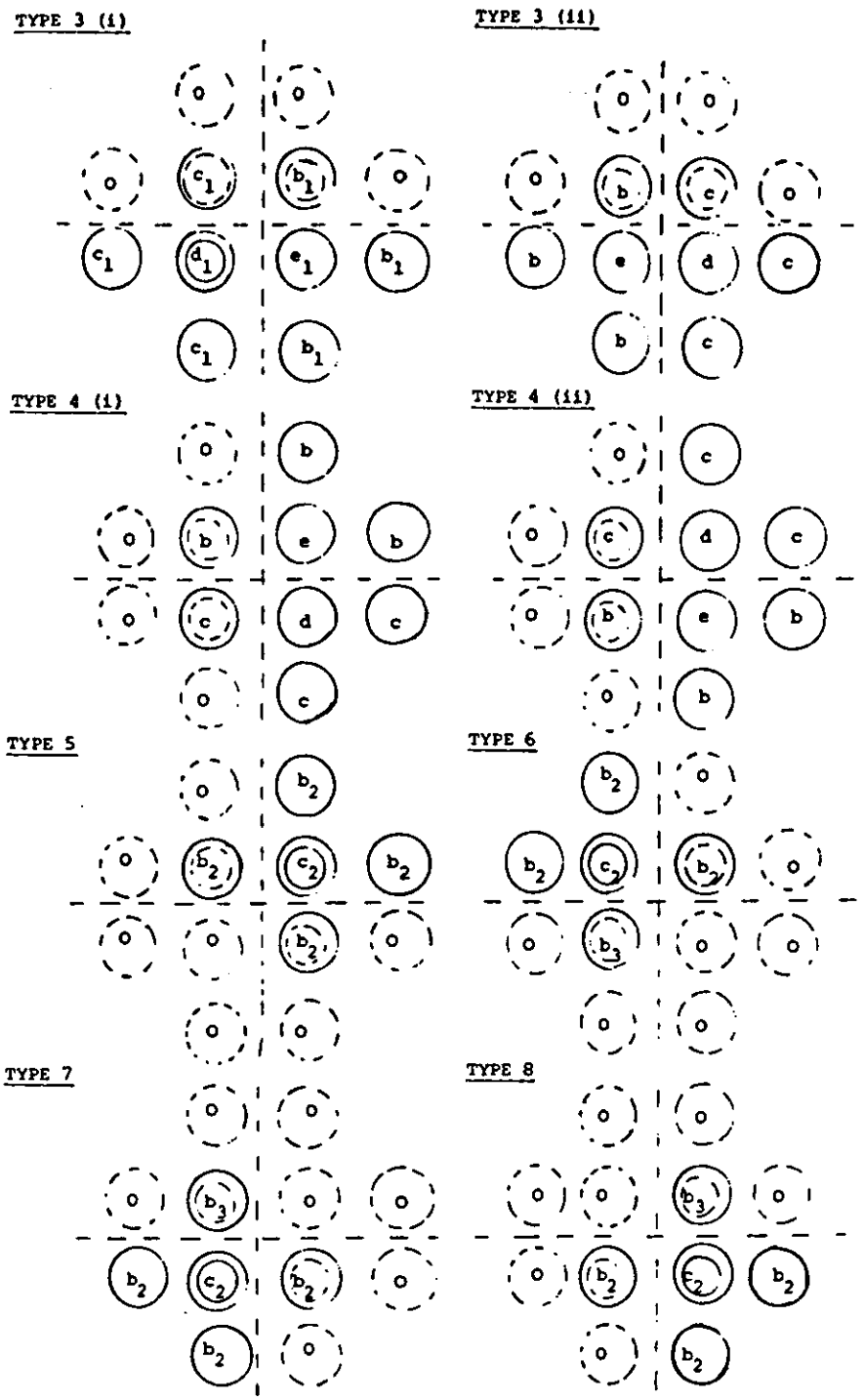


FIGURE 8.5.2: Molecule templates continued

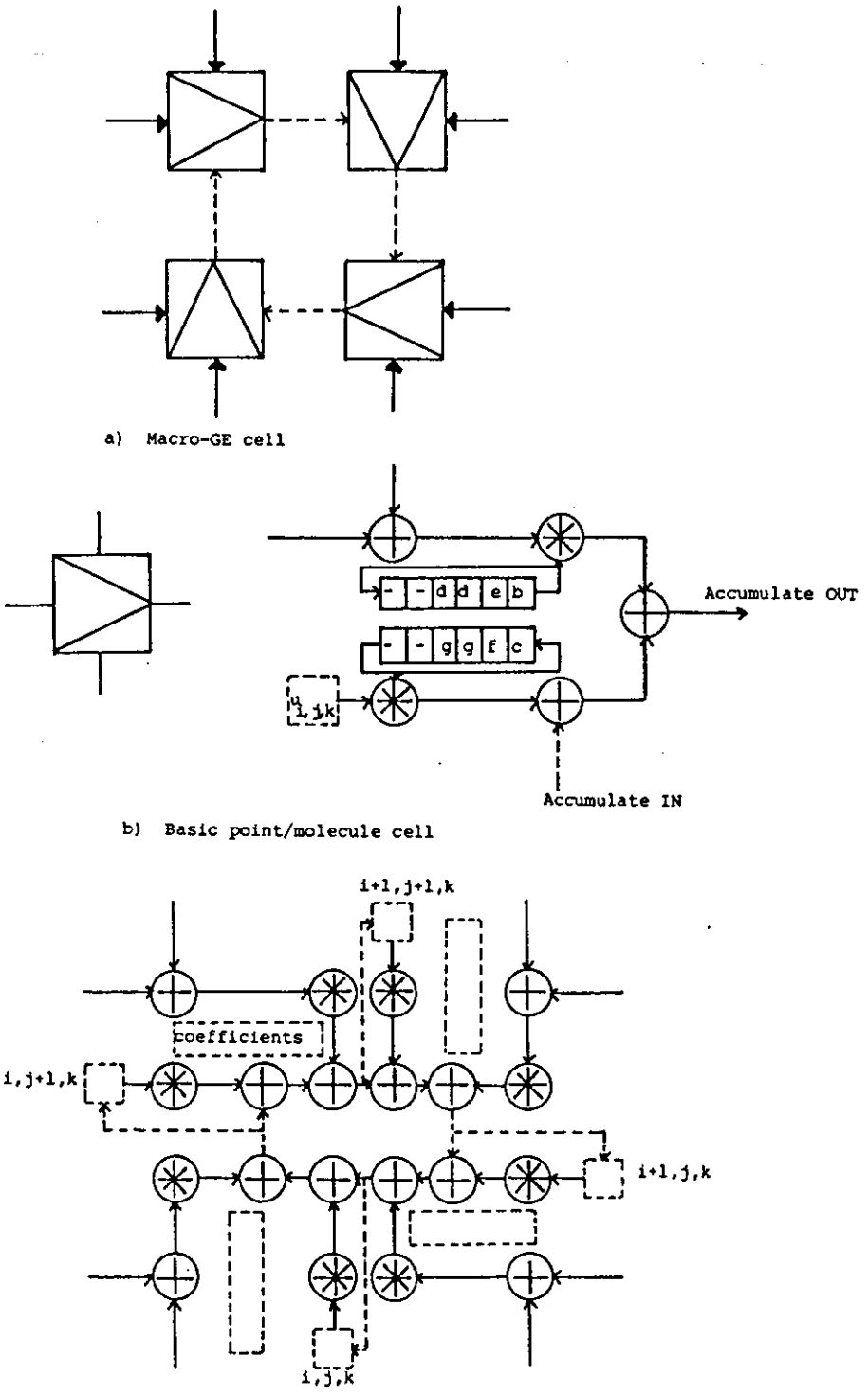
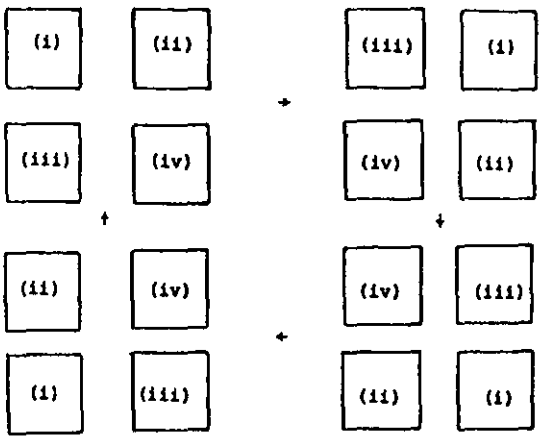


FIGURE 8.5.3: Area efficient macro-cell layout

and if the point cell contains a point approximation all values are nearest neighbour. Now assign the computation such that a point cell computes only the portion of the molecule associated with the quadrant fitting its position in the macro-cell. Molecule computations require the accumulation of all parts of the template which is performed as a 4 step systolic ring computation around the basic point cells. For purposes of illustration evaluating all four Type O molecules simultaneously,



Accumulation of molecule terms on systolic ring

Thus after 4 steps all the molecules have accumulated all their template parts. On the fifth step the point approximation registers can be over-written and the systolic ring values cleared. Fig.(8.5.3b) shows the coefficient ordering for the top left point cell, notice the dummy value for loading and an additional delay which is used later. The delay through a basic point cell is 2 mults + 1 addition = ips + mult, and allowing 6 steps per group we require at most 12 ips cycles. Fig. (8.5.3c) illustrates an area efficient layout for the macro-cell.

Now consider the solution of the 2-D problem using this new cell.

Fig.(8.5.4) illustrates a linear array of macro cells or a bi-linear array of basic cells for m=8, together with additional connections for

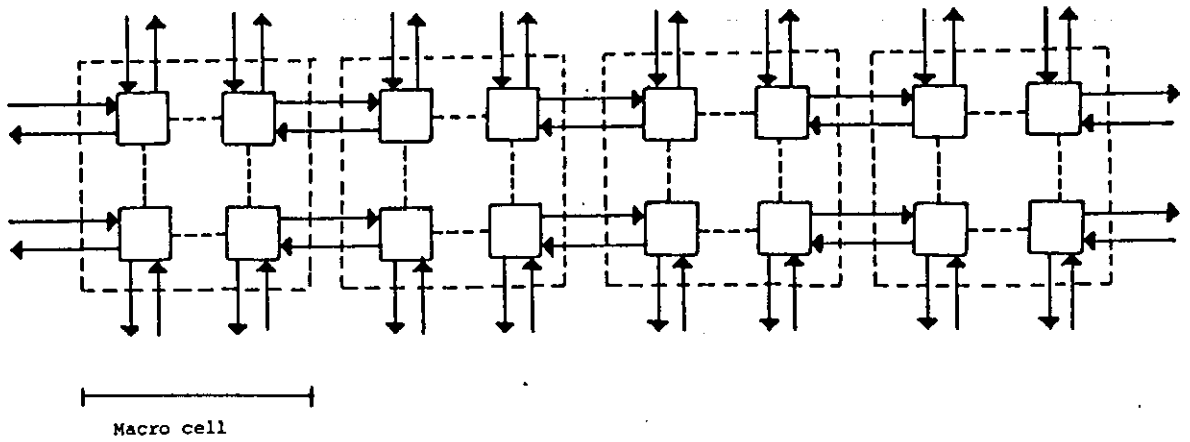


FIGURE 8.5.4: A four cell (m=8 point) array for approximation of the 2-D parabolic problem.
N.B. additional links included for register load/unload operations.

input and output and communication with other cells and are trivial to include. Using the array we can compute two lines in the x or y direction every 12 ips cycles (including loading time), a new time level every $\frac{1}{2}m(12)=6m$ ips cycles, and t_z levels in,

$$T = 6t_z m \ . \qquad (8.5.1)$$

The application of the array resembles the marching principle applied by the LAMP array (Fig.(8.3.6)) where the processing array is viewed as marching systolically up the rows (y-direction) or along columns (x-direction). A slight problem occurs when different types of molecule are encountered during the marching process. For instance, if we compute GER(x) initially we load the array with coefficients for molecules of Type 0 and 3, and march left to right (columnwise) until we reach the last two columns when we must reload the coefficients with Types 7 and 2. Likewise with GEL(y) we load initially with Types 4 and 0

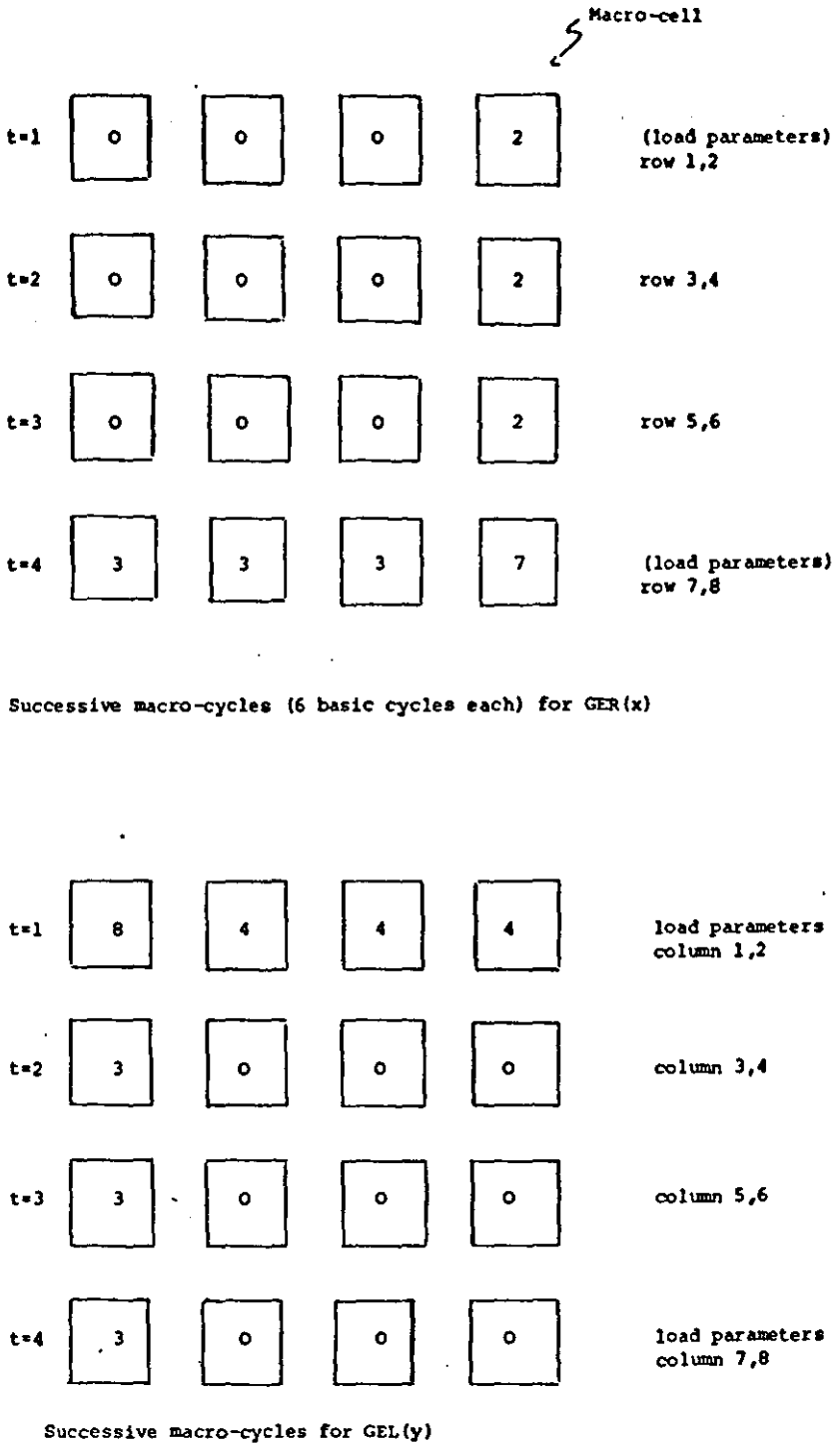


FIGURE 8.5.5: Cell typing for row and columnwise systolic marching in 2-dimensions

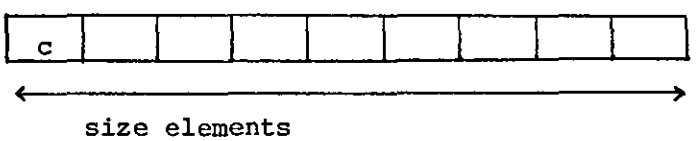
and march row-wise bottom to top reloading with Types 8 and 3 on the last row. Consequently, any of the basic schemes can be computed with only two loads which requires only constant time (each cell loaded in parallel) hence,

$$T = 6t_z m + c \qquad (8.5.2)$$

with c>0 a constant accounting for loading delays. Alternating schemes require no special treatment, and are selected by loading the correct cell type coefficients at the start of a pass through the region.

Throughout the descriptions we have assumed a row-wise march, for a column-wise march the templates must be rotated 90° clockwise before deriving the loading scheme.

A program simulating the bi-linear array is given in the Appendix. Testing was performed using specially constructed mesh points which test cell operation but are not true 2-D problems. In order to run the program two files "odds" and "evens" are required. "odds" contains the molecule coefficients for loading the bottom tier of the array with data corresponding to odd rows of the test grid. Likewise "evens" contains coefficients for the top tier cells and even row data. Each line of the files has the form,



where c is a control value

$$c = \begin{cases} 1 \text{ load meml} \\ 2 \text{ load mem2} \end{cases} \text{ coefficients}$$
$$0 \text{ load groups and compute molecules}$$
$$6 \text{ stop}$$

An example grid was tested for two cases:

1. A molecule with constant coefficients

   (testing ring accumulation)

2. A molecule with different coefficients

   (testing coefficient shifting on grid cells)

The results are given below.

TEST EXAMPLE

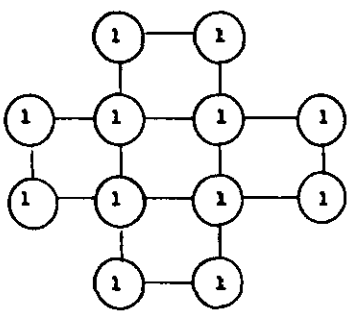
Consider the 8×8 grid

row								
8	1	1	1	1	1	1	1	1
7	1	2	2	2	2	2	2	1
6	1	2	3	3	3	3	2	1
5	1	2	3	4	4	3	2	1
4	1	2	3	4	4	3	2	1
3	1	2	3	3	3	3	2	1
2	1	2	2	2	2	2	2	1
1	1	1	1	1	1	1	1	1

Size = 8 size 2 = 4

TEST 1: Constant coefficients

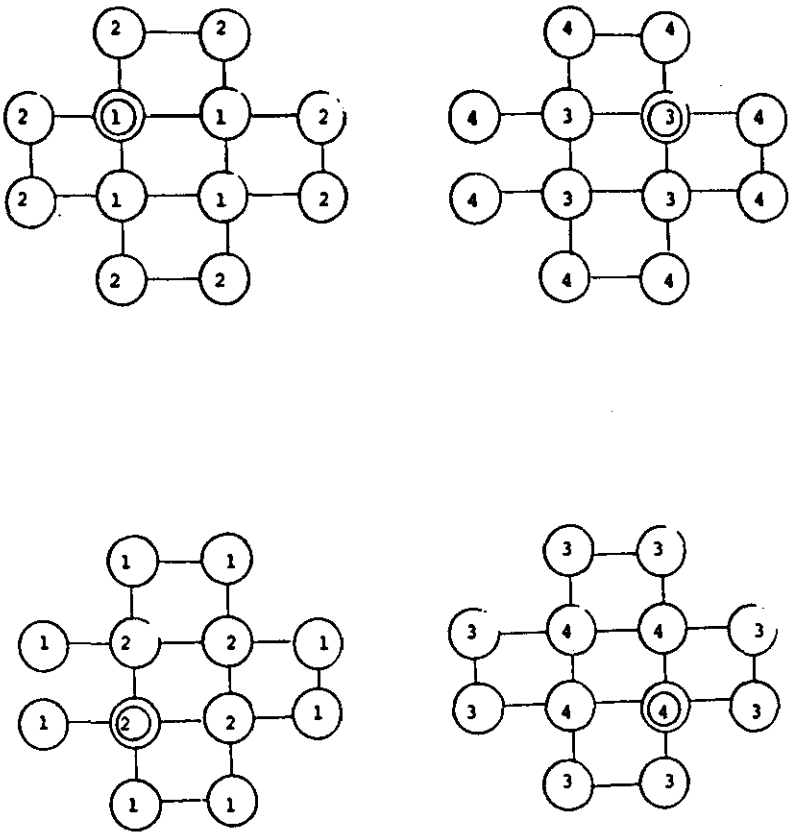
Each point cell computes the same template of form



Thus testing the nearest neighbour communication between macro-cells and the systolic ring accumulation.

TEST 2: Variable coefficients

Assign a different molecule to each point cell in a macro-cell. Each cell computes the molecule associated with its macro-cell position below.



Giving a coefficient ordering which ensures that mem1 and mem2 (see program) never have the same coefficient simultaneously. Thus ensuring that coefficients are loaded and referenced correctly.

INPUT FILES FOR TESTS

TEST 1

```
1 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
1 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
1 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
1 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
2 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
2 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
2 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
2 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
0 1.0 2.0 3.0 3.0 3.0 3.0 2.0 1.0
0 1.0 2.0 3.0 4.0 4.0 3.0 2.0 1.0
0 1.0 2.0 2.0 2.0 2.0 2.0 2.0 1.0
0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
6 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
```

ODDS

```
1 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
1 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
1 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
1 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
2 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
2 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
2 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
2 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0 1.0 2.0 2.0 2.0 2.0 2.0 2.0 1.0
0 1.0 2.0 3.0 4.0 4.0 3.0 2.0 1.0
0 1.0 2.0 3.0 3.0 3.0 3.0 2.0 1.0
0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
6 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
```

EVENS

TEST 2

```
1 2.0 1.0 2.0 1.0 2.0 1.0 2.0 1.0
1 4.0 2.0 4.0 2.0 4.0 2.0 4.0 2.0
1 3.0 4.0 3.0 4.0 3.0 4.0 3.0 4.0
1 1.0 3.0 1.0 3.0 1.0 3.0 1.0 3.0
2 1.0 2.0 1.0 2.0 1.0 2.0 1.0 2.0
2 3.0 1.0 3.0 1.0 3.0 1.0 3.0 1.0
2 4.0 3.0 4.0 3.0 4.0 3.0 4.0 3.0
2 2.0 4.0 2.0 4.0 2.0 4.0 2.0 4.0
0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
0 1.0 2.0 3.0 3.0 3.0 3.0 2.0 1.0
0 1.0 2.0 3.0 4.0 4.0 3.0 2.0 1.0
0 1.0 2.0 2.0 2.0 2.0 2.0 2.0 1.0
0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
6 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
```

ODDS

```
1 4.0 3.0 4.0 3.0 4.0 3.0 4.0 3.0
1 3.0 1.0 3.0 1.0 3.0 1.0 3.0 1.0
1 1.0 2.0 1.0 2.0 1.0 2.0 1.0 2.0
1 2.0 4.0 2.0 4.0 2.0 4.0 2.0 4.0
2 3.0 4.0 3.0 4.0 3.0 4.0 3.0 4.0
2 4.0 2.0 4.0 2.0 4.0 2.0 4.0 2.0
2 2.0 1.0 2.0 1.0 2.0 1.0 2.0 1.0
2 1.0 3.0 1.0 3.0 1.0 3.0 1.0 3.0
0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0 1.0 2.0 2.0 2.0 2.0 2.0 2.0 1.0
0 1.0 2.0 3.0 4.0 4.0 3.0 2.0 1.0
0 1.0 2.0 3.0 3.0 3.0 3.0 2.0 1.0
0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
6 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
```

EVENS

```
OCCAM - Start run
 0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00
 0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00

 0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00
 0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00

 0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00   LOAD MEM1
 0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00

 0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00
 0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00

 0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00
 0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00

 0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00   LOAD MEM2
 0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00

 0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00
 0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00

 0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00
 0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00   SETUP
                                                  (RESULT INVALID)
 0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00
 0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00

 4.00  8.00  4.00  8.00  4.00  8.00  4.00  8.00
 2.00  6.00  2.00  6.00  2.00  6.00  2.00  6.00

 1.00  2.00  2.00  2.00  2.00  2.00  2.00  1.00   LOAD ROWS 1 AND 2
 1.00  1.00  1.00  1.00  1.00  1.00  1.00  1.00

17.00 39.00 30.00 66.00 30.00 66.00 17.00 39.00
16.00 38.00 24.00 60.00 24.00 60.00 16.00 38.00   result

 1.00  2.00  3.00  4.00  4.00  3.00  2.00  1.00
 1.00  2.00  3.00  3.00  3.00  3.00  2.00  1.00   LOADS ROW 3 AND 4

30.00 66.00 57.00 127.00 57.00 127.00 30.00 66.00
24.00 60.00 48.00 118.00 48.00 118.00 24.00 60.00  result

 1.00  2.00  3.00  3.00  3.00  3.00  2.00  1.00
 1.00  2.00  3.00  4.00  4.00  3.00  2.00  1.00   LOAD ROWS 5 AND 6

30.00 66.00 57.00 127.00 57.00 127.00 30.00 66.00
24.00 60.00 48.00 118.00 48.00 118.00 24.00 60.00  result

 1.00  1.00  1.00  1.00  1.00  1.00  1.00  1.00
 1.00  2.00  2.00  2.00  2.00  2.00  2.00  1.00   LOAD ROWS 7 AND 8

17.00 39.00 30.00 66.00 30.00 66.00 17.00 39.00
16.00 38.00 24.00 60.00 24.00 60.00 16.00 38.00  result

OCCAM - Run finished
```

SNAPSHOTS OF ARRAY OUTPUT FOR TEST 1

```
OCCAM - Start run
 0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00
 0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00

 0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00
 0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00

 0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00   LOAD MEM1
 0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00

 0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00
 0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00

 0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00
 0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00   LOAD MEM2

 0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00
 0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00

 0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00
 0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00

 0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00   SETUP
 0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00   (INVALID RESULTS)

 2.00  2.00  2.00  2.00  2.00  2.00  2.00  2.00
 2.00  2.00  2.00  2.00  2.00  2.00  2.00  2.00

 1.00  2.00  2.00  2.00  2.00  2.00  2.00  1.00   LOAD ROWS 1 AND 2
 1.00  1.00  1.00  1.00  1.00  1.00  1.00  1.00

11.00 11.00 18.00 18.00 18.00 18.00 11.00 11.00
11.00 11.00 18.00 18.00 18.00 18.00 11.00 11.00   result

 1.00  2.00  3.00  4.00  4.00  3.00  2.00  1.00
 1.00  2.00  3.00  3.00  3.00  3.00  2.00  1.00   LOAD ROWS 3 AND 4

18.00 18.00 35.00 35.00 35.00 35.00 18.00 18.00
18.00 18.00 35.00 35.00 35.00 35.00 18.00 18.00  result

 1.00  2.00  3.00  3.00  3.00  3.00  2.00  1.00
 1.00  2.00  3.00  4.00  4.00  3.00  2.00  1.00   LOAD ROWS 5 AND 6

18.00 18.00 35.00 35.00 35.00 35.00 18.00 18.00
18.00 18.00 35.00 35.00 35.00 35.00 18.00 18.00  result

 1.00  1.00  1.00  1.00  1.00  1.00  1.00  1.00
 1.00  2.00  2.00  2.00  2.00  2.00  2.00  1.00   LOAD ROWS 7 AND 8

11.00 11.00 18.00 18.00 18.00 18.00 11.00 11.00
11.00 11.00 18.00 18.00 18.00 18.00 11.00 11.00  result

OCCAM - Run finished
```

SNAPSHOTS OF ARRAY OUTPUT FOR TEST 2

8.6 A FAST ALTERNATING GROUP EXPLICIT (AGE) ARRAY

It can be shown that the finite difference approximations (8.1.10) are unconditionally stable for all r>0 (see Saul'ev [64]). Analysis on the group explicit methods (Evans & Abdullah [83b]) indicates that (8.1.21) is stable for $r \leq 1$ as are the GER, GEL, GEU, GEC schemes, while the alternating schemes SAGE and DAGE are unconditionally stable for all r>0. So far the designs presented have attempted to improve array cell efficiency by adapting the decoupled structure of the GE matrices.

Observing the above stability properties we can consider a choice of grid spacings $h=\Delta x$, $\ell=\Delta t$ such that the number of arithmetic operations involved in computing the molecules (8.4.3) is reduced. Thus creating an accelerated (FAST) AGE scheme with optimised basic cell schemes. For the 1-D case a useful choice of r=1, reduces (8.1.22) to the system,

$$\begin{bmatrix} 2 & -1 \\ -1 & 2 \end{bmatrix} \begin{bmatrix} u_{i,k+1} \\ u_{i+1,k+1} \end{bmatrix} = \begin{bmatrix} u_{i-1,k} \\ u_{i+2,k} \end{bmatrix} \tag{8.6.1}$$

or in explicit form,

$$\begin{bmatrix} u_{i,k+1} \\ u_{i+1,k+1} \end{bmatrix} = \frac{1}{3} \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} u_{i-1,k} \\ u_{i+2,k} \end{bmatrix} \tag{8.6.2}$$

with the right boundary,

$$u_{m-1,k+1} = \tfrac{1}{2}\{u_{m,k+1}+u_{m-2,k}\} , \tag{8.6.3}$$

and left boundary,

$$u_{1,k+1} = \tfrac{1}{2}\{u_{0,k+1}+u_{2,k}\} \tag{8.6.4}$$

The various GE schemes are then constructed by substituting r=1 in (8.1.27)-(8.1.34). It follows that the generic structure of the AGE array in Section (8.4) remains unchanged for the FAST AGE except for internal arrangements of the basic cell.

A new basic GE-cell is shown in Fig.(8.6.1), and must compute (8.6.2). Observe that the cell contains registers to hold $u_{i,k}$ and $u_{i+1,k}$ values, even though they play no role in the cell computation, but are passed to adjacent cells for inclusion in the computation there.

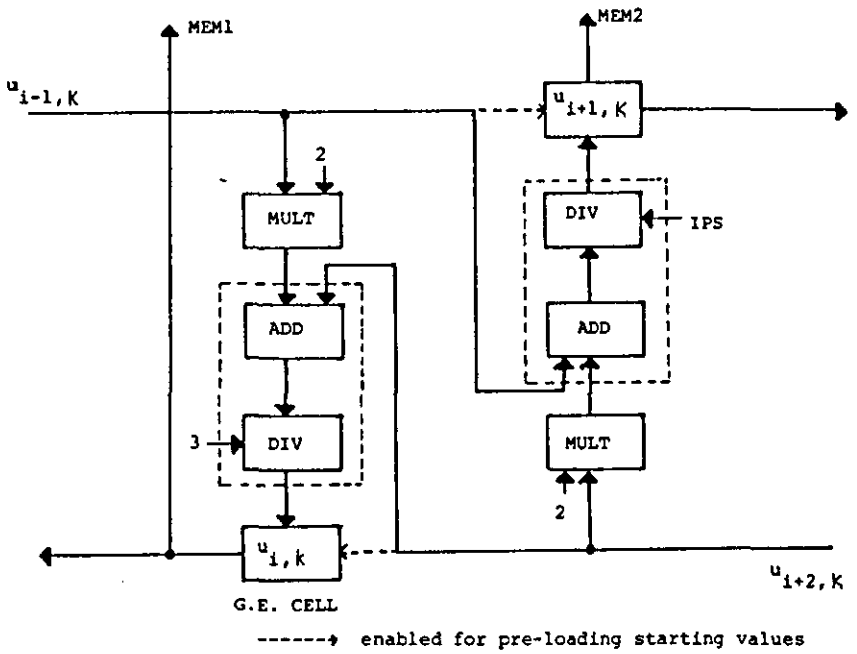


FIGURE 8.6.1: Fast AGE cell

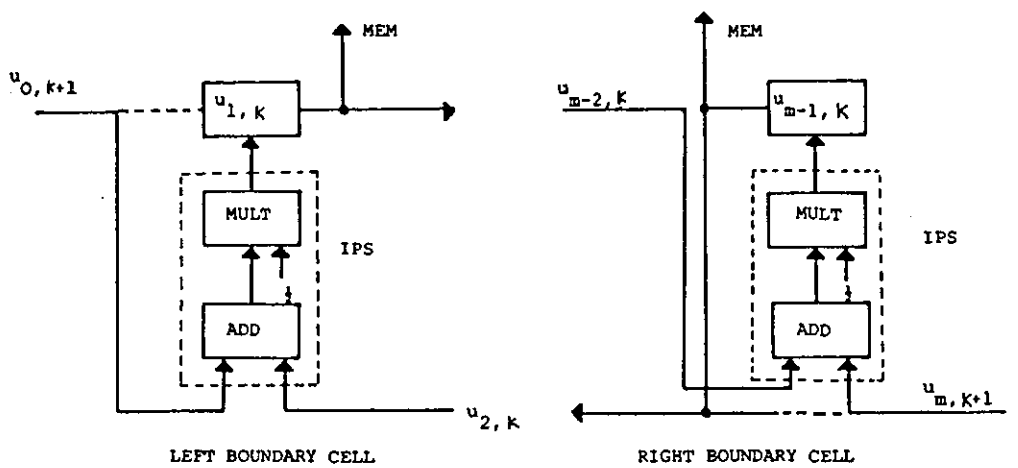


FIGURE 8.6.2: Boundary cells

Also notice that the matrix coefficients can be hardwired into each half of the cell, and that results $u_{i+1,k}$ and $u_{i,k}$ of the previous step are latched to avoid overwriting of inputs before calculations are completed. The multiplier (for fixed point arithmetic at least) can be implemented by a simple bit shift operation and hence consumes negligible area. Consequently, the half cell requires a single adder and divider (≡ inner product cell) and has a cycle time of a single ips. Comparing this to Fig.(8.4.3) indicates that we have a simpler structure requiring no control program or coefficient store, and only a single non-planarity. The cell time of a single ips is also a significant improvement over the 5 ips required previously, and offers an immediate speedup by a factor of 5. An extra connection is added to facilitate preloading and is enabled by a control value broadcast to the array. For a GER scheme the pre-loading consists of two sequences of starting values,

$u_{1,0}, u_{3,0}, \ldots, u_{m-1,0}$ entering and moving left ←

$u_{2,0}, u_{4,0}, \ldots, u_{m-2,0}$ entering and moving right →

yielding the timing,

$$T = t_z + \tfrac{1}{2}m \quad \text{ips cycles} , \tag{8.6.5}$$

for computing t_z levels. Adopting the buffered scheme in Fig.(8.4.6) requires an extra adder to include the $\bar{g}(x,t)$ and with the buffer controls,

c_1	c_2	GE and buffer control
0	0	Normal cell computation
0	1	Preload value (shift left and right)
1	0	Freeze array + output row of memory
1	1	Freeze array + shift up memory

where c_1=1 relates to a stopped array, c_1=0 to a computing array,

$$T = 2t_z + (\bar{t}_z + 1)(\text{freeze-time}) + \tfrac{1}{2}m . \tag{8.6.6}$$

yielding a speedup $S_p \tilde{} 3$ over (8.4.12) for the general equation (8.4.10). Finally for completeness the boundary cells associated with (8.6.3) and (8.6.4) are implemented as shown in Fig.(8.6.2) (a method of combining GE and boundary cells is discussed shortly).

Now consider the SAGE method, which requires shifting of data right and left to alternate between GER(GEL) and GEL(GER). In the previous schemes for general r an extra cycle had to be added to achieve shifting. For the FAST AGE the shift can be hardwired, by tracing the input/output paths of a cell on successive cycles as shown in Fig.(8.6.3) which produce the cell in Fig.(8.6.4). Marked on the cell are switching points A and B, which indicate where a switch between data paths implements a shift. For instance, the SAGE cell switches have the following interpretations,

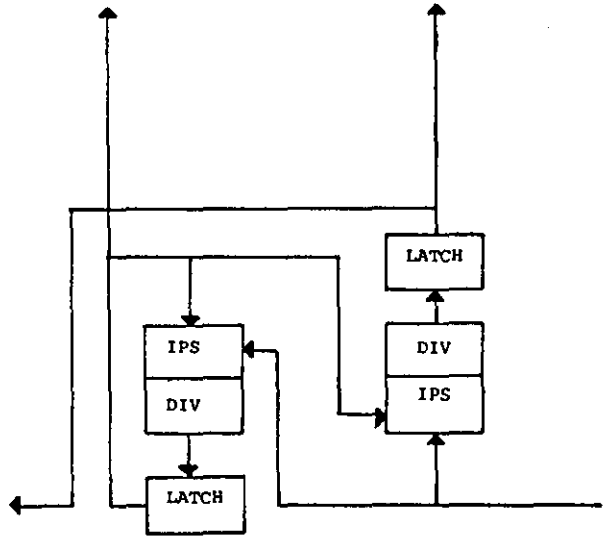
$$A = \begin{cases} \underline{\text{ON}} \text{ act as feedback of latch contents, and} \\ \text{send } u_{i,k} \text{ left} \\ \underline{\text{OFF}} \text{ read } u_{i-1,k} \text{ into left ips arrangement} \end{cases}$$

$$B = \begin{cases} \underline{\text{ON}} \text{ act as feedback loop for } u_{i+1,k}, \text{ and send} \\ u_{i+1,k} \text{ right} \\ \underline{\text{OFF}} \text{ read } u_{i+2,k} \text{ into right ips arrangement} \end{cases}$$

As A and B are mutually exclusive a 1-bit control can implement SAGE shifting.

REMARK: The switches allow erroneous calculations, not part of the algorithm, as these values never affect computation they can be ignored and allow the above simplified control structure.

The final task is to develop a generic boundary cell which alternates between ordinary GE calculations and the modified asymmetric forms (8.6.3) and (8.6.4). Unfortunately the hardwired nature of the

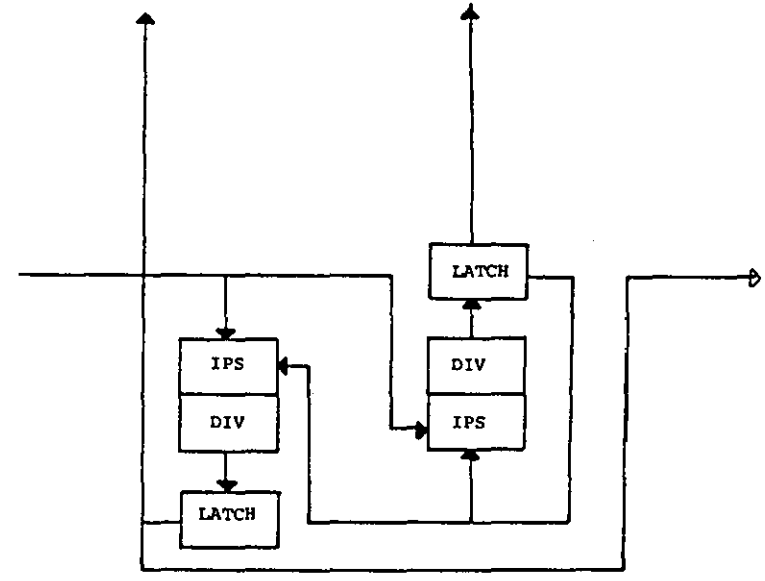


MODEL

t-1: $u_{i,k}$ $u_{i+1,k}$

t: $u_{i,k+1}$ $u_{i+1,k+1}$ $u_{i+1,k+1}$ $u_{i+2,k+1}$ $u_{i+1,k+1}$ $u_{i+3,k+1}$

GEL GE CHANGES TO GER GE



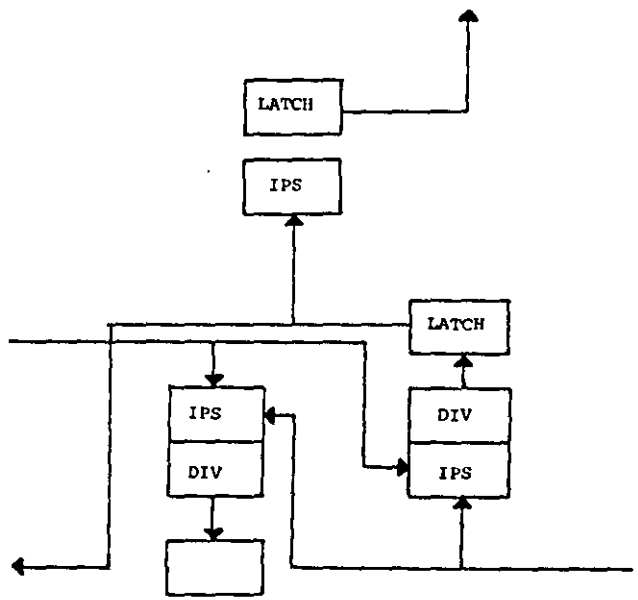
MODEL

t-1: $u_{i,k}$ $u_{i+1,k}$

t: $u_{i,k+1}$ $u_{i+1,k+1}$ $u_{i-2,k+1}$ $u_{i-1,k+1}$ $u_{i,k+1}$ $u_{i,k+1}$

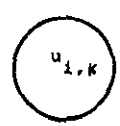
GER GE CHANGES TO GEL GE

a) SAGE GE CELL INTERNAL SWITCHING

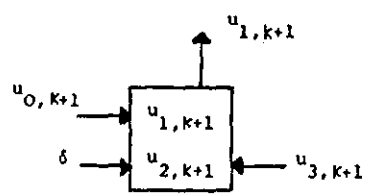


MODEL:

t-1:

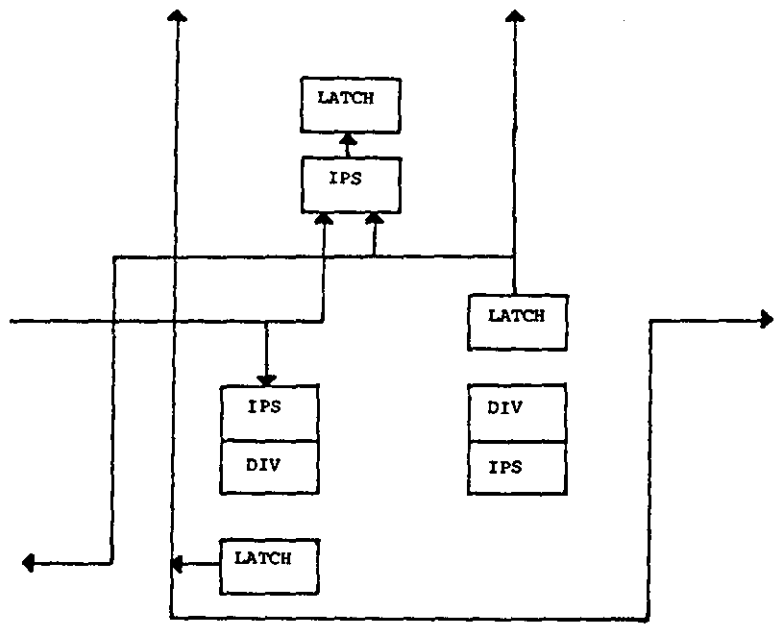


t:



LEFT BOUNDARY CHANGES FROM GEL BOUNDARY TO GE CELL

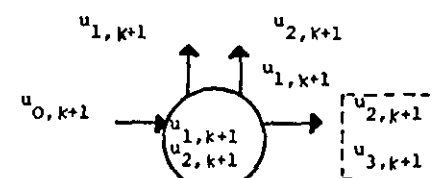
b) LEFT BOUNDARY SAGE CELL INTERNAL SWITCHING



MODEL:

t-1:

t:



LEFT BOUNDARY CHANGES FROM GER GE CELL TO GEL LEFT BOUNDARY CELL

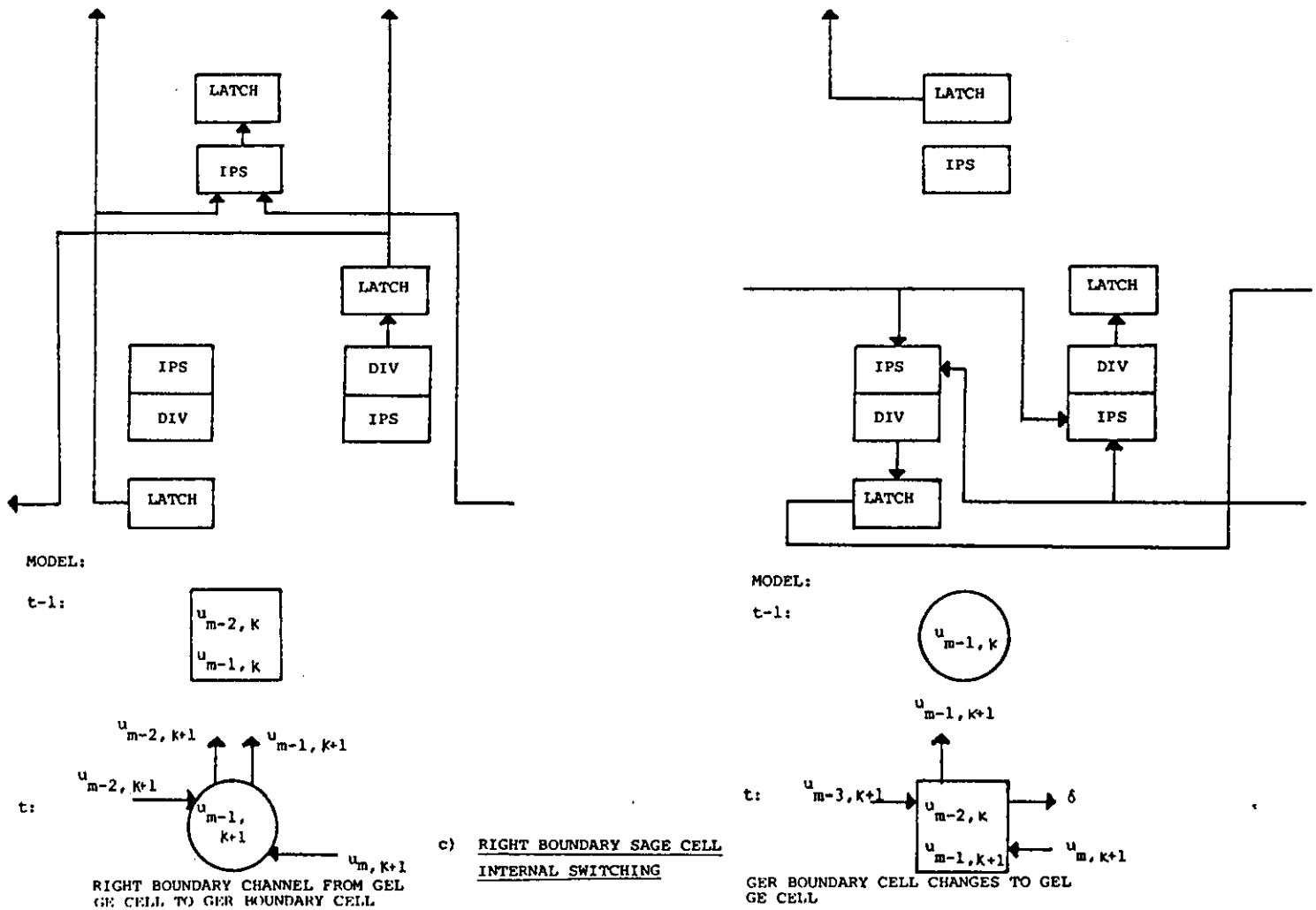


FIGURE 8.6.3: Cell switching diagrams

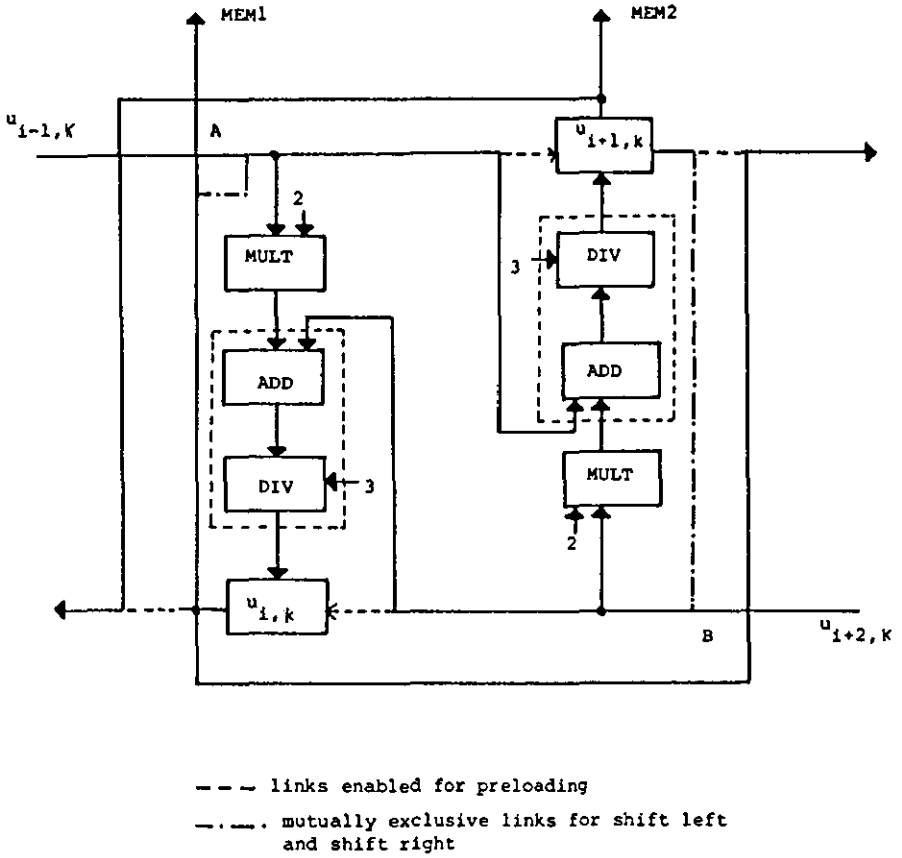


FIGURE 8.6.4: SAGE GE cell

FAST AGE cell means that the unification is not possible and a hybrid cell which switches between the two cell types as shown in Fig.(8.6.5) must be adopted.

We conclude that the FAST AGE scheme including the SAGE algorithm produce a significant reduction in the complexity of basic cells, removing micro-program control and internal coefficient registers, when compared with the general algorithms with r>0. The main advantage of these arrays is that they are closer to hard-systolic frames than previous proposals. While the general arrays allow any value of r, we point out that it is often the case that $r \lesssim 4,5$ is chosen. Consequently if the number of intervals in the x-direction is held constant, the FAST

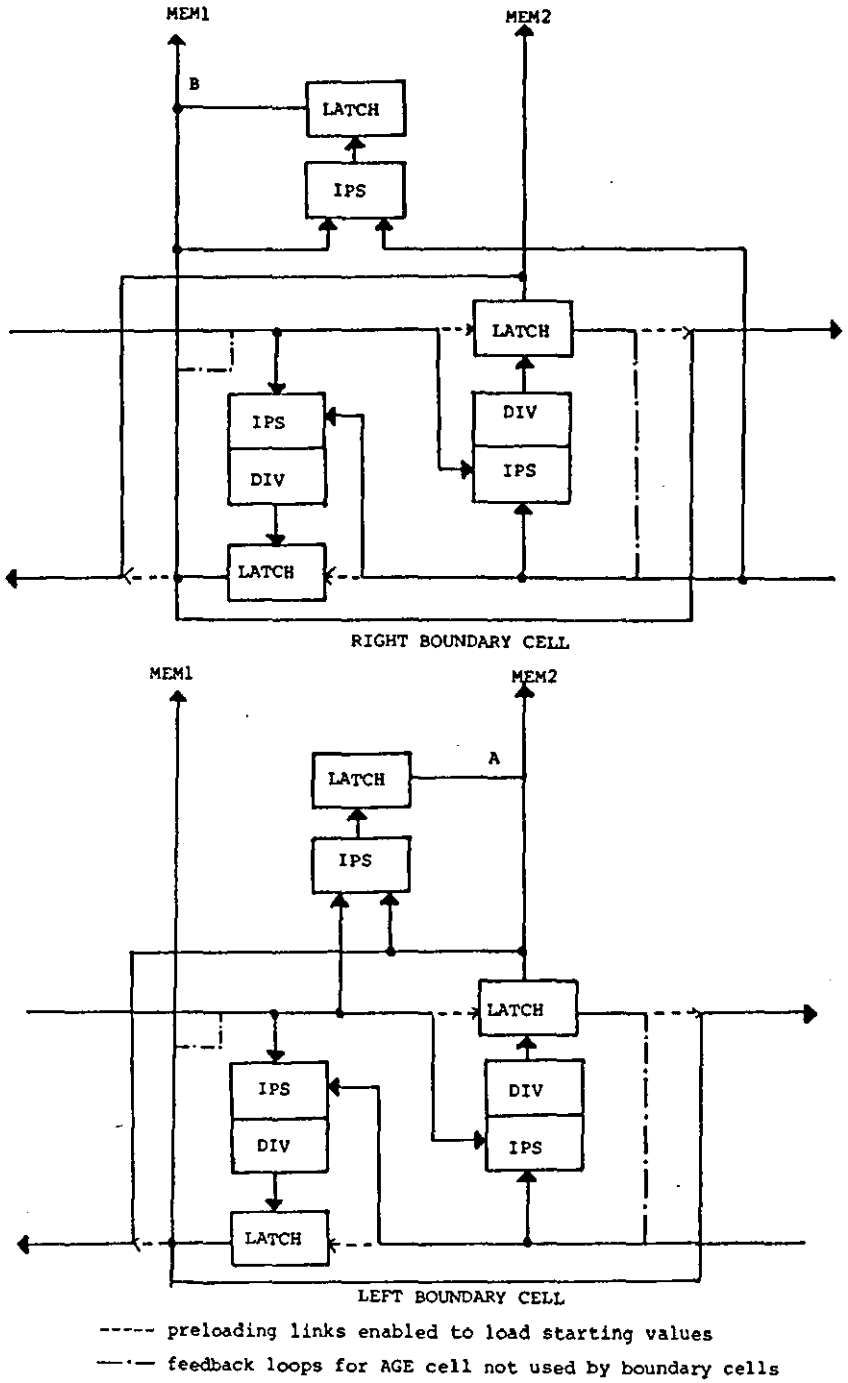

MEM1
MEM2
B
LATCH
IPS
LATCH
IPS
DIV
DIV
IPS
LATCH
RIGHT BOUNDARY CELL
MEM1
MEM2
A
LATCH
IPS
LATCH
IPS
DIV
DIV
IPS
LATCH
LEFT BOUNDARY CELL
----- preloading links enabled to load starting values
—·— feedback loops for AGE cell not used by boundary cells

FIGURE 8.6.5

AGE must compute an extra 3 or 4 levels for each time level of the unconditional scheme. It follows that we can use the FAST AGE speed-up to offset the extra levels being calculated. For equation (8.1.1) a factor of five speed-up is obtained using the FAST AGE indicating that the required extra levels can be accommodated while for equation (8.4.10) the speed-up of 3 is less dramatic. An additional problem is the use of buffers to control host interface complexity, which implies that the FAST scheme requires more freeze time than the unconditionally stable arrays. A possible solution to this is to adopt the fractional splitting technique with the form,

$$\text{SAGE:} \qquad \left.\begin{aligned} (I+rG_1)u_{k+\frac{1}{2}} &= (I-rG_2)u_k + b_1 \\ (I+rG_2)u_{k+1} &= (I-rG_1)u_{k+\frac{1}{2}} + b_2 \end{aligned}\right\} \tag{8.6.7}$$

or

$$\text{SAGE:} \qquad \left.\begin{aligned} (I+rG_1)u_{k+1/4} &= (I-rG_2)u_k + b_1 \\ (I+rG_2)u_{k+\frac{1}{2}} &= (I-rG_1)u_{k+1/4} + b_2 \\ (I+rG_1)u_{k+3/4} &= (I-rG_2)u_{k+\frac{1}{2}} + b_1 \\ (I+rG_2)u_{k+1} &= (I-rG_1)u_{k+3/4} + b_2 \end{aligned}\right\} \tag{8.6.8}$$

For 1 and 3 artificial levels respectively, inhibiting the cell buffer output for intermediate levels by simply overwriting the cell results internally.

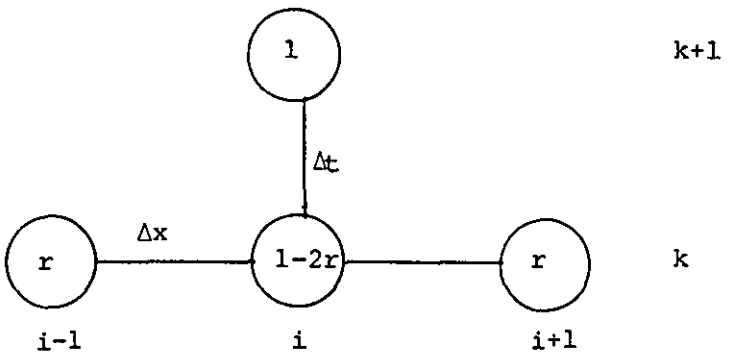
8.7 SYSTOLIC HOPSCOTCH SCHEMES

Now so far in this chapter we have introduced the concept of systolic marching with the relatively complicated asymmetric and group explicit molecules. In this section we examine the possibility of reducing array computation time by the use of simpler computational molecules which possess similar features to the GE methods for parallel

evaluation of time levels. We also examine the possibilities of array compaction derived from methods which produce only partial solutions to the problem.

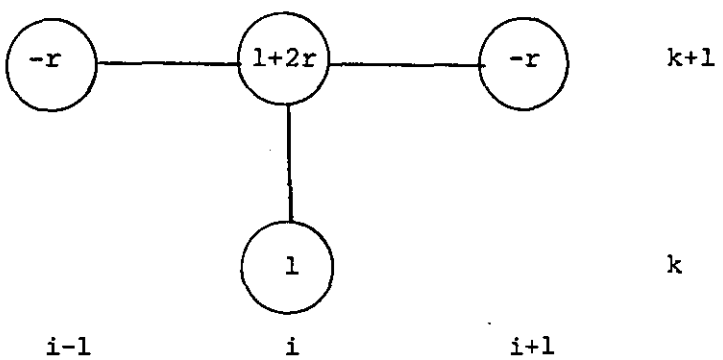
Recall the definitions of the classical implicit and explicit finite difference formulas used in the definition of (2.5.1.14) and (2.5.1.15) for solving (8.1.1) which have the molecule definitions:

a) explicit



$$u_{i,k+1} = ru_{i-1,k}+(1-2r)u_{i,k}+ru_{i+1,k} \qquad (8.7.1)$$

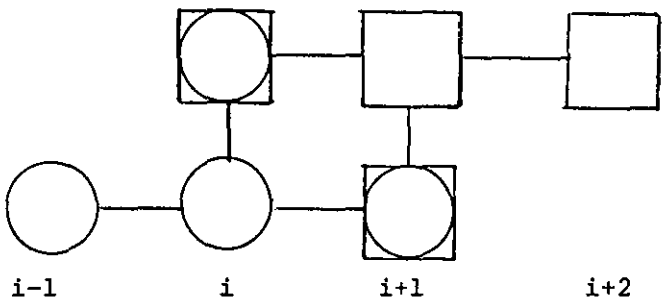
b) implicit



$$-ru_{i-1,k+1}+(1+2r)u_{i,k+1}-ru_{i+1,k+1} = u_{i,k} \qquad (8.7.2)$$

(8.7.2) is unconditionally stable for all $r>0$, while (8.7.1) is stable only for $r\leq\frac{1}{2}$. It is known that both schemes have truncation error $R_{ik}=O(\ell+h^2)$.

Using the implicit and explicit formulas together in an alternating fashion removes the stability problem and implies a hybrid molecule of the form,



(8.7.3)

Compared with the GE molecule (8.1.21). Equation (8.7.3) has three unknowns and two defining equations, and produces a 3×3 under-determined linear system (in contrast to the 2×2 system (8.1.22)). It follows that (8.7.3) cannot be easily converted to explicit form and demands that the component molecules are evaluated sequentially. Converting (8.7.2) to explicit allows (8.7.3) to be expressed algebraically as,

$$\left.\begin{aligned} &\text{a)}\quad u_{i,k+1} = ru_{i-1,k}+(1-2r)u_{i,k}+ru_{i+1,k} \\ &\text{b)}\quad u_{i+1,k+1} = \frac{1}{1+2r}\{u_{i+1,k}+ru_{i,k+1}+ru_{i+2,k+1}\} \end{aligned}\right\} \qquad (8.7.4)$$

or

$$\left.\begin{aligned} &\text{a)}\quad u_{i,k+1} = A_1u_{i-1,k}+A_2u_{i,k}+A_3u_{i+1,k}+E_i \\ &\text{b)}\quad u_{i+1,k+1} = B_1u_{i+1,k}+B_2u_{i,k+1}+B_3u_{i+2,k+1}+E_{i+1} \end{aligned}\right\} \qquad (8.7.5)$$

where $A_1=A_3=r$, $A_2=(1-2r)$, $B_1=\frac{1}{1+2r}$, $B_2=B_3=\frac{r}{1+2r}$ and $E_{i,k}$ and $E_{i+1,k}$ are terms involving $\bar{g}(x_i,k)$ or $\bar{g}(x_{i+1},k)$ for the more general form (8.4.10). Thus cell computation is derived as follows,

$$\left.\begin{aligned} t_1 &= A_2u_{i,k}+E_{i,k} \\ t_1 &= A_3u_{i+1,k}+t_1 \\ t_1 &= (A_1u_{i-1,k}+t_1) = u_{i,k+1} \quad \text{(overwriting } u_{i,k}\text{)} \end{aligned}\right\} \text{1st molecule}$$

$$\left.\begin{aligned} t_2 &= B_1u_{i+1,k}+E_{i+1,k} \\ t_2 &= B_2u_{i,k+1}+t_2 \\ t_2 &= B_3u_{i+2,k+1}+t_2 = u_{i+1,k+1} \quad \text{(overwriting } u_{i+1,k}\text{)} \end{aligned}\right\} \text{2nd molecule}$$

and the array operation is indicated by snapshots in Fig.(8.7.1).

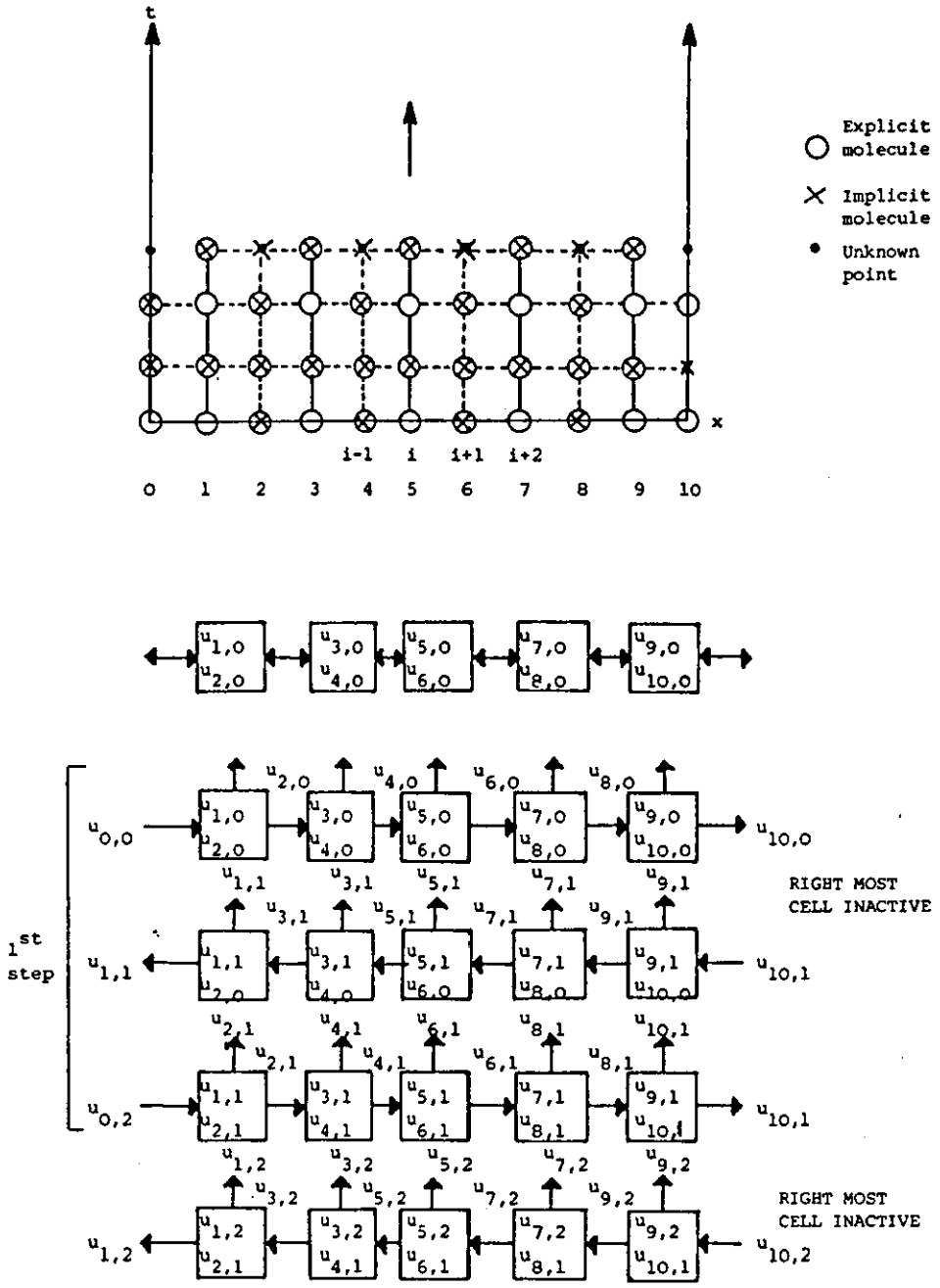


FIGURE 8.7.1: Systolic array for alternative explicit/implicit scheme

We require 6 ips cycles to overwrite the two points contained by each cell the same as an AGE scheme. Notice however that AGE cells require two ips cells, the scheme here only one, which halves the hardware requirements (to ½m ips cells). The ODD-EVEN cell (Fig.8.7.2) requires a single inner product cell with additional switching logic and a cyclic queue of six coefficients which can be tagged with control bits to select the correct ips input operands. The above technique of alternating formulas on the grid is termed the ODD-EVEN hopscotch method, we conclude that it is superior to the AGE schemes from a systolic viewpoint.

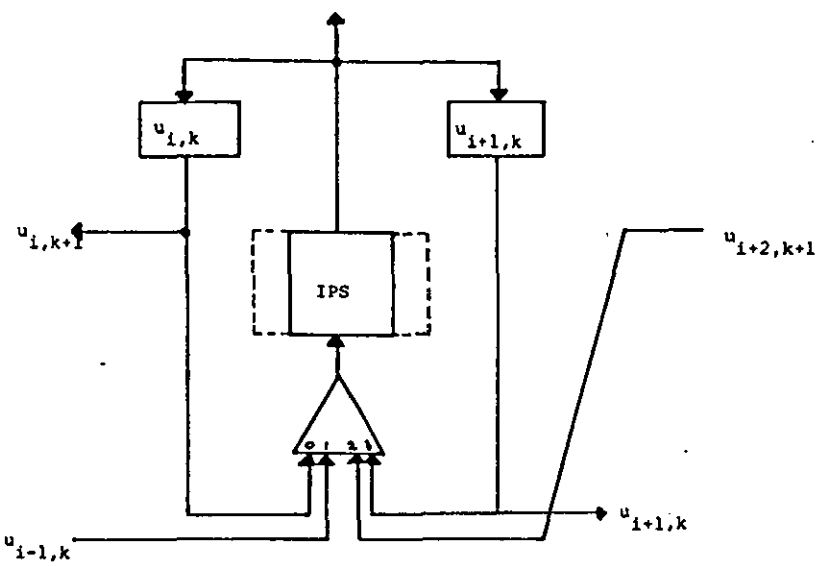


FIGURE 8.7.2: ODD-EVEN hopscotch cell

The idea of hopscotch was expanded by Gourlay [70]* and works on the principle that not all the points in a region need to be calculated.

From earlier work by Gordon, "Non-symmetric difference equations", J.Soc.Indust.Appl.Math., 1973.

The points omitted are placed in areas where they are easily obtainable later from the values already computed. Using this idea a number of strategies can be concocted according to the way formulas are applied and the number of points omitted. We concentrate on two simple forms, the 1-point and 2-point hopscotch methods.

1-Point Hopscotch

Notice that like the FAST AGE for $r=\frac{1}{2}$ the classical explicit formula (8.7.1) simplifies to,

$$u_{i,k+1} = \tfrac{1}{2}(u_{i-1,k}+u_{i+1,k}) \tag{8.7.6}$$

or generally,

$$u_{i,k+1} = \tfrac{1}{2}(u_{i-1,k}+u_{i+1,k}+D_{ik}) \tag{8.7.7}$$

with $D_{ik}=2\bar{g}(x,k)$. Fig.(8.7.3) illustrates the application of this modified molecule to the solution region and the associated array snapshots. Clearly the new array cell consists of only a single adder and a shifter arrangement (for divide by 2), demands a cycle time of approximately $\frac{1}{2}$ ips cycle, and still covers two grid points. It follows that the 1-point scheme requires a time,

$$T = 0.5t_z + m, \quad \text{ips cycles}, \tag{8.7.8}$$

including preloading time, and yields a speed-up over the ODD-EVEN scheme of $S_p=6/0.5=12$. Like the FAST AGE we have lost the desirable property of unconditional stability, but fixing $r=\frac{1}{2}$ and m,

$$\tfrac{1}{2} = \ell/h^2 \quad \text{, with h fixed,}$$

and (8.7.8) implies that we can compute at most 11 extra intermediate levels before the ODD-EVEN scheme competes time-wise. Hence,

$$r_{max} = 12(\tfrac{1}{2}) = (12\ell)/h^2 \tag{8.7.9}$$

implying that the unconditional method must use $r>6$ to out-perform the 1-point hopscotch. As $r\simeq 4,5$ is usual we conclude that the simplified

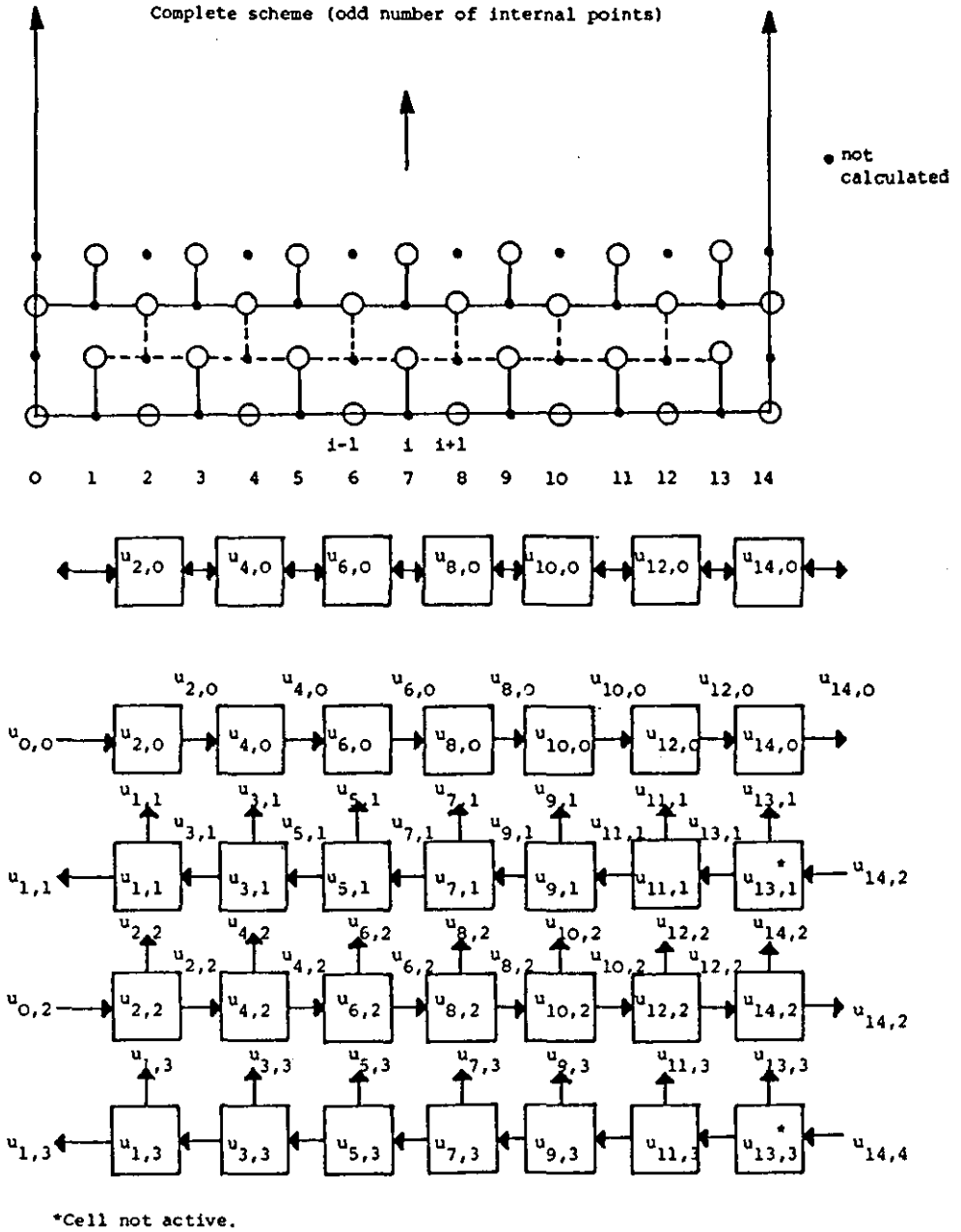


FIGURE 8.7.3: Systolic array for 1-point hopscotch scheme

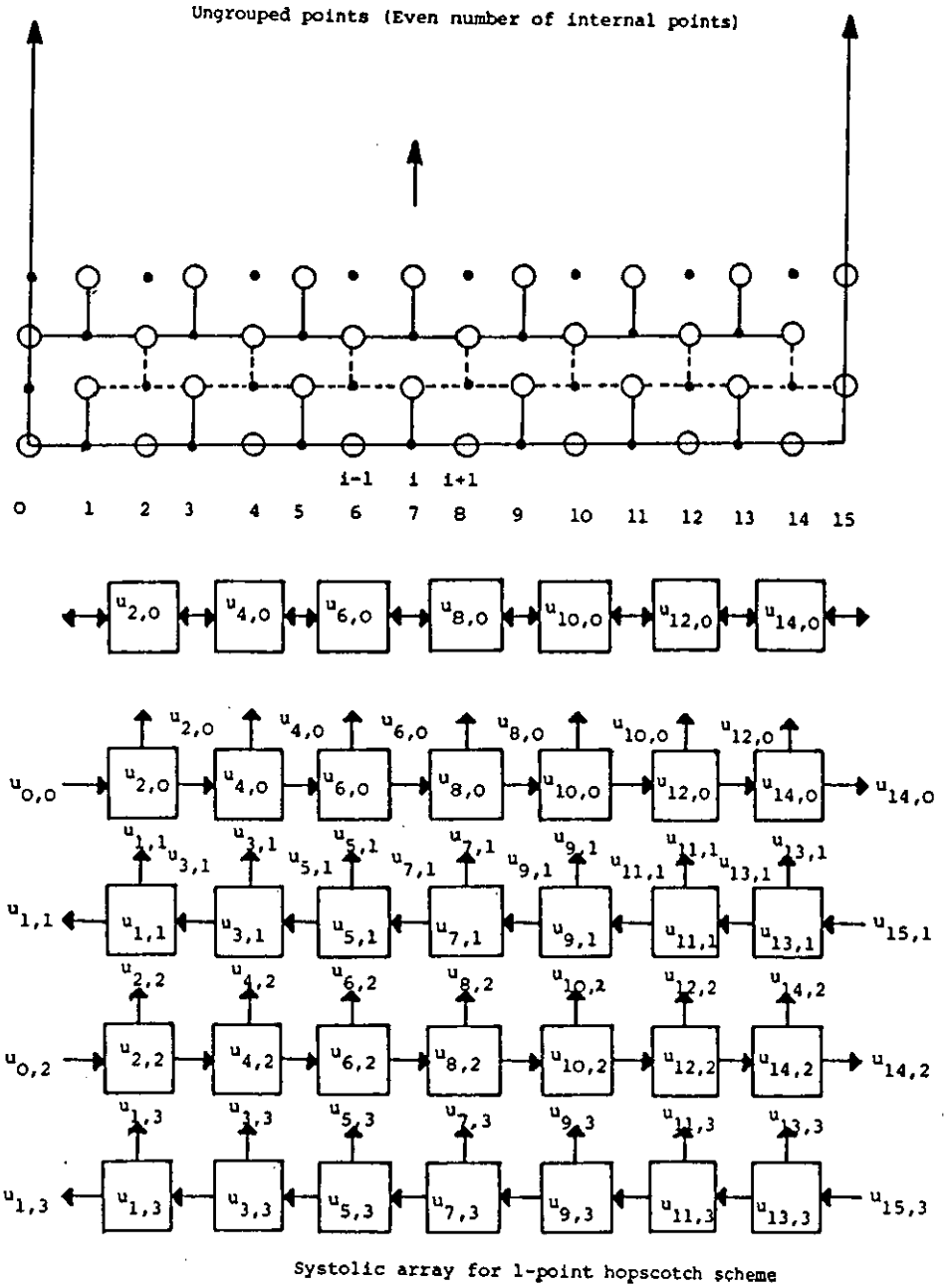


FIGURE 8.7.3: (cont.)

array is extremely desirable. There is, however, a small problem with the 1-point scheme which can be explained simply from the snapshots in Fig.(8.7.3). When the number of internal points in the x-direction is even, all the cells compute all of the time and apart from shifting, the distinction between steps disappears. When the number of internal points is odd the rightmost cell must be inactive on alternate cycles. The shifting of data left and right is useful because it provides a natural method of input boundary values, but when the array shifts left the right boundary must be input even though it is not used until the array shifts right. The difficulty is removed by adding a control tagged to the boundary input which disables the cell while loading the boundary value - and adds negligible hardware.

2-Point Hopscotch

Although the 1-point hopscotch is inherently simpler than the AGE or even the FAST AGE the value $r=\frac{1}{2}$ is required rather than r=1 to retain stability. A relevant question is to ask if the hopscotch can be used to reduce hardware in the FAST AGE scheme? The answer is yes, and the array uses the so-called 2-point block hopscotch shown in Fig.(8.7.4), (two more starting positions can be derived by interchanging circles and crosses on each time level). The immediate consequences of the array is the reduction to m/4 cells or $\frac{1}{2}$m ips cells compared with $\frac{1}{2}$m (or m ips) cells used in the FAST AGE, while computation time remains unchanged. The communication characteristics are also simplified and if we allow bi-directional links the simplified FAST AGE cell of Fig.(8.7.5) is apparent. Finally, when we have an ODD number of internal points boundary cells must be incorporated into the array, and have a form

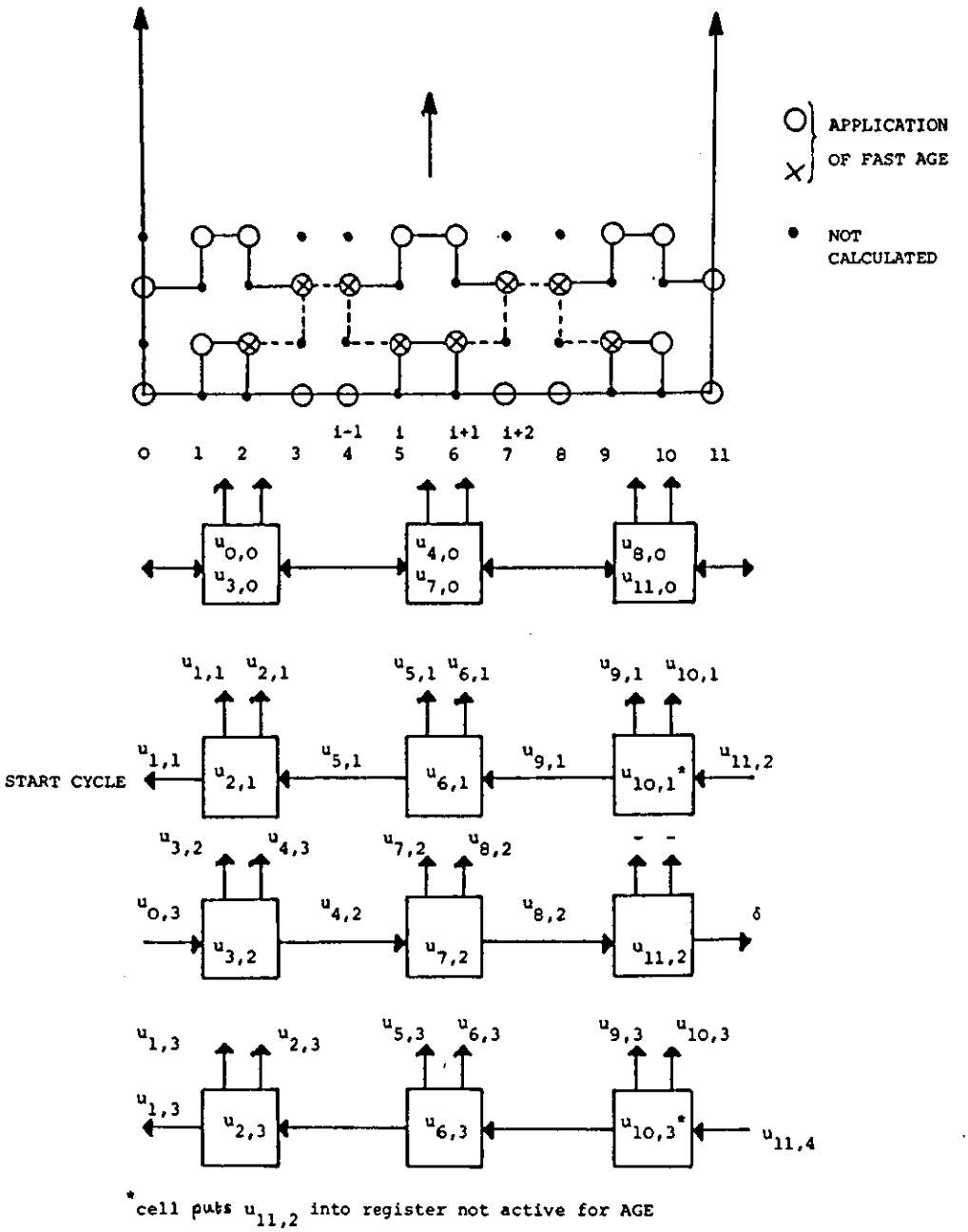


FIGURE 8.7.4: Complete 2-point block hopscotch (even internal points)

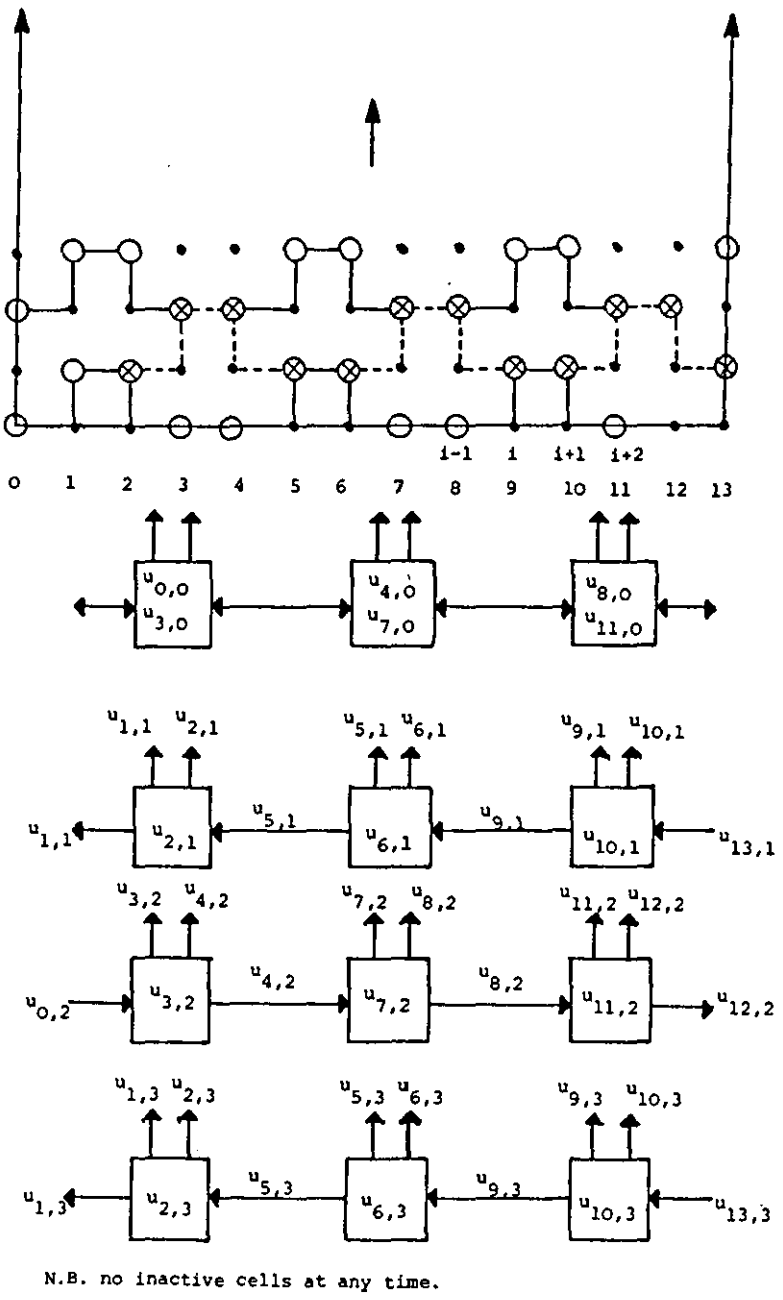
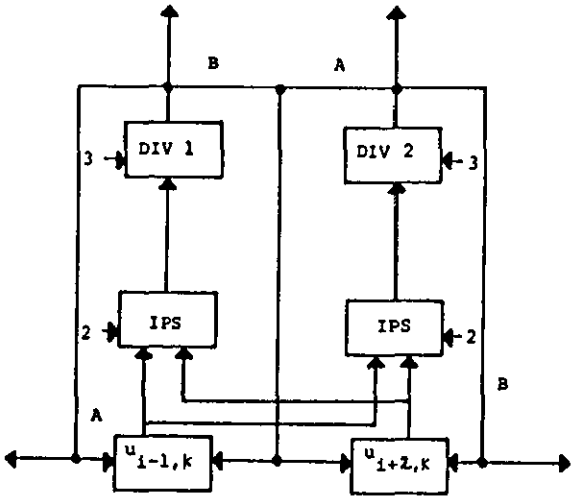


FIGURE 8.7.4: Ungrouped point 2-point block hopscotch



Control switching

c_1	c_2	A	B	C	DATA DIRECTION
0	0	X	X	X	"FREEZE"
0	1	X	✓	✓	RIGHT
1	0	✓	X	✓	LEFT
1	1	X	X	✓	"PRELOAD" (LEFT OR RIGHT)

N.B. C switch in combination with A and B

If $\bar{A} \wedge \bar{B} \wedge C$ connect $u_{i-1,K}$ and $u_{i+2,K}$

$A \wedge \bar{B} \wedge C$ connect DIV2 to $u_{i-1,k}$

$\bar{A} \wedge B \wedge C$ connect DIV1 to $u_{i+2,K}$

$\bar{C}$ Isolated $u_{i-1,k}, u_{i+2,K}$ while unloading.

FIGURE 8.7.5: Simplified FAST AGE cell for 2-point block hopscotch

similar to a 1-point hopscotch cell.

The area efficiency of the hopscotch designs is achieved by omitting some calculations and are in a sense incomplete systolic arrays. The speed-up associated with the simple and compact design is used to offset the cost of extra level computations due to the loss of unconditional stability. But suppose we need all the solutions. Notice that not all the initial conditions and boundary values are incorporated into a

Hopscotch scheme. It follows that for each hopscotch method there are a number of starting positions (or loading orderings) for the array. If we select two starting positions which if used generate all the points, the arrays described can be used in multipass to compute the whole grid. Notice however that only speed-up is then halved. Alternatively, we can define two separate arrays and operate them in parallel retaining the speed-up but doubling the hardware. As hopscotch method reduces hardware over the complete schemes this is a particularly attractive tradeoff.

8.8 A HARD-SYSTOLIC HOPSCOTCH SOLVER

The arrays produced in the previous sections have improved speed and reduced cell complexity edging the designs from soft-systolic to hybrid and finally hard systolic frames. In this section we propose methods for actual VLSI implementation. The simplest formula is the 1-point hopscotch and attention is focussed on this array. Discussions so far have proposed the chip organisation of Fig.(8.4.6) for solving (8.4.10) reducing input/output connections by a buffering strategy. The buffers are emptied and loaded during an array freeze operation and operate in mutually exclusive fashion. Clearly the double buffer method is highly inefficient, but is useful for the complex AGE strategies by simplifying non-planarity problems at the array buffer interface. For the simpler hopscotch schemes it is possible to combine the two buffers reducing memory requirements by half, and ensure that the combined memory is always full. A 4-cell example of the new array is shown in Fig.(8.8.1), the buffer can be interpreted as a collection of horse-shoe segments, with one segment allocated to each cell. After buffer loading

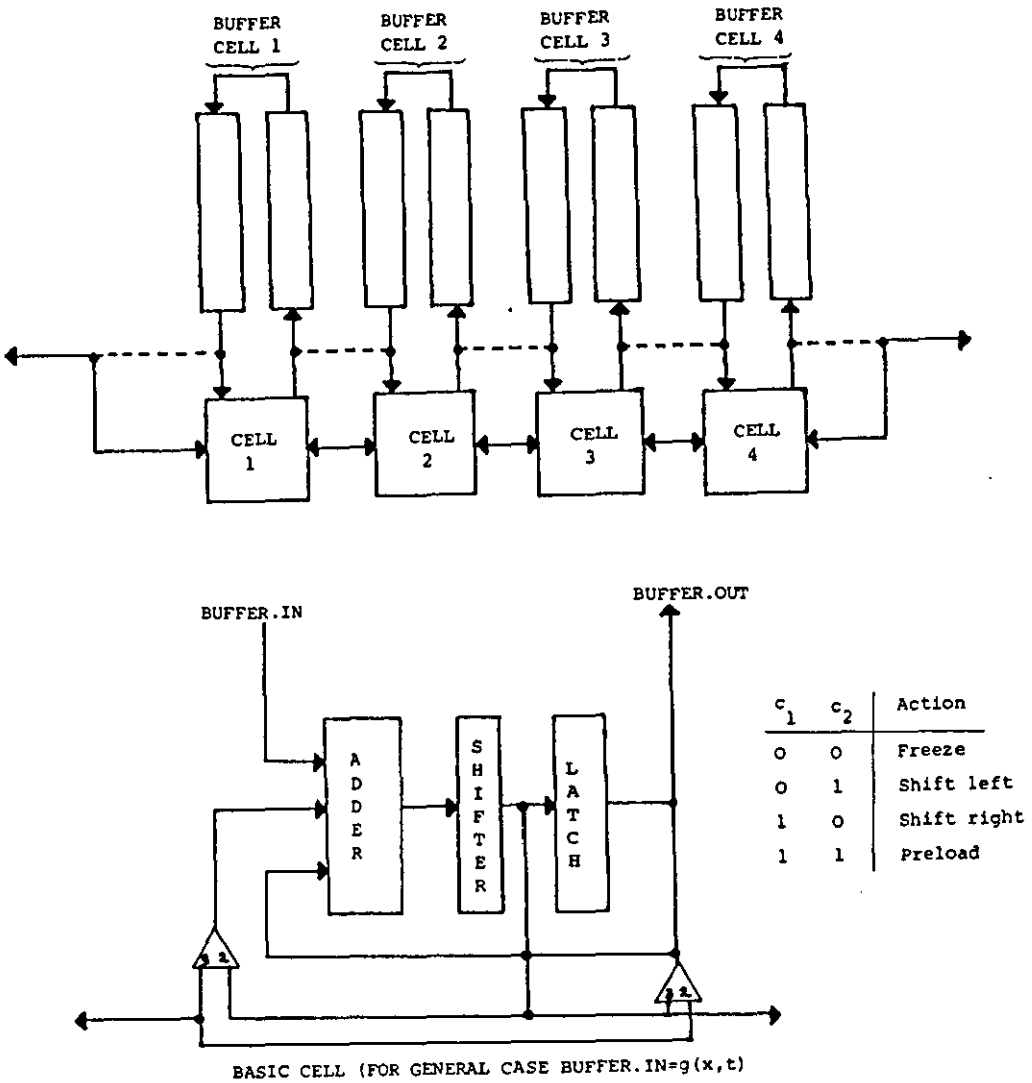


FIGURE 8.8.1: Example 4-cell arrangement

the segment of cell i contains the $\overline{g}(ih,t)$ values in the correct order for the computation of k time levels (where k is the buffer segment size). As computation progresses $\overline{g}(ih,t)$ values are fed into the cell creating holes at the end of the buffer queue which are filled by cell results being output. Hence, the buffer stays full throughout the computation. After the last $\overline{g}(ih,t_k)$ value has left the segment the array is frozen (isolating the new starting values inside the cell) and the inter segment

connections are enabled creating a giant shift register. The results of k time levels and the $\overline{g}(x,t)$ of the next k levels are output and loaded in pipeline fashion from right to left. Once the lead $\overline{g}(x,t)$ value reaches the leftmost buffer register all results have been output, the $\overline{g}(x,t)$ are in position and the array is unfrozen.

In a VLSI design we must decide at the outset the type of the final packaged chip. For purposes of illustration, we shall consider a 64-pin chip and use 28-bit fixed point arithmetic. This allows easy specification of bit-parallel computation and implies the following pin assignments:

	28 pins	INPUT
	28 pins	OUTPUT
	VDD	
	GND	
	CLK	
	c_1, c_2, c_3	CONTROL
TOTAL	62	PINS

using Fig.(8.8.1) as a basis for the design a floorplan and GND,VDD,CLK network arise naturally. We require some random logic to control the chip, and adopt a control broadcast strategy (as pipelining is not possible). Fig.(8.8.1b) illustrates the 1-point hopscotch cell and allows the definition of the following controls.

$\text{Freeze} = \overline{c_1} \wedge \overline{c_2}$, $\overline{\text{Freeze}} = c_1 \vee c_2$

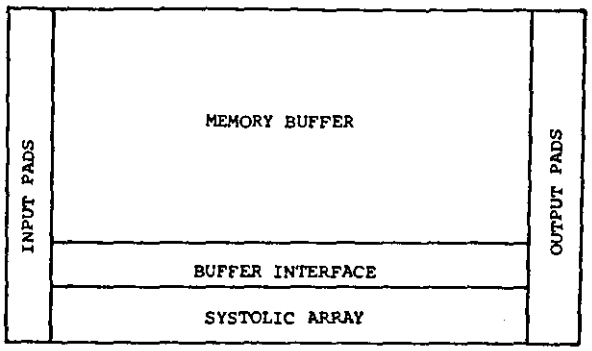
$\text{sh}\ell = c_2$ (shift left), $\text{shr} = c_1 \wedge \overline{c_2}$ (shift right)

$\text{preload} = c_1 \wedge c_2$

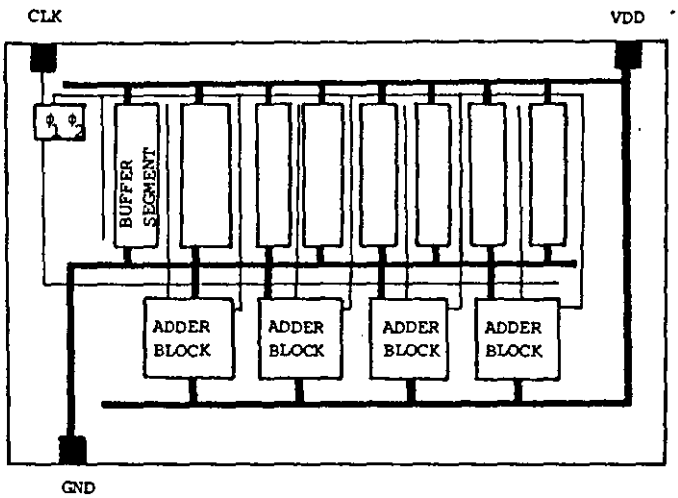
$\text{LD} = \overline{\text{freeze}} \wedge \text{preload}$, $\overline{\text{LD}} = \text{freeze} \vee \overline{\text{preload}}$

$\text{Disable} = \text{preload} \vee c_3$ (for rightmost inactive cell)

We assume that the reader is familiar with material in Mead & Conway [79]



a) Floorplan



b) VDD, GND, AND CLK NET for 1-point hopscotch

FIGURE 8.8.2: VLSI structure of 1-point hopscotch scheme

and Ullman [84] and adopt a two phase non-overlapping clock ϕ_1 and ϕ_2 (such that $\phi_1 \wedge \phi_2$ = false) and generate the signals from a single clock input CLK. (Standard circuits are available for this). The basic adder block is formed from half adders as shown in Fig.(8.8.3) with the divide by two implied by the output connections. The buffer segment cell interface is shown in Fig.(8.8.4a) and a complete 1-bit slice of the cell given in Fig.(8.8.4b). We consider 2's complement arithmetic so that no end around carries are required (which occur with 1's complement). The cell operation is simply two full adders where B_i=0 for (8.1.1) and $B_i \neq 0$ for (8.4.10). The feedback from latch to adder is implemented on a bit-by-bit basis avoiding nasty corner turning layouts to run around the adder (we go through it). The slice is area efficient as all control lines run vertically and data horizontally, except for carries and the adder result shift. The two data paths in Fig.(8.8.1b) are compressed, running through the adder/shifter and combine with the buffer interface. Notice that when the array is in freeze mode no shifting between cells is necessary. The starting value is isolated in the latch, it follows that the buffer can borrow the intercell connections removing the need for an additional path. Fig.(8.8.5) illustrates a complete 4-bit cell with a 10 stage buffer. Each adder block uses 4-bit slices, in a full design we would require 28 slices. This raises the questions 'how many cells could we place on a chip?','how many cells will we be satisfied with?' Assuming a four cell design covers 2 points in the x-direction, allowing 8 points per chip, an arbitrarily large solver could be constructed by chaining chips together using comb layouts to bound the clock and control wire length([Fisher [84]).

Now, a constant conflict in the design of systolic arrays is the

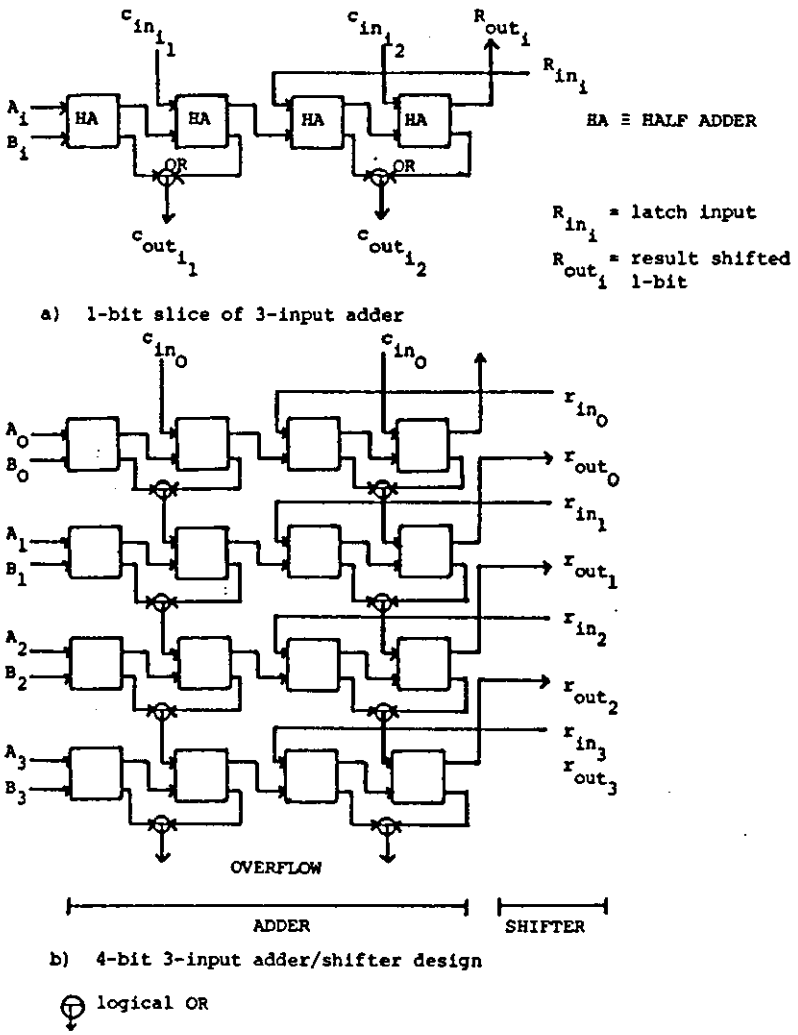
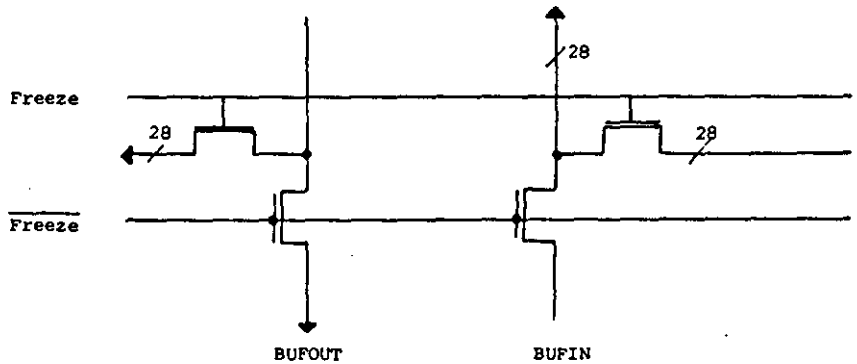
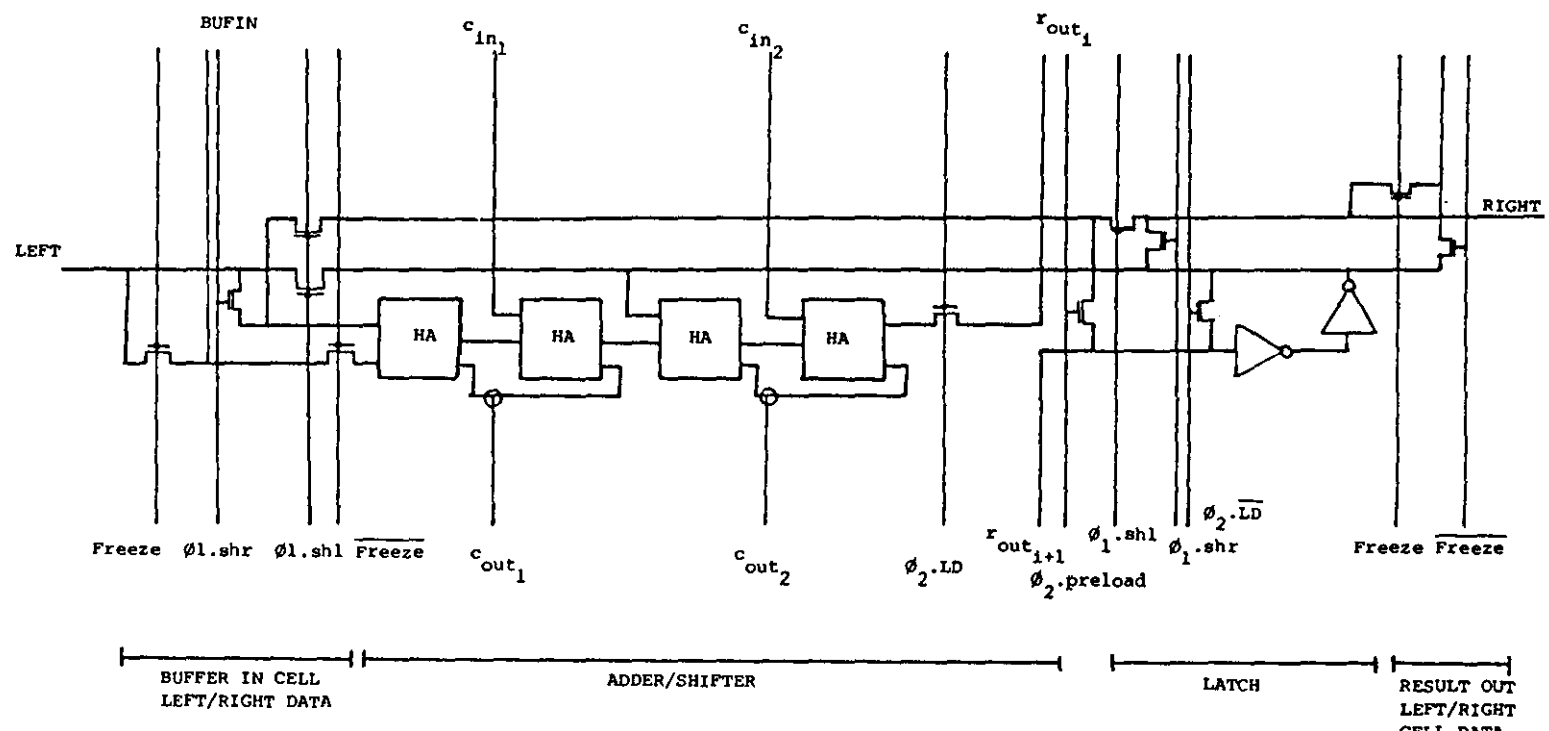


FIGURE 8.8.3: Single adder block



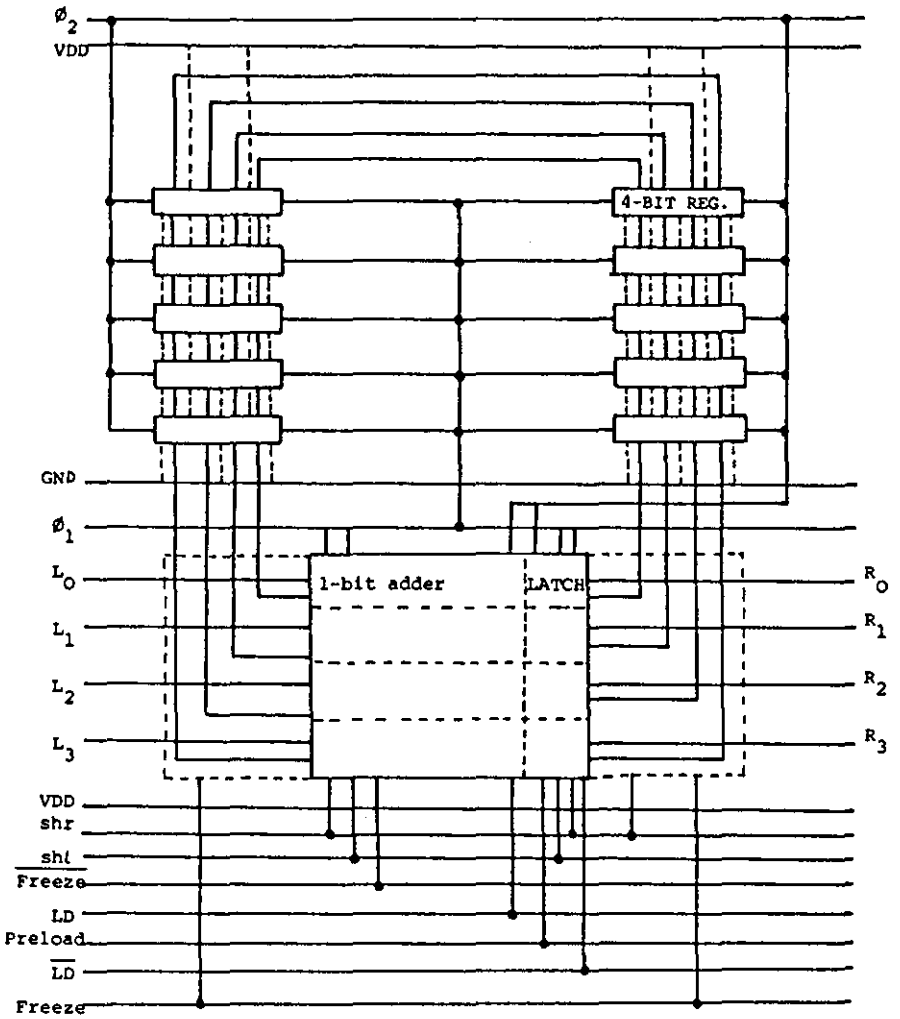
a) Switching for data paths for buffer output and array computation



b) 1-bit slice of 1-point hopscotch cell

Rightmost cell exchanges $Ø_2$.preload for $Ø_2$ disable.

FIGURE 8.8.4: Buffer interface and 1-bit adder slice



N.B. control inside adder block not shown for clarity see 1-bit slice.

FIGURE 8.8.5: Connections for a 4-bit, hopscotch cell uses 10 register buffer

choice of bit serial or bit parallel computation. Bit parallel is the best regarding speed, while bit serial reduces pin requirements. In the hopscotch schemes the conflict appears in the form of an I/O bottleneck caused by data buffering. The buffer allows the array to compute at full speed until the buffer is empty, and keeps the number of pins to an

acceptable level. However, the freezing of the array while results are output and the buffer re-loaded makes the hopscotch pay an increasingly heavy price as the number of chained chips increases. We remove the I/O bottleneck by considering a bit serial design for the 1-point hopscotch cell.

The bit serial cell is shown in Fig.(8.8.6), all the lines are 1-bit wide and the main switching is illustrated, there are additional control signals which are discussed later but omitted for clarity. The cell requires one input for g(x,t) one output for the new time level and four connections are introduced to simplify intercell shifting. Register u holds the starting value and overwrites its contents with the new result which is also sent to the output. The g(x,t) line doubles as the pre-loading connection, at start up time the adders are disabled while u is filled (using the st.ld control) from g(x,t). In addition the g(x,t) line is used to set a Cell Status Bit (CSB) which is discussed later.

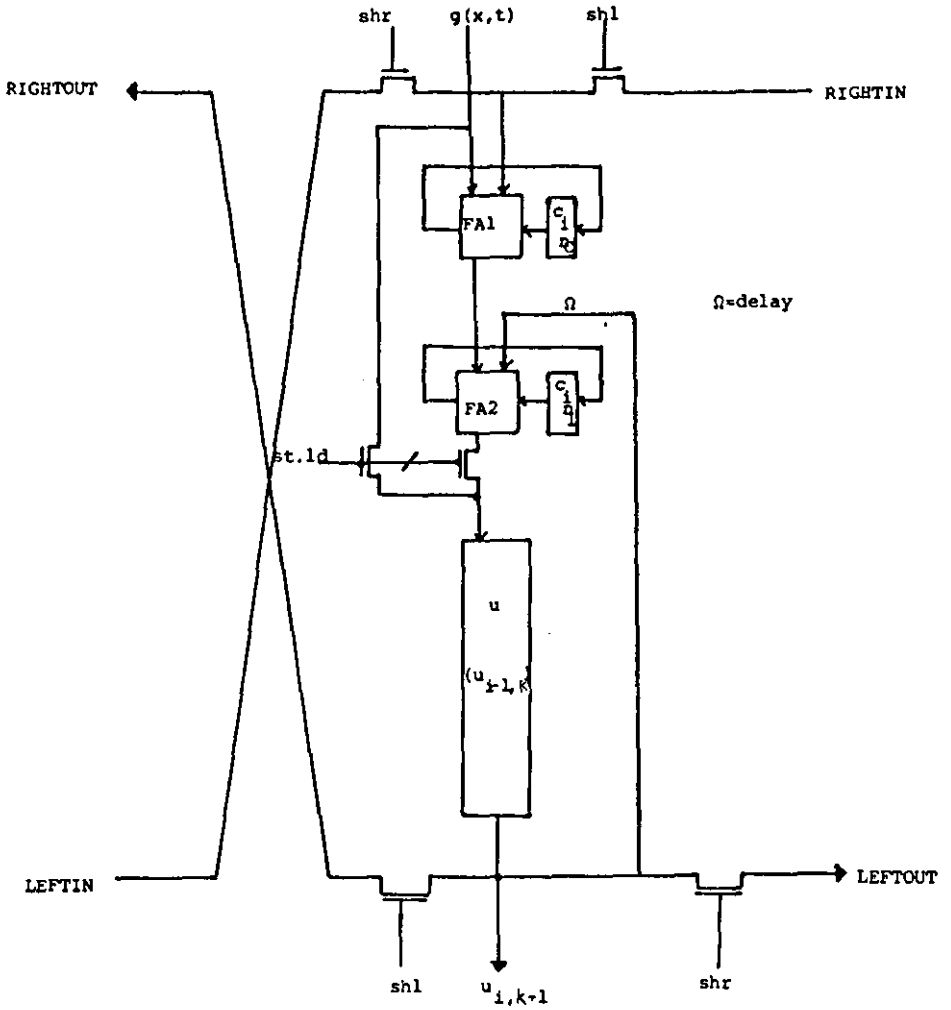
Suppose that the fixed point numbers have ℓ-bits, at start of a cycle u contains the current level value of cell i with the lowest significant bit (lsb) at the bottom and the most significant bit (msb) at the top. Computation is achieved as follows:

(i) The lsb of u loops back to FA2 where it is delayed, at the same time the lsb of cell i-1 or i+1 enters FA1 together with the correct D_{ik} accumulating part of (8.7.7).

(ii) The lsb of FA1 result enters FA2 with the delayed lsb of u from cell i completing the summation in (8.7.7).

(iii) After a further ℓ steps the msb of the sum enters u leaving only the divide by 2.

(iv) The adders carries are cleared and the lowest significant



C_1	C_2	COMMAND	LABEL
0	0	Load starting values	ST.LD
0	1	Shift left	shl
1	0	Shift right	shr
1	1	Load cell status bit	CSB

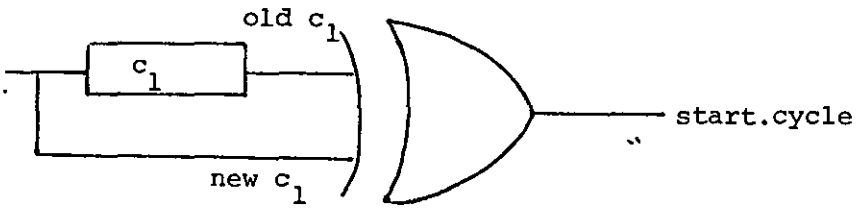
CONTROL TABLE

FA=1-bit full adder
u=register

FIGURE 8.8.6: Bit serial 1-pt. hopscotch cell

ℓ-1 bits of u are shifted down, with the msb (sign bit) overwriting itself. Finally, the left/right shift is switched ready for the next cell cycle. It follows that a single cell cycle requires ℓ+3 bit cycles, where a bit cycle is the delay through FA1 or FA2.

Shifting is easily implemented by the monitoring changes in the shl (or shr) command and generating the start cycle command using the status signal



Furthermore by pipelining the start-cycle command through FA1 and FA2 the adders can be neatly reset.

A simple 4-bit example is shown in Fig.(8.8.7) where,

$$u_{i,k+1} = \tfrac{1}{2}(u_{i-1,k}+u_{i+1,k}) + g(x,k)$$
$$= (\lfloor 3/2 \rfloor + 2/2 \div 1) = 3$$

using integer arithmetic for convenience. Note that after ℓ+3=7 cycles the u register contains 0011_2=3 and at the start of the next cycle is divided by 2. Thus, the output results must be multiplied by two to yield correct results or alternatively the bit shifted out during the divide collected.

The command table in Fig.(8.8.6) includes the instruction load cell status bit (CSB), it provides a more flexible array. When the 1-cell cycle 1-time level designs are used the number of points along the x-direction must be chosen to fill the array, otherwise the boundary values input to the ends of the array will be out of synchronisation and produce

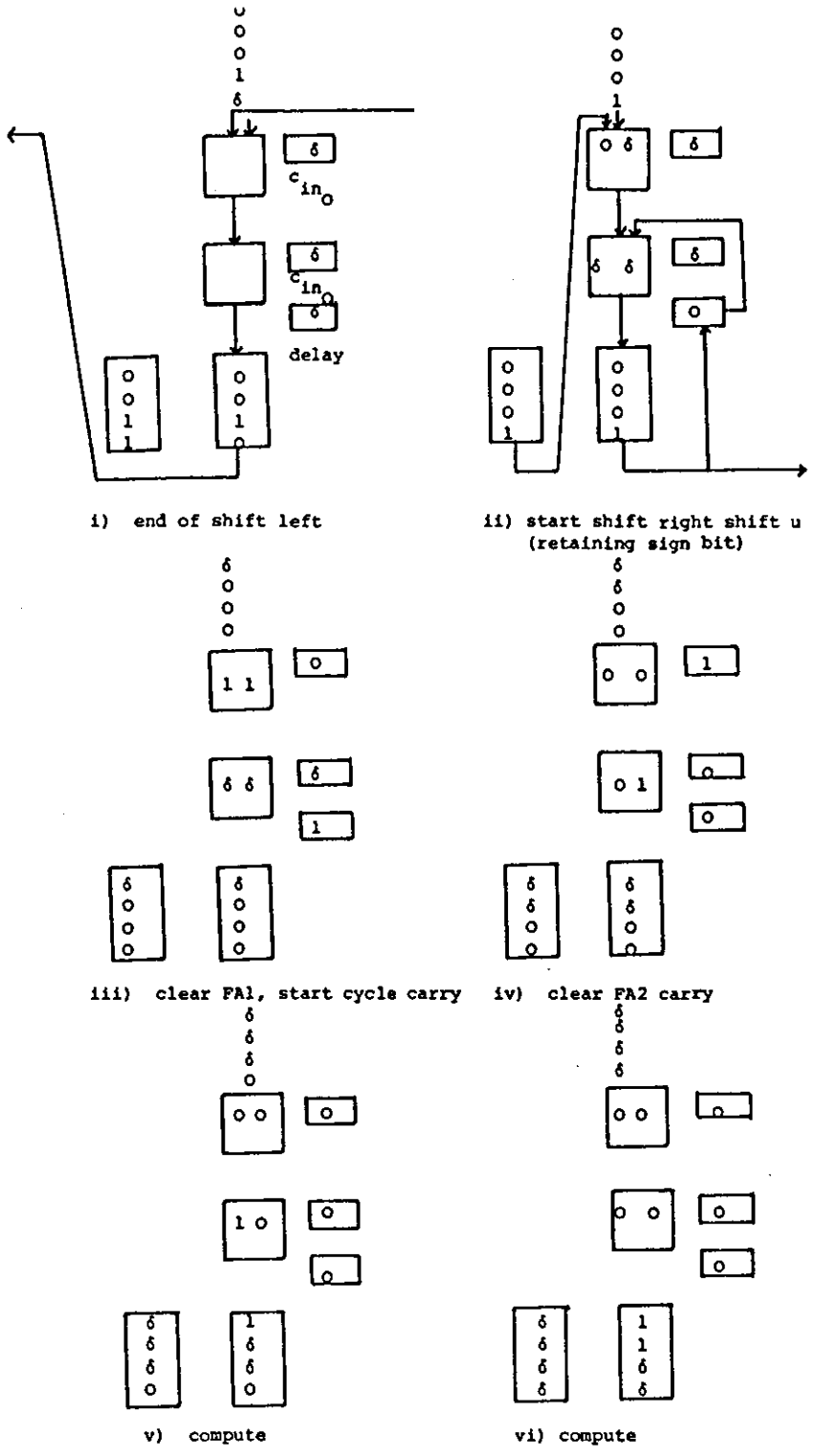


FIGURE 8.8.7: Bit serial computation 4-bit integers

erroneous results. The CSB indicates whether a cell is active or passive (i.e. used or not). An active cell will receive initial values and compute normally, a passive cell plays no part in calculation and is used to filter boundary results through the passive cells. To simplify the control it is best to left justify active cells and cause passive cells to always shift left. This added flexibility allows any sized problem up to and including the capacity of the array constructed to be solved. Extra control is required but to prevent the adders from modifying boundary values these can all be derived from the CSB.

In Fig.(8.8.8) the layout of a bit serial array is indicated for a

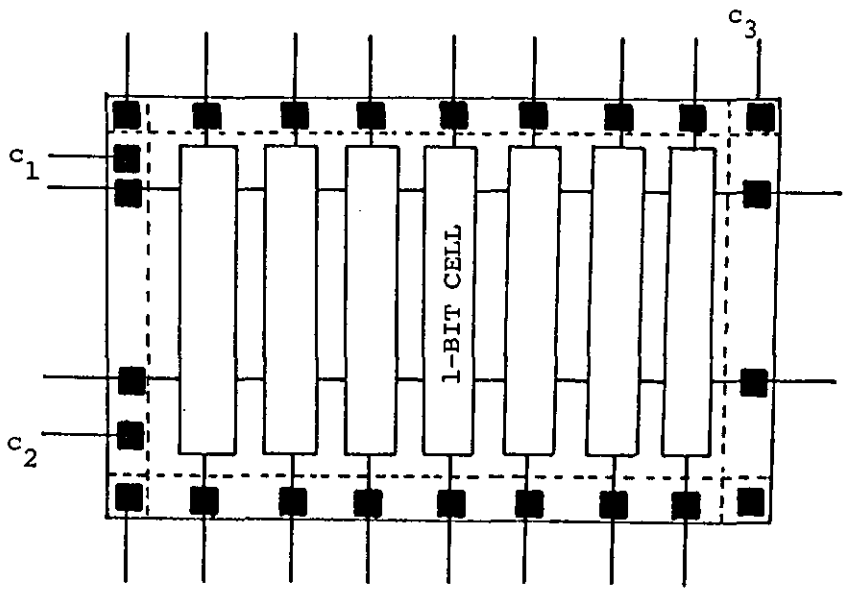


FIGURE 8.8.8: Floorplan

64 pin chip and the following pin allocations,

	3	VDD, GND, CLK
	3	c_1, c_2, c_3
	2	left boundary input/output
	2	right boundary input/output
	27	g(x,t) input
	27	result output
TOTAL	64	pins.

A 27 cell chip is feasible - allowing up to 54 grid point columns by a single chip.

Finally, consider the tradeoff between bit serial and bit parallel hopscotch arrays - when is the bit serial faster? The bit parallel case has a timing t_1 as follows,

$$t_1 = \text{levels} + \left\{\left\lceil \frac{\text{levels}}{\text{bufsize}} \right\rceil + 1\right\} (\text{Bufsize*cells}) \qquad (8.8.1)$$

because we must:

(i) Load the buffers with starting values. If Bufsize=number of buffer registers in a single segment, because shifting occurs right to left, the total loading/unloading time is (Bufsize*cells) cycles.

(ii) The results of levels are stored before output, thus $\left\lceil \frac{\text{levels}}{\text{Bufsize}} \right\rceil + 1$ is the number of times buffers must be loaded/unloaded.

(iii) Each level computed requires a single cell cycle.

For the bit serial case we have,

$$t_2 = \tfrac{1}{2}(\text{levels*word length}) \qquad (8.8.2)$$

where the wordlength is $\ell+3$, and one level is produced every $\ell+3$ cycles.

REMARK: The bit parallel cycle time is two additions the bit serial using two level pipelining only 1 hence the $\frac{1}{2}$ in (8.8.2). Next put L=levels, B=Bufsize, W=wordlength, C=no. of cells. For bit serial to outperform bit parallel,

$$t_2 \leq t_1$$

$$\tfrac{1}{2}(L*W) \leq L + \left\{\left\lceil \frac{L}{B} \right\rceil + 1\right\} (B*C)$$

or

$$\frac{(L*W)}{2C} \leq \frac{L}{C} + \left\{\left\lceil \frac{L}{B} \right\rceil + 1\right\} B$$

$$\frac{L*W}{2C} - \frac{L}{C} \leq \left(\frac{L}{B} + 2\right) B$$

$$\frac{L}{C}\left(\frac{W}{2} - 1\right) \leq L + 2B \qquad (8.8.3)$$

and fixing the wordlength at W=36 we have after some manipulation

$$\frac{L}{2}\left(\frac{17}{C} - 1\right) \leq B \qquad (8.8.4)$$

Consequently if $\frac{17}{C} \leq 0$ the bit serial is always better than bit parallel and the following table illustrates the effect of C>0

L \ C	2	4	6	8	10	12	14	16
2	8	4	2	2	1	1	1	1
4	15	7	4	3	2	1	1	1
6	30	13	8	5	3	2	1	1
8	60	26	15	9	6	4	2	1
16	120	52	30	18	12	7	4	1
32	240	104	59	36	23	14	7	2
64	480	208	118	72	45	27	14	4
128	960	416	235	144	90	54	28	8
512	1920	832	469	288	180	107	55	16

MAXIMUM NUMBER OF BUFFER REGISTERS PER CELL BEFORE BIT SERIAL BECOMES BETTER THAN BIT PARALLEL

8.9 SYSTOLIC GROUP EXPLICIT METHODS FOR HYPERBOLIC EQUATIONS

It should be clear that the techniques discussed above for parabolic equations carry over to other P.D.E.'s like elliptic and hyperbolic equations, provided that suitable hybrid molecules can be found. Recent developments by Sahimi [86] have extended the Group Explicit (GE) principle to simple first order hyperbolic equations, and indicate more flexible array designs which are briefly outlined below.

The basic hyperbolic equation we shall consider is of the form,

$$\frac{\partial u}{\partial t} + \frac{\partial u}{\partial x} = 0, 0 \leq x \leq 1 \ , \ t \geq 0 \ . \qquad (8.9.1)$$

As before, we consider the solution in the infinite rectangular strip

bounded by $0 \leq x \leq 1$, with finite difference approximations made at the intersecting points on a grid superimposed on the region with spacing $\Delta x = h$ along the x-axis, and $\ell = \Delta t$ along the t axis. We shall only briefly discuss the derivation of the new GE form for hyperbolic equations, the interested reader is referred to Sahimi [86].

The hyperbolic form (8.9.1) is expressed as a weighted finite difference analogue equation at a particular grid point, say $(x_i, t_j+\theta) = (i\Delta x, (j+\theta)\Delta t)$ $0 \leq \theta \leq 1$ as follows,

$$-\lambda[\theta\{(1-w)u_{i+1,j+1}+(2w-1)u_{i,j+1}-wu_{i-1,j+1}\}+(1-\theta)\{(1-w)u_{i+1,j}+ (2w-1)u_{ij}-wu_{i-1,j}\}] = u_{i,j+1}-u_{ij} \ , \tag{8.9.2}$$

with $\lambda = \frac{\Delta t}{\Delta x} = \frac{\ell}{h}$.

When w=1 we produce the equation,

$$(1+\lambda\theta)u_{i,j+1}-\lambda\theta u_{i-1,j+1} = (1-\lambda(1-\theta))u_{ij}+\lambda(1-\theta)u_{i-1,j} \tag{8.9.3}$$

and with w=0

$$(1-\lambda\theta)u_{i,j+1}+\lambda\theta u_{i+1,j+1} = (1+\lambda(1-\theta))u_{ij}-\lambda(1-\theta)u_{i+1,j} \tag{8.9.4}$$

At the point $((i-1)\Delta x, (j+\theta)\Delta t)$ (8.9.4) becomes,

$$\lambda\theta u_{i,j+1}+(1-\lambda\theta)u_{i-1,j+1} = -\lambda(1-\theta)u_{ij}+(1+\lambda(1-\theta))u_{i-1,j} \tag{8.9.5}$$

coupling equations (8.9.3) and (8.9.5) in groups of two adjacent points (i-1,j+1) and (i,j+1), etc., leads to the group explicit form,

$$Au_{j+1} = Bu_j \ , \tag{8.9.6}$$

where,

$$A = \begin{bmatrix} -\lambda\theta & (1+\lambda\theta) \\ (1-\lambda\theta) & \lambda\theta \end{bmatrix} \ , \quad B = \begin{bmatrix} \lambda(1-\theta) & 1-\lambda(1-\theta) \\ 1+\lambda(1-\theta) & -\lambda(1-\theta) \end{bmatrix}$$

$$u_{j+1} = \begin{bmatrix} u_{i-1,j+1} \\ u_{i,j+1} \end{bmatrix} \ , \ u_j = \begin{bmatrix} u_{i-1,j} \\ u_{i,j} \end{bmatrix}$$

which in explicit form yields,

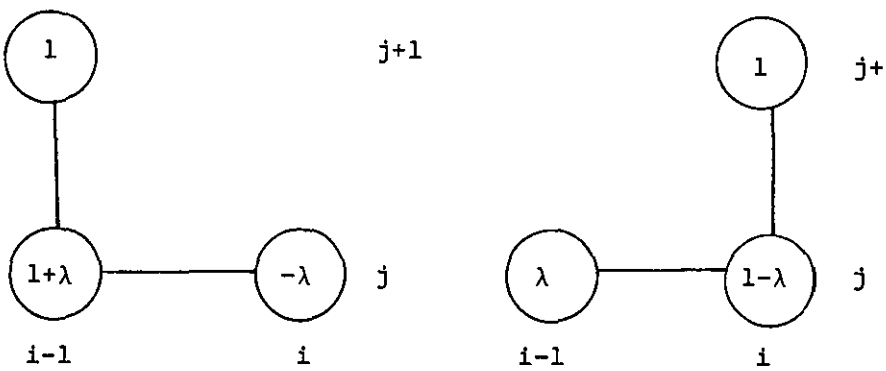
$$u_{j+1} = A^{-1}Bu_j \ , \tag{8.9.7}$$

where,

$$A^{-1}B = \begin{bmatrix} (1+\lambda) & -\lambda \\ \lambda & (1-\lambda) \end{bmatrix}$$

representing the molecule rules,

a)



(8.9.8)

and the equations:

$$\left.\begin{aligned} &\text{a)}\quad u_{i-1,j+1} = (1+\lambda)u_{i-1,j} - \lambda u_{ij} \\ &\text{b)}\quad u_{i,j+1} = \lambda u_{i-1,j} + (1-\lambda)u_{ij} \end{aligned}\right\} \ , \tag{8.9.9}$$

Thus, in a similar manner to (8.1.27)-(8.1.34) we can define various GE schemes. If there are m points in the region along x including the right boundary ungrouped points occuring in the (m-1)th or 1st positions and are computed by,

$$\begin{aligned} &\text{a)}\quad u_{m-1,j+1} = [(1+\lambda(1-\theta))u_{m-1,j} - \lambda(1-\theta)u_{m,j} - \lambda\theta u_{m,j+1}]/(1-\lambda\theta) \\ &\text{b)}\quad u_{1,j+1} = [\lambda(1-\theta)u_{0,j} + (1-\lambda(1-\theta))u_{1,j} + \lambda\theta u_{0,j+1}]/(1+\lambda\theta) \end{aligned} \tag{8.9.10}$$

Next define (m-1)*(m-1) matrices,

$$G_1 = \begin{bmatrix} G^{(1)} & & & \\ & G^{(2)} & & \\ & & \ddots & \\ & & & G^{(\frac{1}{2}(m-2))} \end{bmatrix}_{-1} \ , \quad G_2 = \begin{bmatrix} 1 & & & \\ & G^{(1)} & & \\ & & \ddots & \\ & & & G^{(\frac{1}{2}(m-2))} \end{bmatrix}$$

$$\hat{G}_1 = \begin{bmatrix} 1 & & & \\ & G^{(1)} & & \\ & & \ddots & \\ & & & G^{(\frac{1}{2}(m-3))} \end{bmatrix}^{-1}, \quad \hat{G}_2 = \begin{bmatrix} G^{(1)} & & & \\ & G^{(2)} & & \\ & & \ddots & \\ & & & G^{(\frac{1}{2}(m-1))} \end{bmatrix}$$

with,

$$G^{(i)} = \begin{bmatrix} -1 & 1 \\ -1 & 1 \end{bmatrix}$$

and the following schemes can be derived

m Even:

There are (m-1) internal points - an odd number requiring boundary cells hence we have,

(i) Group Explicit ungrouped Right point (GER) scheme

$$(I+\lambda\theta G_1)u_{j+1} = (I-\lambda(1-\theta)G_1)u_j + b_1 ,$$
$$b_1 = (0,0,\ldots,-\lambda(1-\theta)u_{m,j}-\lambda\theta u_{m,j+1})^T \qquad (8.9.11)$$

(ii) Group Explicit ungrouped Left point (GEL) scheme

$$(I+\lambda\theta G_2)u_{j+1} = (I-\lambda(1-\theta)G_2)u_j + b_2$$
$$b_2 = (\lambda(1-\theta)u_{0,j}+\lambda\theta u_{0,j+1},0,\ldots,0)^T \qquad (8.9.12)$$

(iii) Single Alternating Group Explicit (SAGE) scheme

$$\left.\begin{aligned} (I+\lambda\theta G_1)u_{j+1} &= (I-\lambda(1-\theta)G_1)u_j + b_1 \\ (I+\lambda\theta G_2)u_{j+2} &= (I-\lambda(1-\theta)G_2)u_{j+1} + b_2 \end{aligned}\right\} j=0(2),\ldots \qquad (8.9.13)$$

(iv) Double Alternating Group Explicit (DAGE) scheme

$$\left.\begin{aligned} (I+\lambda\theta G_1)u_{j+1} &= (I-\lambda(1-\theta)G_1)u_j + b_1 \\ (I+\lambda\theta G_2)u_{j+2} &= (I-\lambda(1-\theta)G_2)u_{j+1} + b_2 \\ (I+\lambda\theta G_2)u_{j+3} &= (I-\lambda(1-\theta)G_2)u_{j+2} + b_2 \\ (I+\lambda\theta G_1)u_{j+4} &= (I-\lambda(1-\theta)G_1)u_{j+3} + b_1 \end{aligned}\right\} j=0(4)\ldots \qquad (8.9.14)$$

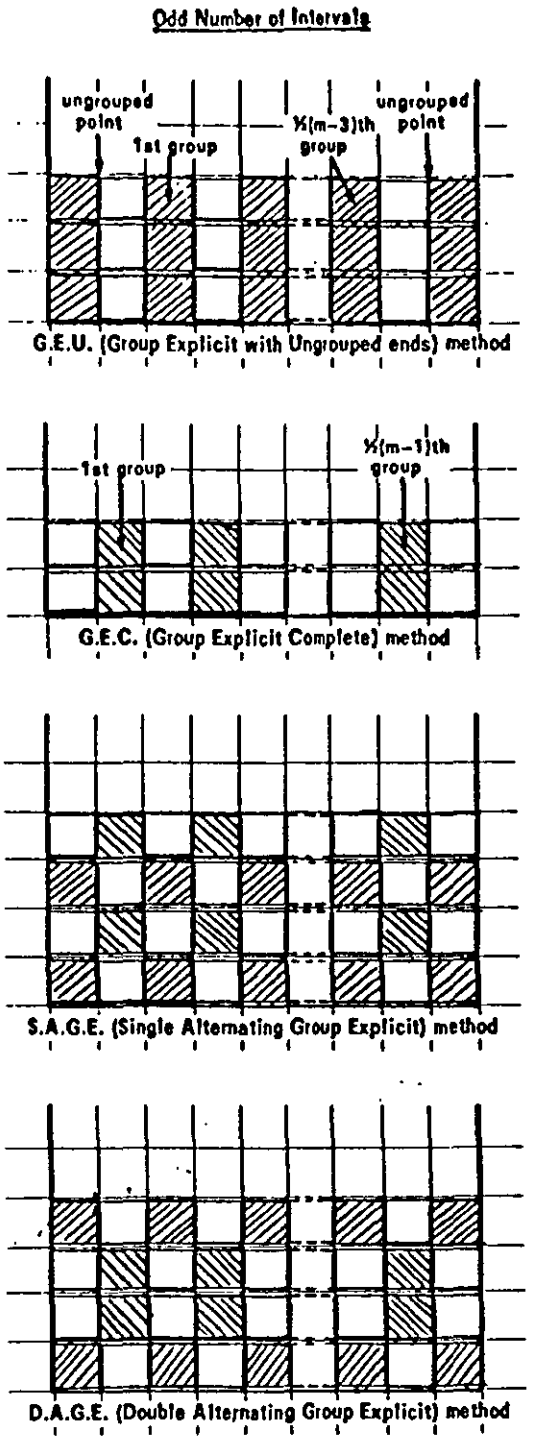


FIGURE 8.9.1: GE computation for hyperbolic equations

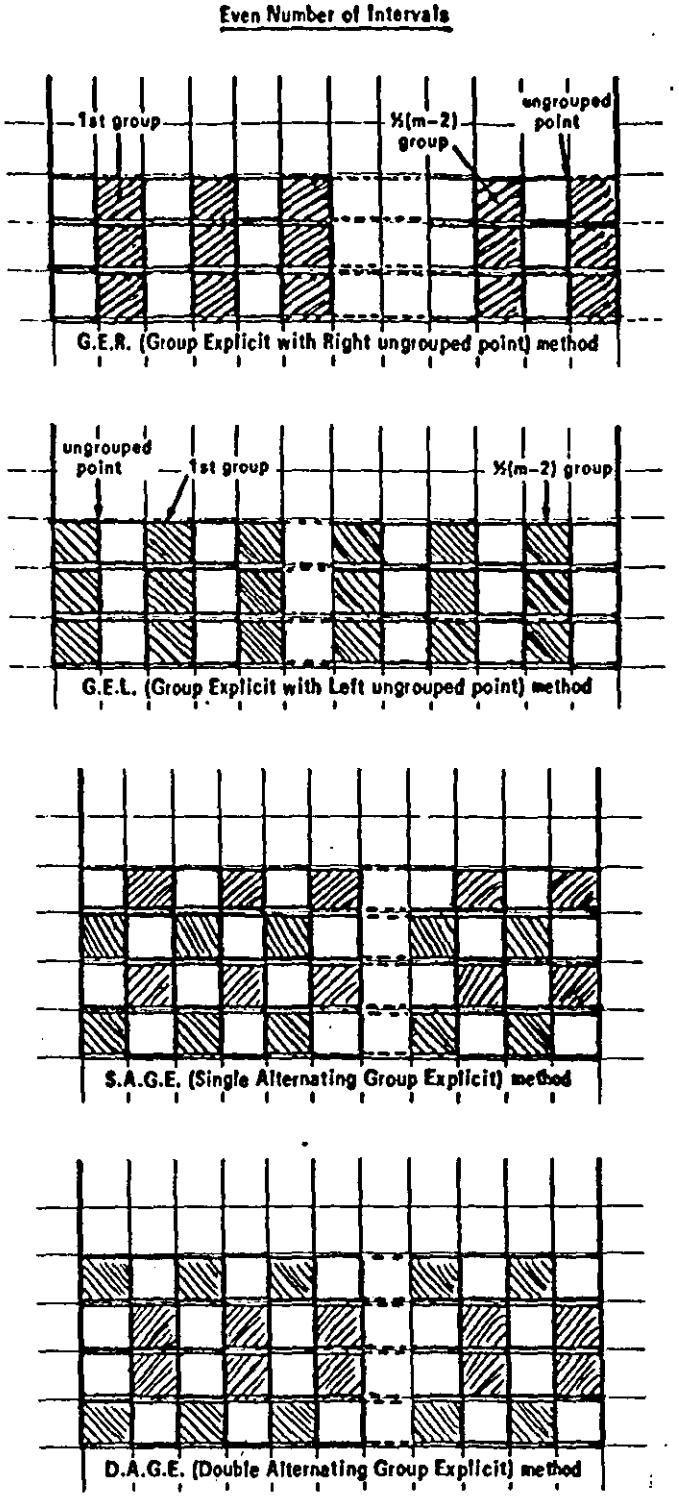


FIGURE 8.9.1(cont).

m Odd:

There are (m-1) internal points which this time is even, yielding,

(i) Group Explicit Ungrouped points (GEU) scheme

$$\left.\begin{aligned}(I+\lambda\theta\hat{G}_1)u_{j+1} &= (I-\lambda(1-\theta)\hat{G}_1)u_j + b_3\\ b_3 &= (\lambda(1-\theta)u_{0,j}+\lambda\theta u_{0,j+1},0,\ldots,0,-\lambda(1-\theta)u_{m,j}-\lambda\theta u_{m,j+1})^T\end{aligned}\right\} \quad (8.9.15)$$

(ii) Group Explicit Complete (GEC)

$$(I+\lambda\theta\hat{G}_2)u_{j+1} = (I-\lambda(1-\theta)\hat{G}_2)u_j \quad (8.9.16)$$

(iii) SAGE

$$\left.\begin{aligned}(I+\lambda\theta\hat{G}_1)u_{j+1} &= (I-\lambda(1-\theta)\hat{G}_1)u_j + b_3\\ (I+\lambda\theta\hat{G}_2)u_{j+2} &= (I-\lambda(1-\theta)\hat{G}_2)u_{j+1}\end{aligned}\right\} \; j=0(2)\ldots \quad (8.9.17)$$

(iv) DAGE

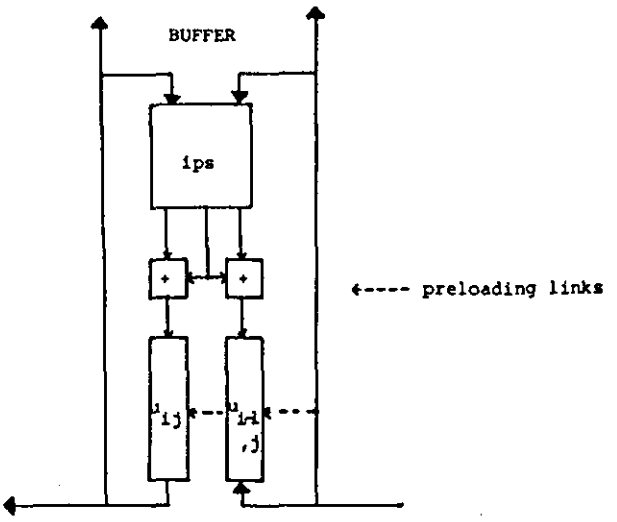
$$\left.\begin{aligned}(I+\lambda\theta\hat{G}_1)u_{j+1} &= (I-\lambda(1-\theta)\hat{G}_1)u_j + b_3\\ (I+\lambda\theta\hat{G}_2)u_{j+2} &= (I-\lambda(1-\theta)\hat{G}_2)u_{j+1}\\ (I+\lambda\theta\hat{G}_2)u_{j+3} &= (I-\lambda(1-\theta)\hat{G}_2)u_{j+2}\\ (I+\lambda\theta\hat{G}_1)u_{j+4} &= (I-\lambda(1-\theta)\hat{G}_1)u_{j+3} + b_3\end{aligned}\right\} j=0(4)\ldots \quad (8.9.18)$$

An example of the group computation ordering is given in Fig.(8.9.1) illustrating the independent nature of non-alternating GE schemes, which is attractive from a systolic viewpoint.

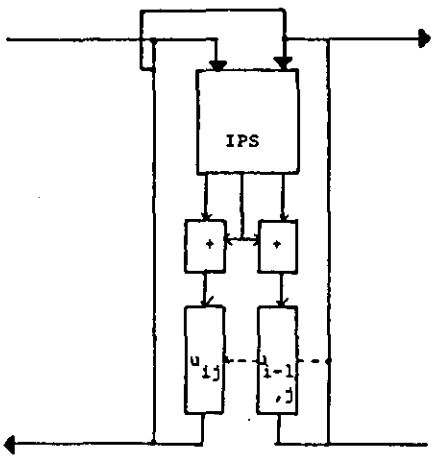
First we develop a basic cell for our design, based like the parabolic scheme on a single level single cycle implementation using a linearly connected array of $\frac{1}{2}(m)$ GE cells. Clearly (8.9.9) can be expressed as,

$$\left.\begin{aligned}u_{i-1,j+1} &= u_{i-1,j} + \lambda\Gamma\\ u_{i,j+1} &= u_{ij} + \lambda\Gamma\\ \Gamma &= u_{i-1,j}-u_{ij}\end{aligned}\right\} \quad (8.9.19)$$

resulting in the simple arrangement of Fig.(8.9.2a), from which a number



a) GER, GEL, GEU, GEC cell



b) SAGE, DAGE cell

FIGURE 8.9.2: Hyperbolic GE-cell

of immediate benefits over the parabolic GE-cell are apparent:

(i) The GE cell requires a single ips cell and two adders (rather than two ips used in the parabolic scheme).

(ii) No control program is required for queueing up the cell operands.

(iii) Only a single register is required to hold coefficient data.

(iv) No points are borrowed from adjacent GE cells.

The cell modifications are due mainly to simplifications in the finite difference formula (hyperbolic rather than parabolic) and saves hardware and time. For example, the hyperbolic scheme requires only 2 ips cycles to compute a group compared with at most 6 ips for the parabolic case yielding the immediate speedup $S_p=3$.

REMARK: Remember that these comparisons indicate only the suitability of the GE problem to systolic arrays, not the choice of hyperbolic or parabolic equations for a problem.

Thus the hyperbolic GE method seems better suited to systolic computation. The amount of area consumed by the hyperbolic cell for data routing is also less than that of the parabolic form. Consider the GER, GEL, GEU, and GEC schemes for both hyperbolic and parabolic methods. In the hyperbolic schemes the individual groups are disjoint, i.e. there are no overlapping points in adjacent groups, whereas the parabolic schemes computation can only proceed with point sharing. Consequently the hyperbolic array requires only single uni-directional connections which are activated only in loading (starting values) and unloading results. The removal of left and right data shuffling also removes all dataflow control yielding a simple and compact cell.

We still have the problem of boundary cells, and a trivial observation of (8.9.10) shows that ungrouped point calculations require more time than full groups. In a parabolic array this degrades performance as ungrouped points supply data to adjacent group cells. However the independent nature of the hyperbolic scheme, together with the concept of an incomplete array suggests an alternative partition of calculations between host/array. By assigning the boundary calculations to the host machine the GE-array is reduced to a GEC scheme again simplifying the

design. In particular:

(i) The array can be built with a fixed number of cells (on a chip)

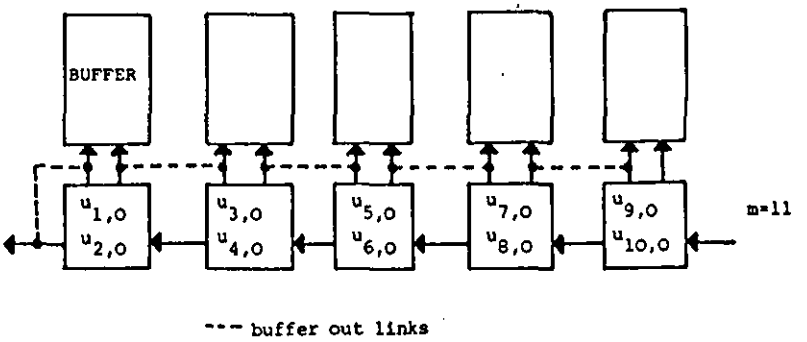
(ii) The rectangular solution strip can be decomposed into k strips which fit the array in (i) and computed sequentially by multi-pass.

(iii) We can linearly connect k chips to solve the complete problem in parallel.

(iv) If a strip has k'<k groups, unused cells can be padded with dummy values, and the independence property ensures adjacent (true groups) are not contaminated by invalid computations.

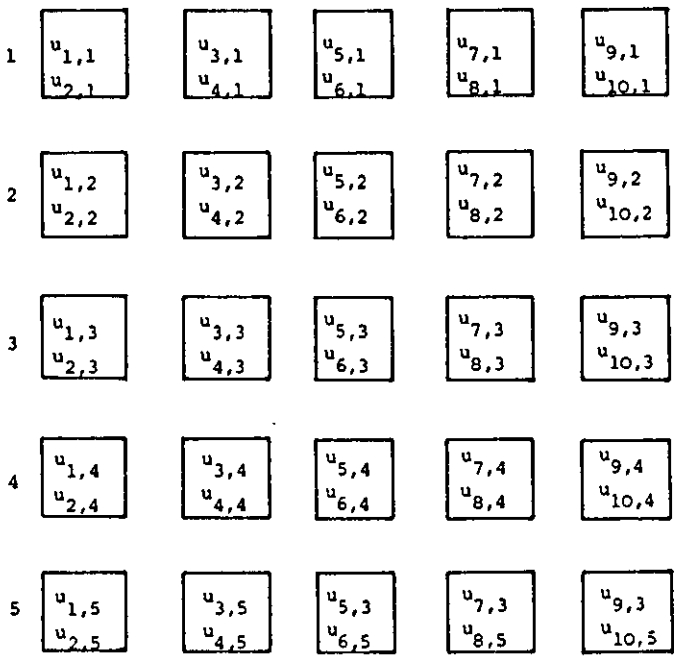
These advantages are far superior to those for parabolic schemes, which force the array to be proportional to the width of the solution strip. Indeed the hyperbolic GE schemes produce an array independent of both the time levels and number of points along the x-axis spawning a number of alternative connection strategies varying speed against area.

Snapshots of array operation for a buffer GE-array are shown in Fig.(8.9.3). We consider the computation of simple hyperbolic schemes using a "bag-of" approach. Essentially we suppose a finite number p of GEC chips each implementing a k cell array. We examine the computations associated with applying a single chip, or a parallel connection of p chips using buffered or unbuffered array designs. Apart from the case when only the final time level is required (and buffers can be removed) buffering implies bit parallel and non-buffering bit serial schemes respectively. A k cell chip can compute a strip k groups wide containing 2k points, a region wider than this uses a number of strips each k cells wide. Our "bag of" chips must therefore contain at least,



--- buffer out links

a) GEC array (with starting values) for producing all results



b) Five cycles of the hyperbolic GEC scheme

REMARK: Notice the independence of the computation

FIGURE 8.9.3: Operating hyperbolic GE array

$$p = \left\lceil \frac{\text{number of groups}}{k} \right\rceil$$

identical GEC chips, to solve the whole region in parallel. We consider the 1-chip versus p-chip bag under five cases:

(i) A GEC array with buffers

(ii) A GEC array without buffers

(iii) A one chip arrangement of (i) and (ii)

(iv) A p-chip arrangement of (i) and (ii)

(v) A single time level approach.

We further suppose that for the various schemes the number of groups is

$$\left.\begin{matrix}\text{GER}\\ \text{GEL}\end{matrix}\right\} = \tfrac{1}{2}(m-1), \quad \text{GEU} = \tfrac{1}{2}(m-3), \quad \text{GEC} = \tfrac{1}{2}(m-2)$$

Buffered GEC:

a) 1-chip sequential (multipass) scheme

The cost of the array is derived as follows:

loading λ parameter into k cell chip	2k cycles
loading initial values into k cells	2k cycles
time to fill buffer with 2b registers	b cycles

The strip is computed to time level $t=t_z$ by filling the buffer $\lceil t/b \rceil$ times and consequently emptying this many times, and the buffer segment of each cell uses 2b cycles to empty. Thus, for k cells the time is 2bk for buffer load/empty;

REMARK: Cost unit time is equivalent to two ips cycles, thus ips timings are obtained by multiplying by 2 to yield an ips cycle timing. A single group strip therefore has a time

$$\begin{aligned} T_O &= (\text{load time}) + (\text{number of times buffer empties}) * \\ &\qquad [\text{cost of buffer empty} + \text{cost of filling buffer}] \\ &= 4k + \lceil t/b \rceil \{2bk+b\} \\ &\leq t(2k+1) + 2k(2+b) + b \end{aligned} \tag{8.9.20}$$

Thus for a multipass buffered scheme,

$$T_{sb} \leq p[t(2k+1)+2k(2+b)+b] \tag{8.9.21}$$

b) p-chip parallel scheme

There are two ways to connect the p chips in parallel.

Case (a): p-chips are chained together creating a multi-chip GEC array. This is equivalent to a single chip with kp cells. Thus,

$$T_{PCB} = 4kp + \lceil t/b \rceil \{2bkp + b\} \tag{8.9.22}$$

follows from (8.9.20).

Case (b): Each strip is solved in parallel on its own chip isolated from the rest yielding,

$$T_{PIS} = T_O \leq t(2k+1)+2k(2+b)+b \tag{8.9.23}$$

Unbuffered GEC:

In this set up we assume enough chip pins to carry the results of k cells directly off the chip.

a) Multipass scheme:

No buffer loading is required, and starting values can be loaded in parallel, hence,

$$T_1 = (\text{time for loading}) + t$$
$$= 2 + t . \tag{8.9.24}$$

Applying a single chip p times gives,

$$T_{sub} = p(2+t) . \tag{8.9.25}$$

b) Parallel scheme:

Case (i) with chips connected serially (sequential loading)

$$T_{pcub} = 4kp + t \tag{8.9.26}$$

Case (ii) with chips isolated

$$T_{piub} = 2 + t . \tag{8.9.27}$$

As noted earlier unbuffered schemes imply bit serial computation. If we substitute s cycles for each cycle of a bit parallel scheme where s is proportional to the word length (8.9.25)-(8.9.27) are revised to yield,

$T_{sub} = sp(2+t)$, $T_{pcub} = s(4kp+t)$ and $T_{piub} = s(2+t)$.

Single time level:

In this case no buffers are required for the bit parallel scheme and a bit serial scheme uses its output pins only once. By shifting results sequentially left or right off the array the computations require

$$T_2 = \text{(time for load)} + \text{(computation for t levels)} + \text{(time of unload)}$$
$$= 4k + t + 2k = 6k+t$$

Hence multipass (T_{sol}), parallel (T_{pcol}) and parallel but independent (T_{piol}) are given by,

$$\left.\begin{aligned} T_{sol} &= p(t+6k) \\ T_{pcol} &= t+6pk \\ T_{piol} &= t+2 \end{aligned}\right\} \qquad (8.9.28)$$

cycles respectively.

The isolation of groups computing columns up the solution region admits many strategies for interleaving different hyperbolic problems on the same hardware. The p-chip 'bag of' is the simplest approach which allows p problems to be solved simultaneously with different λ parameters in each chip - and solving problems with greater than k groups in multipass. A more interesting problem is interleaving on a single chip or serially connected group of chips, as indicated by Fig.(8.9.4). A straightforward approach is simply to queue problemsone behind the other in multipass, or along the cells, filling as many cells as possible. The problem here is that the last user (problem) has to wait for other problems in front to be filtered through. A more flexible approach would be a simple interleaving strategy to give all users a reasonable response time. Clearly the time of array operation is computed by substituting k=(sum of all groups) in the above timings.

The hyperbolic scheme is also well suited to fault tolerance as

Fig.(8.9.5) demonstrates. For buffered and serial loading schemes there is only a single unidirectional line used only in preloading. This single line makes re-routing around faulty cells simple, and allows chip performance in conjunction with multipass computation to degrade gracefully. For a bit serial or unbuffered approach where loading/unloading proceeds in parallel no re-routing is required, and faulty cells can simply be discarded.

Finally we consider the Alternating Group Explicit forms of the hyperbolic GE method. Like the parabolic AGE the hyperbolic forms can be implemented with linear arrays that shift group data left or right, overwriting points in adjacent cells to provide alternation. A simple cell for hyperbolic AGE is shown in Fig.(8.9.2b), clearly the alternation of GER and GEL schemes provides a patchwork pattern across the region (see Fig.(8.9.1)). This re-establishes the group dependency relationships demanding that:

(i) All level t is computed before level t+1 can start.

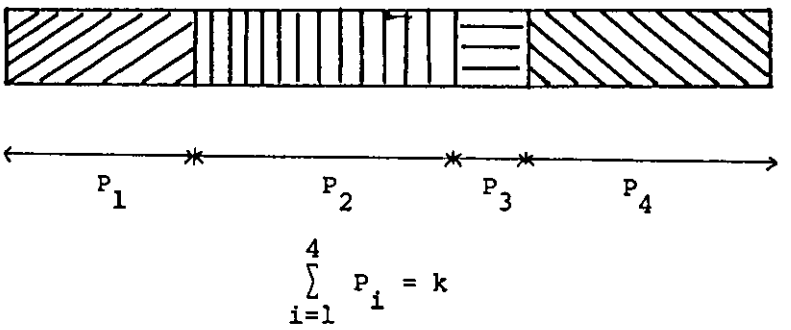
(ii) Boundary calculations must be included in the array (requiring a larger cell cycle time)

and preventing:

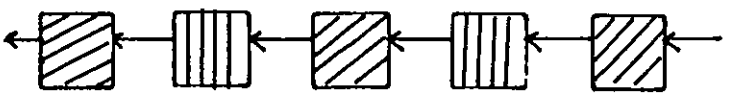
(i) Interleaving of problems

(ii) Multipass computation

The latter features an extremely attractive implementation characteristics. A final design must balance the use of flexible, fast, and fault tolerant non-alternating GE arrays against the restrictive AGE method with unconditional stability (larger stepsizes) hence reduced cells and level calculations. We conclude that hyperbolic GE methods offer greater opportunities for hard-systolic devices than the corresponding parabolic problems.

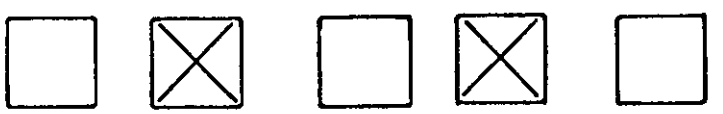


a) Queueing of multiple problem instances

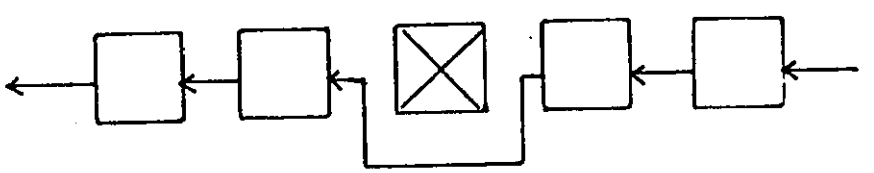


b) Interleaving of problems

FIGURE 8.9.4: Problem interleaving



a) Faulty cells



b) Routing around a faulty cell

FIGURE 8.9.5: Fault tolerance

8.10 SUMMARY

The main objective of this chapter has been to challenge two basic premises currently adopted for solving P.D.E.'s, which can be summarized in the form of the following two questions:

(i) Is the algorithmic or geometric interpretation of an algorithm the best approach to solving a P.D.E. in parallel(in particular by systolic arrays)?

(ii) Can a fast, area efficient parallel and conditionally stable design outperform slower unconditionally stable alternatives?

To illustrate the discussion we considered the solution of the 1-D (heat conduction) and 2-D (unsteady diffusion) parabolic problems, with particular emphasis placed on the 1-D case due to its simpler form.

In order to answer the first question we discarded a linear algebraic formulation of the P.D.E. problem interpreting the grid points of the solution region as a tableau of elements to be generated or modified by use of computational molecules relating the grid elements. In the case of the 1-D problem an open ended table was apparent, implying table generating techniques (from Chapter 7) were applicable and gave rise to three main types of array.

(i) Non-stationary array: A design using a cascaded linear array based on the intuitive iterative algorithmic solution of linear systems, and which acted as a benchmark for other 1-D designs. The principle attribute of the array being the non-stationary movement of successive time-level approximation from the same grid column through the array.

(ii) Column-by-column array: Here column i of grid point approximations remained stationary and tied to cell i of the linear array. The problem was similar to the generation of an open ended trapezium type table.

Basic cells used the asymmetric molecules of Saul'ev [64], to implement a systolic marching technique.

(iii) Row-by-row array: Again a stationary array with columns 2i-1 and 2i, i=1(1)½(m) tied to cell i of the array. The problem used a rectangular table generation pattern producing a single table row (or time level) every cell cycle. Basic cells utilised the unified Group Explicit (GE) molecule of Evans & Abdullah [83b]. Similarly for the 2-D case three types of design were considered for manipulating the regions grid points.

(i) A non-stationary array: Another cascaded iteration array derived from the algorithmic formulation of an iterative matrix problem derived from the 2-D finite difference approximation and used for benchmarking 2-D systolic designs.

(ii) A wavefront processor: Using the 2-D asymmetric approximations and occurrring in two forms:

a) A 2-D mesh with highly efficient pipelining of wavefronts, and useful for fast calculation up to a certain level t_z with no intermediate output.

b) A 1-D linear asymmetric marching processor (LAMP) which reduced hardware by multipass simulation of a) with each pass a single wavefront. This array had the natural capability of outputting all intermediate time levels up to and including t_z.

(iii) A mesh scheme: which adapted the Group Explicit molecules to achieve full parallel operation of processors, and appeared in two forms:

a) A 2-D mesh of reduced instruction set processors, evaluating a single table update in one cell cycle.

b) A 1-D array of macro GE-cells (incorporating a systolic ring) and computing molecules according to a generalised molecule

template. Table updates were achieved by multipass with one update per pass.

For t_z levels and m divisions along the x and y-directions the 1-D case had m-1 and the 2-D case $(m-1)^2$ initial values. The cascaded schemes required $O(m+t_z)$ ips cycles and $O(t_z)$ basic ips cells in the 1-D case, and $O(m^2+t_z)$ ips cycles and $O(mt_z)$ basic ips cells (neglecting synchronising delay cells) for the 2-D case. In contrast the asymmetric molecule arrays required $O(m)$ and $O(m^2)$ basic ips cells for the 1-D, 2-D (LAMP) and 2-D wavefront scheme respectively, and apart from the LAMP array with $O(mt_z)$ time had the same order of magnitude in timings. Thus, the algorithmic schemes favoured wide regions with few time levels, the geometric forms narrow regions with many levels to optimise the array speed/area tradeoff. Consequently, by fixing m, for any substantial calculations with a significant number of time levels and adopting a geometric approach area savings followed immediately. The actual cell savings depended upon the number of iteration arrays included in the cascaded array. For a fixed number of iteration arrays ($\tilde{t}_z>0$) the algorithmic approach times become $O(\left\lfloor \frac{t_z}{\tilde{t}_z} \right\rfloor (m+\tilde{t}_z))$ and $O(\left\lfloor \frac{t_z}{\tilde{t}_z} \right\rfloor (m^2+\tilde{t}_z))$ for the 1-D and 2-D cases. It follows that for finite hardware the geometric schemes run faster and seriously challenge the intuitive algorithmic arrays.

Now having established the answer to our first question above the next logical step was to produce the 'best' geometric array. For purposes of argument we define the 'best' array to be one which:

1. improves the accuracy of grid point approximations (i.e. reduces approximation error)
2. reduces computation time and array area further.

1. is controlled by the truncation error terms associated with the finite

difference approximation used to model the P.D.E. For instance the asymmetric forms had truncation errors $O(\alpha\frac{1}{h}+h^2+\ell)$ and $O(\alpha h+\beta h+\ell+h^2)$ for the 1-D and 2-D problems respectively, where h=stepsize in the x-direction and ℓ the time step, α,β suitably chosen parameters. 2. is dictated by the simplicity of the computational molecule and the sizes of t_z and m. Clearly reducing t_z and m to speedup and compact the arrays requires increases in ℓ and h which in turn increases the approximation error of the asymmetric computations. Likewise reducing h and ℓ to provide more accurate results increases array time and area. Hence 1. and 2. provided conflicting goals. We resolved the problem by alternating the application of asymmetric formulas which retained the simple molecule-cell structures and improved accuracy because of truncation error term cancellations (due to opposite signs). For the 1-D case alternation had the unfortunate effect of sequentialising array computations yielding low processor efficiency and making the algorithmic scheme attractive again. To overcome this difficulty the Group Explicit (GE) methods of Evans & Abdullah [83b] were adopted and new arrays developed, using a hybrid molecule consisting of unified asymmetric molecules. The loosely coupled structure of the GE methods allowed parallel operation of array cells yielding high efficiency while retaining truncation term cancellations to maintain accuracy, at the expense of a more complex basic cell. Various types of array corresponding to positioning of grouped and ungrouped mesh points were developed, and alternating Group Explicit (AGE) arrays devised implementing an unconditionally stable method. Simple data shuffling and cycling operations were then introduced and the principle of cell unification applied to derive generic 1-D and unified 2-D arrays, implementing all the GE techniques. The former 1-D scheme adopted simple

left/right data shifts, the 2-D method adopted a universal molecule template evaluated by accumulating terms on a systolic ring. As the 1-D molecule fitted the 2-D molecule the unified array could also be used to simulate the 1-D computation.

Next we produced FAST arrays based on restricted choices of ℓ and h, with $r=\ell/h^2$, the above asymmetric formulae can be shown to be unconditionally stable for $r\geq 0$. Likewise the simple GE schemes are conditionally stable for $r\leq 1$, and the AGE methods again unconditionally stable. By restricting r=1 approximating equations and hence molecules are simplified with terms disappearing altogether and coefficients involving r becoming constant. The FAST AGE array followed naturally, yielding a speedup $S_p=6$ over the general arrays, from simplifying cell computation. However fixing r also fixed the truncation error. Consequently the array speed increase was used to offset the larger stepsizes achieveable by the general scheme by constructing more accurate approximations to a number of intermediate levels. It followed that for $r\leq 5$ the FAST AGE outperformed the general AGE array.

Fixed r values, where molecule terms disappeared also gave rise to incomplete versions of the P.D.E. solvers, using the hopscotch technique (Gourlay [70]), in which only part of the whole solution region was produced. Omitted grid-points being placed in positions where they could be easily derived from array results. The technique was discussed and arrays described for the ODD-EVEN, 1-point and 2-point (FAST AGE) hopscotch schemes. The ODD-EVEN method produced an unconditionally stable array by using a unified 2-point molecule derived from sequential application of the classical explicit and implicit formulae. The array computed at the same speed as the AGE array but had truncation error of

$O(\ell+h^2)$ making it less accurate. The 1-point scheme adopted the classical explicit molecule (conditionally stable for $r\leq\frac{1}{2}$) with $r=\frac{1}{2}$ and truncation error $O(\ell+h^2)$, and produced a speedup of $S_p=12$ over the ODD-EVEN and AGE arrays while using only $\frac{1}{2}m$ inner product cells. The speed-up was again interpreted as a method of computing more accurate intermediate levels allowing the conditionally stable scheme in some instances to outperform the unconditionally stable schemes. The same array compaction technique was then applied to the 2-point or fast AGE scheme also saving half the hardware.

Next the simple form of the 1-point hopscotch scheme was exploited to derive proposals for a hard-systolic implementation of the parabolic solver by bit parallel and bit serial computation strategies using fixed point arithmetic. The former scheme required buffering of input and output. The latter scheme producing an area efficient unbuffered cell structure - suggesting a FAST chip based design was possible.

Finally we considered the extension of the method to a simple first order hyperbolic equation which exhibited attractive VLSI design features. The derived group explicit molecule produced a fully decoupled approach to computation where pairs of individual table columns could be evaluated independently. This resulted in a decoupled collection of GE cells requiring communication only for the loading of initial values. It followed that a hyperbolic equation could be solved by multipass on a fixed sized architecture (independent of both t_z and m), and that a collection of problems could be solved in parallel by interleaving group columns of different instances on the same array. These attributes together with a uni-directional loading strategy combined to indicate a fault tolerant design which would degrade gracefully as individual cells became faulty.

We conclude that the geometric approach to solving P.D.E.'s is not only suited to soft-systolic frames but produces genuine proposals for hard-systolic implementations.

CHAPTER 9

TOWARDS A GENERAL SYSTOLIC COMPUTER

"I could have done it in a much more complicated way", said the Red Queen, immensely proud.

LEWIS CARROLL.

The recent trends towards the development of more general systolic architectures such as the WARP system (H.T. Kung [84a]), wavefront array processor (S.Y. Kung [84]) and the unified arrays of the previous chapters emphasise the importance of developing soft-systolic algorithms in the form of micro-programs and generic arrays, for related problems.

The aim of this chapter is to investigate the compatibility of some well known computing structures, by use of simulation techniques and virtual machines, to a common architecture. To this end a soft-systolic program simulation system (SSPS) is introduced as a working model of a virtual machine (the Instruction Systolic Array (ISA)) with the power to simulate many hard-systolic, wavefront SIMD and some MIMD algorithms.

The emphasis is not on producing special purpose systolic algorithms which require restricted special purpose architectures, but executing parallel programs on a fixed underlying architecture systolically. Consequently, specially constructed algorithms are devised for the virtual machine which map easily onto the real machine environment.

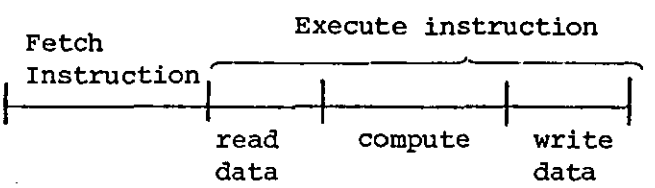
9.1 THE INSTRUCTION SYSTOLIC ARRAY

In Lang [85] the Instruction Systolic Array (ISA) was proposed as a new parallel architecture capable of exploiting VLSI technology. Contrasting with conventional systolic arrays, the ISA pumps instructions rather than data through a mesh connected array of processors. Each processor is capable of executing instructions from some instruction set. Consequently, while a systolic array realizes only one special algorithm (or a collection of related problems in a

generic array), an ISA can implement a wide range of parallel algorithms defined as ISA programs.

Now, in the MIMD concept of parallelism, all the processors of a given array (denoted PA) can execute different instructions. If the array consists of n^2 independent processors each containing a control store, a PA program can consist of up to n^2 different programs. These programs must be distributed over the array before the PA program can be executed, in contrast on an ISA the program is executed as it filters through the array. Consequently it is easier to execute a pipelined sequence of programs using an ISA than a PA.

More formally, define a basic model for a parallel computer as a mesh-connected array of n^2 identical processors, synchronised by a global clock. The processors execute instructions from the same instruction set, with the execution time of all instructions bounded by the most complex operation. In addition each processor contains some local memory and a communication register (CR). An instruction cycle has the form,



with communication occurring in mutually exclusive fashion to prevent overwriting of the communication register before a processors' nearest neighbours have had chance to read it. During the read phase, if a processor requires data from one of its nearest neighbours it simply takes the data from the relevant CR. Consequently only five processors (including the PE containing CR) can read from the same register simultaneously.

Three types of parallel machines which differ only in the way

control information reaches processors are now easily defined.

(i) The Processor Array (PA)

Where each processor has its own control store (as mentioned above).

(ii) The Instruction Broadcast Array (IBA)

Processors use a simple control unit but no control store. Instructions are broadcast to all the cells in the same column, and selector information (0,1) is broadcast to processors of a row. If I_j is the instruction of column j, and s_i is the selector of row i processor p_{ij} performs operations according to,

$$p_{ij} = \begin{cases} I_j \text{ iff } s_i=1 \\ \text{no-op iff } s_i=0 \end{cases}$$

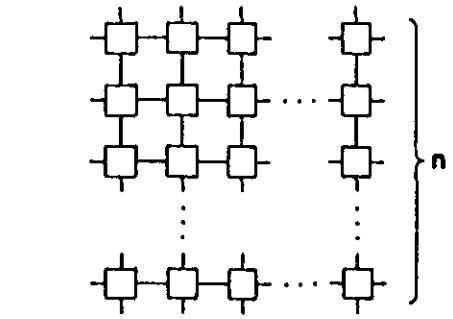
(iii) The Instruction Systolic Array (ISA)

Identical to the IBA except that instructions and selectors are retimed so that they are pumped systolically through the array. Instructions moving row-wise north-south, and selectors column-wise west-east, where no-op is an operation contained in the instruction set I which does not modify the processors memory contents.

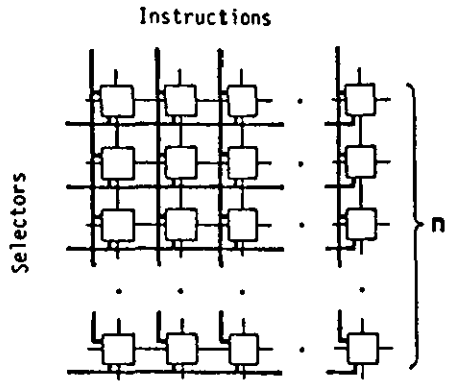
Next, let PA_n, IBA_n and ISA_n be arrays with side n (see Fig.(9.1.1)) and define the concept of a program on each machine as follows.

A program on a PA_n: is a sequence $p^{(1)},p^{(2)},\ldots,p^{(r)}$ of $n\times n$ matrices over I, such that for all $i,j\leqslant n$ and $t\leqslant r$ the instruction executed by processor (i,j) at time t is $p_{ij}^{(t)}$.

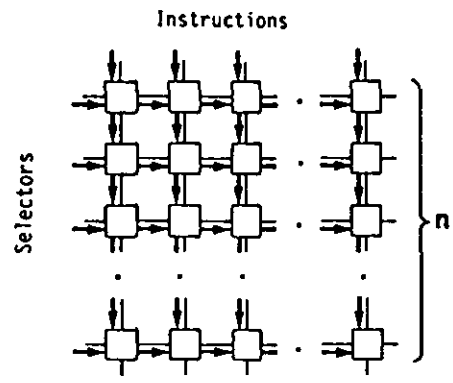
A program on an IBA_n: is a sequence $p^{(1)},p^{(2)},\ldots,p^{(r)}$ of n-tuples (vectors) over I and a sequence $s^{(1)},\ldots,s^{(r)}$ of n tuples (vectors) over {0,1} such that for all $i,j\leqslant n$ and $t\leqslant r$ $p_j^{(t)}$ is the instruction broadcast to column j and $s_i^{(t)}$ is the selector information broadcast to processors in row i at time t.



a) Processor array (PA_n)



b) Instruction broadcase array (IBA)



c) Instruction Systolic array

FIGURE 9.1.1: Three models of parallel architectures

Alternatively processor p(i,j) executes according to,

$$p(i,j) = \begin{cases} p_j^{(t)} \text{ iff } s_i^{(t)}=1 \\ \text{NO-OP otherwise} \end{cases}$$

A program on an ISA_n: is again a sequence $p^{(1)},\ldots,p^{(r)}$ of n-tuples over I and sequences $s^{(1)},\ldots,s^{(r)}$ of n-tuples over {0,1}. But for $i,j\leq n$ and $t\leq r$, $p^{(t)}$ is the row of instructions entering the ith row at t+i-1, and $s^{(t)}$ is the column of selector information entering the jth column of the ISA_n at time t+j-1. That is, processor p(i,j) performs as,

$$p(i,j) = \begin{cases} p_j^{(t+i-1)} \text{ iff } s_i^{(t-j+1)} = 1 \\ \text{NO-OP otherwise} \end{cases}$$

at time t. Finally program execution terminates after the last row of instructions $p^{(r)}$ has entered the first row of ISA processors. Thus if $p^{(r)}$ must filter through to the last row, n-1 rows of no-ops must be appended to the program.

REMARK: The definitions are easily extended to rectangular grids denoted $PA_{m,n}$, $IBA_{m,n}$ and $ISA_{m,n}$ with simple modifications to i,j indices, where $m\neq n$.

Now denote p as a program on a PA_n, IBA_n or ISA_n then the execution time of p is equal to the length of the program and denoted by T(p). Furthermore if $C^{(k)}=(c_{ij})$ is the n×n matrix where $c_{ij}^{(k)}$=CR contents of processor p(i,j), program p is simulated by a sequence of snapshots $c_{ij}^{(k)}$, k=0(1)r where $C^{(0)}$ is the starting state of the grid and $C^{(r)}=C^{(T(p))}$ is the final result image.

Input and output to the grid occurs whenever processors read or write to non-existant processors around the mesh boundary. The input of a program is then defined as a sequence of 4n-tuples (an n-tuple for each boundary) which are read during the execution of p. Likewise the

output of p is a sequence of 4n-1 tuples output from the boundaries during execution of p. Observe that there can be at most T(p) input, and T(p) output (4n-1) tuples.

To conclude the descriptions of the machines we define the idea of program equivalence. For two programs p and q on PA_n, IBA_n, or ISA_n we can call programs equivalent if they contain one of the following attributes.

(i) INTERNAL EQUIVALENCE: If for the same initial conditions $C^{(0)}$, $C^{T(p)} \equiv C^{T(q)}$ for identical input sequences.

(ii) EXTERNAL EQUIVALENCE: If for the same initial conditions $C^{(0)}$, the boundary output sequences are equivalent.

(iii) STRUCTURAL EQUIVALENCE: The intermediate operations of p and q can be mapped onto each other.

Using these definitions Kunde, Lang, Schimmler, Schmeck & Schroder [85] have derived bounds on simulating a program on one architecture by an equivalent program on another. For completeness we state these results as properties characterising program equivalences, in the following table.

Property	Program Transformation			Comment
	Length	p on → q on	Time relation	
1	-	ISA_n or IBA_n → PA_n	T(p)=T(q)	Generally no speed-up
2	-	PA_n → IBA_n	$T(q) \leq (n+1)T(p)$	Worst case IBA simulation
3	r>0	PA_n → IBA_n	$T(q) \geq (n+1)T(P)$	
4	-	IBA_n → ISA_n	$T(q) \geq 3T(p) + 2n-2$	Asymptotic time complexity the same
5	r>0	IBA_n → ISA_n	$T(q)$ is in $\Omega(T(p))$	
6	-	PA_n → ISA_n	$T(q) \leq (n+2)T(p) + 2n-2$	Bound on ISA simulation of PA
7	r>0	PA_n → ISA_n	$T(q) \geq (n+2)T(p)$	
8	-	ISA_n → IBA_n	$T(q) \leq (n+1)T(p) - n^2+n$	Reverse of 4,5 does not hold
9	r≥n	ISA_n → IBA_n	$T(q)=\Omega(n)T(p)$	

TABLE 9.1.1: Summary of program transformations on PA_n, IBA_n and ISA_n

The main result is that an arbitrary program that runs on an n×n mesh connected parallel computer in k steps can be transformed into an ISA program with O(nk) steps. The basic technique to the proofs is to simulate the PA program by the IBA and then retime the equivalent IBA program to produce the ISA version. An intuitive method for simulating a program p on a PA by a program on the IBA is to simulate every $p^{(t)}$ step of p by n steps on the IBA. Generally for the ith step only the ith row of the array is selected and the ith row of $p^{(t)}$ broadcast to the array. However, if row i reads data from row i-1 of the array it

is possible that the updated contents of the communication registers instead of the old values will be used. Consequently, to preserve computation it is necessary to save the contents of the old communication register until all a processors neighbours have had a chance to read it. The solution is to augment the IBA processors with a register R and flag F, such that results of calculations are placed in R and the Flag F set at the end of an instruction cycle. A special copy (C) command is introduced which copies R to CR and resets F. Thus $p^{(t)}$ is simulated by n+1 steps including a copy command at the end of the step to overwrite all the communication registers.

REMARK: As the ISA is simply a retimed IBA, the copy command must also be incorporated into ISA programs.

Fig.(9.1.2) illustrates the control flow for an ISA program.

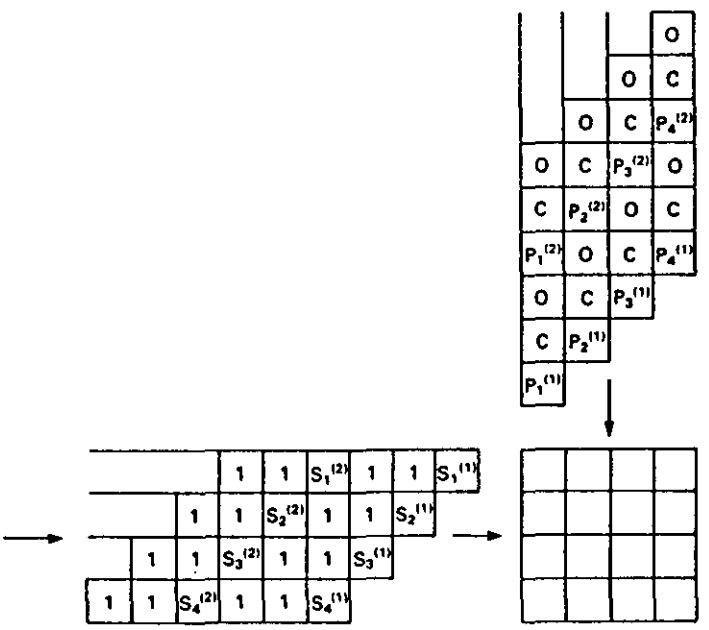


FIGURE 9.1.2: Control flow in an ISA program

It should be clear programs transformed from PA_n or IBA_n are essentially intuitive mappings and that faster more efficient algorithms may be derived by dealing with the ISA_n from the outset. Where special ISA

programs prove difficult to design or unwieldly it is comforting to know that a straightforward program transformation from an existing architecture is available.

Finally, to conclude this section we consider a simple example of transposing an n×n matrix where $n=2^k$ on the ISA_n (Lang[85]). The algorithm proceeds iteratively transposing $2^j \times 2^j$-subarrays j=1(1)k, and is defined as follows, for j=1, a 2×2 subarray is transposed by exchanging elements in the upper right and lower left corners. This requires three steps:

(i) swap 1st row elements

(ii) swap 1st col. elements

(iii) swap 1st row elements (again)

For j>1, a $2^j \times 2^j$ block array is transposed in 4 steps:

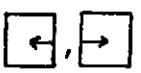
(i) transpose the $2^{j-1} \times 2^{j-1}$ sub-arrays

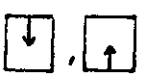
(ii) exchange the two upper sub-arrays

(iii) exchange the two left sub-arrays

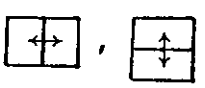
(iv) exchange the two upper sub-arrays.

Fig.(9.1.3) illustrates the development of an ISA program for transposing an 8×8 matrix, where,


: read right (left) processors CR value and place it in own CR.


: as above except read upper (lower) processors


: no-op

Thus  are equivalent to swapping processor elements.

The program is easily generalised to the n×n case.

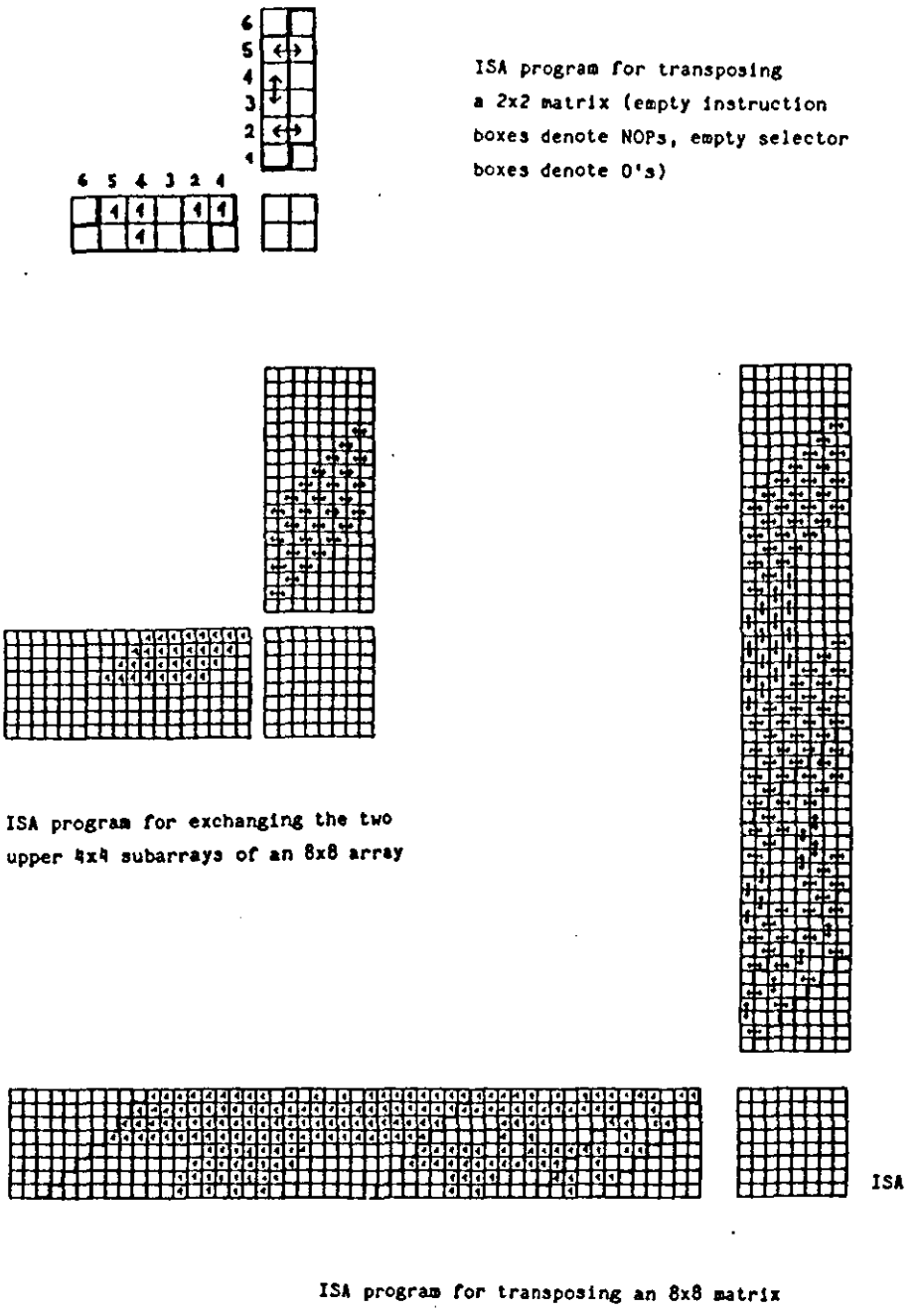


FIGURE 9.1.3: ISA transpose program

9.2 THE n-SPACE ISA AND MULTI-TASKING OF SOFT-SYSTOLIC PROGRAMS

If we consider the classification of parallel computers by Flynn [72] the PA_n, IBA_n and ISA_n have to be classed as MIMD machines. This follows because a number of different instructions can be executed simultaneously on different rows and columns hence data streams of the

mesh. Furthermore, since the processors in an IBA_n or ISA_n do not require central stores they lie closer to SIMD-type architectures than MIMD machines. Thus, the ISA represents a type of hybrid machine, somewhere between SIMD and full-MIMD. It follows that soft-systolic programs and frames can be easily extended via the ISA to link with these wide classes of problems, and from this viewpoint it is essential to characterise the power of the ISA machine.

We can define a full SIMD-program on a PA as a sequence of instruction matrices which consist only of identical instructions, implying that these problems are easier to simulate on the IBA and ISA. Hence,

Theorem 9.2.1a: (Kunde, Lang, Schimmler, Schmeck, Schroder [85])

For every full SIMD-program on a PA there is an equivalent program on an IBA having the same time complexity.

Proof:

Each program vector $p^{(t)}$ in the IBA program is a simple repetition of the instruction in step t, with all selectors 1.

Theorem 9.2.1b: (Kunde, Lang, Schimmler, Schmeck [85])

For each full SIMD-program p on a PA_n or IBA_n there is an equivalent program q on the ISA_n with $T(q) \leq 3T(p)+2n-2$.

Proof:

The simulation of SIMD-programs on an ISA introduces the same problem for arbitrary program simulation, as instructions executed simultaneously by neighbouring processors of the PA or IBA will be executed consecutively on the ISA. Thus, the IBA_n ISA_n transformation in Table (9.1.1) suffices.

Next we can define a PARTIAL SIMD-program on a PA or IBA, where

the instruction matrices consist of just two types of instructions, a no-op and one from the processor instruction set I.

Theorem 9.2.2: For every r there is a partial SIMD-program p on a PA_n with $T(p)=r$ such that for any equivalent program q on an IBA_n, $T(q) \geq (n+1)T(p)$.

Proof:

A partial-SIMD program is a simplified MIMD-program and as such requires a full transformation like property 3 of Table (9.1.1), yielding the time immediately.

This implies that partial-SIMD programs cannot be simulated faster on an IBA_n than arbitrary programs. However a subset of partial SIMD-programs can be simulated with the same speed as full SIMD-programs. These problems are termed vector-orientated SIMD programs and like the partial SIMD programs consist of just two instructions (including the no-op). But in addition the no-op occurs only in complete rows or columns of the array.

Theorem 9.2.3: (Kunde, Lang, Schimmler, Schmeck [85])

For every vector-oriented SIMD-program on a PA there is an equivalent partial SIMD-program on an IBA having the same time complexity.

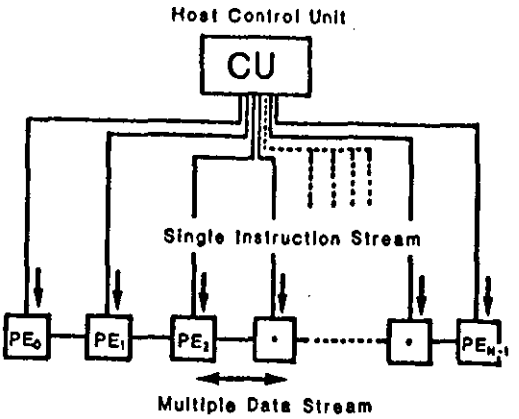
Proof:

To transform a PA program step $p^{(t)}$ with an instruction b≠no-op in it to an equivalent IBA step we set,

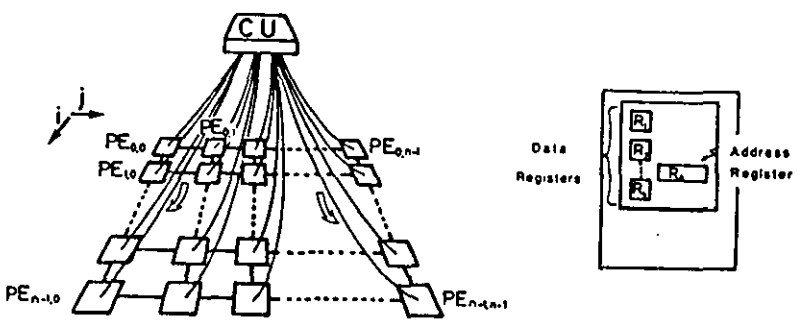
$$p_j^{(t)} = \begin{cases} b \text{ if column } j \text{ in } p^{(t)} \text{ is not a complete no-op column} \\ \text{no-op otherwise} \end{cases}$$

and put,

$$s_i^{(t)} = \begin{cases} 1 \text{ if row } i \text{ is not a complete no-op row} \\ 0 \text{ otherwise} \end{cases}$$



a) 1-D organization of SIMD computer



b) 2-D simple SIMD parallel computer c) Processing element

FIGURE 9.2.1: Organization of SIMD machines (from Umeo[82])

Next consider the class of simple-SIMD algorithms, defined in Umeo [85a] as ones in which the SIMD processing surface is limited to a linear array of processors. The class of simple-SIMD algorithms consists of many interesting problems including sorting, image-processing, and graph algorithms as well as other conventional SIMD algorithms. In Umeo [82], Umeo & Sugata [82], Umeo, Morita, Sugata [82] and Umeo [85a,b] the mapping of simple SIMD and 2-D SIMD algorithms onto

systolic arrays has been investigated and characterized, the conclusion was that SIMD algorithms can be simulated on systolic arrays without much loss of efficiency. The complexity of the systolic array simulation is measured by summing the systolic cycles required to load data, execute the programs and output the results, and is achieved as follows (for a simple SIMD algorithm).

Theorem 9.2.4: (Umeo [85])

For any simple SIMD machine M with time complexity T(n) there exists a systolic array A which simulates M in $2T(n)+3n+O(1)$ steps.

Proof:

Without loss of generality we assume that M has n processing elements (PE's) each with a single data register (the method is easily extended to more data registers). Let a_i be the data preloaded into PE_i and I_t the instruction broadcast to each PE by the SIMD machines control unit. Then $1 \leq t \leq T(n)$ and the array is organised as follows:

(i) There is a buffer $B \equiv 1$ input B_{in} and 1 output register B_{out}.

(ii) A total of n+1 systolic cells c_i, $i=O(1)n$ containing:

a) an address register R_a

b) working registers R_i, $i=1(1)4$

c) a processor to decode and execute I_t

d) auxiliary registers:

I_1 to pipeline the data instruction right

I_2, I_3 book-keeping registers, with I_3 doubling up as the data output register shifting results left.

(iii) c_n acts as a boundary cell

(iv) Data to A is supplied as a joint instruction, data format,

$$a_0 a_1 \ldots a_{n-1} I_1 \delta\ I_2\ \delta \ldots \delta\ I_{T(n)-1} \delta\ T_{T(n)} \gamma$$

γ=end of input

δ=spacing dummy input

Data is input at the rate of 1 symbol per step through B_{in}. Initially $B_{in}=a_0$, and each instruction is input at the rate of one every two steps. The cells can be in one of three states loading, computing, or output, and the current state is stored in R_4. The state is controlled by signals tagged to symbols moving into the cells. When the terminator reaches cell c_n a reset symbol is propagated left to clear the array for a new problem.

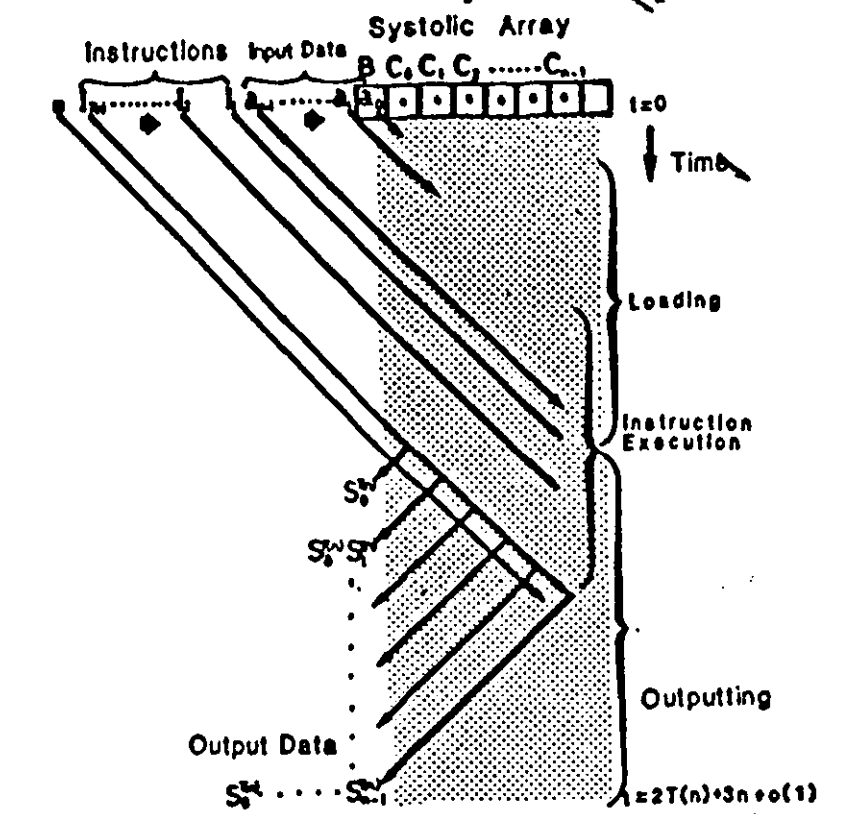


FIGURE 9.2.2: Time-space diagram for the systolic simulation of simple SIMD machine (Umeo [82]).

A primitive mask facility is provided by adding an address field to the data. This address is compared with the cell addresses and if equal masks out the cell for that cycle.

We can easily fit the simulation of the systolic array on a single column of ISA cells as follows.

Theorem (9.2.5): Any simple SIMD machine M with time complexity T(n) can be simulated by a virtual systolic array A on the ISA in $2T(n)+3n+O(1)$.

Proof:

Map the systolic array A in Theorem (9.2.4) onto a column of ISA cells assuming the ISA has an n×n grid. Next set the selectors permanently true for each row i=1(1)n and define the systolic array cell as a virtual processor. For this we just implement the interpretations of I_t and perform the address comparison, to implement the mask. The column structure and data/instruction format is as shown below, the instruction P_n sets the address register of the cell it enters to the data input with P_n (i.e. 1 in cell 1) and the value incremented before being passed on.

Thus each cell has its address register set before any of the real instructions reach them.

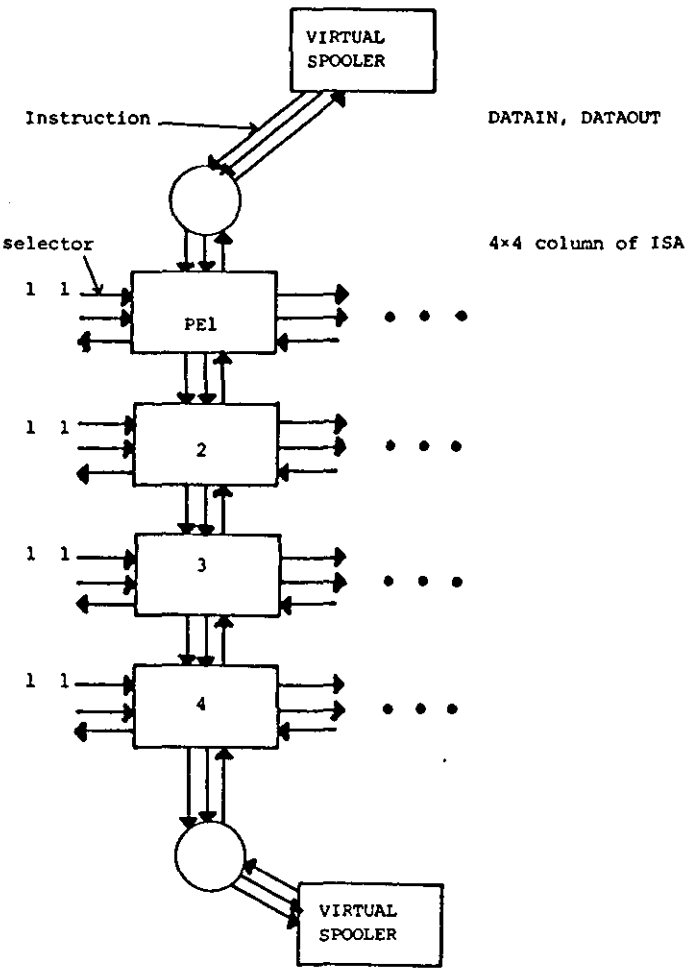
From this theorem it is clear that no horizontal data movement can occur, and the above result can be extended as follows.

Theorem (9.2.6): An n×n ISA grid can simulate n simple SIMD machines M_i with time complexities $T_i(n)$ for i=1(1)n in a total time

$$T=2 \max_{1\leq i\leq n}(T_i(n))+4n+O(1).$$

Proof:

Make an ISA_n program where each instruction column (including data loading) represent a separate simple SIMD program. For an ISA_n grid it



DATA FORMAT

Instructions $P_n\ P_{n-1} \cdots P_0\ \delta I_1\ \delta I_2\ \delta \cdots \delta I_{T(n)-1}\ \delta I_{T(n)}\ \gamma$

Data $1\ \ a_n a_{n-1} \cdots a_0\ \delta M_1\ \delta M_2\ \delta \cdots \delta M_{T(n)-1}\ \delta M_{T(n)}\ \delta$

Here P_n = set address to datain
increment datain
set state data

P_{n-1}-P_1 = read data, P_0 = set state instructions

I_t t=1(1)T(n) Instruction from processor instruction set

δ = padding element

M_i = i=1(1)n address of cell to mask out (address zero disables mask)

γ = terminator, set state output

FIGURE 9.2.3: Mapping of systolic array to ISA column

follows that the time must be bounded by the cost of the longest running program plus an extra n cycles to push selector vectors through all the columns of the grid before starting each simulated machine.

It also follows that Umeo's simulation of simple SIMD machines is a special case of vector-orientated SIMD programs representing a single task (program) systolic simulation theorem. Alternatively the columns of no-ops in vector-orientated SIMD programs can be interpreted as vacant columns not running a simulation. Thus Theorem (9.2.6) is a multiple task (soft) systolic simulation theorem. Consequently the results of Umeo [85] can be extended to the multi-tasking of simple SIMD soft-systolic simulations to provide a multi-programmed environment using the ISA.

Theorem (9.2.7): (The multi-sequential task systolic simulation theorem).

Let M_i, i=1(1)k be any simple SIMD machines, each with time complexity $T_i(n)$ with the same instruction set (or a subset) then there exists a systolic array A which simulates the M_i's, i=1(1)k in,

$$T = 2\sum_{i=1}^{k} T_i(n) + 3kn + O(1) \text{ steps.}$$

Proof:

Simply produce large streams of column data and instructions with the form,

$$\underbrace{a_0^1 a_1^1 \ldots a_{n-1}^1 I_1^1 \delta\, I_2^1 \ldots \delta I_{T_1(n)}^1}_{M1}\ \underbrace{\delta \ldots \delta}_{2n}\ \underbrace{a_0^2 a_1^2 \ldots a_{n-1}^2 I_1^2 \delta I_2^2\ \delta \ldots}_{M2}$$

$$\vdots$$

$$\underbrace{\delta\delta\delta \ldots \delta}_{2n}\ \underbrace{a_0^k a_1^k \ldots a_{n-1}^k I_1^k \delta I_2^k\ \delta \ldots\ I_{T_k(n)}^k}_{M_k}$$

Now each ISA column simulates a sequence of pipelined machines.

Theorem (9.2.8): (The multi-parallel task systolic simulation theorem)

Let M_{ij}, i=1(1)k, j=1(1)n be any simple SIMD machine, each with a time complexity $T_{ij}(n)$ with the same instruction set then the ISA can simulate the M_{ij} in a time,

$$T = \max_{1\leq j\leq n} \left(2 \sum_{i=1}^{k} T_{ij}(n)\right) + n(3k+1) + O(1)$$

Proof:

By extension of Theorem (9.2.6).

Clearly from Theorem (9.2.5) any machine is simulated in $2T(n)+3n+O(1)$, thus a whole column of k machines requires,

$$\sum_{i=1}^{k} [2\ T_{ij}(n)]+3nk+O(1) = 2 \sum_{i=1}^{k} T_{ij}(n) + 3nk+O(1)$$

for some $1\leq j\leq n$. Allowing n steps to filter selectors through the array produces the timing,

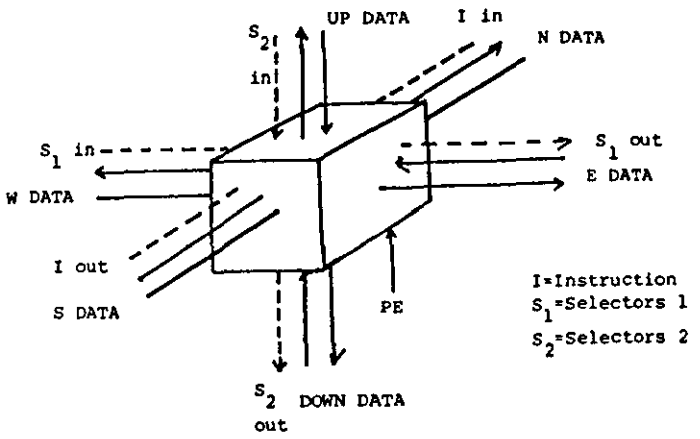
$$T = \max_{1\leq j\leq n} \left(2 \sum_{i=1}^{k} T_{ij}(n)\right) + n(3k+1) + O(1).$$

REMARK: Theorems (9.2.7) and (9.2.8) can be speeded-up by overlapping input and output of machines and interleaving two machine sequences so as to fill the neutral instruction elements.

We conclude that the instruction systolic array is at least as powerful as SIMD-machines. Infact for many cases where the original array machine simulated is of SIMD-type the ISA can simulate it with O(1) delays. Clearly if the MIMD program can be re-written as a collection of simple SIMD programs we can simulate the MIMD problem using multi-tasking. This has repercussions for traditional systolic arrays with a collection of cell types which can be partitioned to yield SIMD-type procedures even though the array should be classed as MIMD (e.g. back-substitution arrays with two cell types).

The ISA as described is capable of simulating only restricted forms of the MIMD model which have simple control flow. For more complex program traces allowing conditionals and loops whose terminator conditions are set by internal loop calculations the ISA encounters difficulties. This implies that the SIMD structure of the ISA is too restrictive. However, the control information delivered to each processor consists of two parts, the instructions propagated down columns and selectors propagated along rows. A simple generalisation of the control scheme is observed directly. Processor (i,j) can be re-defined to execute composite instructions of the form $a_i b_j$ where a_i is a prefix arriving along the row i and b_j along column j.

If A and B are two sets of instructions such that $|A|=k_1$ and $|B|=k_2$ and $o \in A$, $o \in B$ denotes AB=no-op then at most $(k_1-1)(k_2-1)+1$ instructions can be encoded. An $ISA_n(k_1,k_2)$ represents the modified mesh array and $ISA_n(2,k_2)$ is clearly the original ISA. The generalised scheme gives a better opportunity for implementing MIMD algorithms, because we can now implement simple conditionals by using vertical instructions as true program branches and horizontal instructions as false (else) branches. The increased number of no-op combinations providing more flexible masking facilities. By logical extension we can create more complex control arrangements, increasing the dimensionality of the array, providing further directions in which to pump instructions. For example, the 3-space ISA follows naturally by extending the 2-D mesh to an orthogonally connected cube. A processor now contains 6 data inputs, 6 data outputs, and three instruction input/output connections as illustrated overleaf.



Control flow occurs in three orthogonal directions, and instructions are constructed from three sets A_1, A_2, A_3 with $|A_1|=k_1$ $|A_2|=k_2$ and $|A_3|=k_3$, allowing at most $k_1.k_2.k_3$ different instructions (including no-ops). These three sets could then be used to form composite instructions or evaluate conditionals of the form,

```
IF COND1 THEN
    {INSTRUCTION SEQUENCE}          - Encoded by A1 streams
ELSE
 {IF COND2 THEN
    {INSTRUCTION SEQUENCE}          - Encoded by A2 streams
  ELSE
    {INSTRUCTION SEQUENCE}          - Encoded by A3 streams
 }
```

Allowing more complex MIMD programs to be implemented.

Generalising the concept produces an n-space ISA with no geometrical interpretation, but which produces control flow in n mutually orthogonal directions defining instructions sets A_i, i=1(1)n of size $|A_i|=k_i$ and $\prod_{i=1}^{n} k_i$ instructions. Clearly the analysis of control flow for programs becomes increasingly complex, as does the connection network of processors

- limiting the technique severely. The interesting thing about the 3-space ISA is that it easily reflects the SIMD and MIMD type program mappings. For instance, consider any plane of processors of the form (i,c,k) where c is a constant, and i,k=1(1)n, choose $|A_1|=|A_2|=2$ to produce selectors setting s_2 to be a sequence of matrices containing only 1's - the 2-D $ISA_n(2,k_3)$ is produced. Likewise restricting s_1 to a matrix also full of ones and restricting the processors to (c_1,c_2,k) where $c_1,c_2>0$ are constants produces a simple SIMD simulator (or 1-D array).

A plane in the directions (i,j,c) for c>0 and i,j=1(1)n and s_1 and s_2 full of 1's produces the popular PA_n model of an MIMD machine. The PA_n program p then consists of a sequence of T(p) matrices extending in the k direction and coincides with the program store of normal MIMD machines.

Any other plane (or planes) will simulate an MIMD machine allowing the grids to become triangular and rectangular. Notice however that the orthogonal nature of the grid requires diagonal processor connections to be simulated by passing values through adjacent processors.

By extending the theorems on multitasking soft-systolic simulation it follows trivially that the 3-space ISA can simultaneously:

(i) execute n^2 - simple SIMD programs

(ii) execute n 2-D or full SIMD programs

(simply put (i,c,k) c>0 in Fig.(9.2.4) to produce n planes of 2-D ISA_n machines).

Thus from the relationships in Table (9.1.1) n PA programs can also be executed in parallel.

Next consider the case when we have a 3-space PA (a cube of n^3

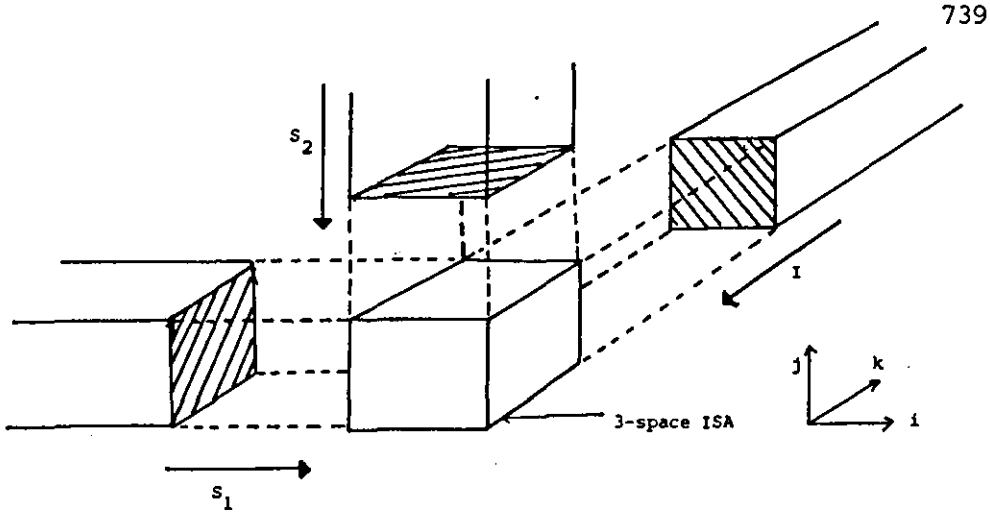


FIGURE 9.2.4: 3-space ISA_n

processors each with a control store). There can be at most n^3 programs making up the PA program residing in the independent processors and which must be distributed through the array before operation begins. On execution the array program is a sequence $p^{(1)},p^{(2)},\ldots,p^{(r)}$ of 3-D arrays over I with processor (i,j,k) performing $p_{ijk}^{(t)}$ at time t.

Theorem (9.2.9): For every program p on the 3-space PA_n there is an equivalent program q on the 3-space ISA_n requiring,

$$T(q) \leq (n+2)T(p)+3n-2 \text{ steps.}$$

Proof:

First consider a step $p^{(t)}$ of the 3-space PA_n program p. This can be considered as a collection of 2-space PA_n programs $\hat{p}_i$, i=1(1)n, with a general program corresponding to the sequence of n×n matrices $\hat{p}_i^{(1)},\hat{p}_i^{(2)},\ldots,\hat{p}_i^{(r)}$ over I corresponding to the mesh planes (i,c,k), c=1(1)n. Using Table (9.1.1) each of these programs is simulated by the 2-space ISA_n representing the (i,c,k) plane of the 3-space ISA_n in $(n+2)T(\hat{p}_i) \leq T(\hat{q}_i) \leq (n+2)T(\hat{p}_i)+2n-2$ steps, where $\hat{q}_i$, i=1(1)n is the equivalent 2-space ISA_n program. Now to allow for the skew of the second selector inputs S2 successive planes from top to bottom of the cube in Fig.(9.2.4) are delayed to retain synchronisation producing

a time bound,

$$(n+2)T(p)+n \lesssim T(q) \lesssim (n+2)T(p)+3n-2.$$

Theorem (9.2.10): For every program p on the 3-space ISA_n there exists an equivalent program q on the 2-space ISA_n requiring

$$T(q) \lesssim n(n+2)T(p)+3n-2 \ .$$

Proof:

The essential idea behind the proof is to interleave the computation of the n 2-space ISA_n comprising the 3-space ISA_n on a single 2-space ISA_n. This is achieved by unifying the cuboid planes as follows. First assume each 2-space ISA_n processor contains a vector CR(j), j=1(1)n of registers for book-keeping. A single plane can simulate all the others by defining CR(j) in processor (i,k), i,k=1(1)n as the communication register of processor (i,j,k) of the 3-space cube. Notice that this guarantees that a 2-D ISA can reproduce the communication with a processors six nearest neighbours in a 3-D space design. We then simply compute the 3-space ISA_n programs on Theorem (9.2.9) in the order,

$$\hat{p}_1^{(j)}, \hat{p}_2^{(j)}, \hat{p}_3^{(j)}, \ldots, \hat{p}_n^{(j)} \ , \ j=1(1)r \ ,$$

using the copy command to overwrite the correct CR(j) registers. As successive planes no-longer compute in parallel the 3-space ISA_n increases to nT(p) in the 2-space ISA_n and direct substitution into Theorem (9.2.9) yields,

$$T(q) \lesssim n(n+2)T(p)+3n-2 \ .$$

Finally, to complete the characterisation of the ISA_n and its power to simulate various programs soft-systolically we consider the wavefront array processor (S.Y. Kung [84], see Fig.(3.5.2.1)). The wavefront machine is again a 2-D mesh of n^2 processors with nearest neighbour communication, and the order of activation and subsequent

computation of processors act like waves propagating across the array. For simplicity waves act on the Huygen's principle such that no two waves can pass or interfere with each other from the same source and prohibits backtracking of waves. S.Y. Kung has noted that the wavefront processor is a trade-off between the general purpose dataflow multi-processors and the dedicated systolic array. This implies that the wavefront array processor is related strongly to the ISA. Indeed, the input of selectors and instructions forms successive wavefronts across the mesh (see Fig.(9.1.2)). Furthermore if we suppose composite instruction sets A_1 and B_1, wavefront processor programs can be interpreted as interference of horizontal and vertical component values such that a diagonal wave is given by instructions,

$$w_i = \begin{cases} a_i b_j & \text{constructive} \\ \text{no-op} & \text{destructive} \end{cases}$$

where a destructive wave is any composite instruction involving a no-op instruction. Hence,

Theorem (9.2.11): For any wavefront array program p with time T(p) and r wavefronts there is an equivalent program q on the 2-space ISA_n which requires $T(q) \leq 3rT(p)+2n-1$.

Proof:

Each wavefront processor wave requires 2n-1 cell cycles to propagate across the mesh, and is mapped directly into n-tuples of instructions and selectors such that each wavefront is replaced by three wavefronts of the form,

$$\left.\begin{array}{l} p_j^{(t)} = I, \quad p_j^{(t+1)} = c, \quad p_j^{(t+2)} = O \\ s_j^{(t)} = s_j^{(t+1)} = s_j^{(t+2)} = 1 \end{array}\right\} \quad j=1(1)n$$

on the ISA with I the instruction executed by wavefront processor cells.

Two extra wavefronts are added to overwrite the communication registers after each instruction wave producing 3rT(p) wavefronts to pass over the ISA mesh. The timing follows immediately, with 2n-1 additional cycles to push the last wave off the mesh. The less than condition results from the fact that for some wavefront algorithms the communication registers can be overwritten directly, dispensing with $p^{(t+1)}$ and $p^{(t+2)}$ instruction n-tuples.

Notice that the converse argument that all ISA_n programs can be simulated by a wavefront array processor does not apply. This follows because in general the wavefront array processor executes only a single instruction I, whereas the ISA in general must contain at least three types of instruction (I,c,no-op) for partial-SIMD simulations.

Next consider the ISA_n simulation of multiple wavefronts in different orientations. Intuitively, this corresponds to the embedded wavefront mesh and systolic control ring arrangement used for rank annihilation, assignment, and simplex algorithms in previous chapters. The multiple wavefront computation can be envisaged as a series of n×n instruction matrices or snapshots as for the general PA_n program. The simulation program for the ISA is then constructed in the same manner as for arbitrary programs. It follows that multiple wavefront programs are simply special cases of partial SIMD algorithms and Theorem (9.2.2) can be applied directly. Fig.(9.2.5) illustrates a backtracking technique for deriving the ISA_n program form.

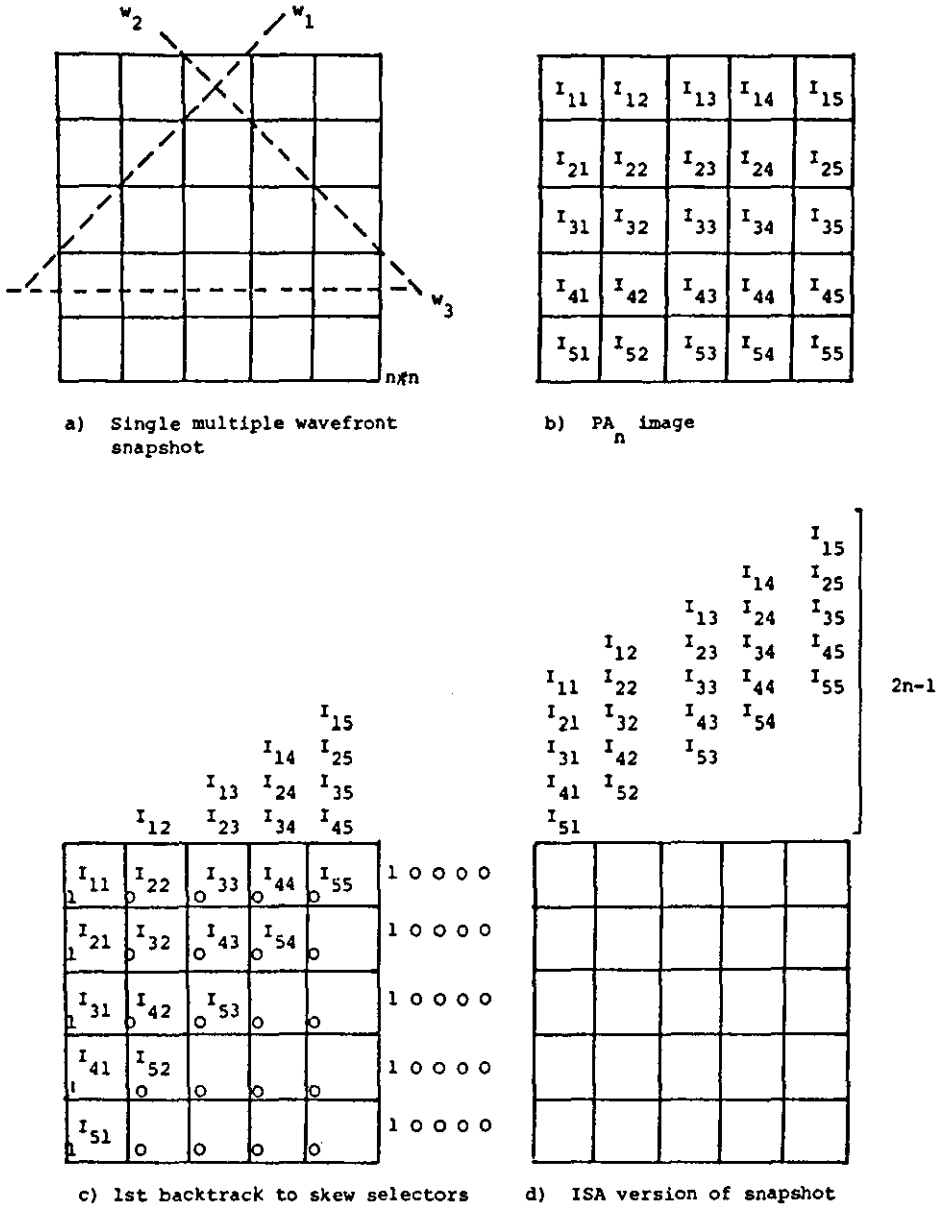
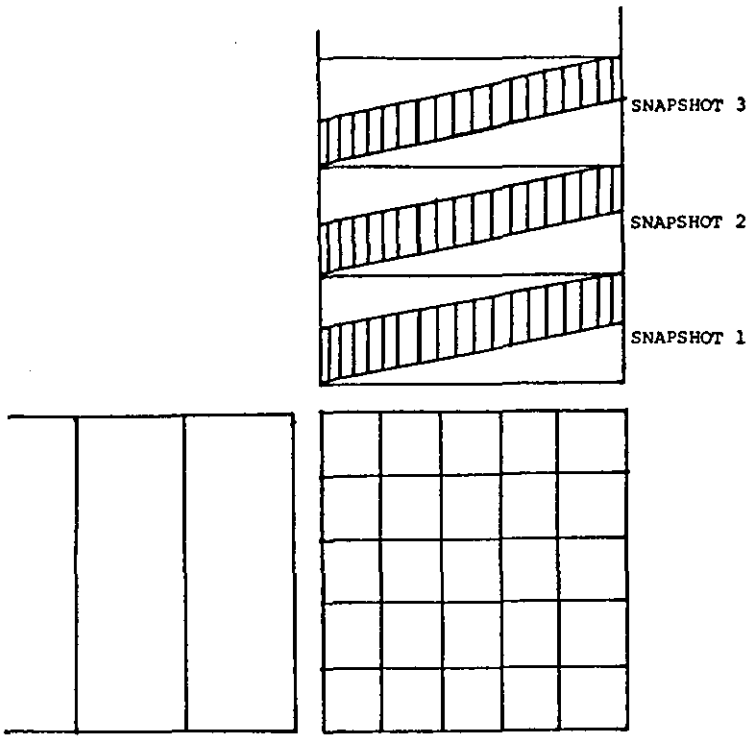
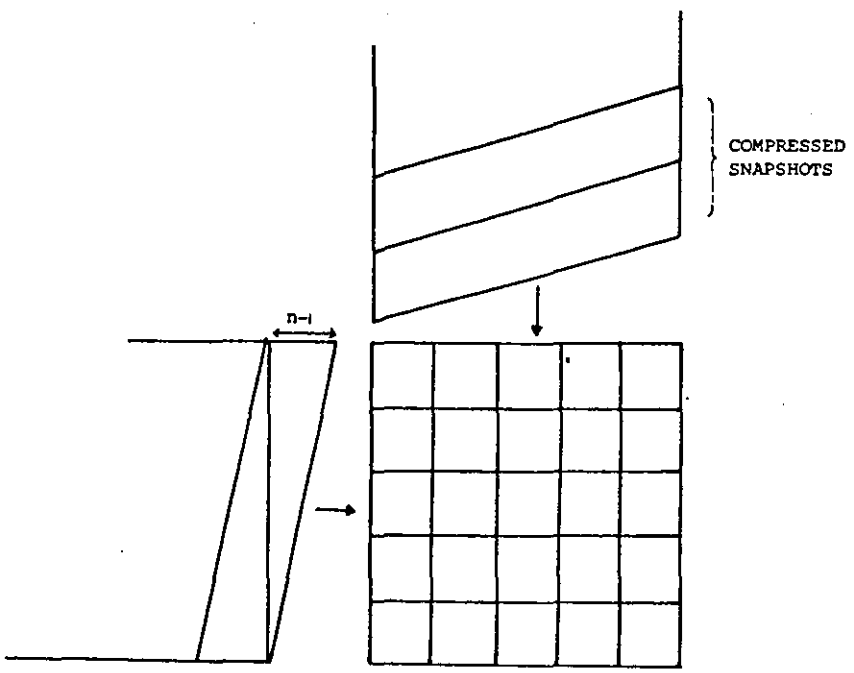


FIGURE 9.2.5: Simulation of multiple wavefront algorithms



e) Multiple sequence of snapshots (i.e. simulate wavefronts)



f) Compacted form of ISA wavefront program

FIGURE 9.2.5: Simulation of multiple wavefront algorithms (cont.)

9.3 THE SOFT-SYSTOLIC PROGRAM SIMULATION SYSTEM (SSPS)

The basic design problem for a general systolic array simulator is to provide a fixed architecture which is capable of simulating the arbitrary graph structure of an array, while also mapping parallel processors to achieve parallelism. Throughout this thesis we have envisaged systolic arrays as soft-systolic programs written in OCCAM with the implicit understanding that OCCAM can be executed effectively on transputer networks to provide parallelism. The problem with this scheme is that it may be better to write a dedicated transputer based version of a method rather than simulate a systolic array version of the algorithm. Thus as we accept the idea of programmable arrays the effectiveness of the special purpose systolic approach to specific algorithms falls off. The essential problem is the emphasis placed on dataflow which demands a different OCCAM program structure for each design. The ISA on the other hand places emphasis on the systolic movement of instructions fixing the data communication and processor structure, and the chances of producing a fast and economic systolic simulator, with an alternative perspective on the meaning of a 'systolic computer'. In this section we consider a soft-systolic program simulator implemented on the VAX machine running under UNIX, at Loughborough University and solve a number of common problems to demonstrate its flexibility. The system can be used to develop special purpose algorithms with a regular form and opens up the possibility of a soft-systolic design workstation for development of simple systolic processing systems.

An overview of the system is shown in Fig.(9.3.1), and the main sections are briefly reviewed below.

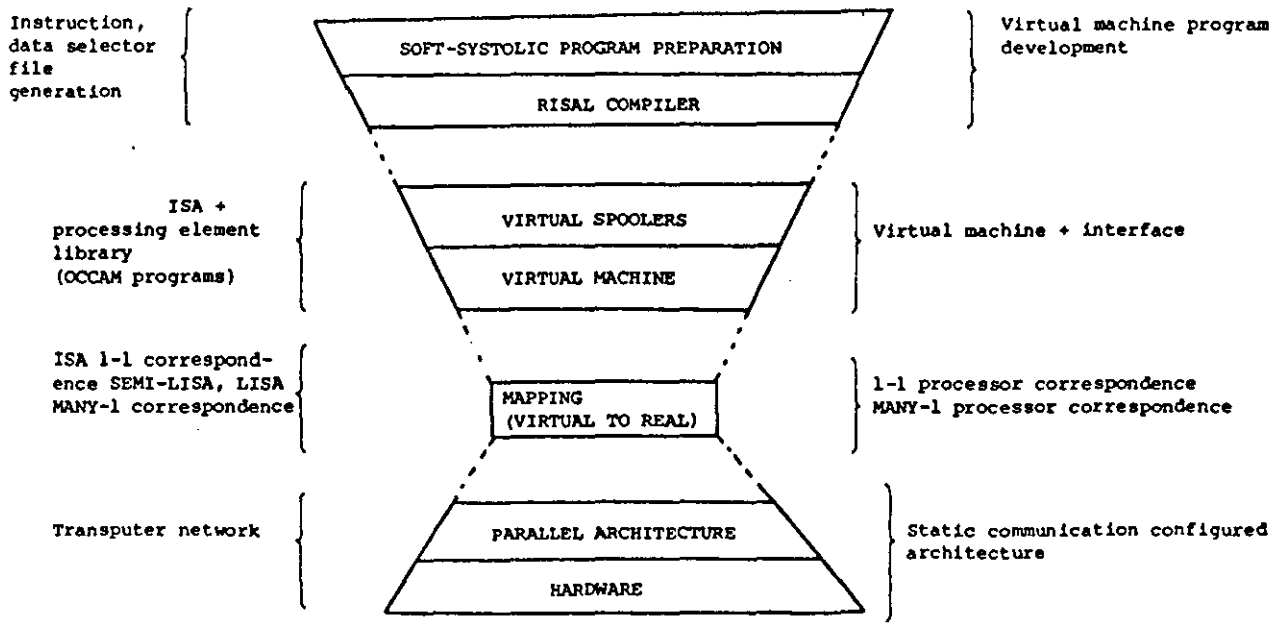


FIGURE 9.3.1: Organisation of soft-systolic program simulator

Program and machine preparation:

The soft-systolic preparation section comprises of the usual operating system facilities for the creation and modification of files during the development of new programs and ISA processor elements. We allow any concurrent high level language to be used to model the soft-systolic program.

RISAL compiler:

The RISAL compiler is adopted to transform the soft-systolic program description into a form suitable for the virtual machine (simulating the algorithm) to run.

Virtual machine:

The virtual machine consists of three basic sections:

a) An ISA network of data and control paths

b) A set of virtual spoolers for driving the ISA computation and opening up the communication bandwidth of the array.

c) A collection of processing element (PE) descriptions for creating specific ISA grids.

Virtual to real mapping:

Here we define a library of processor plugs which allow a number of virtual processors to be essentially plugged into a single real processing element of the underlying architecture. Thus, allowing a large virtual grid to be mapped onto a smaller real grid.

The real architecture:

For simplicity we assume that this is a square orthogonally connected grid of processors such as a transputer network, capable of executing any of the virtual PE's and mapping plugs.

Now clearly the complete design and implementation of the proposed system above would occupy a thesis by itself. Consequently, to demonstrate the feasibility of the system we concentrate on the virtual machine and the RISAL compiler, which forms the core of the design.

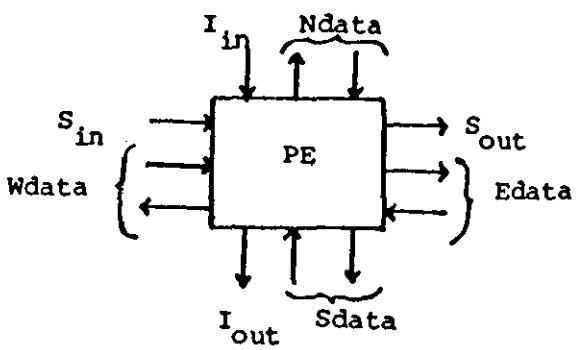
To remain consistent with the rest of the thesis and to retain the possibility of a straightforward mapping of virtual machine to real processor architecture we implemented the ISA in OCCAM. Using the powerful system features of Unix coupled with Loughborough OCCAM the ISA was easily specified as a two part design consisting of:

1. PE library files
2. Grid architecture and virtual spoolers.

The virtual spoolers played the role of buffers for the ISA array interface with higher levels of the system, allowing the bandwidth

of the input to meet that of the ISA. The grid architecture was a simple specification of network connections between processors, the PE libraries simply containing cell descriptions which responded to ISA instructions with different characteristics. Loughborough OCCAM allows the precomputation of library PE's and the grid connection network, which could be simply linked when the virtual machine was required to run - effectively plugging in the correct PE's. Thus a user of the system can develop programs and new PE's with only an abstract working knowledge of the ISA grid.

The virtual grid architecture is shown in Fig.(9.3.2) based on the cell structure



NDATA = North input/output
E " = East " "
S " = South " "
W " = West " "

S = selector in and out

I = instruction in and out

for a 4×4 case. The correct channels can be hooked up by a simple computation using the grid PE position of the form,

```
PROC loc(VALUE i,j, VAR r)=
  SEQ
     r:=(((i-1)*(n+1))+j)-1:
```

The PE to fit the locations is called as a library routine

```
EXTERNAL PROC PE(CHAN wn,we,ws,ww,rn, re, rs, rw, in, is, sw, se  )=
```

and the library PE section uses the PE definitions

```
LIBRARY PROC PE(CHAN wn, we, ws, ww, rn, re, rs, rw, in, is, sw, se  )=
    -- code for cell here.
```

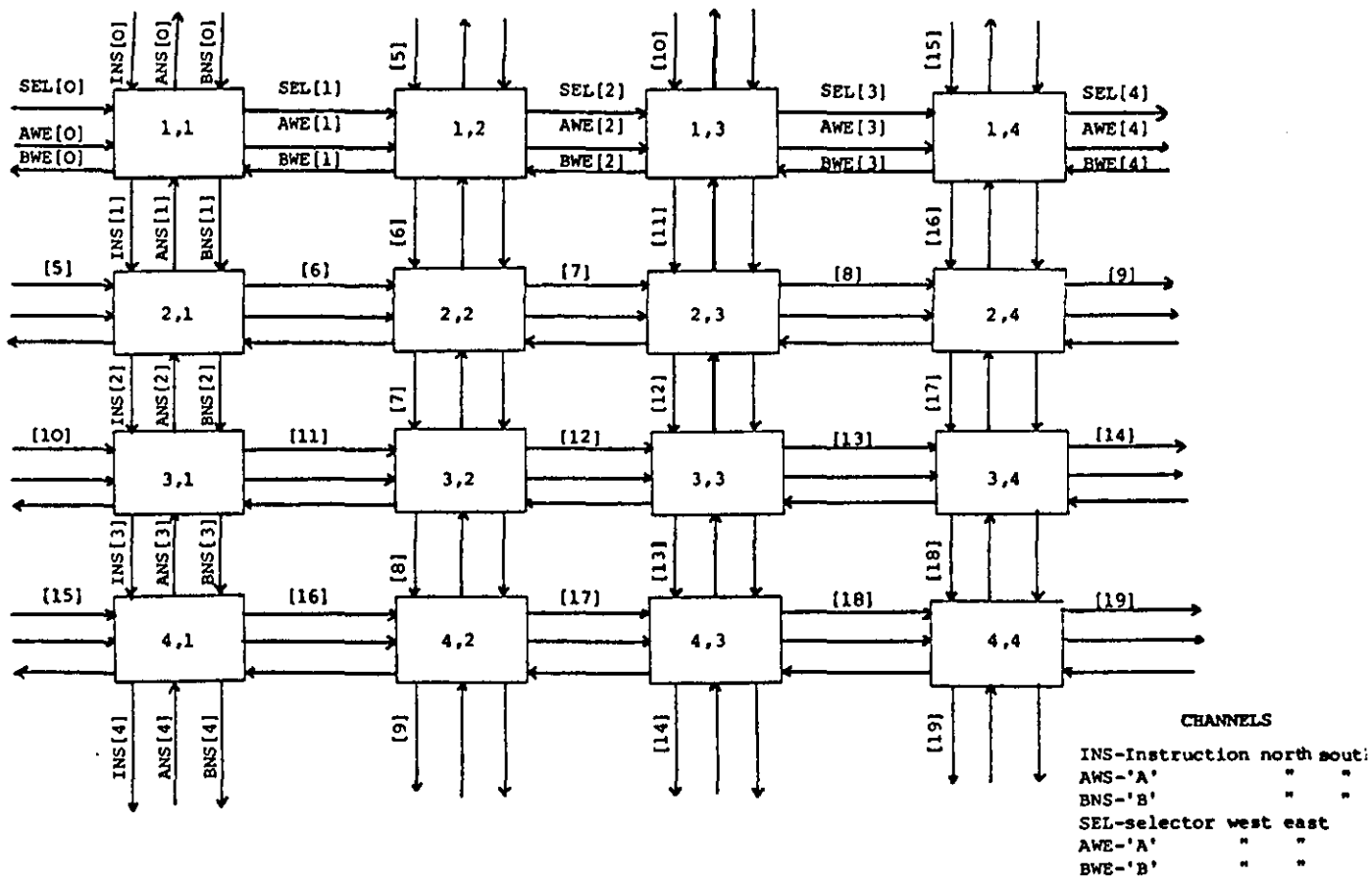
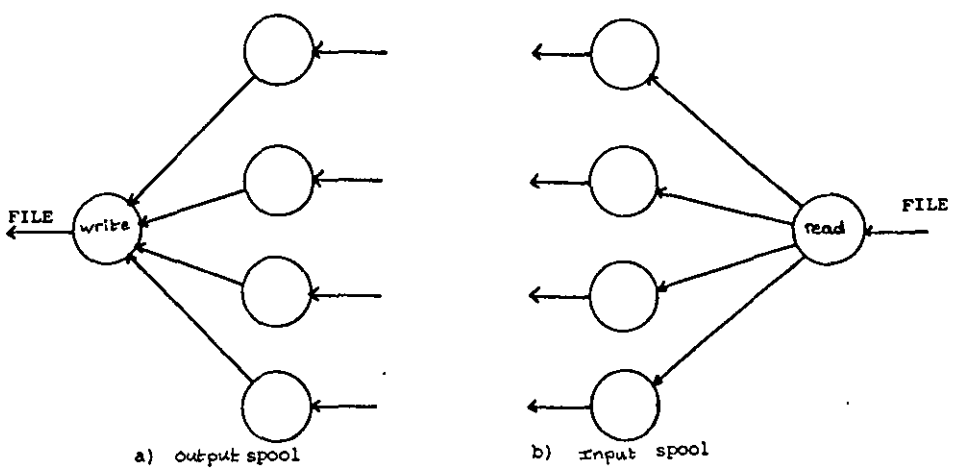


FIGURE 9.3.2: Channel specification for ISA grid

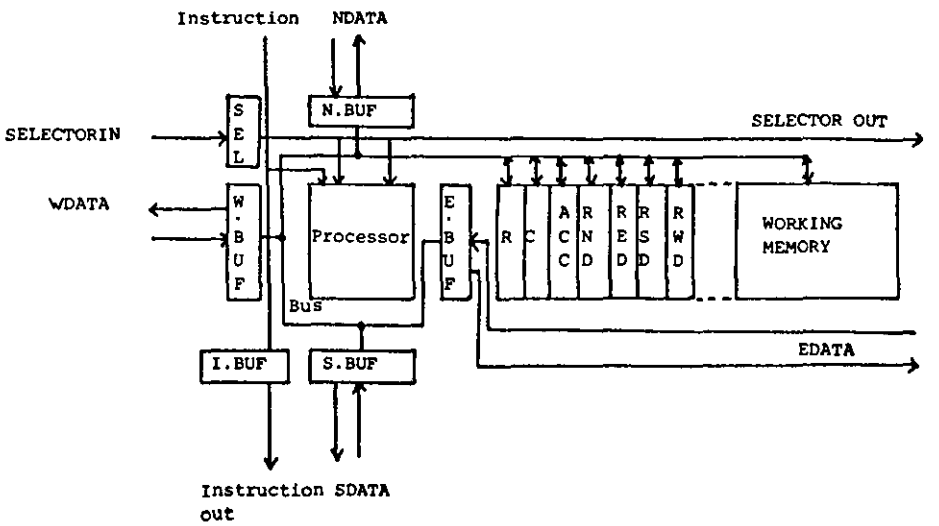
The actual code is given in the Appendix. Included with the ISA grid specification is the data and instruction spooler code. The spoolers are concurrent processes representing buffers for data and instructions input to the boundary cells of the grid. The spoolers also include data output and instruction/selector garbage collection for values falling off the grid. The interface between the virtual machine and the program/PE development section is assumed to be of narrow bandwidth. Infact all data and instructions are assumed to be placed in three files DATA IN, SELECTOR, INSTRUCT, and output is dumped in DATAOUT to represent virtual spool files. The virtual spoolers read these files sequentially and convert the input into a parallel form for the ISA. Likewise for the ISA output the spooler converts the output back into a single stream output sequentially to DATAOUT. The reading of input and writing of output data is performed in parallel with ISA execution. Clearly this is the place where any bottlenecks are likely to occur



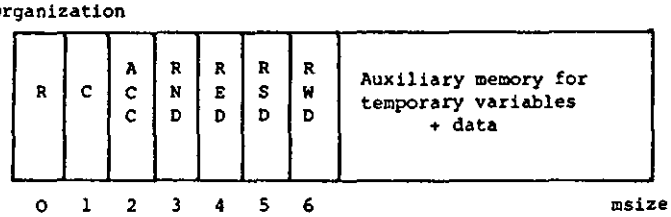
VIRTUAL SPOOLERS (for n=4)

especially for large n. The spoolers can also be used to pad out unused cells with dummy values, when the ISA program running is smaller than the total number of virtual processors. Hence the system with a bounded number of processors can simulate smaller networks without difficulty.

Next we consider the description of a very general processing element which allows the simulation of a wide range of algorithms, and also indicates the method of pumping instructions and selectors through the array. By changing or reducing instruction definitions a range of virtual PE's can be easily developed. The structure of the PE is shown in Fig.(9.3.3) and consists of a central processing element



Memory Organization



R = result register - holds result of computation until c has been read

C = communication register

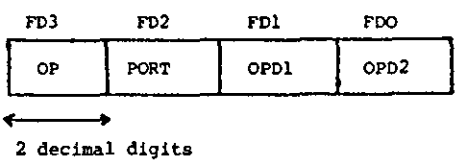
RND = Register north data input

RED = " east " "

RSD = " south " "

RWD = " west " "

Instruction format



Instruction enable = (I<>0) AND select

FIGURE 9.3.3: Basic PE

enabled by the selector and any instruction (other than no-op), A simple bus connects the port input buffers to memory, which contains port value storage registers (RND, RED, RSD, RWD) as well as working memory for data and results. R acts as the accumulator and is backed up by ACC (a secondary accumulator for complex computations), and C is the communication register. The processor embodies all the principles of the ISA cell. Communication is achieved by first loading the output buffers with C and then reading input and output in parallel. The input buffers are then read sequentially to memory to complete the communication phase. Various masks can be constructed for input buffers to prevent overwriting of old buffers avoiding unnecessary movement of data in the memory. The port mask is defined as part of the processor instruction which comprises four fields each 2-decimal digits wide, allowing the implementation of up to 100 instructions or internal memory addresses. The port specification also allows 100 combinations of input/output but only 16 have been used. One possible extension is to utilise the extra slots to allow multiple communication registers in each cell.
REMARK: These operations can be implemented more effectively by using bit logic and slices but Loughborough OCCAM is restricted in this respect. Furthermore, a 2-digit field also allows a wide range of Library PE's to be developed.

Fig.(9.3.4) indicates the operation codes and read masks for our trial cell, (a high bit indicates that a value read will be sent to memory, a low bit that it is not).

The resulting instructions are easily decoded by the OCCAM code

```
SEQ j=[0 FOR 4]
  SEQ                          i=instruction integer
    fd[j]:=i\100
    i:=i/100
```

Processor Operation Codes

OP	CODE	COMMENT
00	NULL	No operation
01	COPY	Mov R to C
02	ADD	R:=A+B
03	SUB	R:=A-B
04	MULT	R:=A*B
05	DIV	R:=A/B
06	MIN	R:=MIN(A,B)
07	MAX	R:=MAX(A,B)
08	DATA	C:=A
09	MOV	Mem[FDO]:=A

A=MEM[FD1] B=MEM[FDO]

Port Controllers

W	S	E	N	INPUTS VALID	
0	0	0	0	No valid data	
0	0	0	1	N	valid
0	0	1	0	E	valid
0	0	1	1	N,E	valid
0	1	0	0	S	valid
0	1	0	1	S,N	valid
0	1	1	0	S,E	valid
0	1	1	1	S,E,N	valid
1	0	0	0	W	valid
1	0	0	1	W,N	valid
1	0	1	0	W,E	valid
1	0	1	1	W,E,N	valid
1	1	0	0	W,S	valid
1	1	0	1	W,S,N	valid
1	1	1	0	W,S,E	valid
1	1	1	1	W,S,E,N	valid

FIGURE 9.3.4

and the port mask with port:=fd[2]

```
SEQ i=[0 FOR 4]
  SEQ
    P[i]:=PORT\2
    PORT:=PORT/2
```

The full PE is given in the program appendix.

Having defined a general PE definition, some simple test programs were developed using a format akin to machine code - making the ISA

program difficult to modify or relate to the abstract algorithm. Consequently the Replicating Instruction Systolic Array Language (RISAL) was devised to provide a very primitive program environment but adequate for testing. RISAL accepts instructions in an assembler like form, but is fairly permissive about the format of statements, subject of course to syntax (the core of which is shown in Fig.(9.3.5)). RISAL also performs a proportion of semantic rules which permit selectors, instructions and data to be converted to ISA form by the same program. Each instruction, selector or data command can be prefixed by a replicating command which generates the following instruction a specified number of times. Checks are performed to ensure that enough data, instructions, or selector inputs are generated for the correct virtual grid size. As a simple example,

```
DATA n,03,00
```

reads the north data port and moves the value into the communication register for the PE defined previously.

```
DATA n,03,00; DATA n,03,00; DATA n,03,00; DATA n,03,00:
```

issues the same command to 4 columns of a 4×4 grid simultaneously and is equivalent to the replicated form,

```
REP(4) DATA n,03,00:
```

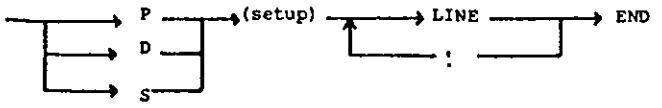
More complex test examples are given below to clarify the syntax and usefulness of the REP command for large arrays.

The structure of a file input to RISAL must identify the following properties for the simulation:

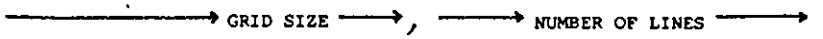
(i) instruction (p), selector (s), or data (d) file.

(ii) the size of the grid - the instruction and selector values can be different for rectangular grids.

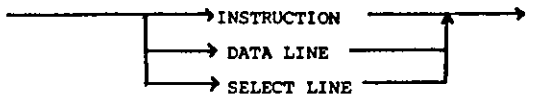
1. RISAL FILE



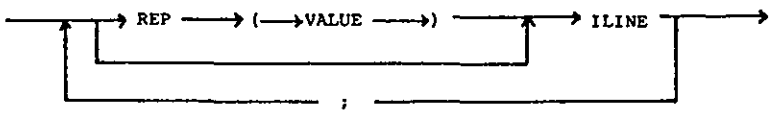
2. SETUP



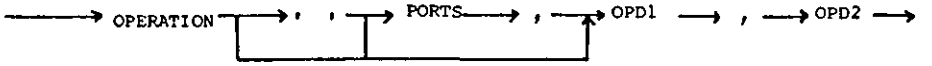
3. LINE



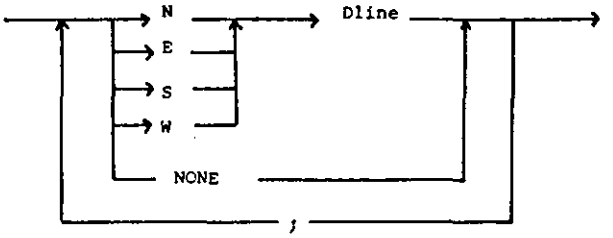
4. INSTRUCTION



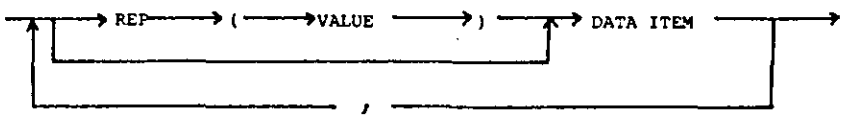
5. ILINE



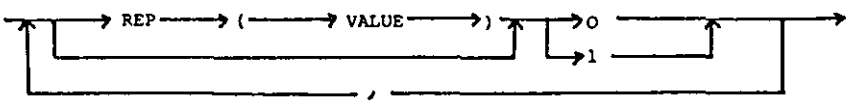
6. DATA LINE



7. Dline



8. SELECT LINE



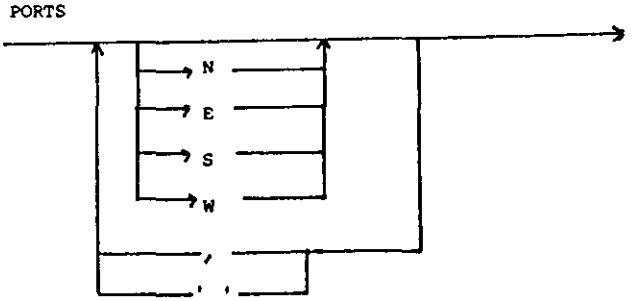
9. VALUE=Integer<gridsize

10. Grid size=Maximum number of columns or rows of processors

11. DATA ITEM=REAL (but can be extended to other more complex types)

12. OPERATION=RESERVED (Mnemonic) keyword for operation

13. PORTS



14. OPD1, OPD2 } Integers in range 0... msize-1
(msize=size of PE private memory)

FIGURE 9.3.5: RISAL syntax diagrams

(iii) the program length - which provides the OCCAM ISA with a primitive shut down facility.

The choice of p,d, or s directs the RISAL compiler to fix the syntax for the particular type of file. The data file is more complex than the rest, as it requires the specification of all four boundaries on the ISA grid. We could define a single file for each boundary but this complicates checking for missing data. Instead we define a single file and sequentially buffer boundary input/output to allow input data to be more easily matched with the ISA instruction sequence. For large grids this method becomes impractical and adding a preprocessor to separate data out into temporary files appears to be the best alternative.

Finally, a special data command NONE is included to mask out a complete boundary, e.g.

```
n  1.0,1.0,2.0,3.0;
e  3.0,rep(3) 0.0;
s  rep(4) 0.0;
   none  :
```

inputs (1.0,1.0,2.0,3.0) to the north grid boundary, (3.0,0.0,0.0,0.0) and (0.0,0.0,0.0,0.0) to the east and south boundaries respectively with west masked out and defaulting to (0.0,0.0,0.0,0.0).

REMARK: Data must always be read in the order n,e,s,w. RISAL checks this,

ISA programs are produced and executed as follows:

(i) Develop three files

I1 = instructions

S1 = selectors

D1 = data

(ii) Run RISAL to check syntax and generate the files

'Instruct', 'Selector', 'Data in'

(iii) All bugs are now semantic errors in the instruction flow of the ISA program,

Compile ISA.occ (virtual grid + spoolers)
Compile PEm.occ (m=library element number)
} if not compiled already

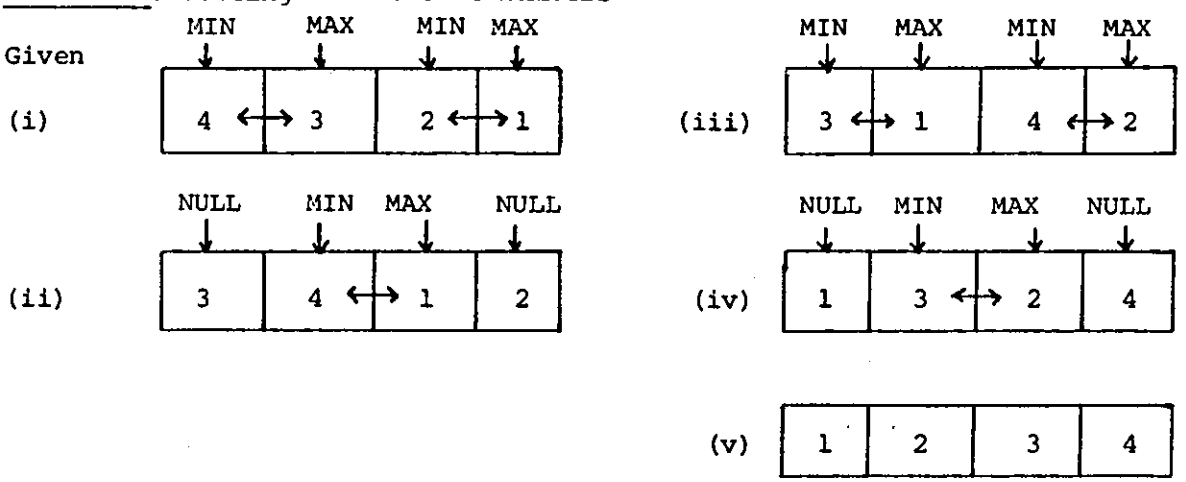
Link two programs (plug in PEm)

(iv) Execute virtual ISA, results placed in file 'dataout'

It is up to the user to ensure that RISAL places its output in the required files and that PEm.occ exists.

The procedure above is quite simple, and was used successfully to produce the following test examples which perform correctly on the virtual grid. The first two examples are quite straightforward, the second two are more involved.

EXAMPLE 1: Sorting a list of 4 numbers



ISA PROGRAM - Sorting a list of 4 numbers

(i) Program

```
p(4,14)
rep(4) null ,0,0:
rep(4) null ,0,0:
rep(4) null ,0,0:
rep(4) data n,3,0:
min e,4,1; max w,6,1; min e,4,1; max w,6,1:
rep(4) copy ,0,0:
null ,0,0; min e,4,1; max w,6,1; null ,0,0:
rep(4) copy ,0,0:
min e,4,1; max w,6,1; min e,4,1; max w,6,1:
rep(4) copy ,0,0:
null ,0,0; min e,4,1; max w,6,1; null ,0,0:
rep(4) copy ,0,0:
rep(4) copy ,0,0:
rep(4) null ,0,0
end
```

(ii) Data

```
d(4,14)
none;none;none;none:
none;none;none;none:
none;none;none;none:
n 4.0,3.0,2.0,1.0;
none;none;none:
none;none;none;none:
none;none;none;none:
none;none;none;none:
none;none;none;none:
none;none;none;none:
none;none;none;none:
none;none;none;none:
none;none;none;none:
none;none;none;none
end
```

(iii) Selector

```
s(4,14)
1,rep(3)0 :
1,rep(3)0 :
1,rep(3)0 :
1,rep(3)0 :
1,rep(3)0 :
1,rep(3)0 :
1,rep(3)0 :
1,rep(3)0 :
1,rep(3)0 :
1,rep(3)0 :
1,rep(3)0 :
1,rep(3)0 :
1,rep(3)0 :
1,rep(3)0
end
```

EXAMPLE 2: 2×2 matrix transpose (see Fig.(9.1.3))

(i) Program

```
p(4,13)
{ load matrix }
data n,03,0; rep(3)null n,0,0:
rep(2)data n,03,0; rep(2)null n,0,0:
null n,0,0;data n,03,0; rep(2) null n,0,0:
{ tranpose }
data e,04,0; data w,06,0; rep(2) null n,0,0:
data n,03,0; rep(3) null n,0,0:
data s,05,0; rep(3) null n,0,0:
data e,04,0;data w,06,00; rep(2) null n,0,0 :
{ read out }
data s,05,00; data s,05,00; rep(2)null n,0,0:
rep(4) null n,0,0:
rep(4) null n,0,0:
rep(4) null n,0,0:
rep(4) null n,0,0:
rep(4) null n,0,0
end
```

(ii) Data

```
d(4,13)
n 6.0,rep(3)0.0;
none; none; none:
n 8.0,2.0,rep(2)0.0;
none; none; none :
n 0.0,5.0,0.0,0.0 ;
none;none;none :
none;none;none;none:
none;none;none;none:
none;none;none;none:
none;none;none;none:
none;none;none;none:
none;none;none;none:
none;none;none;none:
none;none;none;none
end
```

(iii) Selector

```
s(4,13)
1,rep(3)0 :
rep(2)1, rep(2)0 :
1, rep(3)0 :
1, rep(3) 0 :
rep(4) 0 :
rep(2)1, rep(2)0 :
1,rep(3) 0 :
rep(4) 0 :
rep(2) 1, rep(2) 0:
1, rep(3) 0 :
1, rep(3) 0 :
1, rep(3) 0 :
rep(4) 0
end
```

EXAMPLE 3: 4×4 matrix transpose

This is a more complex transposition problem incorporating the use of the 2×2 problem defined earlier. Trace the programs through to ensure that,

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}^T = \begin{bmatrix} 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \\ 4 & 8 & 12 & 16 \end{bmatrix}$$

EXAMPLE 4: 4×4 LU decomposition

Trace through the program to show that:

$$\begin{bmatrix} 2 & 3 & 3 & 2 \\ 4 & 1 & 2 & 3 \\ 2 & 2 & 5 & 1 \\ 3 & 4 & 1 & 2 \end{bmatrix} = \begin{bmatrix} 1 & & & \\ 2 & 1 & & \\ 1 & 0.2 & 1 & \\ 1.5 & 0.1 & -1.107143 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 2 & 3 & 3 & 2 \\ & -5 & -4 & -1 \\ & & 2.8 & -0.8 \\ & & & -1.785714 \end{bmatrix}$$

REMARK: If an extra column was used to supply the r.h.s. of a linear system the factors can be ignored producing the Gaussian Elimination solution.

```
p(4,34)
data n,3,0; rep(3) null ,0,0:
rep(2) data n,3,0; rep(2) null ,0,0:
rep(3) data n,3,0; nulll ,0,0:
rep(4) data n,3,0 :
null ,0,0; rep(3) data n,3,0:
rep(2) null,0,0; rep(2) data n,3,0:
rep(3) null ,0,0; data n,3,0:
rep(4) null ,0,0:
data e,4,0; data w,6,0; rep(2) null ,0,0:
data n,3,0; rep(3) null,0,0:
data s,5,0; null,0,0; data e,4,0; data w,6,0:
data e,4,0; data w,6,0; data n,3,0; null ,0,0:
rep(2) null,0,0; data s,5,0; null ,0,0:
data e,4,0; data w,6,0; data e,4,0; data w,6,0:
null ,0,0; data e,4,0; data w,6,0; null ,0,0:
data e,4,0; data w,6,0; data e,4,0; data w,6,0 :
data n,3,0; data e,4,0;data w,6,0; null ,0,0:
data s,5,0; data n,3,0; data e,4,0; data w,6,0:
data n,3,0; data s,5,0; rep(2) null ,0,0:
data s,5,0; data n,3,0; rep(2) null ,0,0:
null ,0,0; data s,5,0; rep(2) null ,0,0:
data e,4,0; data w,6,0; rep(2) null ,0,0:
null ,0,0; data e,4,0; data w,6,0; null ,0,0:
data e,4,0; data w,6,0; data e,4,0; data w,6,0:
null ,0,0; data e,4,0; data w,6,0; null ,0,0:
data s,5,0; null ,0,0; data e,4,0; data w,6,0:
rep(2) data s,5,0; rep(2) null ,0,0:
rep(3) data s,5,0; null ,0,0:
rep(4) data s,5,0:
rep(4) data s,5,0:
null ,0,0; rep(3) data s,5,0:
rep(2) null ,0,0; rep(2) data s,5,0:
rep(3) null ,0,0; data s,5,0:
rep(4) null ,0,0
end
```

(i) Program

```
d(4,34)
n 13.0,0.0,0.0,0.0;
none;none;none:
n 9.0,14.0,0.0,0.0;
none;none;none:
n 5.0,10.0,15.0,0.0;
none;none;none:
n 1.0,6.0,11.0,16.0;
none;none;none:
n 0.0,2.0,7.0,12.0;
none;none;none:
n 0.0,0.0,3.0,8.0;
none;none;none:
n 0.0,0.0,0.0,4.0;
none;none;none:
none;none;none;none:
none;none;none;none:
none;none;none;none:
none;none;none;none:
none;none;none;none:
none;none;none;none:
none;none;none;none:
none;none;none;none:
none;none;none;none:
none;none;none;none:
none;none;none;none:
none;none;none;none:
none;none;none;none:
none;none;none;none:
none;none;none;none:
none;none;none;none:
none;none;none;none:
none;none;none;none:
none;none;none;none:
none;none;none;none:
none;none;none;none:
none;none;none;none:
none;none;none;none:
none;none;none;none:
none;none;none;none
end
```

(ii) Data

```
s(4,34)
1,0,0,0:
1,1,0,0:
1,1,1,0:
1,1,1,1:
0,0,0,0:
0,0,0,0:
0,0,0,0:
1, rep(3) 0 :
1, rep(3) 0:
0,0,1,0:
rep(3) 1, 0:
1, rep(3) 0 :
1,0,1,1:
rep(3) 1, 0:
1,1,0,0:
1,1,0,0:
0,1,0,0:
1,1,0,0:
0,1,1,0:
rep(4)1:
rep(3) 1,0:
rep(4) 1 :
1,1,0,0:
1,1,0,0:
1,1,0,0:
rep(4)0:
rep(4)0:
rep(3) 1,0:
1,1,0,0:
1,0,0,0:
rep(4) 0:
rep(4) 0:
rep(4) 0:
rep(4) 0
end
```

(iii) Selector

<u>EXAMPLE 3</u>: 4×4 matrix transpose program

```
{ 4*4 lu decomposition }
p(4,39)
{ load matrix}
data n,3,0; rep(3) null n,0,0 :
rep(2) data n,3,0; rep(2) null n,0,0:
rep(3) data n,3,0; null n,0,0 :
rep(4) data n,3,0:
{  start factorisation}
mov s,1,7; rep(3) data n,3,0:
data n,3,0; mov s,1,7; rep(2) data n,3,0:
div ,7,3; data n,3,0; mov s,1,7; data n,3,0:
copy ,0,0; null ,0,0; data n,3,0; mov s,1,7:
null ,0,0; data w,6,0; null ,0,0; data n,3,0 :
null ,0,0; mult ,3,6; data w,6,0; null ,0,0:
null ,0,0; sub ,7,0 ; mult ,3,6; data w,6,0 :
null ,0,0; copy ,0,0; sub ,7,0; mult ,3,6 :
null ,0,0; mov s,1,7; copy ,0,0; sub ,7,0 :
null ,0,0; data n,3,0; mov s,1,7; copy ,0,0 :
null ,0,0; div ,7,3; data n,3,0; mov s,1,7:
null ,0,0; copy ,0,0; null ,0,0; data n,3,0:
rep(2) null ,0,0; data w,6,0; null ,0,0:
rep(2) null ,0,0; mult ,3,6; data w,6,0:
rep(2) null ,0,0; sub ,7,0; mult ,3,6:
rep(2) null ,0,0; copy ,0,0; sub ,7,0:
rep(2) null ,0,0; mov s,1,7; copy ,0,0:
rep(2) null ,0,0; data n,3,0; mov s,1,7:
rep(2) null ,0,0; div ,7,3; data n,3,0 :
rep(2) null ,0,0; copy ,0,0; null ,0,0 :
rep(3) null ,0,0; data w,6,0:
rep(3) null ,0,0; mult ,3,6:
rep(3) null ,0,0; sub ,7,0:
rep(3) null ,0,0; copy ,0,0:
{ read result}
data s,5,0 ; rep(3) null ,0,0:
rep(2) data s,5,0; rep(2) null,0,0:
rep(3) data s,5,0; null,0,0:
rep(4) data s,5,0:
rep(4) data s,5,0:
rep(4) data s,5,0:
null ,0,0; rep(3) data s,5,0:
rep(2) null ,0,0; rep(2) data s,5,0:
rep(3) null ,0,0; data s,5,0:
rep(4) null ,0,0
end
```

(i) Program

```
s(4,39)
1, rep(3)0 :
rep(2)1, rep(2)0:
rep(3)1, 0:
rep(4)1 :
rep(4)0 :
0 ,1,rep(2)0:
0 ,1,1,0:
0, rep(3)1:
0, rep(3)1:
0, rep(3)1:
0, rep(3)1:
0, rep(3)1:
rep(2) 0, rep(2)1:
rep(2) 0, rep(2)1:
rep(2) 0, rep(2)1:
rep(2) 0, rep(2)1:
rep(2) 0, rep(2)1:
rep(2) 0, rep(2)1:
rep(2) 0, rep(2)1:
rep(2) 0, rep(2)1:
rep(3) 0, 1:
rep(3) 0, 1:
rep(3) 0, 1:
rep(3) 0, 1:
rep(3) 0, 1:
rep(3) 0, 1:
rep(3) 0, 1:
rep(3) 0, 1:
rep(4) 0:
rep(4) 0:
rep(4) 0:
rep(4) 1:
rep(3) 1,0:
rep(2) 1, rep(2) 0 :
1,rep(3) 0:
rep(4) 0:
rep(4) 0:
rep(4) 0:
rep(4) 0
end
```

(ii) Selector

```
d(4,39)
n 3.0, rep(3) 0.0 ;
none; none; none:
n 2.0, 4.0, 0.0, 0.0 ;
none; none; none:
n 4.0, 2.0, 1.0 ,0.0;
none; none; none :
n 2.0, 1.0, 5.0, 2.0;
none; none; none :
n 0.0, 3.0, 2.0, 1.0;
none; none; none:
n 0.0, 0.0, 3.0, 3.0;
none; none; none:
n 0.0, 0.0, 0.0, 2.0;
none; none; none:
none;none;none;none:
none;none;none;none:
none;none;none;none:
none;none;none;none:
none;none;none;none:
none;none;none;none:
none;none;none;none:
none;none;none;none:
none;none;none;none:
none;none;none;none:
none;none;none;none:
none;none;none;none:
none;none;none;none:
none;none;none;none:
none;none;none;none:
none;none;none;none:
none;none;none;none:
none;none;none;none:
none;none;none;none:
none;none;none;none:
none;none;none;none:
none;none;none;none:
none;none;none;none:
none;none;none;none:
none;none;none;none:
none;none;none;none:
none;none;none;none:
none;none;none;none:
none;none;none;none:
none;none;none;none:
none;none;none;none
end
```

(iii) Data

EXAMPLE 4: 4×4 LU decomposition program

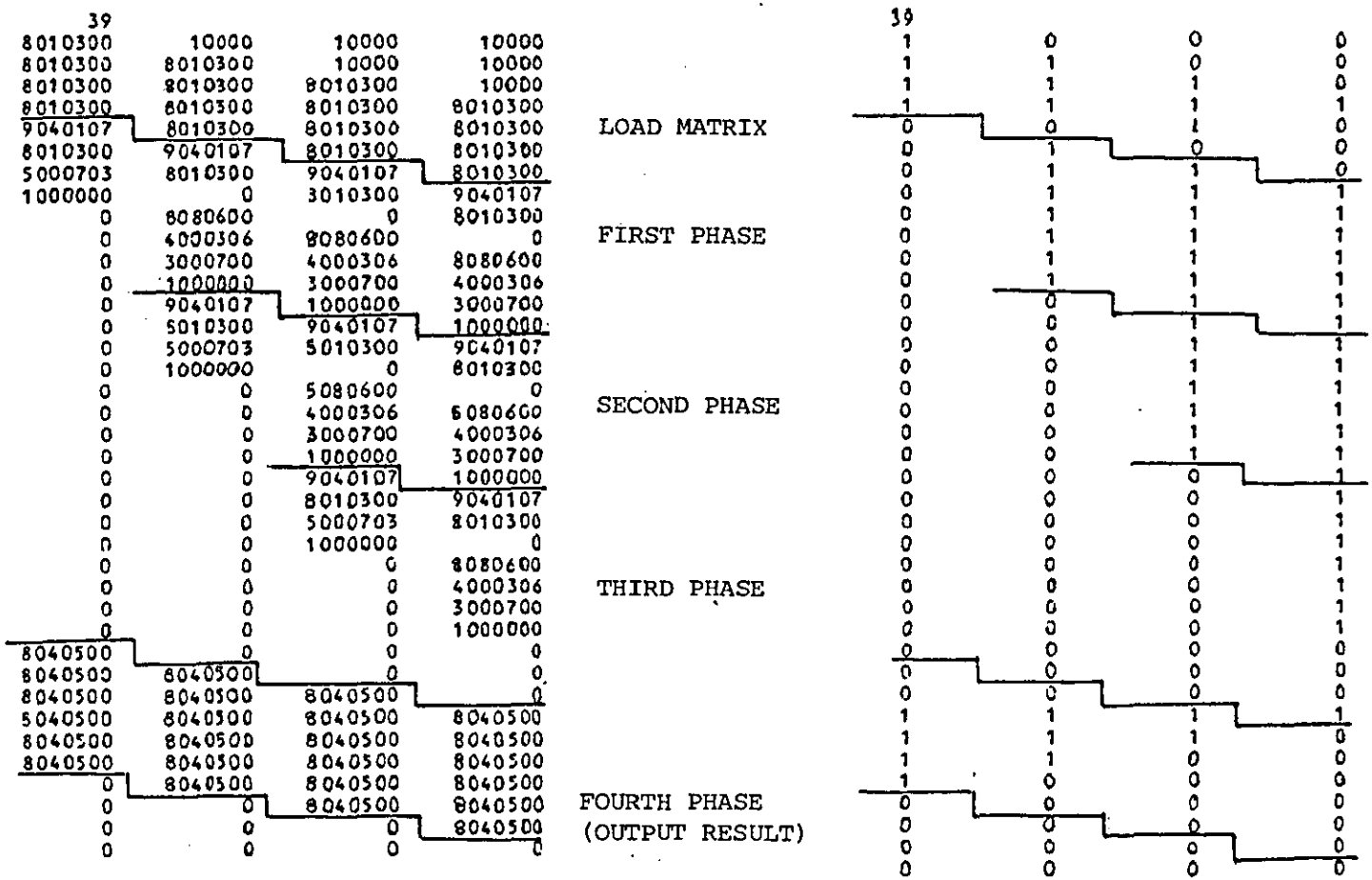
```
   34
8010300        0        0        0
8010300  8010300        0        0
8010300  8010300  8010300        0
8010300  8010300  8010300  8010300
      0  8010300  8010300  8010300
      0        0  8010300  8010300
      0        0        0  8010300
      0        0        0        0
8020400  8080600        0        0
8010300        0        0        0
8040500        0  8020400  8080600
8020400  8080600  8010300        0
      0        0  8040500        0
8020400  8080600  8020400  8080600
      0  8020400  8080600        0
8020400  8080600  8020400  8080600
8010300  8020400  8080600        0
8040500  8010300  8020400  8080600
8010300  8040500        0        0
8040500  8010300        0        0
      0  8040500        0        0
8020400  8080600        0        0
      0  8020400  8080600        0
8020400  8080600  8020400  8080600
      0  8020400  8080600        0
8040500        0  8020400  8080600
8040500  8040500        0        0
8040500  8040500  8040500        0
8040500  8040500  8040500  8040500
8040500  8040500  8040500  8040500
      0  8040500  8040500  8040500
      0        0  8040500  8040500
      0        0        0  8040500
      0        0        0        0
```

(i) Program

```
34
1        0        0        0
1        1        0        0
1        1        1        0
1        1        1        1
0        1        1        1
0        0        1        1
0        0        0        1
1        0        0        0
1        0        0        0
0        1        0        0
1        1        1        0
1        0        1        1
1        0        1        0
1        1        0        0
1        1        0        0
0        1        0        0
1        1        0        0
1        1        1        0
1        1        1        1
1        1        1        0
1        1        0        0
1        1        0        0
0        1        0        0
1        1        0        0
1        0        0        0
0        0        1        0
1        1        0        0
1        1        0        0
0        1        0        0
0        0        0        0
0        0        0        0
0        0        0        0
```

(ii) Selector

EXAMPLE 3: Matrix transpose ISA machine code



(i) Program (ii) Selector

EXAMPLE 4: LU-decomposition ISA machine code

Although RISAL is very primitive it has been useful in illustrating the ISA's capabilities and has suggested some improvements to the design of PE's, the interfacing arrangements such as spooling for the virtual grid and a number of additional features to produce a more robust version of RISAL itself.

To allow a wide flexibility in PE development it was observed that reading operation definitions from a file (in alphabetical order) including operation codes allowed new commands to be enlisted easily inside RISAL and permitted the same codes for different operations in alternative PE's. We remark that care must be taken in using duplicate codes but no real problems were encountered.

For RISAL three main constructs suggested themselves and can be listed as follows:

(i) Replicated line instruction (REPL): of the form REPL(count) [Line] where the line enclosed by [,] is repeated count times e.g.

REPL(7)[data n,03,00; rep(3) null,0,0]

This appears simple to implement with a stack to maintain nested REP operations and storage to hold the full line statement. For large grids this may pose a significant problem.

(ii) Replicated line section (REPS): For example,

data n,03,00; REPS(count)[null,0,0; data n,03,00]; null,0,0:

which would repeat the section of the line in brackets count times. The main difficulty in implementing this statement is keeping track of REP nesting and checking that the correct number of instructions is generated.

(iii) Replicated line shift (REPLS): of the form,

REPLS(count,shift)[line]:

Here a specified line is replicated count times and on each replication is shifted right or left. 'Shift' places according to the sign of the shift. Instructions falling off the end of a line must be neglected and spare places filled with a default operation like null.

Many variations to these basic constructions such as cyclic line shifting, shifting of line sections, and conditional line shifting are also apparent - but amount only to improving the readability of the ISA program.

Next consider the mapping of the design onto some real underlying mesh architecture which preserves operation of the virtual grid. There are basically two types of mapping we can consider with 1-1 or many-1 processor correspondences.

(i) 1-1 correspondence:

This is the simplest mapping, in which each PE in the virtual grid maps onto a single real processing element (e.g. a transputer). Some modifications to the ISA program are required to ensure only 4 outgoing and 4 incoming channels. Notice however that the PE is designed to allow instructions and selectors to be processed sequentially before data communication allowing multiplexing of instructions and data on the same channels.

(ii) MANY-1 correspondence:

For large virtual grids we can consider mapping a number of virtual PE's onto a single PE, to reduce the total number of real PE's and reduce the actual mesh bandwidth. The many-1 mapping is implemented by a special virtual PE definition which acts like a plug adapter fixing onto a real PE and essentially simulates a block of virtual processors. The structure of a plug is shown in Fig.(9.3.6), together

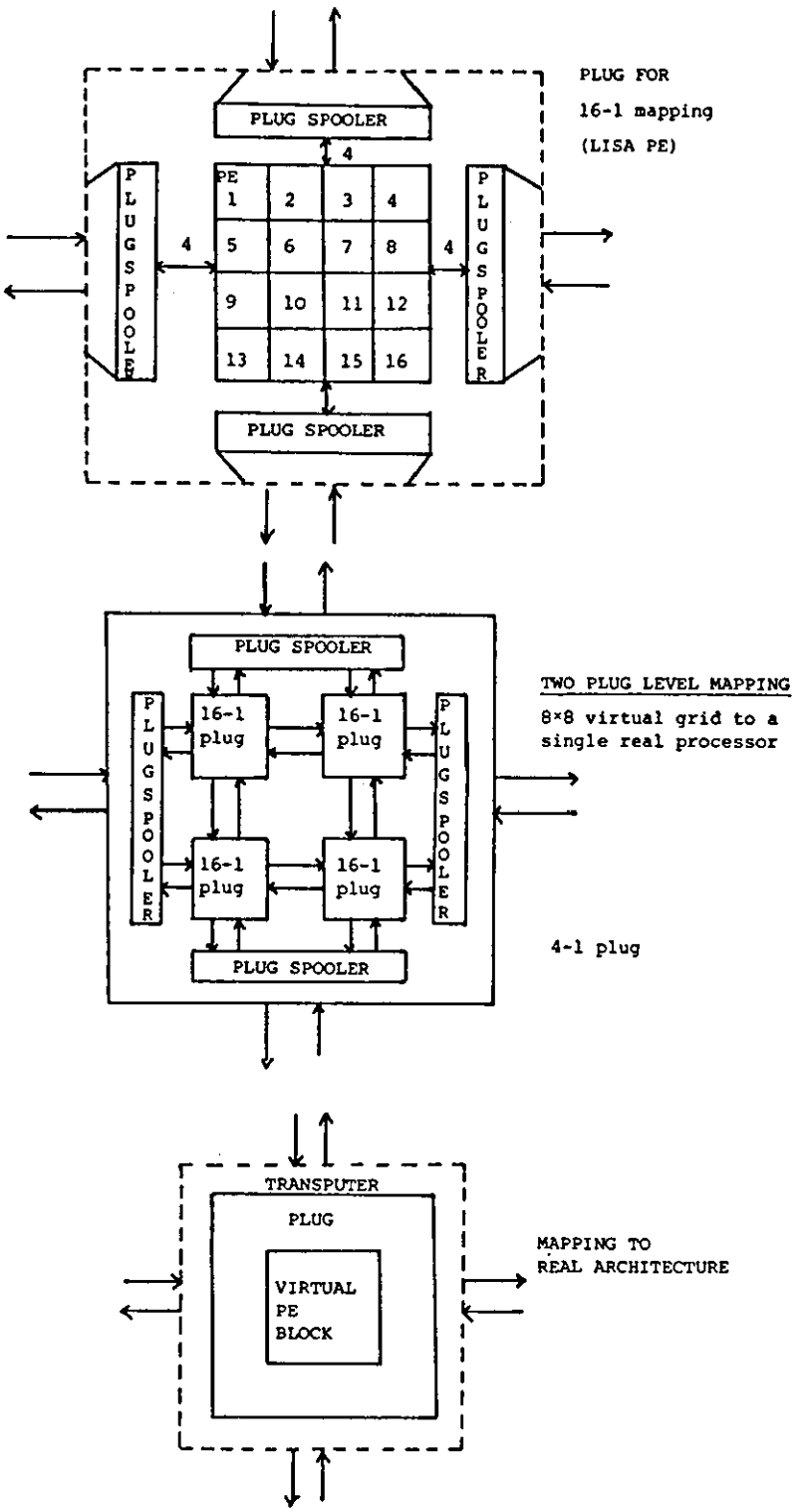


FIGURE 9.3.6: ISA plugs

with nested plug connections. The plug idea solves the spooler problem for large meshes by producing virtual fan-in and fanout communication trees with heights proportional to the number of recursive plugs adopted. The only problem is the reduced efficiency of the real array having to simulate all the plugs and virtual PE's. A second solution to the many-1 mapping is to rewrite the plug definition to accept a modified ISA program. Notice that a plug implies some multiplexing of instruction and selector information on a single cell input. In Section (9.5) we discuss an efficient alternative to virtual plugs.

9.4 SIMULATION OF ARRAYS WITH BOUNDARY AND SPECIAL PROCESSING ELEMENTS

The SSPS uses a virtual instruction systolic array to simulate SIMD, MIMD and systolic algorithms using RISAL. Linear pipeline algorithms are the simplest to implement using only a single ISA row, and a range of simple PE's can be developed to provide efficient implementations of basic cell operations. Systolic arrays with homogeneous cells require the development of a PE with only a single instruction. These algorithms extend easily to 1-D and 2-D SSPS simulations due to their close relation to SIMD algorithms.

More general systolic algorithms have a few different cells making them partial SIMD algorithms, computing multiple instructions on multiple data streams but with instructions fixed over time. In this section we develop the idea of Dynamic Instruction Modification (DIM) which allows a class of systolic algorithms to be simulated in the same time as full SIMD ISA programs. As examples, we consider the familiar triangular Gaussian Elimination, and hexagonal LU decomposition arrays.

Before examining these algorithms we consider the simpler problem

of simulating linear arrays on the SSPS. For simplicity we consider two separate cases:

(i) arrays with only a single cell type

(ii) arrays with multiple cell types.

Single cell types:

Ths mapping is quite simple. Define the virtual ISA_n where n is the number of cells in the linear array, and develop a library PE to accept a single instruction I_1 equivalent to that cell type.

Theorem 9.4.1: If A is a linear array with one cell type and n-cells which computes in a time T(n), it can be simulated on the virtual ISA_n in T(n)+n steps.

Proof: (By construction of the ISA program)

Without loss of generality let n=4, then,

(i) Select a 4×4 ISA grid with PE≡cell then the ISA_4 program below simulates A.

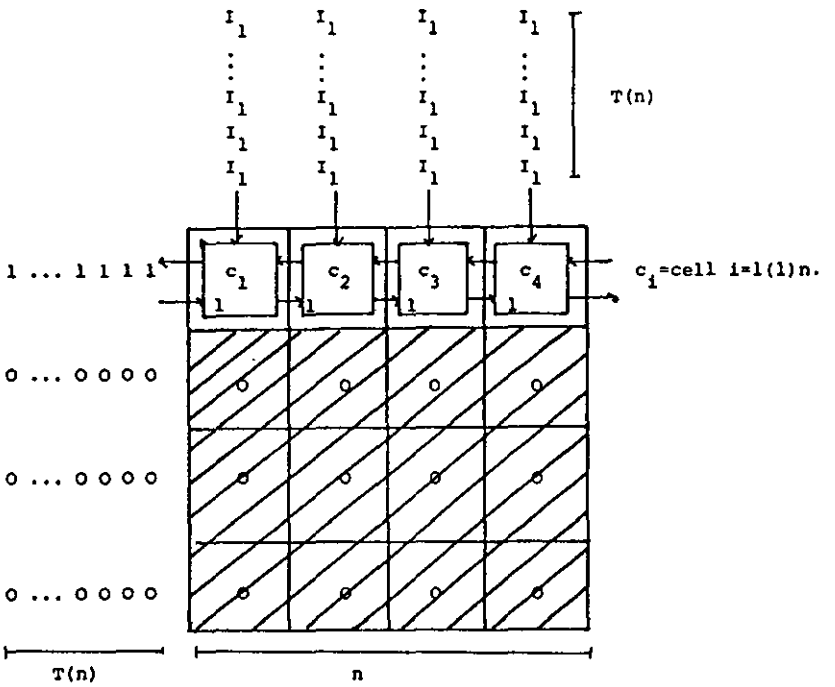


FIGURE 9.4.1: ISA simulation of single cell type 1-D arrays

(ii) placing selectors in the first row of the grid requires n=4 PE cycles.

(iii) the algorithm requires T(n) array/PE cycles hence T(n) instructions and selectors yielding the total time T(n)+n steps.

During the selector setup period no-ops instructions are pumped into the grid. Data can be input from north, west, and east, making the simulation sufficiently general for many pipeline problems. The only significant drawback being the inefficient use of processors. Later we show how to improve efficiency.

Multiple cell types:

As a simple example consider the backsubstitution array where two types of cell are required, and prove the following theorem.

Theorem 9.4.2: If A is a linear systolic array of n cells with multiple cell types and computation time T(n), it can be simulated on the ISA in T(n)+n steps on the ISA.

Proof:

Without loss of generality we can consider just two cell types and proceeding in a similar manner as for Theorem (9.4.1) to derive the ISA program structure

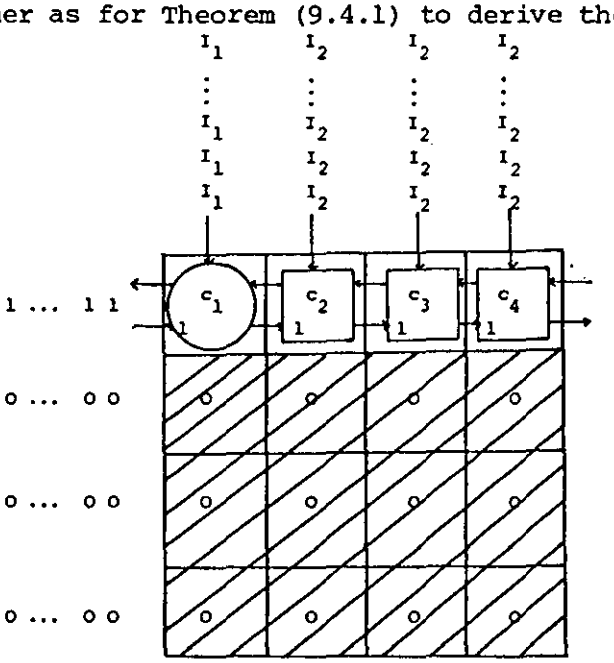


FIGURE 9.4.2: ISA simulation of multiple cell type 1-D arrays

yielding the identical timing.

As before we have a large inefficiency, and we assume that the number of input and outputs fit the ISA grid. For more complex examples the PE definition must be rewritten to multiplex data on the available channels, altering the algorithm time to,

$$cT(n) + n = O(T(n)) .$$

The extension of the proof to many cells is trivial, when each PE is capable of executing any of the cell functions.

For 2-D systolic arrays simulation is split into two types:

i) arrays with a single cell type

ii) " " multiple cell types.

Single cell type 2-D arrays:

Arrays of this type include problems like matrix multiplication and speech recognition (Frison & Quinton [85]), and can be easily simulated by a natural extension of linear single cell techniques.

Theorem 9.4.3: A 2-D systolic array using a single cell type can be simulated by the ISA_n grid in T(n)+2n where T(n) is the time of the original array.

Proof:

Again without loss of generality put n=4, and define a grid with a PE implementing three operations, I_1=cell operation, I_0=the no-op instruction, I_2=a setup operation. I_2 reads data from north to south and decides using the associated input data if the cell is to be switched on (as opposed to selected), e.g.

```
read North A
IF A=0 THEN {change instruction to I1}
ELSE {A=A-1}
```

Next we develop the set up phase of a program using Fig.(9.4.3).

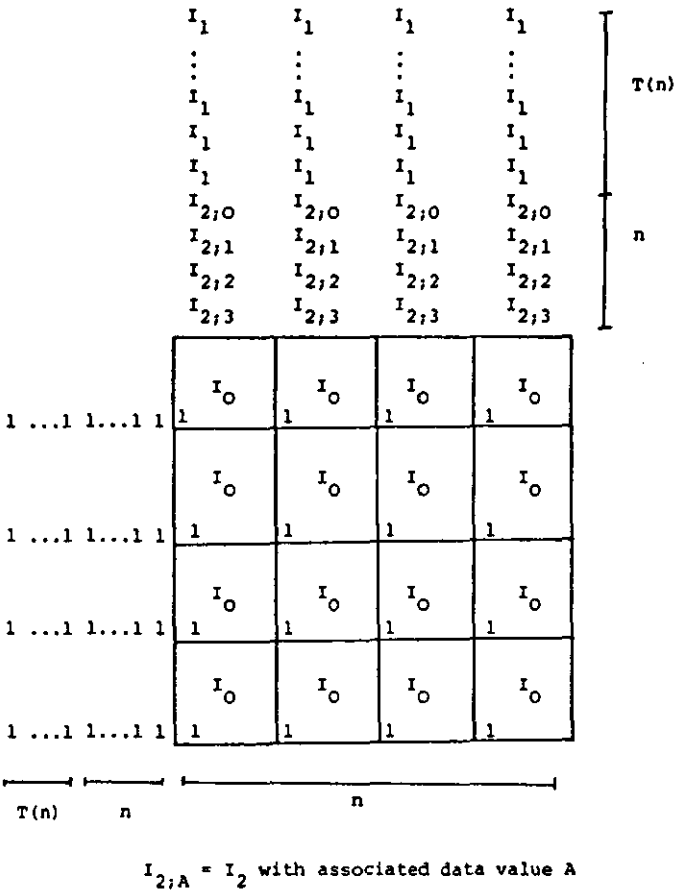


FIGURE 9.4.3: ISA simulation of single cell type 2-D array

as follows:

a) we require n steps to filter selectors and I_0 into the array

b) after a further n steps, all the I_2 associated with address data get modified to I_1 and the array starts computing.

c) compute normally, I_1;c now has real data c associated with it on the north port.

Thus the total time is given by T(n)+2n.

Theorem (9.4.3) allows 2-D single cell type systolic algorithms which do not have a refracted wavefront input format to be setup and run on the ISA. This application is particularly useful if a suitable dummy input to convert algorithms to the refracted form is not available. In addition, Theorem (9.4.3) also illustrates the concept of dynamic

instruction modification (DIM) where essentially instructions moving through the array systolically can be modified. This idea is essential for simulations with multiple cell types.

Multiple cell type 2-D arrays:

Like the linear (1-D) arrays multiple cell type 2-D arrays are simulated in the same manner as the single cell 2-D arrays except that each ISA PE is capable of implementing each cell type. It follows that the simulation time is the same, and we consider here just two examples to illustrate the technique.

The first problem we consider is the triangular Gaussian Elimination array (of Gentleman & H.T. Kung [81]) which represents a significant simulation problem. Notice that if the ISA is to be used, the lower triangular part of the grid will be inactive throughout the computation. The data values flow from north to south, the same direction as instructions, but the distribution of cell types is far from ideal. The different cell types inside each column mean than an instruction flow of a standard ISA cannot possibly mimick the operation of the array without increasing execution time using an arbitrary PA program simulation or modifying dataflow. Here we show that the algorithm can be preserved in time, structure, and dataflow with only the addition of ISA set up time, using DIM. If the array is to be used with high throughput i.e. one problem instance after another, the setup time is an acceptable overhead. The DIM technique is used this time to set the cell type and control the individual cells encountered by entering different instruction states as the instructions move through the array. To understand the DIM operation we need to recall the virtual ISA instruction format, where 4 fields (2 digits wide) were adopted. We

construct a new PE in which the op-code part of the instruction is either active or inactive (no-op) making it similar to a selector. The opd1 and opd2 which normally reference data inside a PE now carry data which can be interpreted as true data or an instruction. For this simulation the nested DIM instructions have the form:

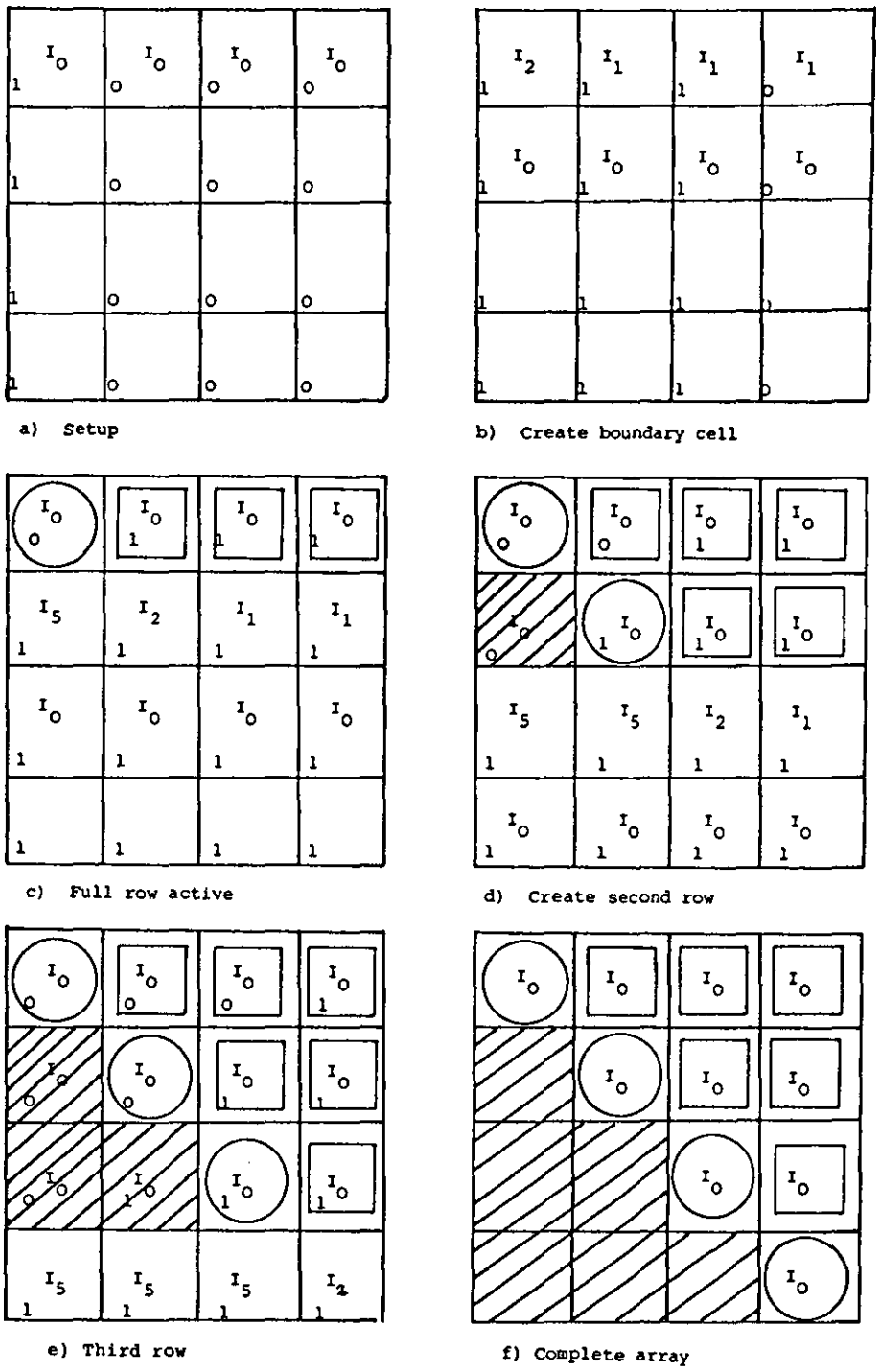


FIGURE 9.4.4: DIM startup procedure for triangularisation array

$$\text{OPD2} = \begin{cases} 5 \text{ null cell} \\ 4 \text{ shift data north to south} \\ 3 \text{ execute cell} \\ 2 \text{ select cell type divider} \\ 1 \text{ select cell type IPS} \end{cases}$$

$$\text{OPD1} = \begin{cases} 1 \text{ an execute cell load north input into save registers} \\ \text{otherwise perform cell operation} \end{cases}$$

$$\text{OP} = \begin{cases} 8 \text{ read data} \\ 0 \text{ null.} \end{cases}$$

We now configure the array on the ISA grid, with instruction modifications part of the virtual PE plugged into the grid, with the format,

$$I_{3,1} = \text{DATA NW,1,0}$$

$$I_{3,0} = \text{DATA NW,0,0}$$

and data for the matrix to be eliminated tagged to instructions the array setup is performed as shown in Fig.(9.4.4) yielding the total timing $T(n)=3n+(n+2)$ steps as follows:

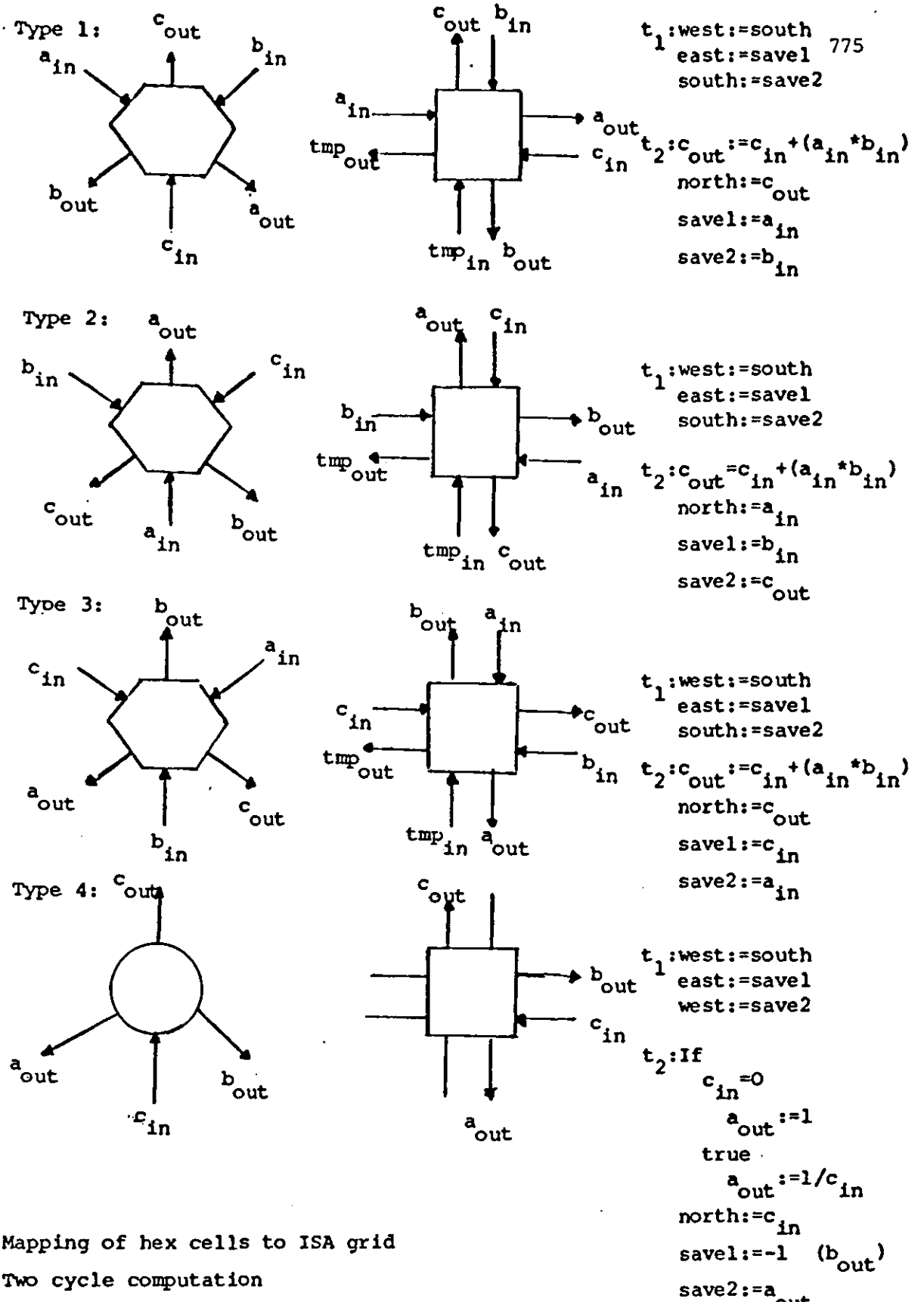
(i) setup =2 steps (as pipelined with computation)

(ii) elimination = 3n steps

(iii) readout result = n.

Notice that the number of steps is the same as for the original systolic array during computation. Even the dedicated array must include n-steps to output the triangularized matrix.

The hexagonal LU-decomposition array represents a similar problem requiring only two cell types. However some cells of the same type are orientated differently and for purposes of simulation must be treated as different cells. In addition the hex array uses a diagonal link which requires special consideration. Fig.(9.4.5) shows the correspondence between hex cells and the grid processors of the ISA.



Mapping of hex cells to ISA grid

Two cycle computation

(i) cycle 1 move c_{out} up and left

(ii) cycle 2 real computation

FIGURE 9.4.5: Cell types for hexagonal LU-decomposition

The reciprocal cell of the hex array is placed in the top left (position 1,1) PE, which automatically defines the layout of the other processors. Hex cells rotated by 120° anti-clockwise appear in the first row, and hex cells rotated by 120° clockwise in the first column of the ISA. The remaining (n-1)*(n-1) 'normal' cells occupying the (n-1)*(n-1) ISA grid consisting of (i,j) i,j=2(1)n ISA rows and columns. Diagonal links are simulated by adding an extra cycle to each basic computation used solely for data movement. A diagonal movement is then achieved by shifting elements north and then west, hence the extra cycle. As a result the normal algorithm time 3n+min(p,q) for a matrix of bandwidth w=p+q-1 is increased to 2(3n+min(p,q)). Using the DIM technique all the cells must be setup and started simultaneously, requiring 2n steps as illustrated by Fig.(9.4.6). The setup time appears extravagant but for repeated use of the array is extremely efficient.

Both the designs were tested using RISAL and the resulting programs are given below, the virtual PE definitions appear in the program appendix with the ISA. Finally, for a banded matrix the LU scheme is easily modified to incorporate more than one null operation code to mask out cells above and below the main grid diagonal.

To conclude this section we return to the inefficient cell usage of linear array schemes, where only a few rows of the ISA grid were used. Fig.(9.4.7) illustrates some efficient linear array layouts for our 4×4 test grid. It should be clear that cells with different orientations of communication but the same cell type can be modelled using a DIM setup scheme to improve processor utilisation. Hence we have shown that systolic arrays with different cell types can be

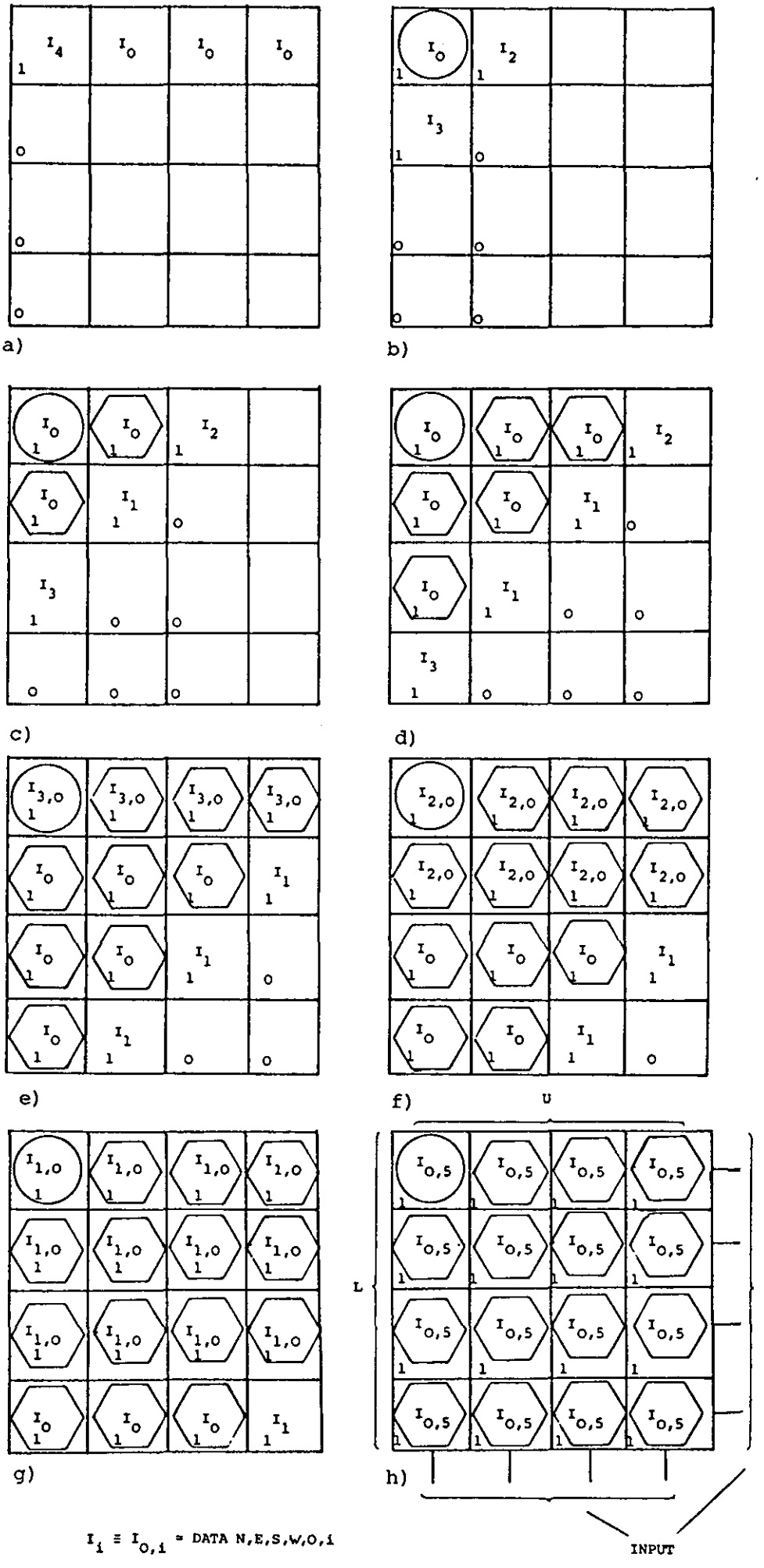


FIGURE 9.4.6: DIM setup procedure for hex LU decomposition simulation

1. GAUSSIAN ELIMINATION

```
p(4,17)
rep(4) null ,0,0 :
data ,0,2; rep(3) data ,0,1 :
data n,1,3; rep(3) null ,0,0 :
data n w,0,3; data n,1,3; rep(2) null ,0,0:
rep(2) data n w,0,3; data n,1,3; null ,0,0:
rep(3) data n w,0,3; data n,1,3:
data s,0,4; rep(3) data n w,0,3:
rep(2) data s,0,4; rep(2) data n w,0,3 :
rep(3) data s,0,4; data n w,0,3:
rep(4) data s,0,4:
rep(4) data s,0,4:
rep(4) data s,0,4:
null ,0,0; rep(3) data s,0,4 :
rep(2) null ,0,0; rep(2) data s,0,4 :
rep(3) null ,0,0; data s,0,4 :
rep(4) null ,0,0:
rep(4) null ,0,0
end
```

RISAL PROGRAMS FOR TRIANGULAR

AND HEXAGONAL ARRAY USING DIM

```
d(4,17)
none;none;none;none :
none;none;none;none :
n 2.0,0.0,0.0,0.0 ;
none;none;none :
n 4.0, 3.0, 0.0,0.0 ;
none;none;none :
n 2.0, 1.0, 3.0, 0.0;
none;none;none :
n 3.0, 2.0, 2.0, 2.0;
none;none;none :
n 0.0, 4.0, 5.0, 3.0;
none;none;none :
n 0.0, 0.0, 1.0, 1.0;
none;none;none :
n 0.0, 0.0, 0.0, 2.0;
none;none;none :
none;none;none;none :
none;none;none;none :
none;none;none;none :
none;none;none;none :
none;none;none;none :
none;none;none;none :
none;none;none;none :
none;none;none;none
end
```

```
s(4,17)
1,1,1,1 :
1,1,1,1 :
1,1,1,1 :
1,1,1,1 :
1,1,1,1 :
1,1,1,1 :
0,1,1,1 :
0,0,1,1 :
0,0,0,1 :
1,1,1,1 :
1,1,1,0 :
1,1,0,0 :
1,0,0,0 :
0,0,0,0 :
0,0,0,0 :
0,0,0,0 :
0,0,0,0
end
```

2. LU-DECOMPOSITION

```
p(4,39)
{ hex LU }
data ,0,4; rep(3) null,0,0:
null,0,0; data,0,2; rep(2) null,0,0:
rep(2) null,0,0; data ,0,2; null ,0,0:
rep(3) null,0,0; data ,0,2 :
{ switch on cells }
rep(4) data n e s w,3,0 :
rep(4) data n e s w,2,0 :
rep(4) data n e s w,1,0 :
rep(4) data n e s w,0,5 :
{ start computation }
rep(4) data n e s w,0,5 :
rep(4) data n e s w,0,5 :
rep(4) data n e s w,0,5 :
rep(4) data n e s w,0,5 :
rep(4) data n e s w,0,5 :
rep(4) data n e s w,0,5 :
rep(4) data n e s w,0,5 :
rep(4) data n e s w,0,5 :
rep(4) data n e s w,0,5 :
rep(4) data n e s w,0,5 :
rep(4) data n e s w,0,5 :
rep(4) data n e s w,0,5 :
rep(4) data n e s w,0,5 :
rep(4) data n e s w,0,5 :
rep(4) data n e s w,0,5 :
rep(4) data n e s w,0,5 :
rep(4) data n e s w,0,5 :
rep(4) data n e s w,0,5 :
rep(4) data n e s w,0,5 :
rep(4) data n e s w,0,5 :
rep(4) data n e s w,0,5 :
rep(4) data n e s w,0,5 :
rep(4) data n e s w,0,5 :
rep(4) data n e s w,0,5 :
rep(4) data n e s w,0,5 :
rep(4) data n e s w,0,5 :
rep(4) data n e s w,0,5 :
rep(4) data n e s w,0,5 :
rep(4) data n e s w,0,5 :
rep(4) data n e s w,0,5 :
rep(4) data n e s w,0,5
end
```

```
s(4,39)
1,0,0,0 :
1,1,0,0 :
1,1,1,0 :
1,1,1,1 :
1,1,1,1 :
1,1,1,1 :
1,1,1,1 :
1,1,1,1 :
1,1,1,1 :
1,1,1,1 :
1,1,1,1 :
1,1,1,1 :
1,1,1,1 :
1,1,1,1 :
1,1,1,1 :
1,1,1,1 :
1,1,1,1 :
1,1,1,1 :
1,1,1,1 :
1,1,1,1 :
1,1,1,1 :
1,1,1,1 :
1,1,1,1 :
1,1,1,1 :
1,1,1,1 :
1,1,1,1 :
1,1,1,1 :
1,1,1,1 :
1,1,1,1 :
1,1,1,1 :
1,1,1,1 :
1,1,1,1 :
1,1,1,1 :
1,1,1,1 :
1,1,1,1 :
1,1,1,1 :
1,1,1,1 :
1,1,1,1 :
1,1,1,1
end
```

```
d(4,39)
none;none;none;none :
none;none;none;none :
none;none;none;none :
none;none;none;none :
none;none;none;none :
none;none;none;none :
none;none;none;none :
none; e 0.0,0.0,0.0,2.0 ; none; none :
none;none;s 0.0,0.0,0.0,0.0; none :
none;none;none;none :
none;none;s 0.0,0.0,0.0,4.0; none :
none; e 0.0,0.0,3.0,0.0; none; none :
none;none;none;none :
none; e 0.0,0.0,0.0,1.0; none; none :
none;none;s 0.0,0.0,2.0,0.0; none :
none; e 0.0,3.0,0.0,0.0; none;none :
none;none;s 0.0,0.0,0.0,2.0; none :
none; e 0.0,0.0,2.0,0.0; none;none :
none;none;s 0.0,3.0,0.0,0.0;none :
none; e 2.0,0.0,0.0,5.0; none;none :
none;none;s 0.0,0.0,4.0,0.0;none :
none; e 0.0,3.0,0.0,0.0; none;none :
none;none;s 0.0,0.0,0.0,1.0;none :
none;e 0.0,0.0,1.0,0.0;none;none:
none;none;none;none :
none; e 0.0,0.0,0.0,2.0; none;none :
none;none;none;none :
none;none;none;none :
none;none;none;none :
none;none;none;none :
none;none;none;none :
none;none;none;none :
none;none;none;none :
none;none;none;none :
none;none;none;none :
none;none;none;none :
none;none;none;none :
none;none;none;none :
none;none;none;none
end
```

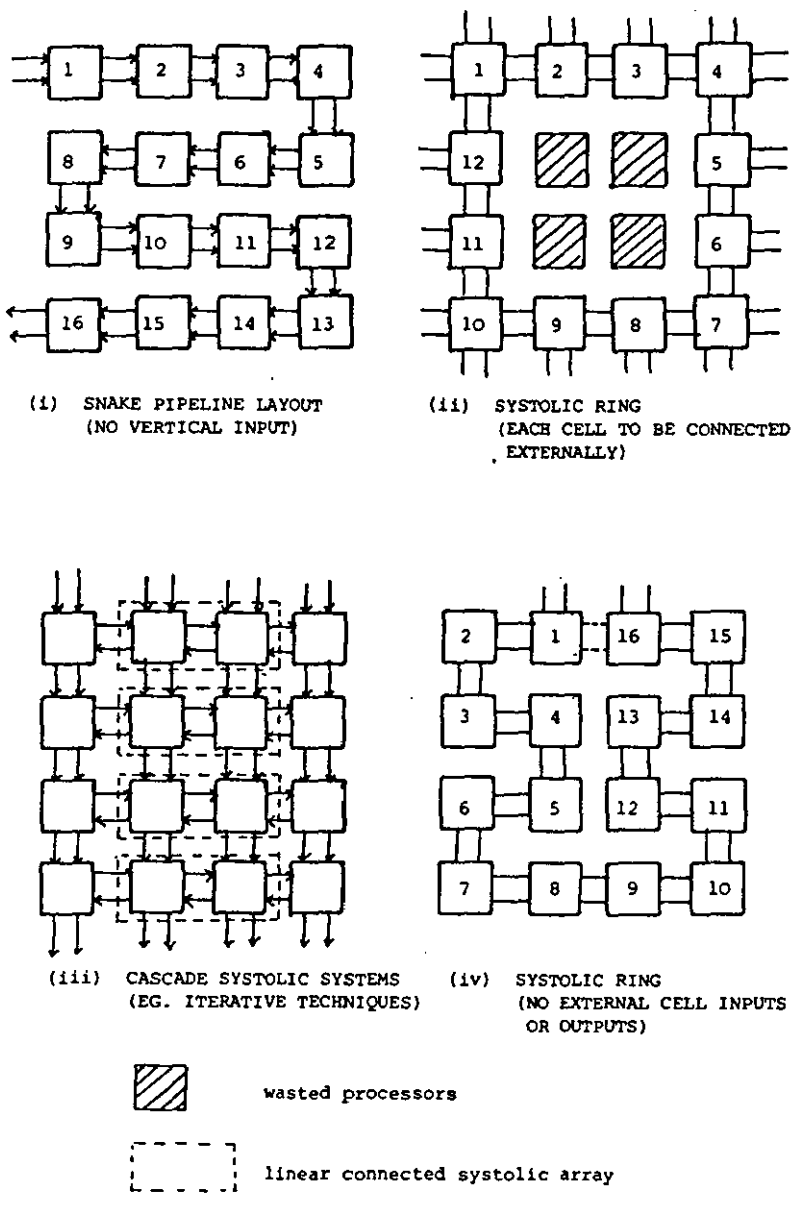


FIGURE 9.4.7: Configuration of arrays on ISA to improve processor utilisation

simulated on the ISA more efficiently than SIMPLE and PARTIAL SIMD programs. For many problems where communication links map directly onto the ISA grid data flow and the number of steps can be preserved. The overall computation time of the array is increased when extra steps are added to multiplex data for more complex communication patterns. Simulations themselves require a setup phase where the systolic array is configured and started on the ISA grid using a Dynamic Instruction Modification (DIM) program. DIM allows an instruction inside the ISA grid to be modified as it passes through a processor. The modifications are predetermined by a virtual processing PE plugged into the ISA grid at run time. DIM offers an additional flexibility to the ISA, by allowing portions of the grid to be allocated to different arrays which can run simultaneously on the grid, or to reconfigure the systolic array 'on the fly'.

9.5 THE LINEAR INSTRUCTION SYSTOLIC ARRAY (LISA)

We now return to the spooler problem of expanding the input and condensing the output interface of the ISA. The method of plugs which establishes a hierarchical many-1 correspondence between virtual processors and the real grid processors produces fanin and fanout spooler trees and is rather naive. The execution time of simulation will increase as the size of the plug and hence number of virtual PE's that are simulated by a single real processor increases. To solve virtual mapping problem we apply the technique of Yang and Lee [86] used to transform a wavefront processor and architecture into a 1-D (i.e. linear) array. This mapping was applied only to single wavefront algorithms involving no backtracking (i.e. Huygen's principle) such that

at time cycle i, only the PE's on the diagonal w(i) are activated (see Fig.9.5.1). We extend Yang and Lee's technique to multiple

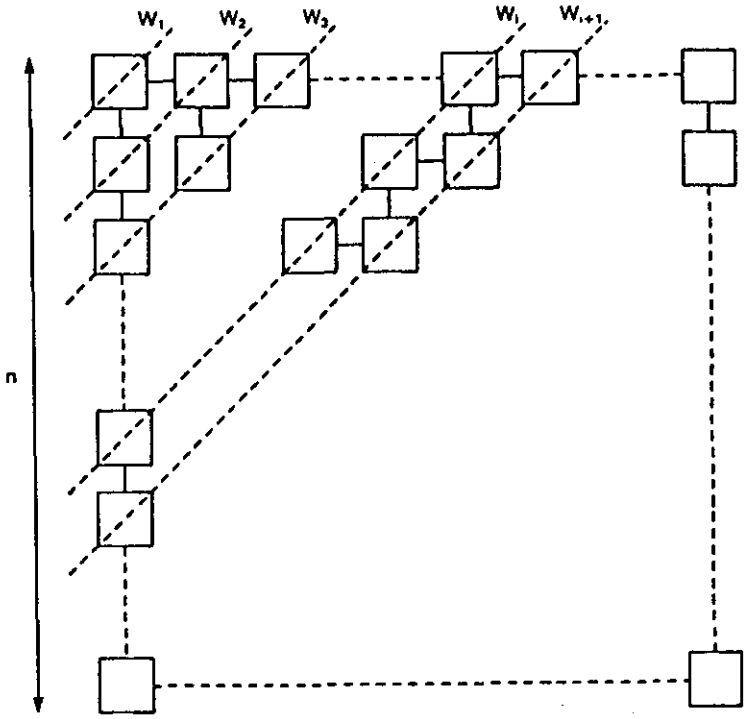


FIGURE 9.5.1: Wavefront array processor

wavefront algorithms involving no backtracking using the ISA. This forms the basis of a more general mapping technique of ISA_n programs to linear systolic arrays. Clearly the existance of such a mapping can be used to simulate plug definitions by linear arrays, hence reducing the overall number of virtual and real processors in the simulator hopefully minimising simulation time.

The mapping proceeds in two phases as follows:

a) convert the wavefront processor to an equivalent ISA

b) k-slow the ISA program mapping the ISA onto a linear array.

First consider a single wavefront algorithm. At any time t there is at most one active wavefront on the wavefront processor. The single

wavefront algorithm is simply encoded as an ISA program with only one instruction triple followed by 2n-1 no-op instructions pushing the wavefront across the processor mesh. That is,

$$p^{(1)} = \langle I_1, I_1, \ldots, I_1 \rangle \qquad s^{(1)} = \langle 1,1,\ldots,1 \rangle$$

$$p^{(2)} = \langle c, c, \ldots, c \rangle \qquad s^{(2)} = \langle 1,1,\ldots,1 \rangle$$

$$p^{(2+i)} = \langle 0,0,\ldots,0 \rangle \qquad s^{(2+i)} = \langle 1,1,\ldots,1 \rangle, \quad i=1(1)2n-1.$$

The progression of the single wavefront pattern is shown at times t=1,4,8 for n=4 in Fig.(9.5.2b). Three more inputs are required to push $p^{(3)}$ off the array and terminate the program under normal ISA rules. To apply Yang & Lee's mapping technique the ISA must be converted to a slowed or Delayed Wavefront Program (DWP).

The DWP is an ISA program which ensures that only a single wavefront is active at t=1,4,8 (Fig.(9.5.2b)). This implies that processors behind the wave must execute a no-op instruction or be de-selected for 2n-1 cycles between each instruction making the DWP extremely large. Next we convert the ISA into a Linear Instruction Systolic Array (LISA) by defining a new processor structure or LISA cell. Fig.(9.5.2a) illustrates the ISA→LISA conversion. For the simulation of an n×n, $N=n^2$ 2-D ISA we use just n-LISA cells connected in a pipeline as shown in Fig.(9.5.3) .

Each LISA cell contains 4 bi-directional input/output lines for data and a select line passing horizontally through the cell from west to east, and an instruction line north to south. It follows that:

(1) LISA(i) corresponds to the ith row of the original 2-D ISA array.

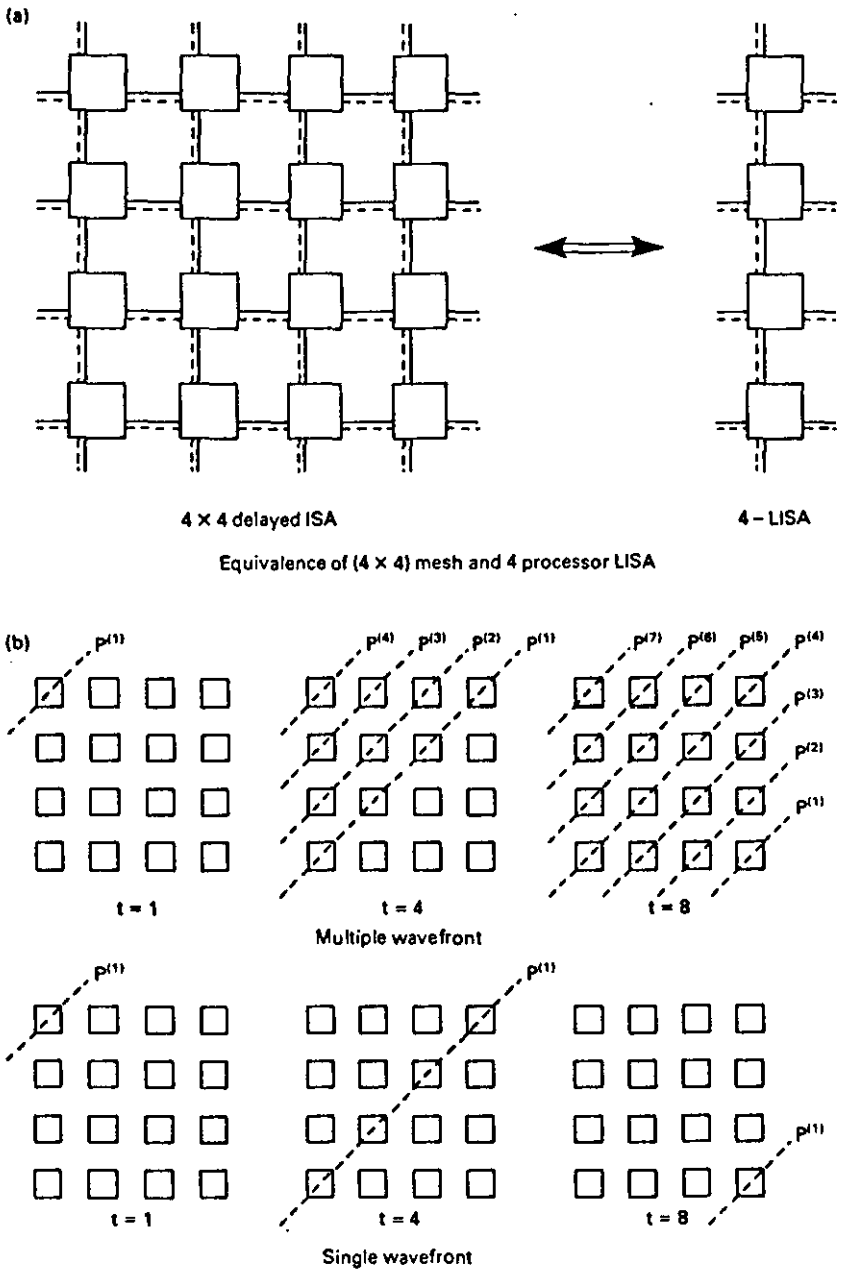


FIGURE 9.5.2: ISA to LISA mapping

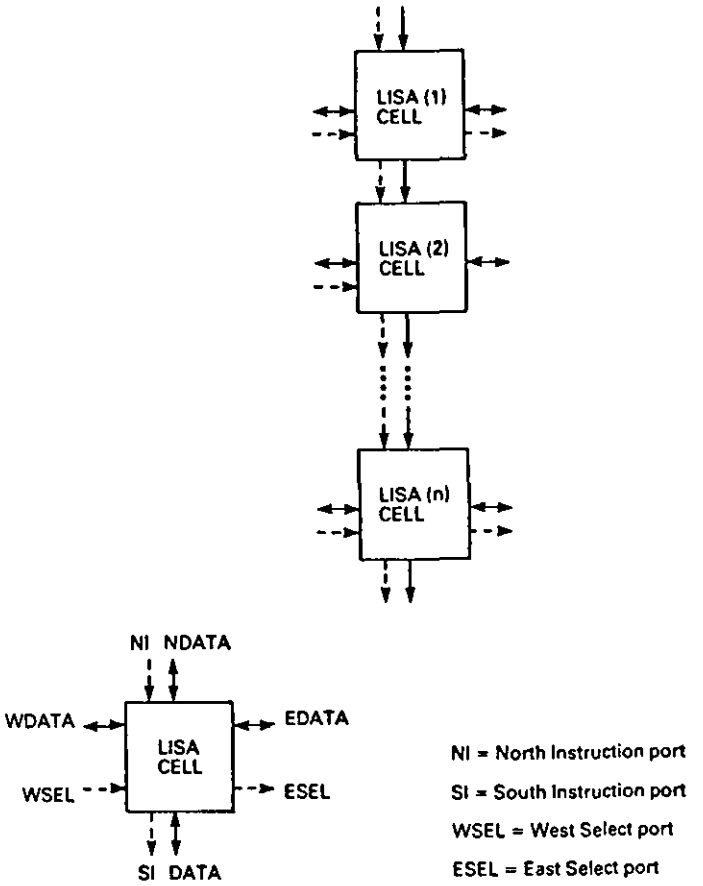


FIGURE 9.5.3: Linear instruction systolic array

(2) If LISA(i) acts as the (i,j)th ISA processor then LISA(i-1) and LISA(i+1) act as the (i-1,j+1)th and (i+1,j-1)th ISA processor respectively.

(3) If LISA(i) acts as the (i,j)th processor of ISA at time t, then at time t+1 the same processor acts as the (i,j+1)th ISA processor.

(4) Since each ISA cell can communicate in four directions we have to consider communication problems in the linear array. The north direction is omitted because the processors behind

the wavefront execute no-ops. Since LISA(i) simulates row i of the ISA east and west data is easily obtained. This leaves communication south, as LISA(i) acts as (i,j)th ISA processor at time t the data moved to LISA(i+1) at time t+1 is the data required by ISA processor (i+1,j).

Considering these factors the following cell is produced,

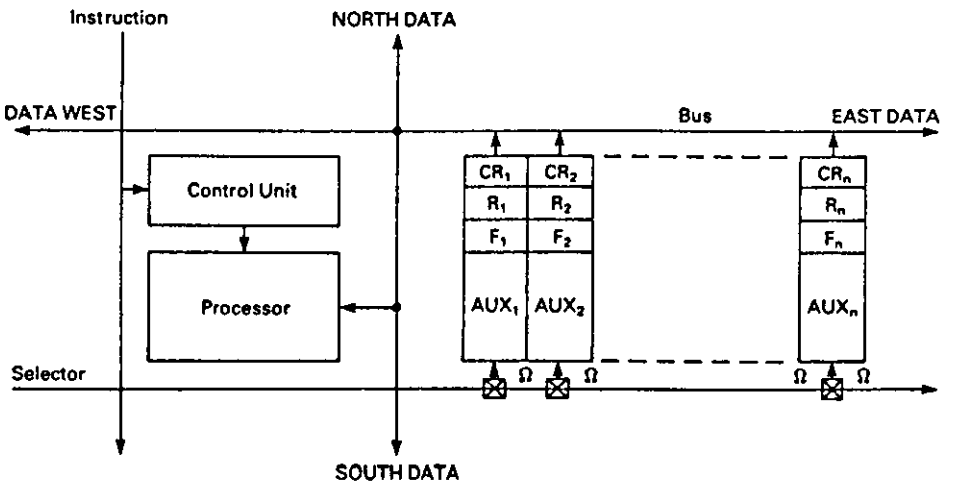


FIGURE 9.5.4: General LISA cell n×n mesh

Each cell contains sufficient memory for each cell in a row of the ISA array, and,

CR_j = communication register of jth cell in the row

R_j = computation save register of jth cell in the row

F_j = flag of jth cell in row indicating overwrite of CR_j

AUX_j = auxiliary memory for storing extra data and book-keeping.

It follows that only one section of the memory (CR_i,R_i,F_i,AUX_i) can be used at a particular time for a single wavefront algorithm, hence bus selection of memory is performed by the delay of the selector signal pumped through the array. The instruction signal enables the communication directions.

The resulting $LISA_n$ array is much more area efficient as only n control and processor units are required in contrast to the n^2 used by the ISA_n. Observe that memory requirements remain the same.

Returning now to the mapping problem it is trivial to see that the single wavefront ISA_n program is simulated by $LISA_n$, leaving only the problem of placing the next wavefront of a DWP onto the array while it retains the same values. Clearly at time t+1=n+1 the wavefront has passed the anti-diagonal of ISA PE's consequently LISA(1) of the $LISA_n$ array must be free. So rather than waiting for the current wavefront to leave the array altogether we can start the next command $p^{(2)}$ (the copy command) after another (n+1) cycles we can enter $p^{(3)}$ etc.

Now the DWP is produced easily from consideration of the LISA dataflow and the instruction/selector paths to produce the format in Fig.(9.5.5).

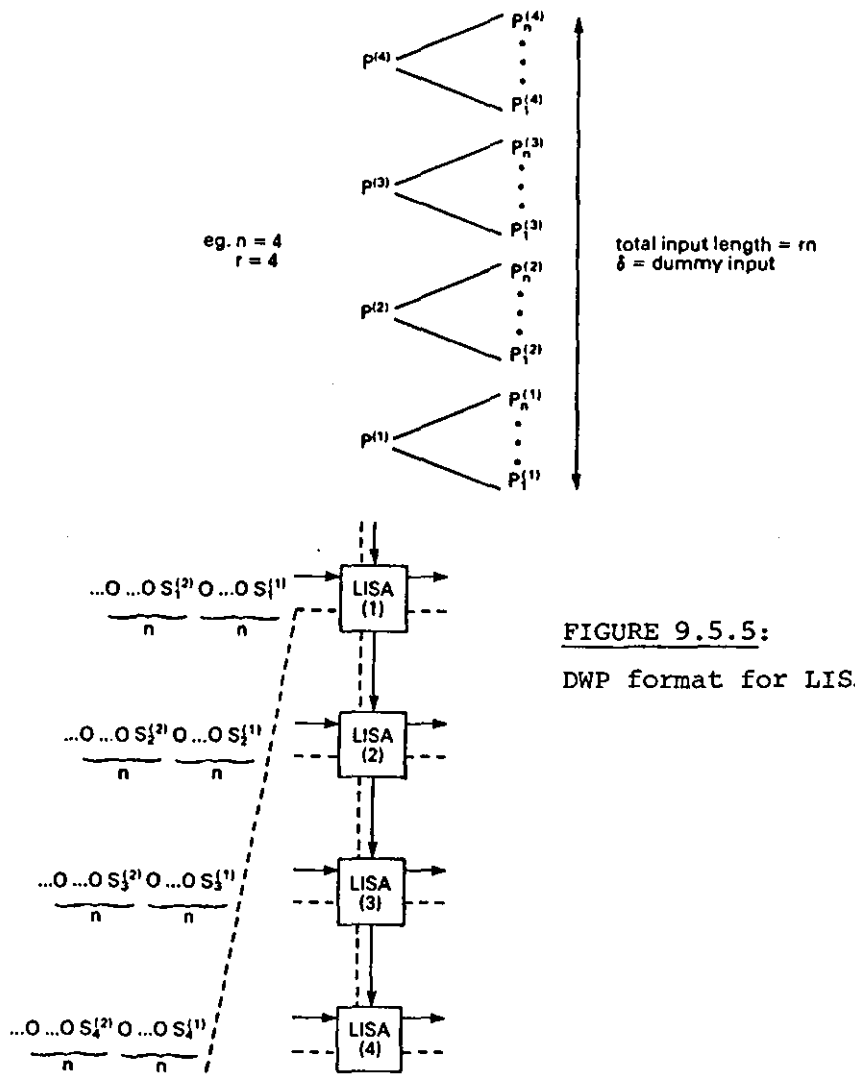
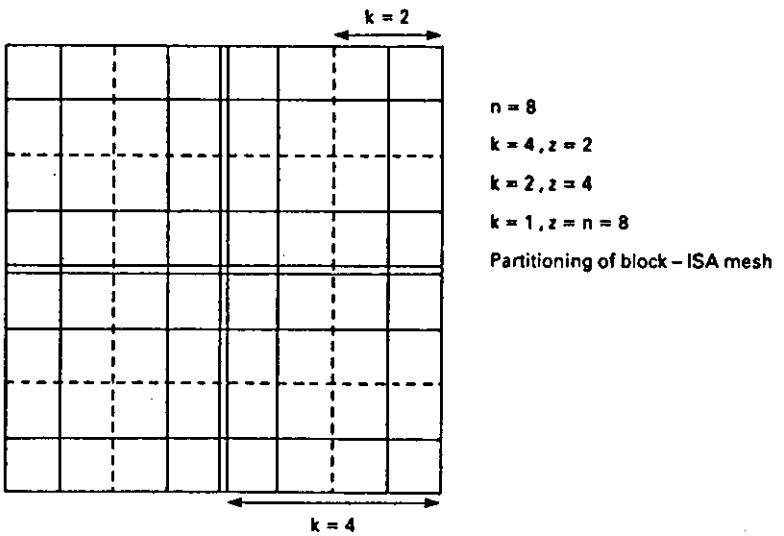


FIGURE 9.5.5:
DWP format for LISA pipeline

illustrating that any ISA program can be translated into a DWP for the LISA pipeline. Clearly if we have a large mesh size n and a long program (r) the DWP becomes very long and simulation slow making the method impractical. From a theoretical viewpoint an additional transformation applied to ISA results of Table (9.1.1) produces a timing for LISA by substituting nr for r or nT(p) for the length of a program p on an ISA_n. Also observe that the size of a LISA cell is dependent upon n the mesh size for defining the amount of simulation storage.

A flexible and expandable ISA architecture can be constructed by using LISA building blocks. Considering the n×n ISA_n mesh as a matrix of PE's we choose a block size k and partition the mesh into k×k blocks as follows:



The essential idea being to maximise the block size and minimise area by trading the number of additional instructions to the DWP, with the size of the LISA pipeline. Notice that the number of vertical LISA inputs is independent of the block size while the horizontal (selector) lines are related to k, an additional design constraint. In the example above choosing k=4 produces an ISA_n with four $LISA_4$ pipelines simulating a N=64 processor mesh. The choice of k=4 is arbitrary, but based on restricting the simulation storage of each LISA cell, while removing 3/4

of the processor and control units of the original grid. Furthermore we would like to retain the possibilities of chip based LISA implementations.

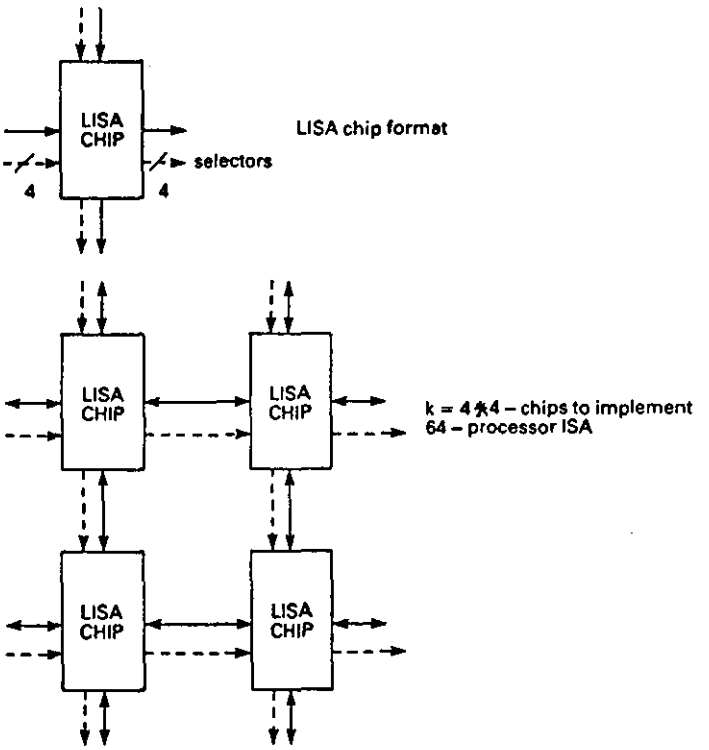


FIGURE 9.5.6: LISA-based construction of ISA mesh

For a single LISA pipe we use only one data input and one data output line horizontally and vertically. The immediate objection to this policy is that perhaps more than one PE will communicate across a block boundary at the same time. Simple analysis shows that at most two ISA processors can be active across horizontal or vertical block boundaries in the same instance. Furthermore they are always in adjacent rows or columns and can be easily resolved by an additional delay in the DWP. Fig.(9.5.7) illustrates the DWP data movement, and it is trivially observed that the maximum delay between instructions is dependent on the block size k. Any program on the ISA has an equivalent DWP on the block ISA which is at most k times as long as the ISA program, or k-slowed. For example, the instruction $p^{(1)}=(p_1^{(1)},p_2^{(1)},\ldots,p_8^{(1)})$ is partitioned into n/k, k sized components and input into the block partition ISA as illustrated by Fig.(9.5.8).

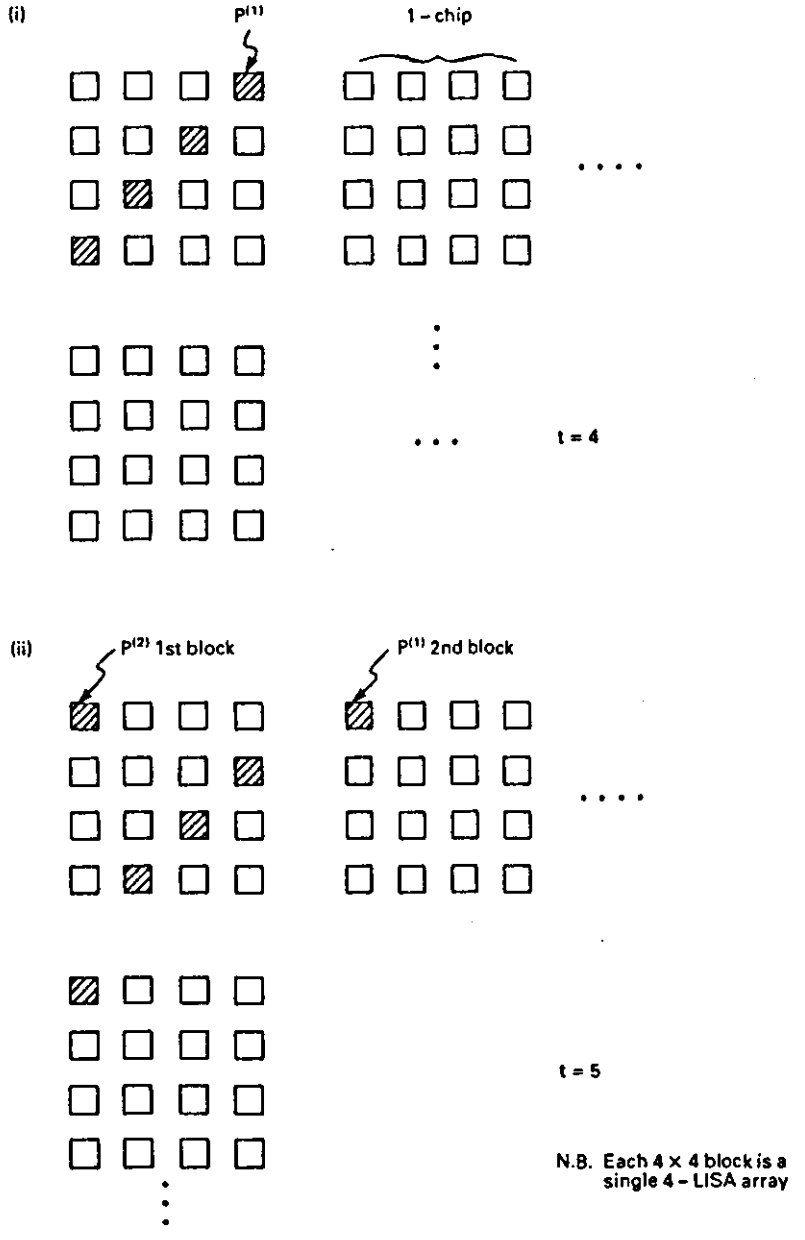


FIGURE 9.5.7: Block - ISA wavefronts

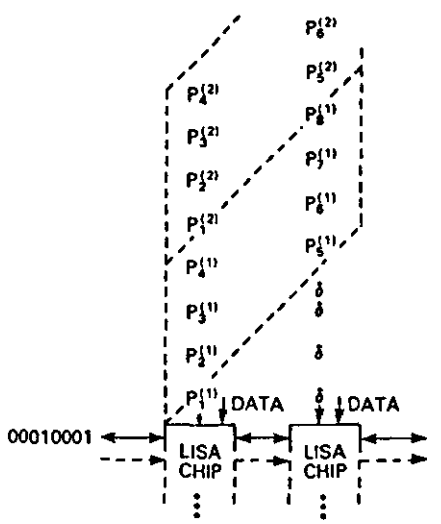


FIGURE 9.5.8: Instruction input format for block ISA

It follows that the LISA design can be used in two roles for our simulator,

(i) As a hardware component to minimise the total number of real processors required to simulate the full ISA grid.

(ii) As a software plug which minimises the additional simulation time by running a collection of virtual PE's on a single real PE.

It should be clear that a k=4 block partitioning corresponds to a 2×2 matrix or $LISA_4$ which act as 16-4 plugs.

9.6 SUMMARY

In this chapter we have described and characterized the instruction systolic array a flexible and powerful parallel architecture, capable of simulating full, partial and vector orientated SIMD programs, the wave-front array processor of S.Y. Kung and many dedicated systolic arrays. A number of theorems were presented relating program size and complexity of MIMD-type mesh machine algorithms and the ISA by a series of program transformations.

Using this flexible architecture as a virtual machine programmed in OCCAM we developed a soft-systolic simulator where the emphasis was on executing programs systolically rather than systolic movement of data. An overall system structure was defined and the virtual machine discussed in detail. A primitive assembler/compiler for a special language the Replicating Instruction Systolic Array Language (RISAL) was devised for experimentation with the machine.

Using RISAL a number of simple test examples were constructed suggesting extensions to the language and highlighting potential problems

with the structure of the virtual machine. In particular the difficulty of expanding the machines interface from the limited host machine bandwidth using a spooler arrangement.

We indicated that the virtual ISA could be used directly for multi-tasking simulation using vector orientated programs and a systolic array simulation (Umeo [85]). For dedicated systolic arrays like the matrix triangulariser (Gentleman & H.T. Kung [81]), and the hexagonal LU-decomposition array (Leiserson [81]) where cells consisted of multiple types (including single cell type arrays as a special case), the method of Dynamic Instruction Modification (DIM) was introduced. A DIM ISA program allowed instructions to be modified as they were pumped systolically through the grid permitting the setup and stationary operation of instructions forming dedicated arrays on the ISA grid. The above examples were tested using RISAL programs.

Finally we considered a further program transformation from ISA to a Linear ISA (LISA) a 1-D array of processors. The essential idea was to minimise hardware by arranging for only one ISA program wavefront to be active on the mesh at a time, using a delayed wavefront program (DWP). The cost of additional delay instruction wavefronts produced large simulation programs but was offset by block partitioning of the virtual grid. For k×k blocks a k-cell LISA was used to simulate each block, and a k-slowed DWP equivalent to an ISA program produced. For k=4, 4×4 block partitioning saved 75% of the processors (excluding their internal memory) and introduced only a constant delay factor over the ISA for program execution.

To overcome the host/array interface spooling problem and to achieve the mapping of large virtual grids to a finite real machine architecture

the concept of processor plugs was introduced, using a many-1 mapping of portions of the ISA to a single PE. The nested use of plugs created fanin and fanout spooling trees for distributing instructions from a sequential host to the parallel ISA. Unfortunately, for large virtual grids simulation time increased as more plugs were introduced to simulate portions of the virtual grid. Finally we swapped the plugs for the LISA design to produce efficient grid mappings, providing the capability to simulate any (reasonably) sized virtual ISA.

We conclude that the principles discussed briefly here can form the basis for a soft-systolic simulator using an orthogonal connected mesh of processors. The wide range of algorithms which the ISA can simulate make it suitable for a virtual simulating grid, while the use of RISAL, DIM and DWP permit the implementation testing of ISA programs and dedicated systolic arrays (with regular communication structures). Future work would include the implementation of the ISA program on a fixed network of transputers (possibly a meiko computing surface) and the much needed development of a robust version of RISAL.

CHAPTER 10

CONCLUSIONS AND SUGGESTIONS FOR FURTHER WORK

"There is nothing so powerful as an idea whose time has come"

DORIS GRANT.

This thesis has concentrated on the introduction of the soft-systolic paradigm for the development of novel systolic algorithms, resulting from the relaxation of constraints imposed on array designers by VLSI technology. The algorithms and concepts presented have been discussed in a semi-formal manner which partitions neatly into three distinct but related parts. In this final chapter, rather than enumerating the results, advantages and disadvantages of each design, we take the opportunity to stand back from detailed descriptions, and characterise the major properties of these parts and the relationships between them.

PART I consisted mainly of survey and background (mathematical) material necessary for the ready comprehension of the thesis and provided a stock of accepted designs to act as benchmarks for new arrays. In addition, the concept of a systolic frame was introduced. Systolic frames are intended to lend a semi-formal structure consisting of axioms and heuristics to characterise classes of systolic designs and their interrelationships. A single algorithm/array is a single element of a frame, whose abstract computational capabilities are characterised by the axioms and its implementation by technology based heuristics. Different elements (designs) of a frame are imagined as different arrays/algorithms produced by re-timing, re-placement or synthesis operations. A subframe is simply a collection of designs over the same axioms and heuristics which solve the same problem (and are clearly a subset of a more general frame). Closed subframes follow naturally as collections of designs for the same problem, such that applying re-timing, replacement or synthesis techniques to a particular design produces another design in the same subframe. Additional frame types can also

be identified such as an anchored subframe (where at least one design is formally verified as correct), free or constrained frames (where heuristics are omitted or defined respectively), and finally regular or irregular frames (where axioms dictate the degree of structure in the connection topology). For our purposes we classified systolic frames into three main groups, soft-, hybrid- and hard-systolic according to a single heuristic 'programmability'. Consequently all the designs presented in this thesis are members of constrained frames with definition soft, hybrid, or hard prefixed to determine the degree of programming required to implement the design; soft-systolic algorithms requiring simulation (or total programming), hybrid a mixture of micro-programming and special hardware, and hard-systolic implicitly programmed by circuit connections.

Recent trends in the literature towards formal transformational approaches to deriving systolic algorithms indicate that the days of ad-hoc design are numbered. From this viewpoint we believe that the formal definition of the systolic frame concept will play an increasing role in determining classes of systolic algorithms and legal transformations between them. In particular relationships between soft, hybrid and hard frames must be explored to determine if simulated arrays map easily into real implementations. If nothing else this thesis has tested the viability of adopting these more flexible (but controlled) attitudes to defining new systolic schemes. Adequately defining the frame concept in itself is non-trivial and represents a sizeable amount of further research into systolic algorithm properties, we have contented ourselves with a less formal outline of frames by which investigations could be directed.

PART II concentrated on improvements to linear algebra arrays, using a soft-systolic frame under 3-D (flexible layout) and optical processing (long wire) heuristics. Improvements to existing designs as well as new arrays were introduced and theorems about them presented which compared favourably with corresponding results for traditional architectures of Chapter 3. The essential point to observe is that all the designs are members of linear algebra subframes with new arrays produced in three main ways:

1. Changing the length of input data streams: by
   a) filling dummy or neutral elements
      (e.g. double pipes, and problem interleaving)
   b) modifying the host/array interface
      (e.g. block partitioning, D^z-pipes ($z>0$)).
2. Re-organising the underlying array structure: by
   a) Partitioning the problem or array into loosely coupled or decoupled subsections which can run sequentially or in parallel using multipass or multilayer configurations.
   b) Modifying internal cell structures to improve array efficiency (i.e. multi-layer tree layouts, two-level pipelining, block-ips arrangements).
3. Deriving new arrays from new algorithms:
   a) Increasing the parallelism of the solution technique (Rank-annihilation, circulant system solvers (BATS)).
   b) Incomplete Arrays with minimal area and time providing fast approximations to a problem solution which are 'cleaned up' outside the array. (Preconditioning preprocessors).

1. and 2. correspond to the application of retiming and replacement to

existing designs to produce additional members of the subframe. Clearly the method is ad-hoc and exhaustive depending on the ability to permute data (e.g. QI-methods) or instructions (block partitioning, multi-pass etc.) Already there are signs that the rate of improvements to existing arrays is declining indicating that designs are settling to form a few standard arrangements. We suggest that these 'standard' arrays are elements of closed subframes, which by varying heuristics we have re-opened. Clearly, the amount of design possible by the sole use of techniques 1. and 2. is limited and the frame will eventually close again under the new constraints. On the other hand 3. is a more systematic approach based on improved algorithms arising from theoretical developments, and creates subframe elements directly without recourse to 1. and 2. Although once the element is established 1. and 2. can be applied to produce additional possibly improved designs. For example, this is what occurred with the rank annihilation Toeplitz solver, and the BATS array using the Audish & Evans factorisation. In contrast, incomplete arrays result from the relaxation of axioms in a systolic frame and in particular the requirement that exact solutions (within the bounds of rounding errors) are produced by the array. This approach immediately circumvents the AT^2 (Savage [81]) lower bound argument on area and time of arrays used to determine optimum array designs, producing dramatic cell reductions (e.g. >75%) and decreased computation time. The essential feature of an incomplete array is the redistribution of computation between host and array, according to some percentage weighting. For example, a compacted area efficient and fast array may produce an approximation very close to the correct answer, which is then refined by the host to produce the exact solution. Thus emphasis is no

longer on full array calculation but accelerated accurate approximations to a problem, which allows a single array to be adopted for a range of problem instances. This is in contrast to a multipass scheme which satisfies the AT^2 bound and reduces area by iterating calculations, thus increasing computation time to supply an exact answer. We described incomplete schemes mainly for iterative solutions to linear systems in the form of preconditioning, where the fixing or refining method was clearly defined and supported by theoretical arguments. Clearly incomplete techniques should be extended to other methods and arrays. We might also conjecture that as the number of iterations associated with good preconditioners continues to drop, that compacted and fast incomplete arrays may outperform direct (complete) designs from the viewpoint of economic viability, where the solution of large linear systems is concerned.

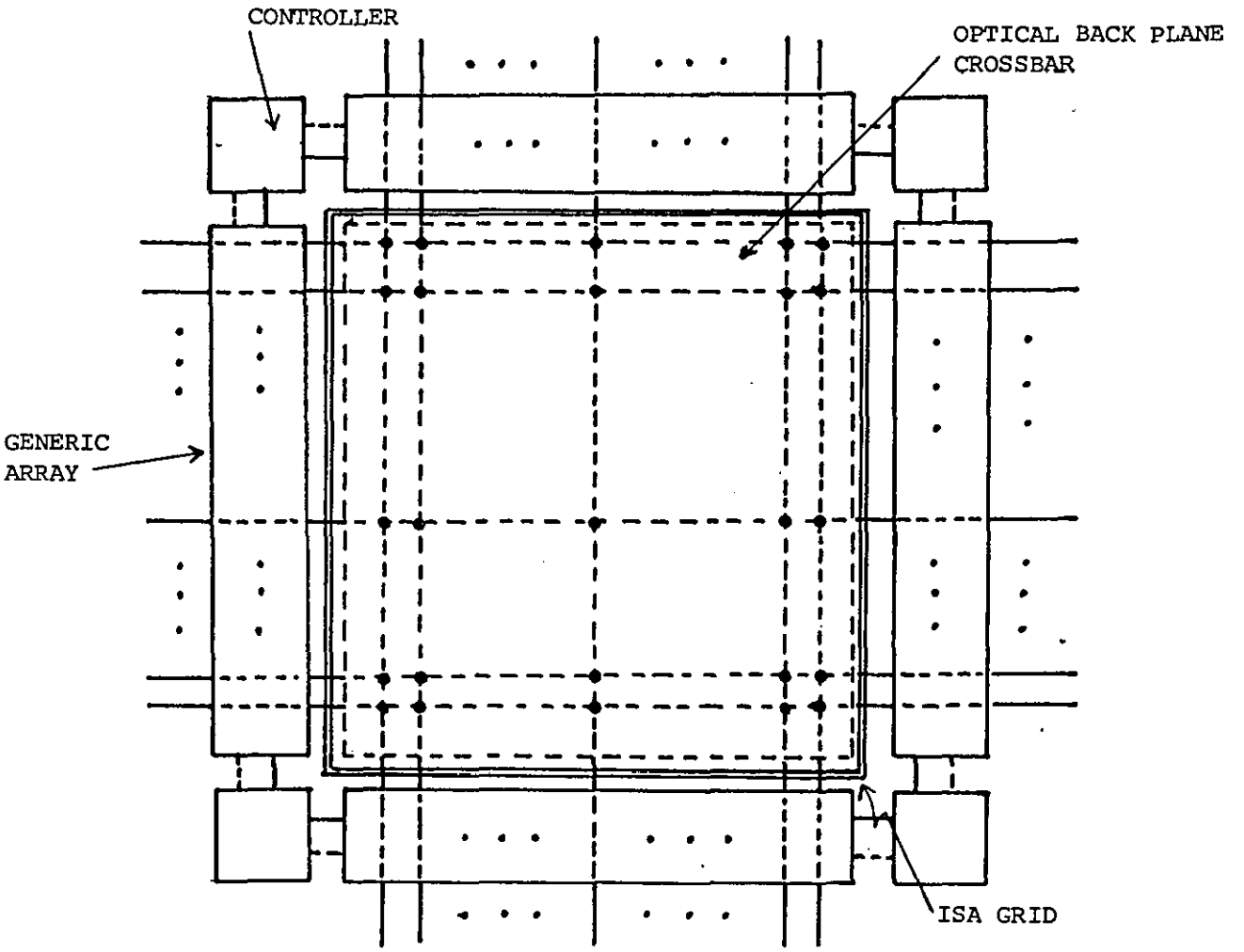
The purpose of PART III of the thesis was twofold, but was chiefly about producing generic arrays which trade-off speed vs. power, speed vs. area, functionality vs. area, memory tradeoffs and the number of external connections. As a secondary issue to produce examples and support discussions we considered the use of computational molecules or rules for deriving systolic forms using an alternative recurrent formulation to that explicit in linear algebra. Indeed, we characterized by means of templates,arrays for generating and manipulating tables of elements, creating different arrays based on row-by-row or column-by-column and combined non-stationary schemes (e.g. QD-array). These can be likened to the row, column or diagonal ordering of matrix inputs in traditional linear algebra schemes. In the guise of algorithmic vs. geometric interpretations of P.D.E. problems we implicitly defined relations

between linear algebra arrays and computational rule (or Table) arrays and indicated that a wide class of useful new arrays existed. A number of new table based designs were discussed, including the more complex simplex, revised simplex, and assignment problem which graphically illustrated the difficulties of manipulating large tables. This problem was not unlike the dense matrix problems encountered with arrays based on linear algebra.

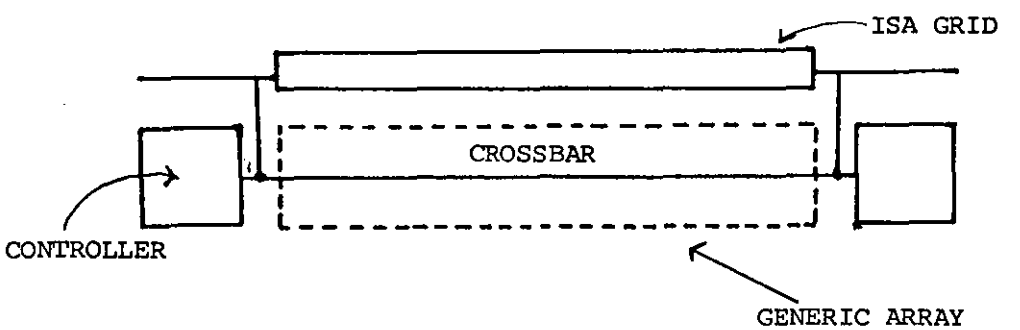
From the viewpoint of systolic frames the emphasis in PART III shifted from identifying particular subframe elements and instead addressed the relations between collections of similar designs which formed similar sub-frames belonging to different global frames. That is, the correspondences between designs for the same problems in soft, hybrid and hard systolic frames. To meet the demanding tradeoffs above two opposing forces are readily observed. In one direction many simple hard designs map to a single generic array increasing the programmability factor of a design shifting it from hard, through hybrid to soft-systolic subframes. In the other direction restricting the generality of a problem constrains the array, mapping it from soft, to hybrid, and hard subframes in a one to many transformation increasing the options for implementation. We attempted to cancel these two forces by the principle of array unification producing a hybrid type device which attempted to minimise cell hardware whilst increasing programmability using a set of simple control switches. In the Unified Systolic Array for Differencing (USAD) we achieved this balance although this was a result of the simple structure of the problems selected. A similar balance with less success was developed for the systolic marching and Alternating Group Explicit (AGE) arrays. Here the optimisation of

hardware and increased programmability by use of switches was offset by the complexity of the computational rule employed (and its coefficients) which complicated cells particularly when alternating was necessary to retain stability. When unbalanced the two opposite forces migrate designs to one extreme or the other in terms of programmability. For generic arrays ultimately all hard, hybrid designs collapse onto a single unified array. Such a case is illustrated by the soft-systolic simulator in which the highest factor of programmability where arrays are simulated by programs executed systolically is achieved. For hard systolic designs a generic array maps onto a single well-defined and restricted special array. This was illustrated by the mapping of the generic P.D.E. solver down to the bit serial Hopscotch scheme. In this case even the solution method had to be modified along the way, to achieve a simple cell form with a low programmability factor. The mechanics of these shifts between soft, hybrid, and hard frames is further complicated by the collusion of improvement methods like problem interleaving and incomplete techniques identified in PART II. Indeed the PDE solvers adopted interleaving for asymmetric marching, while FAST AGE and HOPSCOTCH arrays used incomplete techniques to balance partial table production and approximation accuracy against cell area and array speedup.

To conclude, the lesson to be learnt from these observations is that a successful general purpose systolic computer must rely on a balance between the two extremes - implying a hybrid-systolic structure with limited programmability supported by specialist hardware. This makes it promising to investigate new MIMD architectures which incorporate VLSI structures into architectural design perhaps orientating them to particular problem domains. Based on the work in this thesis such a



a) Systolic Control Ring Architecture



b) Sideview of back plane organisation

FIGURE 10.1: SCRIP machine organisation

machine may take the form of a Systolic Control Ring Instruction Processor (or SCRIP) machine illustrated by Fig.(10.1). SCRIP incorporates an ISA_n mesh inside a Systolic Control Ring composed of simple controllers and generic linear arrays. Thus the ISA using RISAL programs can be adopted for general systolic simulation, the Control Ring to implement optimised multiple wavefront algorithms (like rank annihilation, Assignment problems, etc.) while the boundary arrays provide high performance computation of specified problem classes. A back plane crossbar is added to allow the flexibility of implementing toroidal networks and the independent use of boundary arrays and the ISA mesh, thus increasing parallelism and allowing a multi-user environment. Hence SCRIP is simply a 'bag' of useful arrays on a single architecture. The development of SCRIP type machines poses interesting and exciting problems for the future, and presents an attractive alternative to the SYS-PACK type machine of Chapter 1.

REFERENCES

ATHALE & LEE [84], *"Optical processing using outer product concepts"*, Proceedings IEEE, Vol. 72, No.7, July 84, pp.931-941.

AUDISH [81], *"The numerical solution of banded linear systems by generalized factorisation procedures"*, Ph.D. Thesis 1981, Loughborough University of Technology, (LUT).

AUDISH & EVANS [85a], *"On the parallel solution of certain circulant banded linear systems"*, Int.Jour.Comp.Math. 18 (1985), pp.83-90.

AUDISH & EVANS [85b], *"A parallel circulant linear solver"*, Comp.Stud. Report 238, L.U.T. 1985.

BARRODALE & ROBERTS [73], *"An improved algorithm for discrete L_1 linear approximation"*, SIAM J. Numer.Anal. Vol.10, No.5, Oct.1973.

BARRODALE & ROBERTS [78], *"An efficient algorithm for discrete L_1 linear approximation with linear constraints"*, SIAM J.Numer.Anal. Vol.15, No.3, June 1978, pp.603-611.

BERZINS, BUCKLEY & DEW [83], *"Systolic matrix iterative algorithms"*, Parallel Computing 83, pg.483-488, Eds. Feilmeier, Joubert & Schendel.

BOCKER [84], *"Algebraic operations performable with electro-optical engagement processors"*, SPIE, 1984, pp.212-220.

BRENNER & LOHMANN [86], *"The digital optical computing program at Erlangen"*, CONPAR 86 - Proceedings of Conf. on Algorithms & Hardware for Parallel Processing, Sept. 1986, pp.69-76. Springer Verlag Lecture Notes in Computer Science 237.

BRENT, KUNG, H.T., LUK [83], *"Some linear time algorithms for systolic arrays"*, CMA-R01-83 and invited paper 9th World Computer Congress Paris, 1983.

BRENT & LUK [84], *"A systolic array for the linear time solution of Toeplitz systems of equations"*, Jour. VLSI & Computer Systems, Vol. 1, Part 1 pp. 1-22 1984/5.

BRUDARU [85], *"Systolic algorithms to solve linear systems by iteration methods"*, Analele stiintifice ale universitatii, 'Al.I.Cuz.' din Iasi Timul XXXI S Ia, Matematica 1985, pp.301-306

BURDEN, FAIRES & REYNOLDS [81], *"Numerical Analysis"*, 2nd Edition, Prindle, Weber & Schmidt Publishers 1981.

BURLISCH & STOER [66], *"Numerical treatment of ordinary differential equations by extrapolation methods"*, Numerische Mathematik, Vol. 8, pp.1-13 1966.

CAULFIELD, GRUNINGER & CHENG [84], *"Using optical processors for linear algebra"*, SPIE, 1984, pp.190-196.

CAULFIELD, RHODES, FOSTER, & HORVITZ [81], *"Optical implementation of systolic array processing"*, Optics Communications Vol. 40, No. 2 1981, pp.86-90.

CHANDY & MISRA [86], *"Systolic algorithms as programs"*, Distributed Computing 1986, Vol. 1, pp .177-183.

CHEN [85], *"On the solution of a class of Toeplitz systems"*, Report yaleu/DCS/RR-417 Aug.1985 Dept.Comp.Stud. Yale Univ. New Haven.

CHVATAL [80], *"Linear programming"*, 1980, W.H. Freeman & Company Publishers.

CIEPIELEWSKI & HARIDI [84], *"Execution of bagof on the OR parallel Token machine"*, Proc. Int.Conf. on Fifth Generation Computer Systems 1984 ed. ICOT, pp. 551-560.

CODENOTTI [86], *"The matrix equation MX+XN=B in the VLSI model"*, Int.Jour.Comp.Math. 1986. Vol. 19, No. 1, pp.93-98.

DEW [84] *"VLSI architectures for problems in numerical computation"*, Workshop on Progress in the Use of Vector and Array Processors, eds. Paddon D.J. & Pryce, J.D., pp.1-24.

DEW, DODSWORTH, MORRIS [85], *"Systolic array architectures for high performance CAD/CAM workstations"*, NATO Advanced Study Institute on Fundamental Algorithms for Computer Graphics, 1985, pp.659-694.

DEW, MANNING, MCEVOY [86], *"A tutorial on systolic array architectures for high performance processors"*, 2nd Int.Electronic Image Week, Nice 1986 and Report 205, Leeds University.

EVANS, HADJIDIMOS & NOUSTOS [79], *"The parallel solution of banded linear equations by the new quadrant interlocking factorisation (QIF) method"*, Tech.Report No. 26 April 79, Dept. Maths. University Ioannina, Greece.

EVANS & HATZOPOULOS [79], *"A parallel linear system solver"*, Intern. J.Comp.Math. 1979, Section B, Vol. 7, pp.227-238.

EVANS & LIPITAKIS [79], *"On sparse LU factorization procedures for the solution of parabolic differential equations in three space dimensions"*, Intern.J.Comp.Math. 1979, Section B, Vol. 7, pp.315-338.

EVANS [80a], *"On the solution of certain symmetric quindiagonal linear systems"*, Intern.J.Comp.Math. Vol. 8, pg.271-284, 1980.

EVANS [80b], *"On the solution of certain Toeplitz tridiagonal linear systems"*, SIAM J.Numer.Anal., Vol. 17, No.5, Oct. 1980, pg.675-680.

EVANS & HADJIDIMOS [80], *"A modification of the quadrant interlocking factorisation parallel method"*, Intern.J. Comp.Math. 1980, Sect. B, Vol. 8, pp.149-166.

EVANS & OKOLIE [81], *"A recursive decoupling algorithm for solving banded linear systems"*, Intern.J.Comp.Math., Vol.10, pp.139-152.

EVANS & SOJOODI-HAGHIGHI [82], *"Parallel iterative methods for solving linear equations"*, Int.J.Comp.Math. 1982, Vol. 11, pp.247-284.

EVANS [83a], *"On the solution of certain Toeplitz quindiagonal linear systems"*, Intern.J.Comp.Math. 1983, Vol. 14, pp.305-324.

EVANS [83b], *"New parallel algorithms for partial differential equations"*, Parallel Computing 83, pp.3-56, eds. Feilmeier, Joubert, Schendel, Pub. North-Holland 84.

EVANS [83c], *"On preconditioned iterative methods for partial differential equations"*, Preconditioning Methods, Theory & Applications, ed. Evans, D.J., Gordon & Breach Publishers 1983.

EVANS [83d], *"Preconditioning methods, theory and applications"*, Gordon & Breach Publishers 1983 (General reading).

EVANS & LIPITAKIS [83], *"An approximate QIF method for parallel computers"*, in EVANS [83d].

EVANS & ABDULLAH [83a], *"A new explicit method for the solution of* $\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}$ *"*, Intern.J. Comp.Math. 1983, Vol. 14, pp.323-353.

EVANS & ABDULLAH [83b], *"Group explicit methods for parabolic equations"*, Intern.J.Comp.Math. 1983, Vol. 14, pp.73-105.

EVANS [85], *"Group explicit iterative methods for solving large linear systems"*, Intern.J.Comp.Math., Vol. 17, pp.81-108.

EVANS & MEGSON [85a], *"Romberg integration using systolic arrays"*, Intern. Report Comp.Stud. 250, Oct. 1985, L.U.T.

EVANS & MEGSON [85b], *"Construction of extrapolation tables by systolic arrays for solving ordinary differential equations"*, Internal report Comp.Stud. 252, Nov. 1985, L.U.T.

EVANS & MEGSON [85c], *"A systolic array for the quotient difference algorithm"*, Int.Report Comp.Stud. 254, Nov. 1985, L.U.T.

EVANS & AUDISH [86], *"The factorisation of constant circulant matrices into cyclic matrices"*, Int. Report Comp.Stud. 266, L.U.T.

EVANS & MEGSON [86a], *"Compact systolic arrays for incomplete factorization methods"*, Int. Report Comp.Stud. 321, Sept.86, L.U.T.

EVANS & MEGSON [86c], *"A highly pipelined systolic array for solving Toeplitz systems"*, Int.Report Comp.Stud. 332, Dec. 86, L.U.T.

EVANS & MEGSON [86d], *"A systolic norm generator"*, Int. Report Comp. Stud. 316, Sept. 1986, L.U.T.

EVANS & MEGSON [86e], *"Matrix inversion by systolic rank annihilation"*, Int. Report Comp.Stud. 295, July 86, L.U.T.

EVANS & MEGSON [86f], *"Systolic arrays for the modified quadrant interlocking factorisation (QIF) method for linear equations"*, Int. Report Comp.Stud. 286, Jun., 1986, L.U.T.

EVANS & MEGSON [86g], *"Hopscotch schemes for the solution of parabolic equations on area efficient systolic arrays"*, Int. Report Comp. Stud. 269 Mar. 1986, L.U.T.

EVANS & MEGSON [86h], *"Improving systolic array efficiency by block partitioning of matrices"*, Int. Report Comp.Stud. 271 Mar.86, L.U.T.

EVANS & MEGSON [86i], *"A systolic array for the FAST AGE (Alternating Group Explicit) method for parabolic equations"*, Int. Report 261, Feb. 1986, L.U.T.

FISHER, KUNG H.T., MONIER, WALKER, DOHI [83], *"Design of the PSC: a programmable systolic chip"*, Proc. of 3rd Caltech Conf. on VLSI, ed. Bryant R. Comp.Science Press, pp.287-302.

FISHER [84], *"Implementation issues for algorithmic VLSI processor arrays"*, Ph.D. Thesis, 1984, CMU, Pittsburgh.

FISHER, KUNG H.T., SCROCKY [84], *"Experience with the CMU programmable chip"*, CMU-CS-85-161. CMU, Pittsburgh.

FOSTER & KUNG H.T. [80], *"The design of special purpose VLSI chips"*, IEEE Computer 13(1), 26-40, Jan, 1980.

FRISON & QUINTON [85], *"An integrated systolic machine for speech recognition"*, VLSI: Algorithms and Architectures, Eds. P. Bertolazzi & F. Luccio, North Holland pg. 175-186, 1985.

FLYNN [72], *"Some computer organisations and their effectiveness"*, IEEE Trans. Comp. C-21, 9(1972), pg. 948-960.

GASS [69], *"Linear programming 3rd edition"*, McGraw-Hill Pub. 1969.

GENTLEMAN & KUNG H.T. [81], *"Matrix triangularization by systolic arrays"*, SPIE, Vol. 298, Real-time Signal Processing IV, 1981, pg.19-26.

GOODMAN, LEONBERGER, KUNG S.Y., RAVINDRA [84], *"Optical interconnections for VLSI systems"*, Proc. of IEEE Special Issue, Vol. 72, No. 7, pg. 850-866.

GOURLAY [70], *"Hopscotch, a fast second order partial differential equation solver"*, JIMA 6 (1970), pg. 375-390.

GUERRA [86], *"A unifying framework for systolic designs"*, Aegean Workshop on Computing July 1986, Springer Verlag Lecture Notes in Comp.Sci., No. 227 - VLSI Algorithms & Architectures, eds. Goos & Hartimanis, pg.46-56.

GUIBAS & LIANG [82], *"Systolic stacks, queues and counters"*, Conf. on Advanced Research in VLSI, MIT, Jan.1982, pg.155-164.

HAYNES, LAU, SIEWIOREK, MIZELL [82], *"A survey of highly parallel computing"*, Computer Jan. 1982, pg.9-23.

HEHNER & HOARE [83], *"A more complete model of communication processes"*, pg.105-120, Theoretical Comp.Sci. 26, 1983, pp.105-120.

HELLER [85], *"Partitioning big matrices for small systolic arrays"*, VLSI & Modern Signal Processing, pg.185-199, eds. Kung S.Y., Whitehouse, Kalaith, Prentice-Hall, 1985.

HELLER & IPSEN [82], *"Systolic networks for orthogonal equivalence transformations and their applications"*, Proc. Advanced Research in VLSI, pp.113-122.

HOARE [78], *"Communicating sequential processors"*, Comm.ACM, 21, pg. 666-677 (1978).

HOPCROFT & ULLMAN [79], *"Introduction to automata theory formal languages and computation"*, Addison Wesley, 1979.

HOROWITZ & SAHNI [78], *"Fundamentals of computer algorithms"*, Pitman Publishing Limited, 1978.

HSU, KUNG H.T., NISHIZAWA, SUSSMAN [84], *"LINC: the link and inter-connection chip"*, CMU-CS-84-159, Pittsburgh.

HUANG [84], *"Architectural considerations involved in the design of an optical digital computer"*, Proc. IEEE, Vol. 72, No. 7, July, 1984 pg.780-786.

HUANG & ABRAHAM [84], *"Algorithm based fault tolerance for matrix operations"*, IEEE Trans. on Computers, Vol. C-33, No.6, 1984, pg. 518-528.

HWANG & CHENG [82], *"Partitioned matrix algorithms for VLSI arithmetic systems"*, IEEE Trans. on Computers Vol. C-31, No.12, 1982, pg.1215-1220.

INMOS [84], *"OCCAM programming manual"*, Hoare, Series Editor, Prentice Hall, (International Series in Computer Science".

INMOS [85], *"Transputer reference manual"*, 1985, INMOS.

INMOS [86], *"The Transputer family"*, 1986, INMOS.

IPSEN [84], *"Stable matrix computations in VLSI"*, Ph.D. Thesis, 1984, Ann Arbor, Michigan.

JOHNSON & REISS [77], *"Numerical Analysis"*, Addison-Wesley 1977, Ch.5.

JOU & ABRAHAM [86], *"Fault tolerant matrix arithmetic and signal processing on highly concurrent computing structures"*, Proc. IEEE Vol. 74, No.5, May 1986, pg.732-741.

KATONA [82], *"String pattern matching algorithms for cellular processors"*, pg.431-436, Parallel Computing 83,eds. Feilmeier, Joubert, Schendel.

KATONA [85], *"A programming language for cellular processors"*, Parallel Computing 85, eds. Feilmeier, Joubert & Schendel, pg.371-376.

KOREN & PRADHAN [86], *"Yield and performance enhancement through redundancy in VLSI and WSI multiprocessor systems"*, Proc. IEEE, Vol. 74, No.5, May 1986, pg.699-711.

KUNDE, LANG, SCHIMMLER, SCHMECK, SCHRODER [85], *"The instruction systolic array and its relation to other models of parallel computers"*, Parallel Computing 85, pg. 491-498, (see above).

KUNG H.T. [86], *"The structure of parallel algorithms"*, Advances in Computers, Vol. 19, pg.66-112.

KUNG H.T., RUANE, YEN [81], *"A two-level pipelined systolic array for convolutions"*, in 'VLSI Systems and Computations', Ed. H.T. Kung, Sproull, Steele, 1981, pg.255-265.

KUNG H.T. [82a], *"Notes on VLSI computation"*, Parallel Processing Systems, ed. D.J. Evans, Cambridge University Press, 1982, pg.339-356.

KUNG H.T. [82b], *"Why systolic architectures?"*, Computer Jan. 1982, pg.37-45.

KUNG H.T. & SONG [82], *"A systolic 2-D convolution chip"*, extended abstract in IEEE Proc. 1981 or Computer Society Workshop on Computer Architecture for Pattern Analysis & Image Database Management 81, pp.159-160.

KUNG H.T. & LIN [83], *"An algebra for VLSI algorithm design"*, CMU-CS-84-100, Apr. 1983 CMU, Pittsburgh.

KUNG H.T. [84a], *"Systolic algorithms for the CMU WARP processor"*, CMUC-CS-84-158 (7th Int.Conf. on Pattern Recognition also).

KUNG H.T. [84b], *"Systolic algorithms"*, Large Scale Scientific Computing, Academic Press, pg. 127-139.

KUNG H.T. & LAM [84], *"Wafer scale integration and two level pipelined implementations of systolic arrays"*, Jour. of Parallel and Distributed Computing 32-63 (1984).

KUNG S.Y., ARUN, RAO, HU [81], *"A matrix dataflow language/architecture for parallel matrix operation based on computational wavefront concept"*, in VLSI Systems & Computations, eds. Kung H.T., Sproull, Steele, 1981, pg. 235-244.

KUNG S.Y. & HU [83], *"A highly concurrent algorithm and pipelined architecture for solving Toeplitz systems"*, IEEE Trans. on Acoustics, Speech & Signal Processing, Vol. ASSP-31, 1983, pg.66-75.

KUNG S.Y. [84], *"On supercomputing with systolic/wavefront array processors"*, Proc. of IEEE, Vol. 72(7), pg.867-884.

KUNG S.Y. [85], *"VLSI array processors"*, pg.4-22, IEEE ASSP July 1985.

KUNG S.Y., WHITEHOUSE, KAILATH [85], *"VLSI and modern signal processing"*, Part II pg. 178-200, Kuhn, Heller (in particular).

LANG [85], *"The instruction systolic array, a parallel architecture for VLSI"*, 1985 report 8502, Institute fur Informatik and Practische Mathematik - Christien Albrechts Universitait, Kiel.

LEISERSON [81], *"Area efficient VLSI computation"*, Ph.D. Thesis, Oct. 1981, CMU, Pittsburgh.

LEISERSON & SAXE [83], *"Optimizing synchronous systems"*, J. VLSI & Computer Systems 1(1), pp.41-67, Comp.Sci. Press 1983.

LEVIN & EVANS [85], *"Parallel matrix inversion algorithms"*, Comp. Stud. 231, May 1985, L.U.T.

LIN & WU [85], *"Area-period tradeoffs for multiplication of rectangular matrices"*, J.Computer & Systems Sciences 30, pg.329-342. 1985.

LIPITAKIS [78], *"Computational and algorithmic techniques for the solution of elliptic and parabolic partial differential equations in two and three space dimensions"*, Ph.D. Thesis, 1978, L.U.T., Comp.Stud.

LIPITAKIS & EVANS [80], *"Solving non-linear elliptic difference equations by extendable sparse factorisation procedures"*, Computing 24, 325-339 (1980).

LIPSCHUTZ [74], *"Linear Algebra"*, Schaums Outline Series McGraw-Hill.

LJUNG [80], *"Fast algorithms for recursive estimation and identification"*, Numerical techniques for Stochastic Systems, eds. F. Archelti & M. Cugiani, North-Holland, 1980, pg.43-49.

LLEWELLYN [64], *"Linear programming"*, Holt, Rinehart & Winston Publishers 1964.

LUK [86], *"An analysis of algorithm based fault tolerant techniques"*, EE-CEG-86-11, Technical Report Engineering Group, Cornell University (to appear in Advanced Algorithms & Architectures for Signal Processing, Proc. SPIE, Vol. 696).

McCANNY & MCWHIRTER [82], *"Implementation of signal processing functions using 1-bit systolic arrays"*, Electronics Letters 18(6), pg.241-243, 1982.

McCANNY & MCWHIRTER [83], *"Yield enhancement of bit level systolic array chips using fault tolerant techniques"*, Electronics Letters 19(14), 2, pg.525-527, 1983.

MCKEOWN [84], *"Iterated interpolation using a systolic array"*, Report CSA/21/1984 UEA Norwich, Mathematics & Algorithms group 7.

MCKEOWN & RAYWARD-SMITH [84], *"A note on the use of systolic arrays for solving network problems"*, Report CSA/22/1984 UEA, Norwich, Mathematical group 8.

MEAD & CONWAY [79], *"Introduction to VLSI design"*, Addison-Wesley (particularly Ch.8).

MEGSON [84], *"Simulating systolic arrays using OCCAM"*, B.Sc. Dissertation, Leeds University, 1984.

MEGSON & EVANS [85a], *"Systolic arrays for finite differences and rational function approximation"*, Int. Report, Comp.Stud. 255, Dec. 85, L.U.T.

MEGSON & EVANS [85b], *"LISA: A parallel processing architecture"*, Int. Report Comp.Stud. 253, Nov. 1985, L.U.T.

MEGSON & EVANS [85c], *"A stable O(V) BATS cell"*, Int. Report Comp. Stud. 249, Oct. 1985, L.U.T.

MEGSON & EVANS [85d], *"Improvements to the O(n) BATS cell"*, Int. Report Comp.Stud. 246, Sept. 1985, L.U.T.

MEGSON & EVANS [85e], *"BATS: Banded and Toeplitz systems systolic array"*, Int. Report 243, July 1985. L.U.T.

MEGSON & EVANS [85f], *"Survey of systolic algorithms for banded linear systems"*, Int. Report Comp.Stud. 241, July 85, L.U.T.

MEGSON & EVANS [85g], *"A recursive decoupling algorithm for the solution of certain circulant linear systems"*, Int. Report Comp.Stud. 242, July 85, L.U.T.

MEGSON & EVANS [85h], *"Soft-systolic pipelined matrix algorithms"*, Int.Rep.Comp.Stud. 234, June 85, L.U.T.

MEGSON & EVANS [85i], *"The design and simulation of systolic arrays"*, Int. Report Comp.Stud. 230, May 1985, L.U.T.

MEGSON & EVANS [86a], *"The unification of systolic differencing algorithms"*, Int. Report Comp.Stud. 86, L.U.T.

MEGSON & EVANS [86b], *"A systolic simplex algorithm"*, Int. Rep. Comp. Stud. 288, June 86, L.U.T.

MEGSON & EVANS [86c], *"A systolic cylinder for the revised simplex algorithm"*, Int.Rep.Comp.Stud. 304, Aug. 86, L.U.T.

MEGSON & EVANS [86d], *"An orthogonal design for the assignment problem"*, Int. Report. Comp.Stud. 294, July 86, L.U.T.

MEGSON & EVANS [86e], *"Systolic preconditioning algorithms"*, Int.Rep. Comp.Stud, 319, Sept. 86, L.U.T.

MEGSON & EVANS [86f], *"Incomplete elimination systolic arrays"*, Int. Rep. Comp.Stud. 325, Oct. 86, L.U.T.

MEGSON & EVANS [86g], *"Matrix power generation using an optical reduced bandwidth systolic array"*, Int. Rep.Comp.Stud. 314, Sept. 86, L.U.T.

MEGSON & EVANS [86h], *"On systolic arrays for complex matrix problems"*, Int. Rep. Comp.Stud. 329, Nov. 1986, L.U.T.

MEGSON & EVANS [86i], *"A systolic group explicit method for 2-D parabolic problems"*, Int. Rep. Comp.Stud. 323, Oct. 1986, L.U.T.

MEGSON & EVANS [86j], *"The solution of parabolic partial equations by systolic marching techniques"*, Int.Rep.Comp.Stud. 317, Sept. 1986, L.U.T.

MEGSON & EVANS [86k], *"Simulation of soft-systolic arrays with boundary and special processing elements"*, Int.Rep.Comp.Stud. 309, Aug. 1986.

MEGSON & EVANS [86ℓ], *"Group explicit systolic arrays for a hyperbolic equation of first order"*, Int. Rep.Comp.Stud. 293, July 1986.

MEGSON & EVANS [86m], *"3*3 block matrix schemes for systolic arrays"*, Int.Rep.Comp.Stud. 278, Apr. 1986.

MEGSON & EVANS [86n], *"The 3-space ISA and multitasking of soft-systolic programs"*, Int.Rep.Comp.Stud. 273, Apr. 1986.

MEGSON & EVANS [86o], *"Pipelining of systolic Hopscotch schemes"*, Int.Rep.Comp.Stud., 270, Mar. 1986.

MEGSON & EVANS [86p], *"The soft-systolic program simulation system(SSPS)"*, Int. Rep.Comp.Stud. 272, Mar. 1986.

MEGSON & EVANS [86q], *"The alternating group explicit (AGE) systolic array for the solution of large linear systems"*, Int. Rep. Comp.Stud. 265, Feb. 1986.

MEGSON & EVANS [86r], *"Systolic arrays for group explicit methods for solving parabolic equations"*, Int.Rep. 259, Jan. 1986.

MELHEM & RHEINBOLDT [84], *"A mathematical model for the verification of systolic networks"*, SIAM J. Comp. Vol. 13, No.3, Aug. 1984, pg.541-565.

MINSKY [67], *"Computation: finite and infinite machines"*, Prentice-Hall, 1967.

MURPHY [78], *"Numerical methods for solving ordinary and partial differential equations"*, Ph.D. Thesis, 1978, L.U.T.

PICKERING [84], *"Solution of quasi-tridiagonal systems of linear equations"*, Int.Jour.Comp.Math., 1984, Vol.15, pg.181-191.

QUINTON, JOINNAULT, GACHET [86], *"A new matrix multiplication systolic array"*, Parallel Algorithms and Architectures, Cosnard et al, Elseiver Science Publishers, North-Holland, 1986. pg 259-268

RALSTON & WILF [60], *"Mathematical methods for digital computers"*, Wiley & Sons Inc., 1960, USA, pg.73-77.

ROBERT & TCHUENTE [84], *"Parallel solution of band triangular systems on VLSI arrays with limited fanout"*, International Workshop on Modelling and Performance evaluation of Parallel Systems, Ed. Becker, 1984, Grenoble, pg. 209-229.

ROBERT [85], *"Block LU decomposition of a band matrix on a systolic array"*, Intern. J.Comp.Math., Vol. 17, pp.295-315, 1985.

ROBERT & TCHUENTE [85], *"Reseaux systoliques pour des problemes de mots"*, RAIRO Informatique Theorique Theoretical Informatics, Vol. 19, No.2, 1985, pg.107-123.

ROBERT & TRYSTRAM [85], *"An orthogonal systolic array for the algebraic path problem"*, Report 553 IMAG 1985, France.

ROBERT [86], *"Algorithmique parallele: reseaux d'automates, architectures systoliques, machines SIMD & MIMD"*, Doctor of Science Thesis, L'institut National Polytechnique de Grenoble, L'universite Scientifique et Medicale de Grenoble.

ROSENBERG [83], *"Three dimensional VLSI: A case study"*, J.ACM, Vol. 30, No.3, July 1983, pp.397-416.

ROTHE [85], *"A systolic array for the algebraic path problem"*, Computing 34, 191-219 (1985).

ROTHE [86], *"On the connection between hexagonal and unidirectional rectangular systolic arrays"*, Proc. Aegean Workshop on Computing (AWOC 86), to appear 87.

RUTIHAUSER [51], *"Solution of eigenvalue problems with the LR-transformation"*, Nat.Bur. Standards, Appl.Math. Ser. 49, 1958, pg.47-81.

SAHIMI [86], *"Numerical methods for solving hyperbolic and parabolic partial differential equations"*, Ph.D. Thesis, 1986, L.U.T.

SAMEH & KUCK [circa 78], *"On stable parallel linear system solvers"*, J.ACM, 25, 1 (1978), pg.81-91.

SAMI & STEFANELLI [86], *"Reconfigurable architectures for VLSI processing arrays"*, Proc. IEEE, Vol. 74, No.5, May 1986, pg.712-722.

SAUL'EV [64], *"Integration of equations of parabolic type by the method of nets"*, International Series of Monographs in Pure & Applied Mathematics, Pergamon Press Ltd. 1964.

SAVAGE [81], *"Area time tradeoffs for matrix multiplication and related problems in VLSI models"*, J.Computer & System Sciences 22, pg. 230-242, 1981.

SCHREIBER [83a], *"Computing generalised inverses and eigenvalues of symmetric matrices using systolic arrays"*, NA-83-03, Nov. 1983, Stanford University, California.

SCHREIBER [83b], *"On systolic array methods for band matrix factorisations"*, TRITA-NA-8316, Dept. of Numerical Analysis and Computing Science, The Royal Institute of Technology, S-100 44, Stockholm Sweden.

SHAPIRO [84], *"Systolic programming a paradigm of parallel processing"*, Proc. of FGCS, 84, and TR CS84-16 Weizmann Institute Applied Math.

SMITH [85], *"Numerical solution of partial differential equations: finite difference methods"*, Oxford Applied Mathematics and Computing Science Series, Oxford Univ. Press.

SNYDER [82], *"Introduction to the configurable highly parallel computer"*, Computer Jan. 1982, pg.47-56.

SOJOODI-HAGHIGHI [81], *"The numerical solution of elliptical and*

parabolic partial differential equations by novel block iterative methods", Ph.D. Thesis, 1981, L.U.T.

SORENSEN [85], "*Analysis of pairwise pivoting in Gaussian Elimination*", IEEE Transactions on Computers, Vol. C-34, No.3, pg. 274-218.

SPEISER & WHITEHOUSE [81], "*Parallel processing algorithms and architectures for real-time signal processing*", SPIE Vol. 298, Real-time Signal Processing IV (1981). pp.2-9.

STONE [73], "*An efficient parallel algorithm for the solution of a tridiagonal linear system of equations*", J.Ass. Computing Machinery, Vol. 20. No. 1, pg.27-38.

SWEET [84], "*Fast Toeplitz orthogonalization*", Numer.Math. 43, pg. 1-21, 1984.

THOMPSON & TUCKER [85], "*Theoretical considerations in algorithm design*", NATO Advanced Study Institute on Fundamental Algorithms for Computer Graphics. (Also Leeds Univ.Comp.Stud. Report 200).

TURKEDJIEV [86], "*Synthesis of systolic algorithms and processor arrays*", pg.165-172, Springer Verlag Lec. Notes in Comp.Science 237, (CONPAR 86).

TYLAVSKY [85], "*Quadrant interlocking factorisation, a form of block LU factorisation*", IEEE, to appear.

UMEO & SUGATA [82], "*Systolic simulation of synchronous SIMD parallel computers*". Faculty of Engineering, Osaka Univ. Japan Report. AL-82-21.

UMEO [82], "*Two dimensional systolic implementation of array algorithms*", Report AL-82-32 (as above).

UMEO, MORITA, SUGATA [82], "*Deterministic one-way simulation of two-way real-time cellular automata and its related problems*", Information Processing Letters, Vol. 14 No.4, 1982, pg.158-161.

UMEO [85a], *"A class of SIMD machines simulated by systolic VLSI arrays"*, VLSI Algorithms and Architectures, eds. Bertolazzi & Luccio, 1985, pg.39-48.

UMEO [85b], *"Time optimum parallel binary address setting schemes for cellular computers"*, IEEE Computer Society Symposium on New Directions in Computing, pg. 293-299.

ULLMAN [84], *"Computational aspects of VLSI"*, Computer Science Press, 1984, Ch.1, Ch.2, Ch.5.

VARGA [62], *"Matrix iterative analysis"*, Prentice-Hall Series in Automatic Computation, 1962.

WANG [81], *"A parallel method for tri-diagonal equations"*, ACM Trans. on Mathematical Software, Vol.7, No.2, 1981, pg.170-183.

WEISER & DAVIS [81], *"A wavefront notation tool for VLSI array design"*, VLSI Systems and Computations, ed. Kung H.T., Sproull, Steele, pg.226-234.

WESTLAKE [68], *"Numerical matrix inversion and solution of linear equations"*, Wiley & Sons, Inc. 1968, USA, pp.23-24.

WHITEHOUSE, SPEISER & BROMLEY [85], *"Signal processing applications of concurrent array processor technology"*, pg. 25-41 of VLSI & Modern Signal Processor, Prentice-Hall, 1985, eds. Kung, S.Y., Whitehouse, Kailaith.

WU & COPPINS [81], *"Linear programming and Its Extensions"*, McGraw-Hill Book Co. 1981.

WYNN [62], *"Acceleration techniques for iterated vector and matrix problems"*, Math. & Comp. 16, pg.301-322, 1962.

YANG & LEE [86], *"The mapping of 2-D array processors to 1-D array processors"*, Parallel Computing, 1986, 3, in press.

APPENDIX I

OCCAM SUMMARY

In OCCAM processes are connected to form concurrent systems, each process can be regarded as a black box with an internal state which can communicate with other processes via point to point communication channels. The processes themselves are finite. Each process starts, performs a number of actions then terminates. An action may be a set of parallel processes to be performed at the same time. As a process is itself composed of processes which may themselves be executed in parallel a process allows internal concurrency which varies with time.

Processes:

All processes are constructed from three primitive processes, assignment, input and output.

Assignment: An assignment is indicated by the symbol :=, for example, v:=e sets variable v to the value of the expression e and then terminates.

Input: An input is indicated by the symbol ?, for example, c?x inputs a value from a channel c assigning it to x and then terminating.

Output: An output is indicated by the symbol ! and c!e outputs the expression e to channel c, and then terminates.

A pair of concurrent processes communicate using a one way channel connecting the two processes. One process outputs a message to the channel, the other process inputs the message from the channel. A particular process can be ready to communicate on one or more of its channels any time between its start and termination, but a communication only takes place when both it and the process sharing one of its channels is ready. Where a number of connected processes are ready simultaneously communication can occur in parallel.

Constructs:

A number of processes can be combined to form a construct which is

itself a process and can be used as a component for other constructs. Each component process is indented by two spaces from the left hand margin indicating which construct it is part of. There are only four basic construct types, sequential, parallel, conditional, and alternative.

SEQ: is the keyword for a sequential construct denoted

```
SEQ
  P1
  P2
  P3
  ...
```

where the component processes P1,P2,P3,... are executed in strict sequence with process P_i finishing before P_{i+1} starts and after P_{i-1} terminates. Sequential constructs are similar to programs written in conventional programming languages.

PAR: is the keyword for a parallel construct of the form

```
PAR
  P1
  P2
  P3
  .
  .
  .
```

and in contrast to SEQ, here all the component processes P1,P2,P3,... are executed concurrently. The PAR construct terminates when all the component processes have finished.

IF: is the keyword for a conditional construct with the appearance

```
IF
   condition 1
        P1
   condition 2
        P2
       ...
```

This means that P1 is executed iff condition 1 is true, otherwise P2 iff condition 2 is true, etc.etc. Notice the strict sequential ordering of tests. Only one of the processes P_i is executed and the IF construct terminates when the process finishes.

ALT: is the keyword for the alternative construct

```
ALT
   input 1
     P1
   input 2
     P2
     :
     :
```

This construct waits until one of input 1, input 2, input 3,... is ready. If input 1 is ready first, input 1 is performed and on completion P1 is executed. Similarly if input i is ready first input i is performed and P_i executed. Only one of the inputs is performed and its corresponding process executed before the construct terminates. If more than one input becomes ready at the same time the one executed is chosen arbitrarily.

Repetition:

There is only one explicit construction for repetition denoted by

```
WHILE condition
  P
```

which repeatedly executes process P until the value of the condition is false. Observe that P itself can be a composition of sequential and parallel constructs.

Replication:

A replicator is used with a constructor to replicate the component process a number of times. With SEQ a standard for loop

```
SEQ i=[0 FOR n]
  P
```

is created executing process P sequentially n times. When used with PAR an array of concurrent processes with the form

```
PAR i=[0 FOR n]
  Pi
```

is created such that n similar processes $P_0, P_1, \ldots, P_{n-1}$ are executed in parallel. Notice that i=0(1)n-1 not n, thus if generally i=[base FOR count]

there are base+count-1 values i takes starting with i=base.

Declarations:

A declaration introduces a new identifier for use in the process that follows it, and defines the meaning the identifier will have within the process. If the new identifier is the same as one already in use, all subsequent occurrences of the identifier in the process will refer to the meaning of the most recent declaration. Declarations are of four basic types VAR, CHAN, DEF and PROC linked to a following process by a colon (:) at the last line of the declaration. The process follows on the next line at the same level of indentation as the keyword declaration. For example,

```
VAR x:
P
```

declares variable x to be used in process P, and

```
CHAN C:
P
```

defines a channel C to be used in communication for P. A variable vector declaration introduces an identifier to be used as a vector of variables, viz.

```
VAR list [16]:
P
```

for a vector named list of 16 variables indexed as list[0],list[1],... list [15]. Likewise a channel vector declaration introduces a new identifier as a vector of channels for communicating between concurrent processes

```
CHAN C[n]:
P
```

DEF associates a name with a constant value, or with a table of constant values, e.g.

```
DEF a=1, b=2:
```

associating a with 1 and b with 2, using these identifiers within a process yields the associated values.

The PROC declaration introduces an identifier to name the process which follows, indented, on the succeeding lines. The process is termed the named process and is itself followed by a process in which the named process will be used. The named process can have parameters which are declared with the declaration of the named process and are called formal parameters. The named process text will be substituted for all occurrences of the process name in subsequent processes, the var and chan variables substituted in place of the formal parameters are called actual parameters. For example,

```
PROC buffer(CHAN in, out) =
  WHILE TRUE
    VAR x :
    SEQ
      in?x
      out!x :

CHAN c,c1,c2 :
PAR
  buffer(c1,c)
  buffer(c,c2)
```

declares two buffer processes executed concurrently, buffer is the named process with formal channel parameters in and out. In the following process C,C1,C2 are actual parameters and on execution the WHILE loop will be textual substituted for occurrence of the name buffer and C,C1,C2 substituted for in and out respectively. The size of a vector is not specified in the formal parameters of a named process and different sized vectors may be used as actual parameters on different substitutions. In addition to the standard declarations VAR and CHAN, a VALUE parameter may also be used, as either an ordinary or vector formal parameters and cannot be changed within a process by assignment or input.

Finally an identifier which is used but not declared in a named process is termed a free identifier. Any free identifier in use when a named process substitution takes place must be the same as a variable already in use. The free variable then takes on the most recent incarnation of the variable at the point where the process substitution takes place.

Program Format:

In OCCAM indentation from the left hand margin indicates program structure. Each process starts on a new line, at an indentation level indicated by the following rules.

Constructs

The construct keyword (and the optional replicator) occupies the first line. Each of the component processes start on a new line and are indented by two spaces more than the keyword.

Conditionals

The condition expression occupies the first line, and the component process starts on the next line indented by two more spaces.

Alt inputs

The expression and its associated input occupy the first line and the component process starts on the next line indenting two more spaces.

Declarations

Each declaration starts on a new line, at the same level of indentation as the process it prefixes, the final line of the declaration being terminated by a colon. Blank lines can be inserted anywhere and are ignored.

A construct can be broken to occupy more than one line, with line breaks occurring after comma, semicolon and before the second operand

of an operator (requiring two operands). The continued line must be more indented than the first line of the construct.

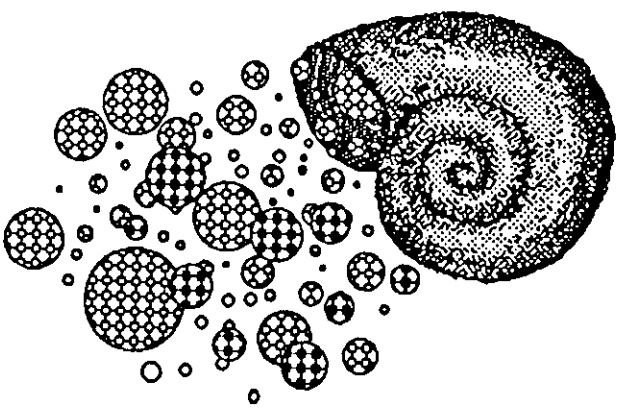
Comments:

Comments are denoted by double hyphen (--) and terminate at the end of a line. All characters of a comment are ignored. A comment may follow an OCCAM construct on the same line or be on a line by itself

This summary of OCCAM is taken from INMOS [84,85] and implements 'proto-OCCAM'. A more sophisticated version OCCAM 2 is now available providing Real, Integer, and Boolean types. We remark that the programming in this thesis was performed on a VAX machine under UNIX using Loughborough OCCAM as implemented by R.P. STALLARD. Appendix II discusses the Loughborough version of OCCAM and particularly its extensions of proto-occam to allow real variables and non-standard OCCAM features. We point out that the programs listed in Appendix III where possible have avoided these non-standard characteristics.

APPENDIX II

Loughborough Occam™ Compiler Version 5.0 Documentation



Help for running the occam compiler

A source 'occam' file (OCCAM and INMOS are trademarks of the INMOS group of companies) must be of the form '*.occ', to compile it to form an 'a.out' command file use the default options. For example to compile 'my_first.occ' :-

```
occam my_first.occ
```

An executable object 'a.out' is produced. As a shortcut you can omit the '.occ' affix and just say 'occam my_first', the compiler will add on the affix for you.

If a program is split into several files these can be separately compiled and linked together using the occam compiler and built in linker.

Each previously compiled occam program is specified in the command line in the form '*.o' e.g. ;-

```
occam main.occ numericlib.o screenlib.o
```

This will compile the source of 'main' and link it in with the pre compiled library occam files 'numericlib.occ' 'screenlib.occ'. The -l option is used to generate new versions of library file objects.

Various switch options are provided, mainly for compiler debugging. Flags can either be put separately ('-g -l') or together and in any order ('-lg', '-gl'). The following switches may be useful :-

-g :

```
occam -g fast.occ
```

Compile the occam program as before but run the resulting program immediately (a compile,load and go option). If flag options are specified that apply to the run of the program these will be passed on as in 'occam -gqc fast'.

-l :

```
occam -l new_lib
```

Compile the program and produce object but do not link the object files together to produce an object program. This option is used for bulding up libraries of routines or to cut down the compilation time for compiling one long program.

-o :

```
occam keep_it -o saverun
```

Compile the program as normal but place the object program in the file 'saverun' rather than the default 'a.out'. Useful for saving several occam object files at the same time.

-x :

```
occam -x old_fashioned.occ
```

Compile according to the strict Inmos occam specification, LUT extensions (see file 'occamversion') currently include :-

Multiple source file cross linking.
Dynamic features.
Variable PAR replicator counts.
Floating point arithmetic.

-c :

```
a.out -c
```

Run the object program with cursor addressable facilities enabled, the standard library procedures 'goto.x.y' and 'clear.screen' require these facilities.

-G : occam -G error_prone

Compiles the file as normal but generates a symbol file as well (in this case it would be 'error_prone.sym'), this is used by the run-time system to inspect the values of variables.

-q :

```
a.out -q
```

Run the object program without producing any characters to the screen other than those output by the program (unless CTRL c used). This enables occam programs to dump output that can be processed by other occam programs.

-F and -M :

```
occam -F num.occ
```

'-F' Includes the floating point library routines to provide a simple real number arithmetic capability. '-M' includes both the floating point and mathematical library routines to provide mathematical library routines.

-I :

This provides the features of the Inmos proto-occam definition (see 'occam version') such as STOP and TIME, it should be used where possible as it is closer to the occam-2 definition.

Full list of compiler option flags

The full (often cryptic) range of switch options are as follows. Several switch flags can be given, in any order and either separately or together.
The mnemonic character giving the switch is highlighted by a capital letter.
They are divided into sections - user defined flags, and system defined options, which are selected by prefixing with '%'.

User Flags

-% The next flag(s) are system flags - switch flag mode.

-c Run the program with Cursor addressable options enabled.
The library routines 'clear.screen' and 'goto.x.y' need this flag set. If used for the compiler must also give the -g option.

-e Produce object/run object for Execution tracing. The resulting object file is then run with the '-e' option. This utility is described in 'tracerinfo'.

-f Force full occam semantic check on use of variables.
A variable (not vectors though) can not be set within a PAR construct if the declaration is outside the PAR. This applies equally to procedure calls that change global variables.

-g Run the resulting object file if compilation succeeded.
The program Goes immediately it is ready to.

-h Print out this 'Help' information.

-i Force an Interrupt immediately before start of execution - immediately displays the debug help menu. This enables break and trace points to be setup prior to anything being executed.

-l Compile but do not link the occam source. Needed when using multiple occam source Library files.

-m Check that every channel Match properly on execution, channels can have only one input and one output process during execution.

-o Produce an Object program with name given by the non-switch argument following this switch. Enables you to choose an object file name other than 'a.out'.

-q Run the program without outputting some non occam program produced messages - e.g. 'OCCAM Start Run'. Must give -g option as well 'q' stands for Quiet. Useful when producing output to be piped or processed by other programs.

-w Suppress the Warning messages from the compiler - when you have seen these warnings once you may find it less irritating to suppress them on subsequent compilations - does not affect error reporting or any other compiler action.

-x Do not permit any local LUT eXtensions in the source text.
See 'occinfo' for information about these - for example recursion and EXTERNAL procedure definitions. Useful if moving an occam program for use on another occam compiler system.

-F Include the standard Floating point library routines.
Provides routines to read or write floating point routines to channels.

-G Produce a symbol table file (with affix '.sym') for use with the 'm' option in the dynamic debugger for symbol value examination.

-I Permit the use of INMOS proto-occam version 2. These changes include the use of 'TIME' instead of 'NOW', the 'STOP' primitve and the use of 'Stopping IF' - an alternative without any TRUE conditions will STOP.

-L Use Long winded load, all the 'C' libraries are added at the last moment rather than using the pre-linked object, this may be useful if a user occam/C library calls a 'C' routine that is not used in the occam run time system. See 'libraryhelp' for more info.

-M Include the Mathematical library and floating point routines.

-O Produce optimized object. May improve run time by 20%.

-R Use Randomized scheduling when running the program - the same scheduler choices will not be made on separate executions.
This gives non-deterministic execution and will be slightly slower but may be useful occasionally.

-S Do not include the Standard I/O routines with the object. This library is included by default, there is no reason not to want to include it unless you want to devise a totally new one.

-T The next argument is a Timing definition file built by the 'timebuild' utility to be used in conjuntion with the '-e' option , supplying '-T' automatically selects '-e'. If this option is not selected the execution timings are taken from the source library file 'times'. Look at the 'timerinfo' help file for more details.

-V The compiler will normally desist reporting errors and warnings after the first fifty or so, with this option all the errors will be reported. May produce Very Verbose output.

-W Give Warning messages about declarations that turn out not to have been used at all. This may highlight misspelt declarations or existence of no longer used procedures.

System Flags

-% Switch back to expecting 'user' mode flag options.
This means you can replace -G%v by -%v%G.

-a Enable Analysis of the usage of channels - this facility is still under test.

-n Check the source occam for syntax errors, but do Not produce any object data from it.

-t Print out the program in the form just after it was Transformed.
Not generallt useful as the program has changed so much.

-v Give Verbose information at each stage when running the compiler - will print out a more accurate description of the system commands it is calling and all the files it accesses.
Also switches on a full print out of the occam link information.

-A Produce the object code ('C' or Assembler) in a permanent file so that it can be inspected.

-C Produce 'C' rather than assembler output from the occam compiler then compile and link it. There will be *.o and *.c containing the object and compiler generated source created in the directory.
The 'C' and assembler code produced will be similar and there is little point in producing 'C' unless to waste time ! (as the 'C' compilation phase takes a long time). If the compiler is ported to a non-VAX system then this option will automatically be selected.

-D Switch on variable name and line number Dumping in the C/Assembler 'object' file so that the object can be tied in with the source.

-H Undocumented feature under test.

-L Produce an occam-'C' interface Library, the two files ending '-c.c' and '.occ' are linked together, the occam can refer directly to the 'C' routines.

-N Run the compiler showing the steps it would execute but without actually doing anything - like '-n' in the UNIX 'make' command. Useful when options start getting complicated. A No operation facility.

-Q Undocumented feature under test.

-S Do not apply some Simplifying transformations on the program. These currently remove constructs with no processes in them and redundant SEQ and PAR headers. These save a small amount of space and time at run and compile time and there is little point in turning off this option.

-X Print out the procedures that have been defined in the link files but has not been referenced - detects eXtra procedures defined across files but not used.

-Y Produce the linker assembler output in a permanent file rather than in a temporary file on '/tmp'. Enables the output from the linker to be debugged.

-Z Get the linker to print out all the definitions it is told about.

Description of the library routines

Standard Library

Provide commonly used routines to read and write to the keyboard and screen channels. The routines are written in 'C' and occam and use standard C or 'curses' I/O routines. There are also general routines for use to pause or abort a program as well as to use the 'C' random number routines. They are available by default to all programs unless the -S compiler flag is used to override their inclusion.

```
EXTERNAL PROC str.to.screen (VALUE s []) :
```

Output the string s (a byte array with byte 0 as the length).
The whole string is guaranteed to be printed in one sequence, two concurrent calls to str.to.screen will not interleave.
Equivalent to the program fragment :-

```
PROC str.to.screen (VALUE s []) =
  SEQ n = [1 for s [BYTE 0]]
    screen ! s [BYTE n] :
```

```
EXTERNAL PROC num.to.screen (VALUE n) :
```

Output a number to the screen. The number can be signed, and uses the minimum number of characters (no leading spaces). Equivalent to the 'C' language 'printf ("%d",n);' statement.

```
EXTERNAL PROC str.to.chan (CHAN c,VALUE s []) :
```

Output the string s to a channel 'c'. The call 'str.to.chan (screen,"fred")' is identical to 'str.to.screen (fred)'. Useful for string output to files.

```
EXTERNAL PROC num.to.chan (CHAN c,VALUE n) :
```

Output ascii string for the number 'n' to channel 'c'. Like 'str.to.chan' but for numbers not channels.

```
EXTERNAL PROC num.to.screen.f (VALUE n,d) :
```

Output a number to the screen in a field of width 'd'. If the number is too big for the field the number is written out in full regardless, the routine call num.to.screen.f (n,1) is equivalent to num.to.screen (n). The routine uses the 'C' language printf format %nd where n is the field width.

```
EXTERNAL PROC goto.x.y (VALUE x,y) :
```

Use the 'curses' package to implement a cursor 'goto' facility. No error checking is made that the move is within the screen area. The x-axis is across the screen and y-axis down, co-ordinate (0,0) is in the top left hand corner of the screen. The first line is used by the run time system to print messages.

```
EXTERNAL PROC clear.screen :
```

Use curses to clear the screen,if cursor addressable option not used this will still try to clear the screen using the curses "CL" termcap defined string.

```
EXTERNAL PROC num.from.keyboard (VAR n) :
```

Read a number from the keyboard and assign to variable 'n'. The routine is not very sophisticated. It will read negative numbers (start '-') and ignore any leading 'space' characters. The number must be followed by a non-digit, this character is read by the routine and not available on a subsequent 'Keyboard ? ch' process. There is no check that the number is too big for the number range. It will expect at least one digit otherwiae it will give an error message.

```
EXTERNAL PROC num.from.chan (CHAN c,VAR n) :
```

Read a number from a channel 'c'. If 'c' is the keyboard this is equivalent to calling 'num.from.keyboard'.

```
EXTERNAL PROC abort.program :
```

Force the program to abort execution. An explanatory message is printed so that the cause will be known.

```
EXTERNAL PROC force.break :
```

Perform the same action as if 'CTRL-C' was pressed at the terminal. The user interface routines can then be run under the menu selection facility provided.

```
EXTERNAL PROC random (VALUE d,VAR n) :
```

Return a pseudo random number in the range 0 to d-1 by using the 'C' 'random ()' function in the variable n. The VALUE of d must not be zero. The sequence of random numbers will be modified if the '-R' run option is used.

```
EXTERNAL PROC init.random (VALUE n) :
```

Initialise the seed for the random number generator for subsequent calls to the procedure 'random'. Uses the 'C' language routine 'srandom ()'.

```
EXTERNAL PROC trace.value (VALUE n) :
```

Print out the integer value of 'n' on the screen with the prefix string 'Trace value : ' - this makes debugging a little easier.

```
EXTERNAL PROC open.file (VALUE path.name [],access [],CHAN io.chan) :
```

Connect the channel 'io.chan' to a UNIX file. The procedure must be provided with the pathname of the file as a string, and the access mode ("r" read access,"w" write access,"a" append access). Subsequent input or output on 'io.chan' will fetch/put a single character from/to the file. Attempts to input past the end of file will receive the value -1.

```
EXTERNAL PROC close.file (CHAN io.chan) :
```

Cease connection of the channel with its currently open file.

```
EXTERNAL PROC open.pipe (VALUE command.name [],access [],CHAN io.chan) :
```

Connect the channel 'io.chan' to a UNIX pipe running command 'command.name'. The procedure must be provided with the UNIX command name and 'r' to read from it, or 'w' to write to it). Subsequent input or output on 'io.chan' will fetch/put a single character from/to the file. Attempts to input past the end of file will receive the value -1.

```
EXTERNAL PROC close.pipe (CHAN io.chan) :
```

Cease connection of the channel with its currently active command.

```
EXTERNAL PROC system.call (VALUE command [],VAR code) :
```

Execute the UNIX command contained in the string 'command' and return the value in 'code' TRUE if the command succeeded without error and FALSE otherwise.

```
EXTERNAL PROC set.timers (VALUE init.value) :
```

Set up the interval timers ITIMER_REAL,ITIMER_VIRTUAL to the given start value. These are used for timing sections of code on the VAX. Uses 'setitimer' call. Note that using 'WAIT' primitive will reset the timer so it can only be used for simple sections of code. It should also be noted that it times the whole program and not a single occam process.

```
EXTERNAL PROC get.real.timer (VAR secs,micro.secs) :
```

Get the current elapsed timer values in seconds and microseconds. Timers count downwards and are not especially accurate. Uses 'getitimer' call.

```
EXTERNAL PROC get.cpu.timer (VAR secs,micro.secs) :
```

Get the current executed CPU timer values in seconds and microseconds. Timers count downwards and are not especially accurate.

Floating Point Library

Routines to perform floating point input/output. They are available by giving the compiler flag '-F' when linking an occam program.
Floating point value can be assigned and transmitted via channels just like normal integer values, see the file 'occamversion' for details as to the language extensions introduced to support them.

Input/Output Routines

EXTERNAL PROC fp.num.to.screen (VALUE FLOAT f) :

Print out the floating point number in 'C' language float format "%6.6f". If the number is too small or too big the standard 'C' action will be taken.

EXTERNAL PROC fp.num.to.screen.f (VALUE FLOAT f,VALUE w,d) :

Print out the floating point number in 'C' real format "%w.df". If the number is too small or too big problems will arise.

EXTERNAL PROC fp.num.to.screen.g (VALUE FLOAT f) :

Print out the floating point number in 'C' real format "%g". This will use the most appropriate format - exponent form if necessary.

EXTERNAL PROC fp.num.to.chan (CHAN c,VALUE FLOAT f) :

Write a number to a channel. If channel is 'screen' this is equivalent to 'fp.num.to.screen'. Useful for writing data to files.

EXTERNAL PROC fp.num.from.keyboard (VAR FLOAT f) :

Read in a floating point number. The number is expected to begin with a digit or '.' (indicating 0.), leading spaces are ignored. The number ends on a non-digit and this character will not be available to subsequent reads from the keyboard channel. The following are valid input numbers followed by the interpreted value for the input.
45.35 (45.35) 0.0004 (0.0004) .0 (0.0) 1. (1.0) 124 (124.0)

EXTERNAL PROC fp.num.from.chan (CHAN c,VAR FLOAT f) :

Read a floating point number from a channel 'c'. If channel is keyboard this is equivalent to 'fp.num.from.keyboard'.

Mathematical Routine Library

Mathematical routines from the UNIX '-lm' library. These are included by specifying the '-M' flag. They are all in single precision even though double precision 'C' routines are called.

EXTERNAL PROC fp.sine (VALUE FLOAT a, VAR FLOAT res) :

Return the sine of 'a' in 'res'. Angles are in radians.

EXTERNAL PROC fp.cosine (VALUE FLOAT a, VAR FLOAT res) :

Return the cosine of 'a' in 'res'. Angles are in radians.

EXTERNAL PROC fp.arc.sine (VALUE FLOAT a, VAR FLOAT res) :

Return the arc sine of 'a' in 'res'. Angles are in radians.

EXTERNAL PROC fp.arc.cosine (VALUE FLOAT a, VAR FLOAT res) :

Return the arc cosine of 'a' in 'res'. Angles are in radians.

EXTERNAL PROC fp.arc.tan (VALUE FLOAT a, VAR FLOAT res) :

Return the arc tangent of 'a' in 'res'. Angles are in radians.

EXTERNAL PROC fp.exp (VALUE FLOAT a, VAR FLOAT res) :

Return e to the power 'a' in 'res'.

EXTERNAL PROC fp.log (VALUE FLOAT a, VAR FLOAT res) :

Natural logarithm of 'a' in 'res'.

EXTERNAL PROC fp.sqrt (VALUE FLOAT a, VAR FLOAT res) :

Square root of 'a' in 'res'. Returns an occam error if 'a' is negative.

The run time system

As you might hope when an occam program is executed it will follow the program execution until one of three things happen.

1] The program terminates
2] CTRL-C is pressed on the keyboard
3] An error is detected.

In the case of (2) and (3) a debug option will be displayed, this allows you to abort the program, ignore the interrupt (continue), and to restart the program again. Other options control the '-e' trace output, provide a 'system' debug option (which is only really useful to someone who knows their way around the compiler), an option to specify which source file you want to debug and the 'screen animated debug'. This later option should be of most use and is described in detail in the next section.

Errors come in two types 'Fatal Errors' and just 'Errors', it is not possible (or wise) to continue execution after the former, but the latter may be ignored if the symptom is expected.

The run time display debugger

This utility that runs under the run time system enables users to look at the status of the processes during execution of a program.

The utility requires the use of a cursor addressable terminal. The system provides selective display of the source file(s) that were compiled to form the program together with a column showing the currently existing processes on those particular lines of the source file.

When initially entered by pressing 'CTRL-C' the program execution will be halted, the execution can be restarted in 'stepped mode' so that the display will be updated every occam scheduler action.

Breakpoints and trace points can be added at selected line numbers. Break points cause the debug display to be automatically entered when any of the process executes any of the source lines on which a break point is set. Trace points cause temporary entry into the debug display before resuming normal execution after five seconds pause.

If a file has been compiled with the '-G' flag then the value of occam variables and the status of channels can be printed. Because an occam program can have several processes running with different values to the same identifiers (e.g. within PAR n = [0 FOR 7],'n' has a different value for each separate process) a single process must be selected as before this facility can be used. When selected a second window within the debug display is opened and the values printed by the program are placed within it.

Straightforward use of the debug display will normally entail running a program and pressing CTRL-C when a dubious section of code is about to be executed and entering the debug display ('z' command). Thereafter the commands 'p' to find the next process, 'f' and 'b' might be used to see whereabouts the process is executing. The program can then be single stepped through using the 'r' command to start execution and 's' command to stop execution. Eventually exit of the debug displayer can be made with the 'x' command.

There are two special markers that are used, '>' on a line indicates the currently selected line and '-' the currently selected process.

The commands where practical have been made similar to those in UNIX 'vi'. (UNIX is a trademark of A.T. & T.).

Available commands

Moving about within the file

↑D- Move forward half a page of source text.
↑F- Move forward a page of source text.
↑U- Move backward half a page of source text.
↑B- Move backward a page of source text.
:<number> - Move to given line <number> in file.
k - (or ↑K) Move down one line.
j - (or ↑J) Move up one line.
/<string> - Find given <string> in file from current position.
n - Find next string occurrence for match string selected by '/' command.
p - Find the next process in the file.

Trace/Breakpoints

b - Add breakpoint at currently selected line.
t - Add tracepoint at currently selected line.
d - Delete the trace/break point at the selected line.
c - Delete all the points in the current file.
C - Delete all the points in all the files.
P - Print process status of the currently selected process
D - Deselect the current debug occam process.
S - Select the current debug occam process.
N - Select next process on the same line, if there are several processes that are shown as executing on the same line then 'S' will make an arbitrary choice, 'N' can be used to override this and step through the processes until the one that is desired is selected.

Symbol inspection

m - Select a symbol to display, if no symbols have been selected before then the symbol window is opened and the value of the variable or the status of a channel.
M - Repeat the previous 'm' command. To find the value of the same variable name again.

Execution control

a - Abort the run.
r - Run debug display if a debug process is selected the debug display will be re-entered every time that process is run, otherwise the debug display will be run each time any process is run.
> - Execute in single step mode. Only a single step is executed.
s - Stop the debug display from running temporarily after a 'r' or 'x' command.
u - Change display step interval (initial step interval is 1), this permits the location of processes to be seen after 'n' steps rather than after each and every time it is executed. Not particularly useful.
x - Exit display debugger, program will proceed normally until a trace/break point is found or '↑C' is pressed.
X - Exit to main '' menu so that program restart,abort,file selection or system debug can be done. Used when you wish to debug a different file or to set things going again after setting up breakpoints.

Miscellaneous

? - Print out this help information.
↑L- (or ↑R) Redraw the current displayed information.
i - Buffer keyboard channel input text for the program.
O - Print overall data about the processes currently executing - how many are in each process status, stack use and clock time.
V - Display the occam program's current screen output temporarily
v - Invoke the 'view' command on the occam source file (this is just like 'vi' but with read only access to the file - This can be used to provide more powerful string search facilities when debugging.

Display key

The column between the line number and the text is used to display the number and status of processes executing on that line. Because of the compilation these may be out by a line or two in some circumstances. Most sequential code will be executed as a single block - so a process will not move through a SEQ block one step at a time necessarily.
The special symbol 'P' does not represent a process, it indicates that a procedure has been called at that point. 'P' therefore represents the 'call point' of the procedure.

The following symbols are used to represent the various process statii :-

* - An active process - may be chosen for execution at any time.
a - Process waiting for one or more ALT guards to become TRUE.
w - Process waiting for a clock time or for input/output.
c - Process is waiting for one or more child PAR processes to terminate.

In addition break and trace points are indicated in the column by giving a 'T' for a trace point and 'B' for a break point.

So a display of :-

316:3*w : occam.s ? razor

indicates that there are three active processes and one process waiting input on line 316.

Keyboard and Screen input/output

Because the debug display routine is fully interactive the screen and keyboard data from the program can not be handled in the same manner as normal. Input for the keyboard must be input using the 'i' command - a whole line can be input and will be buffered up for program input in this way. Screen output should be displayed as it is produced (but a copy of it will be sent to the screen image that will redisplayed on exit from the display debugger) or the 'V' command. Strings can have escapes in them '*n' means newline,'*r' carriage return and '**' space.

Non standard occam features

This compiler to the best of my knowledge (Mr.R.P. Stallard of the Department of Computer Studies, Loughborough University of Technology, U.K.) implements the occam language as defined in the occam programming manual published by INMOS limited subject to a few restrictions and extensions that are described in this file. These differences are intended to make transfer of occam programs from different implementations feasible.

It is intended to be compatible to the INMOS booklet version and the Prentice Hall book definition. OCCAM, INMOS and Transputer are registered trademarks of the INMOS Group of Companies.

INMOS proto-occam language revisions

The following additional features introduced into INMOS occam products can now be selected by the compiler flag option '-I'.

STOP primitive.
TIME channel.
IF on finding none of the conditions TRUE STOPs.

Restrictions

These restrictions are either optional features as described in the published language definition or compiler restrictions unlikely to limit ordinary use of occam.

No configuration section rules.
The operator '>>' uses VAX shift right operator.
No prioritized PAR, all parallel processes have equal priority.
Number of arguments to a procedure limited to 255 maximum.
AFTER returns a time difference not a boolean value.

Extensions

PAR replicator count and base can be variables
A variable number of processes can be created by replicated PAR.

Recursive calls to procedures permitted
A procedure can call itself.

Screen channel can be used by more than one process
The special screen channel can be accessed by any number of different occam processes. This facilitates debugging of occam programs and is not difficult to implement.

Multiple source file compilation
Procedures and Variables can be defined in one file and referenced in another.
The definition is preceded by the new keyword 'LIBRARY' before 'PROC' and the definition must be at the outer level of program nesting.
References to procedures in other files are defined by preceding 'PROC' by 'EXTERNAL' and replacing the '=' start of procedure definition by ':' to indicate end of definition.
e.g.

File main.occ

```
EXTERNAL PROC f (value n) :
SEQ
  f (27)
```

File sub.occ

```
LIBRARY PROC f (value n) =
  SEQ
    num.to.screen (n*102)
    str.to.screen ("Enter next"):
```

The two files can be compiled by :-

```
occam main.occ sub.occ   to compile both together
occam sub.occ -l         to compile sub.occ separately
occam main.occ sub.o     to link in the pre-compiled sub.occ file
```

In 3.0 this has been extended to variables and channels, in the case of vectors of variables and channels the size need not be specified but the type must be :-

Defining file :-

```
LIBRARY CHAN network,comms [56] :
LIBRARY VAR blot [BYTE 4],spot [42] :
LIBRARY VAR FLOAT hyper,bolic [2],active [17] :
```

Referring file :-

```
EXTERNAL CHAN network,comms [] :
EXTERNAL VAR blot [BYTE],spot [],bolic [FLOAT] :
EXTERNAL VAR FLOAT hyper,active [] :
```

Floating point arithmetic

The compiler permits the use of floating point numbers and arithmetic operators. The compiler uses 32 bit VAX floating point throughout.
Floating point numbers are declared by following VAR by the new keyword float :-

```
VAR FLOAT x,y,factor :  — Floating point number declaration
VAR num,ply :           — Normal occam variables.
```

Floating point number constants are supported these may be in two forms with decimal point or with decimal point and exponent :-

```
x := 1.45
y := 2.3e-23 + 3.4e+1  -- Note that the exponent must be given a sign
```

The following operators may be used on floating point numbers (both operands must be floating point)

```
+ - * / < > <= >= = <> - (monadic minus)
```

```
x := 1.3 + (y * factor)
IF
  x > 67.8
    y := -3.4          -- Note use of monadic minus.
```

Parameters to procedures must also have type set to VAR FLOAT or VALUE FLOAT - the actual parameters must be of the same type.

```
PROC sum (VALUE FLOAT a [],b [],VAR FLOAT res [],VALUE n) =
  PAR i = [0 FOR n]
    res [i] := a [i] + b [i] :
VAR FLOAT t [23],s [45],w [32] :
--
sum (t,s,w,12)
```

Floating values may be transmitted along channels - but there are no checks that the sender and receiver both expect floating point values.

Input of floating point numbers can be carried out by calling the library routine 'fp.num.from.keyboard' and output by the routine 'fp.num.to.screen'.

Interconversion of floating point and integers is performed by the assignment operator :-

```
num := x  -- Convert floating 'x' to integer 'num'
y := num  -- Convert integer 'num' to floating 'y'
```

Attempts to use logical and shift operators on floating point numbers are flagged as errors.

APPENDIX III

SELECTED PROGRAM LISTINGS

```
--    program 1 :
--
--    solution of a linear system : occam coding of systolic array.
--
--    The array encoded solves a linear system prsented as a lower
--    triangular n by n band matrix with band width q. Total compu-
--    tation time 2n + q; with an extra cycle to close down the
--    system. Unit time is the cost of execution for an ips cell

--  array dependant parameters and communication channels

def n=4, q=3, total.time = (2*n)+(q+1) :
var xval[n], yval[n], aval[n*n], bval[n] :

chan x[q+1], y[q], a[q], b :

proc cell1( chan xin, xout, yin, yout, ain ) =
  -- inner product cell definiton.
  var a[1], x[1], y[1], xsave, ysave :
  seq
    -- startup values
    seq
      xsave := 0
      ysave := 0
    -- run the cell
    seq i=[ 1 for total.time]
      seq
        par
          -- perform i/o
          xin?x[0]
          yin?y[0]
          ain?a[0]
          xout!xsave
          yout!ysave
        seq
          -- inner product
          ysave := y[0] + (x[0]*a[0])
          xsave := x[0] :

proc cell2( chan xin, xout, yin, bin, ain ) =
  --
  --  solve for x cell.
  --
  var a[1], x[1], y[1], b[1], xsave :
  seq
    xsave := 0
    seq i=[ 1 for total.time ]
      seq
        par
          -- perform i/o
          xin?x[0]
          yin?y[0]
          bin?b[0]
          ain?a[0]
          xout!xsave
        -- compute solution
        xsave := (b[0] / y[0])+a[0] :

proc sourcea( chan outpt, value mem[], z1, z2,  delay1 ) =
  --
  -- matrix values pumped into systolic array by this
  -- control mechanism.
  --
  var toggle, d1, d2, delay2 :
  seq
    -- set starting values;
    seq
      toggle := true          -- dummy or real value switch
      d1 := z1                -- delay of data stream.
      d2 := z2
      if
        d1 > d2
          delay2 := delay1 + (d1 - d2)
        true
          delay2 := delay1 + (d2 - d1)
    -- start pumping
    seq i=[ 1 for total.time ]
      if
        ( i <= delay2 ) or ( d1>n ) or ( d2 > n)
          outpt!0
        true
          if
            toggle
              seq
                -- pump in next value, and
                -- locate next item.
                outpt!mem[(((d1-1)*n)+d2)-1]
                seq
                  d1 := d1 + 1
                  d2 := d2 + 1
                  toggle := false
            true
              seq
                -- pumpin dummy seperator.
                outpt!0
                toggle := true :

proc source( chan out, value memout[], delay ) =
  --
  -- generic source definiton for b,x,y data streams.
  --
  var toggle, j :
  seq
    -- setup values
    toggle := true
    j := 1
    -- pump data until done
    seq i=[ 1 for total.time ]
```

```
    seq
      if
        (i <= delay) or ( j > n)
          -- pass dummy values
          -- for synchronisation.
          seq
            out!0
        toggle
          seq
            -- send memory value
            out!memout[j-1]
            seq
              j := j + 1
              toggle := false
        true
          seq
            -- send dummy seperator
            out!0
            toggle := true :

proc sink( chan in, var memin[], value delay ) =
  --
  --  generic sink : collects garbage values
  --  and stores results in memory.
  --
  var toggle, j, tmp :
  seq
    -- setup values
    seq
      j := 1
      toggle := true
    -- start running
    seq i=[ 1 for total.time]
      seq
        --decide on type of data recieved
        -- e.g garbage or result.
        if
          (i <= delay ) or (j > n)
            -- synchronise
            in?tmp
          toggle
            seq
              -- store a result
              -- next vacant area
              in?memin[j-1]
              seq
                j := j + 1
                toggle := false
          true
            seq
              -- garbage throw away
              in?tmp
              toggle := true :
```

```
proc system( chan a[], x[], y[], b, var xval[], yval[], aval[], bval[] ) =
  --
  -- systolic system definition interms of basic ips cell
  -- and sources.
  --
  par
    par i= [ 1 for q-1 ]
      par
        -- matrix sources and ips cells
        sourcea(a[i], aval, i+1, 1, q-1)
        cell1( x[i-1], x[i], y[i], y[i-1], a[i] )
    -- reciporcal cell
    cell2( x[q], x[0], y[0], b, a[0] )
    -- periphery sources and sinks
    sourcea( a[0], aval, 1, 1, q-1 )
    source( x[q], xval, q-1 )
    source( b, bval, q-1 )
    source( y[q-1], yval, 0 )
    sink( x[q-1], xval, (2*q)-1 ) :

proc getdata( var xval[], yval[], bval[], aval[] ) =
  --
  -- primitive routine to read in data from terminal
  --
  var tmp :
  seq
    screen!'1';'s';'*n'
    -- read in lower triangular matrix
    seq i=[ 1 for n]
      seq
        screen!'*n';'['
        seq j=[ 1 for n]
          seq
            keyboard?tmp
            screen!tmp;'*s'
            aval[(((i-1)*n)+j)-1] := tmp - '0'
        screen!']'
    screen!'*n';'b';'*n';'*n';'['
    -- clear x,y vectors for startup
    seq i=[ 0 for n]
      seq
        keyboard?tmp
        screen!tmp;'*s'
        bval[i] := tmp - '0'
        yval[i] := 0
        xval[i] := 0
    screen!']';'*n';'*n' :

proc putdata( var xval[] ) =
  --
  --  primitive routine to write data to terminal
  --
  seq
    screen!'*n';'x';'['
```

```
    seq i=[ 0 for n]
      screen!xval[i] + '0';'*s'
    screen!']' :

-- main program section

-- performs reading and writing of data to Host computer
-- in this case the user terminal.

-- also creates and starts the system running.
seq
  getdata( xval,yval,bval,aval )
  system( a, x, y, b, xval, yval, aval, bval )
  putdata( xval )
```

```
-- program 2
--
-- Band matrix-vector multiplication
--
-- Systolic array to multiply an n by n band matrix with an
-- n component vector . the matrix has bandwidth  w=p+q-1.

-- problem dependent constants and channels

def n=4,p=2,q=3, w=(p+q)-1 :
def total.time = (2*n)+w :

var xval[n], yval[n], aval[n*n] :
var delay.a, delay.x, delay.y :

chan x[w+1], y[w+1], a[w] :

proc setup =
  --
  -- setup : calculates the delays of inputs
  -- entering the array. Although p and q are constants
  -- if they are modified to create a larger system,
  -- synchronisation delays change
  var xt,yt :
  seq
    -- delay.a computes time cycles
    -- that matrix elements wait
    -- until entering the array
    seq
      xt := p-1
      yt := q-1
      if
        xt > yt
          seq
            -- x has longest distance to travel
            delay.x := 0
            delay.y := xt - yt
            delay.a := xt
        true
          seq
            -- y has longest distance to travel
            delay.y := 0
            delay.x := yt - xt
            delay.a := yt :

proc cell(chan xin,xout, yin,yout, ain ) =
  --
  -- inner product cell
  --
  var a[1],x[1],y[1],xtmp,ytmp :
  seq
    -- dummy values for
    -- startup
    seq
```

```
      xtmp := 0
      ytmp := 0
      -- run the cell
      seq i=[1 for total.time]
        seq
          par
            -- i/o
            xin?x[0]
            yin?y[0]
            ain?a[0]
            xout!xtmp
            yout!ytmp
          seq
            -- perform inner product
            ytmp:= y[0] + (a[0]*x[0])
            xtmp := x[0] :

proc source(chan out ,value mem[], delay) =
  --
  -- generic source.
  --   references area of memory belonging to
  --   host, and pumps required data placed there into
  --   systolic array suitably delayed.
  --
  var j,toggle :
  seq
    -- intialisation
    j := 1
    toggle := true
    -- while running
    seq i=[ 1 for total.time]
      if
        (i <= delay) or (j>n)
          out!0
        toggle
          seq
            -- fetch memory contents
            -- locate next location
            out!mem[j-1]
            j := j + 1
            toggle := false
        true
          seq
            -- dummy spacers
            out!0
            toggle := true :

proc sink( chan in, var mem[], value delay) =
  --
  -- generic sink :
  --   performs data collection; reading out
  --   garbage elements wheqre necessary and placing
  --   valid data back into host memory in correct
  --   position.
  --
  var j,toggle ,tmp :
  seq
    -- intialisation
    j := 1
    toggle := true
    -- while running
    seq i=[1 for total.time]
      if
        ( i <= delay ) or (j > n)
          in?tmp
        toggle
          seq
            -- result to host memory
            in?mem[j-1]
            j := j + 1
            toggle := false
        true
          seq
            -- collect garbage result
            in?tmp
            toggle := true :

proc sourcea(chan outpt,value mem[],z1,z2,delay1 )=
  --
  -- alternative source :
  --   locates ,fetches, and pumps data from band matrix into
  --   the systolic array. Addressing of host memory is more
  --   complex as we need extra delays for matrix inputs
  --
  var delay2,toggle,d1,d2 :
  seq
    -- intialisation
    seq
      toggle := true
      d1 := z1
      d2 := z2
      if
        d1 > d2
          delay2 := d1 - d2
        true
          delay2 := d2 - d1
    -- while running
    seq i=[1 for total.time]
      if
        ( i <= (delay1 + delay2)) or (d1 > n) or (d2 > n)
          -- pad with dummy elements
          -- until data arrives
          outpt!0
        true
          if
            toggle = true
              seq
                -- fetch data, locate next, pump
```

```
            outpt!mem[(((d1-1)*n)+d2)-1]
            seq
              d1 := d1 + 1
              d2 := d2 + 1
              toggle := false
        true
          seq
            -- dummy spacer
            toggle := true
            outpt!0 :

proc alloc.sources.sinks(chan x[], y[], a[], var aval[], yval[],xval[])=
  --
  -- allocation routine : provides the mapping between cells, communication
  -- channels to host environment and systolic array.
  --
  par
    -- create matrix inputs
    par i=[ 1 for w]
      if
        i <= p
          sourcea( a[i-1],aval, 1, (p-i)+1, delay.a)
        true
          sourcea( a[i-1],aval,(i-p)+1,1, delay.a)
    -- x-vector and result y-vector
    -- inputs and outputs
    source(y[w],yval,delay.y)
    source(x[0],xval,delay.x)
    sink(y[0],yval,w+delay.y)
    sink(x[w],xval,w+delay.x) :

proc getdata( var xval[],aval[],yval[] ) =
  --
  -- very primitive input routine : reads in the data from user
  -- terminal inserting it into simulated host memory, in this case
  -- simple arrays
  -- n.b great savings can be made by omitting known zeroes
  --     in the matrix structure this not pursued for simplicity
  --     the main aim was to test the array; not an exercise in data
  --     structuring.
  --
  var tmp :
  seq
    screen!'b';'a';'n';'d';'=';w+'0';'*s';'p';'=';'p+'0';'*s';'q';'=';q+'0';'*n'
    screen!'b';'a';'n';'d';'=';w+'0';'*s';'p';'=';p+'0';'*s';'q';'=';q+'0';'*n'
    screen!'n';'=';n+'0';'*n'
    -- read the band matrix
    seq i=[1 for n]
      seq
        screen!'*n';'['
        seq j=[ 1 for n]
          seq
            keyboard!tmp;'*s'
            aval[(((i-1)*n)+j)-1] := tmp - '0'
        screen!']'
    screen!'*n';'x';'*n';'['
    -- read x and clear y vectors
    seq i=[0 for n]
      seq
        keyboard?tmp
        screen!tmp;'*s'
        xval[i] := tmp - '0'
        yval[i] := 0
    screen!']';'*n' :

proc putdata( var yval[]) =
  --
  -- another primitive i/o routine this time to
  -- output data. Note; both input and output deal only
  -- with single digit values; restrictive but adequate
  -- for test purposes.
  seq
    screen!'*n';'y';'*n';'['
    seq i=[0 for n]
      screen!yval[i]+'0';'*s'
    screen!']' :

-- main

-- allocation and setup of the system;

seq
  setup
  getdata(xval,aval,yval)
  -- the array
  par
    alloc.sources.sinks(x,y,a,aval,yval,xval)
    par i=[1 for w]
      cell(x[i-1],x[i],y[i],y[i-1],a[i-1])
  -- computation complete
  putdata(yval)
```

```
-- program 3
--
--  band matrix multiplier
--
--  program to multiply two n by n band matrices
--  using a hex connected systolic array .
--
--  matrices have band widths w1 = p1 + q1 - 1 and w2 = p2 + q2 -1
--

--  problem dependant constants

def n=4, p1=2, q1=3, w1=(p1+q1)-1 :
def p2=3,q2=2, w2= (p2+q2)-1 :
def maxchan = (((w1+1)*(w2+1))*3)+3, w3 = (w1 + w2) -1  :
def don =true, doff = false :

var delay.a, delay.b, delay.c, total.time, min, strtc :
var a[n*n], b[n*n], c[n*n] :

chan pool[maxchan] :

proc setup( var c[] )  =
  --
  -- setup routine ; calculates delaysof input data for
  -- synchronisation  inside the array ; and total computation
  -- time.
  -- also sets some varibles e.g strtc required to setup the
  -- connection network.
  var ctime :
  seq
    -- computation time
    seq i=[ 1 for n*n]
      c[i-1] := 0
    if
      w1 < w2
        seq
          total.time := (3*n)+w1
          min := w1
      true
        seq
          total.time := ( 3*n ) + w2
          min := w2
    -- offsets and delays
    strtc := w2 + 1
    ctime := q1-1
    if
      q1 > p2
        seq
          strtc := strtc + (q1 - p2)
          ctime := ctime - (q1 - p2)
    -- delays of sources
    if
     (ctime >= (q2-1)) and ( ctime >= (p1 - 1))
        seq
          delay.c := 0
          delay.a := ctime - (q2 - 1)
          delay.b := ctime - (p1 - 1)
      ( (q2 - 1) >= ctime ) and (q2 >= p1 )
        seq
          delay.a := 0
          delay.c := (q2-1 ) - ctime
          delay.b := q2 - q1
      ( (p1 - 1) >= ctime ) and ((p1-1) > (q2-1))
        seq
          delay.b := 0
          delay.c := (p1 - 1) - ctime
          delay.a := p1 - q2 :

proc allocpool( value x, y, r, var c ) =
  --
  -- the array is built onto a rectangular grid; with processors
  -- at intersections; channels are connected by indexing
  -- processors by underlying grid points
  --
  seq
    c := (((((x - 1) * (w2 + 1)) + y) * 3) + r)-1 :

proc cell( chan ain, bin, cin, aout, bout, cout ) =
  --
  -- inner product cell
  -- ( hex in the array)
  --
  var a[1], b[1], c[1], atmp, btmp, ctmp :
  seq
    -- intialisation
    seq
      atmp := 0
      btmp := 0
      ctmp := 0
    -- while running
    seq i=[1 for total.time]
      seq
        par
          -- i/0
          ain?a[0]
          bin?b[0]
          cin?c[0]
          aout!atmp
          bout!btmp
          cout!ctmp
        seq
          -- inner product
          ctmp := c[0] + (a[0]*b[0])
          btmp := b[0]
          atmp := a[0] :

proc source( value mem[], z1, z2, delay, flag, chan outpt ) =
```

```
--
--  generic source
--    locates, fetches , and pumps data into the array
--    for simplicity c is assummed to be a zero array
--    at start of computation.
--
var toggle, toggle1, delay2, d1, d2 :
seq
  -- intialisation; and computation
  -- of source delays
  toggle := true
  toggle1 := false
  d1 := z1
  d2 := z2
  if
    d1 > d2
      delay2 := (d1 - d2)
    true
      delay2 := (d2 - d1)
  if
    flag
      delay2 := (delay2*2) + delay
    true
      delay2 := delay2 + delay
  -- while running
  seq i=[ 1 for total.time ]
    if
      (i <= delay2) or (d1 > n) or ( d2 > n)
        outpt!0
      true
        if
          toggle1
            seq
              -- sceond dummy value
              outpt!0
              toggle := true
              toggle1 := false
          toggle
            seq
              -- fetch , locate next and pump data
              outpt!mem[(((d1-1)*n)+d2)-1]
              seq
                d1 := d1 + 1
                d2 := d2 + 1
                toggle := false
          true
            seq
              -- first dummy value
              outpt!0
              toggle1 := true :

proc sink( var mem[], value  z1, z2, delay, flag, extra, chan inpt) =
  --
  -- generic sink;
  --  almost identical source except that data is
  --  collected and inserted into memory as results
  --  garbage values are removed from the array
  --
  var toggle, toggle1 , delay2, d1, d2 :
  seq
    -- intialisation
    toggle := true
    toggle1 := false
    d1 := z1
    d2 := z2
    if
      d1 > d2
        delay2:= (d1 - d2)
      true
        delay2 := d2 - d1
    if
      flag
        delay2 := (delay2 * 2) + (delay + extra)
      true
        delay2 := delay2 + ( delay + extra )
    -- while running
    seq i=[ 1 for total.time  ]
      if
        ( i <= delay2) or (d1 > n) or (d2 > n)
          inpt?any
        true
          if
            toggle1
              seq
                -- dummy value
                inpt?any
                toggle := true
                toggle1 := false
            toggle
              seq
                -- insert result into memory
                inpt?mem[(((d1-1)*n)+d2)-1]
                seq
                  d1 := d1 + 1
                  d2 := d2 + 1
                  toggle := false
            true
              seq
                -- dummy value
                inpt?any
                toggle1 := true :

proc allocsources( chan pool[], value strtc ) =
  --
  -- allocation of sources to
  -- ends of systolic array
  --
  def xlim = w1 + 1, ylim = w2 + 1 :
```

```
  par
    -- allocate a(i,j) sources
    par x=[1 for w1]
      var idx :
      seq
        allocpool(x+1, ylim, 0, idx)
        if
          x <= q1
            source( a, q1 - (x - 1), 1, delay.a, doff, pool[idx])
          true
            source( a, 1, (x - q1) + 1, delay.a, don, pool[idx])
    -- allocate b(i,j) sources
    par y = [ 1 for w2]
      var idx :
      seq
        allocpool(xlim, y+1, 2, idx)
        if
          y <= p2
            source( b, 1, p2 - (y - 1) , delay.b, doff, pool[idx])
          true
            source( b, (y - p2) + 1, 1, delay.b, don, pool[idx])
    -- allocate c(i,j) sources
    par i=[ 1 for w3]
      var  idx :
      seq
        if
          i <= (w2 - 1)
            allocpool( 1, (w2 - i) + 1, 1, idx )
          i = w2
            allocpool( 1, 1, 1, idx )
          i > w2
            allocpool((i - w2) + 1, 1, 1, idx )
        if
           ((strtc - i)+1) > 1
             seq
               source( c, strtc - i, 1, delay.c, don, pool[idx])
           true
             seq
               source( c, 1, (i - strtc) + 2, delay.c, don, pool[idx]) :

proc allocsinks( chan pool[], value strtc ) =
  --
  -- allocation of sinks to periphery of array
  -- almost the same as the source allocations
  -- and could make one routine but the lack of clarity
  -- in the latter proved a heavy cost in debugging
  --
  def xlim = w1 + 1, ylim = w2 + 1 :
  par
    -- allocate a(i,j) sinks
    par x=[ 1 for w1]
      var idx :
      seq
        allocpool( x+1, 1, 0, idx)
        if
          ( x <= q1)
            sink( a, q1-(x-1), 1, delay.a, doff, 0, pool[idx])
          true
            sink( a, 1, (x - q1) + 1, delay.a, don, 0, pool[idx])
    -- allocate b(i,j) sinks
    par y=[ 1 for w2]
      var idx :
      seq
        allocpool(1, y+1, 2, idx )
        if
          ( y <= p2)
            sink( b, 1, p2 - ( y - 1), delay.b, doff, 0, pool[idx])
          true
            sink( b, (y-p2)+1, 1, delay.b, don, 0, pool[idx])
    -- allocate c(i,j) sinks
    par i=[ 1 for w3]
      var idx, tmp :
      seq
        if
          i <= (w1 - 1)
            allocpool( i+1, ylim, 1, idx )
          i = w1
            allocpool( xlim, ylim, 1, idx )
          i > w1
            allocpool( xlim, ((w3 - i)+2) , 1, idx )
        if
          (( strtc - i) + 1) > 1
            seq
              sink( c, strtc - i, 1, delay.c, don, i, pool[idx])
          true
            seq
              if
                i > w2
                  tmp := (w3 - i) + 1
                true
                  tmp := min
              sink( c, 1, (i - strtc)+ 2, delay.c, don, tmp, pool[idx]) :

proc alloc.cells(chan  pool[]) =
  --
  -- placement of inner product cells
  -- onto the rectangular grid, and connection with
  -- sources and immediate neighbours
  --
  par i = [ 2 for w1]
    par j = [ 2 for w2]
      var id1, id2, id3, id4, id5, id6 :
      seq
        allocpool(i, j, 0, id1 )
        allocpool(i, j, 2, id2 )
        allocpool(i, j, 1, id3 )
        allocpool(i, j-1, 0, id4)
        allocpool(i-1, j, 2, id5)
```

```
        allocpool(i-1, j-1, 1, id6)
        cell(pool[id1], pool[id2], pool[id6], pool[id4], pool[id5], pool[id3] )
:

proc getdata( var mem[] ) =
    --
    -- primitive input routine toread single digit postive
    -- numbers ; restrictive but good enough for test purposes
    --
  var tmp :
  seq
    screen!'*n'
    seq i= [ 1 for n ]
      seq
        screen!'*n';'['
        seq j=[ 1 for n]
          seq
            keyboard?tmp
            screen!tmp;'*s'
            mem[(((i-1)*n)+j)-1] := tmp - '0'
        screen!']'
    screen!'*n' :

proc putdata( value mem[] ) =
  --
  -- primitive routine to output data
  --
  seq
    seq i =[ 1 for n ]
      seq
        screen!'*n';'['
        seq j=[ 1 for n]
          screen!mem[(((i-1)*n)+j)-1] + '0';'*s'
        screen!']' :

-- main routine

-- allcates and creates running array

seq
  getdata( a )
  getdata( b )
  setup( c )
  par
    allocsources(pool, strtc)
    alloc.cells(pool)
    allocsinks(pool, strtc)
  putdata( c )
```

```
-- program 4.
--
-- Systolic array : performing lu-decompostion of an n*n
--                  Matrix
--

-- notes: This program is based on a simulation or soft-systolic
-- implementation of the Hexagonal array presented by h.t. kung
-- and c.e. liserson .
-- esssentially the program is the same as program 3 for
-- band matrix multiplication. the inner product step (ips) cell
-- has been made programmable, in the sense that the various
-- orientations e.g  120 deg rotations can be selected

-- problem dependant constants

def n=4, p1=2, q1=3, w1=(p1+q1)-1 :
def p2=3,q2=2, w2= (p2+q2)-1 :
def maxchan = (((w1+1)*(w2+1))*3)+3, w3 = (w1 + w2) -1  :
def don =true, doff = false :

-- synchronisation and data storage

var delay.a, delay.b, delay.c, total.time, min, strtc :
var a[n*n], b[n*n], c[n*n] :

-- pool of communication channels

chan pool[maxchan] :

proc setup( var c[], d[] ) =
  --
  -- routine to perform necessary calculations for
  -- delay of data on input channels, and the
  -- specification of the array as virtual processors
  -- on a rectangular processing surface.
  --

  var ctime :
  seq
    -- clear vectors
    seq i=[ 1 for n*n]
      seq
        d[i-1] := 0
        c[i-1] := 0
    -- computation time
    if
      w1 < w2
        seq
          total.time := (3*n)+w1
          min := w1
      true
        seq
          total.time := ( 3*n ) + w2
```

```
        min := w2
    -- offsets and delays
    strtc := w2 + 1
    ctime := q1-1
    if
      q1 > p2
        seq
          strtc := strtc + (q1 - p2)
          ctime := ctime - (q1 - p2)
    -- delays of sources
    if
     (ctime >= (q2-1)) and ( ctime >= (p1 - 1))
        seq
          delay.c := 0
          delay.a := ctime - (q2 - 1)
          delay.b := ctime - (p1 - 1)
      ( (q2 - 1) >= ctime ) and (q2 >= p1 )
        seq
          delay.a := 0
          delay.c := (q2-1 ) - ctime
          delay.b := q2 - q1
      ( (p1 - 1) >= ctime ) and ((p1-1) > (q2-1))
        seq
          delay.b := 0
          delay.c := (p1 - 1) - ctime
          delay.a := p1 - q2 :

proc allocpool( value x, y, r, var c ) =
  --
  -- address calculation of pool channel
  --
  seq
    c := (((((x - 1) * (w2 + 1)) + y) * 3) + r)-1 :

proc cell( chan ain, bin, cin, aout, bout, cout, value type ) =
  --
  -- definition of inner product cell
  --
  -- the cell is programmable ; the type selects if the cell
  --  is true ips, rotated by 120 deg clock wise ,anticlockwise
  --  or reciprocal cell
  --
  var a[1], b[1], c[1], atmp, btmp, ctmp :
  seq
    seq
      -- setup start values
      atmp := 0
      btmp := 0
      ctmp := 0
      if
        type = 3
          seq
            atmp := -1
            btmp :=  1
    -- while running
    seq i=[1 for total.time]
      seq
        -- perform i/o
        par
          ain?a[0]
          bin?b[0]
          cin?c[0]
          aout!atmp
          bout!btmp
          cout!ctmp
        -- do computation
        -- depending on type of cell
        seq
          if
            (type = 0) or (type = 2)
               seq
                 ctmp := c[0] + ( a[0]*b[0] )
                 atmp := a[0]
                 btmp := b[0]
            type = 1
              seq
                ctmp := c[0] + ( a[0]*b[0] )
                btmp := ctmp
                atmp := a[0]
            type = 3
              seq
                if
                  c[0] = 0
                    btmp := 1
                  true
                    btmp := 1/c[0]
                atmp := -1
                ctmp := c[0]  :

proc source( value mem[], z1, z2, delay, flag, chan outpt ) =
  --
  -- generic source
  --         pumps data into the array, zero values except
  --         where matrix values are to enter
  --
  var toggle, toggle1, delay2, d1, d2 :
  seq
    -- set switches
    -- calculate number of
    -- dummy inputs
    toggle := true
    toggle1 := false
    d1 := z1
    d2 := z2
    if
      d1 > d2
        delay2 := (d1 - d2)
      true
```

```
      delay2 := (d2 - d1)
    if
      flag
        delay2 := (delay2*2) + delay
      true
        delay2 := delay2 + delay
    -- while running
    seq i=[ 1 for total.time ]
      if
        (i <= delay2) or (d1 > n) or ( d2 > n)
          -- pad with dummy value
          outpt!0
        true
          if
            toggle1
              seq
                -- send dummy value
                outpt!0
                toggle := true
                toggle1 := false
            toggle
              seq
                -- send data element
                outpt!mem[(((d1-1)*n)+d2)-1]
                seq
                  d1 := d1 + 1
                  d2 := d2 + 1
                  toggle := false
            true
              seq
                -- send dummy value
                outpt!0
                toggle1 := true :

proc sink( var mem[], value  z1, z2, delay, flag, extra, chan inpt) =
  --
  -- generic sink
  --          opposite of the generic source; collects l and u factors
  --          results storing them in the correct place in host memory
  --          dummy values and garbage results are disposed of cleanly
  --
  var toggle, toggle1 , delay2, d1, d2 :
  seq
    -- set switches
    toggle := true
    toggle1 := false
    d1 := z1
    d2 := z2
    if
      d1 > d2
        delay2:= (d1 - d2)
      true
        delay2 := d2 - d1
    if
      flag
        delay2 := (delay2 * 2) + (delay + extra)
      true
        delay2 := delay2 + ( delay + extra )
    -- while running do computation
    seq i=[ 1 for total.time  ]
      if
        ( i <= delay2) or (d1 > n) or (d2 > n)
          -- pad dummy values
          inpt?any
        true
          if
            toggle1
              seq
                -- dummy or garbage
                inpt?any
                toggle := true
                toggle1 := false
            toggle
              seq
                -- store this value; and
                -- locate next free space
                inpt?mem[(((d1-1)*n)+d2)-1]
                seq
                  d1 := d1 + 1
                  d2 := d2 + 1
                  toggle := false
            true
              seq
                -- dummy or garbage value
                inpt?any
                toggle1 := true :

proc allocsources( chan pool[], value strtc ) =
  --
  -- assign source processors to grid positions
  --
  def xlim = w1 + 1, ylim = w2 + 1 :
  par
    -- allocate a(i,j) sources
    par x=[1 for w1]
      var idx :
      seq
        allocpool(x+1, ylim, 0, idx)
        if
          x <= q1
            source( a, q1 - (x - 1), 1, delay.a, doff, pool[idx])
          true
            source( a, 1, (x - q1) + 1, delay.a, don, pool[idx])
    -- allocate b(i,j) sources
    par y = [ 1 for w2]
      var idx :
      seq
        allocpool(xlim, y+1, 2, idx)
```

```
      if
        y <= p2
          source( b, 1, p2 - (y - 1) , delay.b, doff, pool[idx])
        true
          source( b, (y - p2) + 1, 1, delay.b, don, pool[idx])
    -- allocate c(i,j) sources
    par i=[ 1 for w3]
      var  idx :
      seq
        if
          i <= (w2 - 1)
            allocpool( 1, (w2 - i) + 1, 1, idx )
          i = w2
            allocpool( 1, 1, 1, idx )
          i > w2
            allocpool((i - w2) + 1, 1, 1, idx )
        if
           ((strtc - i)+1) > 1
             seq
               source( c, strtc - i, 1, delay.c, don, pool[idx])
           true
             seq
               source( c, 1, (i - strtc) + 2, delay.c, don, pool[idx]) :

proc allocsinks( chan pool[], value strtc ) =
  --
  -- allocate sink processors to grid positions
  --
 def xlim = w1 + 1, ylim = w2 + 1 :
 par
   -- allocate a(i,j) sinks
   par x=[ 1 for w1]
     var idx :
     seq
       allocpool( x+1, 1, 0, idx)
       if
         ( x <= q1)
           sink( a, q1-(x-1), 1, delay.a, doff, 0, pool[idx])
         true
           sink( a, 1, (x - q1) + 1, delay.a, don, 0, pool[idx])
   -- allocate b(i,j) sinks
   par y=[ 1 for w2]
     var idx :
     seq
       allocpool(1, y+1, 2, idx )
       if
         ( y <= p2)
           sink( b, 1, p2 - ( y - 1), delay.b, doff, 0, pool[idx])
         true
           sink( b, (y-p2)+1, 1, delay.b, don, 0, pool[idx])
   -- allocate c(i,j) sinks
   par i=[ 1 for w3]
     var idx, tmp :
     seq
       if
         i <= (w1 - 1)
           allocpool( i+1, ylim, 1, idx )
         i = w1
           allocpool( xlim, ylim, 1, idx )
         i > w1
           allocpool( xlim, ((w3 - i)+2) , 1, idx )
       if
         (( strtc - i) + 1) > 1
           seq
             sink( c, strtc - i, 1, delay.c, don, i, pool[idx])
         true
           seq
             if
               i > w2
                 tmp := (w3 - i) + 1
               true
                 tmp := min
             sink( c, 1, (i - strtc)+ 2, delay.c, don, tmp, pool[idx]) :

proc alloc.cells(chan  pool[]) =
  --
  -- allocate ips processors to grid positions
  --
 def xlim = w1 + 1, ylim = w2 + 1 :
 par i = [ 2 for w1]
   par j = [ 2 for w2]
     var id1, id2, id3, id4, id5, id6 :
     seq
       allocpool(i, j, 0, id1 )
       allocpool(i, j, 2, id2 )
       allocpool(i, j, 1, id3 )
       allocpool(i, j-1, 0, id4)
       allocpool(i-1, j, 2, id5)
       allocpool(i-1, j-1, 1, id6)
       if
         (i = xlim ) and ( j = ylim)
           cell(pool[id1], pool[id2], pool[id6], pool[id4], pool[id5],
                pool[id3], 3)
         (j = ylim )
           cell(pool[id2], pool[id6], pool[id1], pool[id5], pool[id3],
                pool[id4], 1)
         (i = xlim)
           cell(pool[id6], pool[id1], pool[id2], pool[id3], pool[id4],
                pool[id5], 2)
         true
           cell(pool[id1], pool[id2], pool[id6], pool[id4], pool[id5],
                pool[id3], 0) :

proc getdata( var mem[] ) =
  --
  -- read in the array to be factored
  --     primtive routine integer values >0 and <=9
  --     but adequate for testing.
```

```
  --
  var tmp :
  seq
    screen!'*n'
    seq i= [ 1 for n ]
      seq
        screen!'*n';'['
        seq j=[ 1 for n]
          seq
            keyboard?tmp
            screen!tmp;'*s'
            mem[(((i-1)*n)+j)-1] := tmp - '0'
        screen!']'
    screen!'*n' :

proc putdata( value mem[] , value flag ) =
  --
  -- write out the factors to the screen
  --
  -- primitive routine uses only integer values
  -- but sufficient for testing
  --
  seq
    seq i =[ 1 for n ]
      seq
        screen!'*n';'['
        seq j=[ 1 for n]
          seq
            if
              (i > j) and (flag = 1)
                screen!'0';'*s'
              (i < j) and (flag = 2)
                screen!'0';'*s'
              ( i = j) and ( flag = 2)
                screen!'1';'*s'
              true
                screen!mem[(((i-1)*n)+j)-1] + '0';'*s'
        screen!']' :

-- main program

-- read data; allocate system; run it; write results
seq
  getdata( c )
  setup( a, b )
  par
    allocsources(pool, strtc)
    alloc.cells(pool)
    allocsinks(pool, strtc)
  screen!'u';'=';'*n'
  putdata( c, 1)
  screen!'*n';'1';'=';'*n'
  putdata( c, 2 )
```

```
-- program 5.

-- systolic array : Double pipe implementation of Matrix Vector
--                  Multiplication algorithm.
--
-- notes          : The Array is based on the Soft-Systolic approach
--                  to simulating systolic arrays.

-- problem dependent constants
def n=5, p=4, q=4, w=(p+q)-1, max.cells = (w/2)+1, total.time = (n+(w/2)+3) :

-- setup and communication variables
var delay.x, delay.y, no.cells.1, no.cells.2 :
var x.vec[ n ], a.mem[ n*n ], y.vec[ n ], null.vec[ n ] :

-- Interface and Floating point Arithmetic Library Routines.
EXTERNAL proc fp.float( value int, var float ) :
EXTERNAL proc fp.add( value f1, f2, var f3 ) :
EXTERNAL proc fp.mult(value f1, f2, var f3 ) :
EXTERNAL proc fp.num.to.screen( value f ) :
EXTERNAL proc fp.num.from.keyboard(var f) :
EXTERNAL proc str.to.screen( value s[] ) :

proc setup =
  --
  -- calculation of synchronisation parameters
  -- and number of cells in each pipe.
  --
  seq
    if
      (p - (2*(p/2))) = 0
        delay.x := (p/2) -1
      true
        delay.x := p/2
    if
      (q-(2*(q/2))) = 0
        delay.y := (q/2) - 1
      true
        delay.y := q/2
    if
      (w -(2*(w/2))) = 0
        seq
          no.cells.1 := w/2
          no.cells.2 := w/2
      true
        seq
          no.cells.1 := (w/2) + 1
          no.cells.2 := w/2
    if
      delay.x > delay.y
        seq
          delay.y := delay.x - delay.y
          delay.x := 0
      true
```

```
        seq
          delay.x := delay.y - delay.x
          delay.y := 0 :

proc ips ( chan xin, yin, xout, yout ,ain ) =
  --
  -- basic inner product step cell
  --
  var a[1], x[1], y[1], xtmp, ytmp, tmp :
  seq
    -- startup values
    fp.float(0,xtmp)
    fp.float(0,ytmp)
    fp.float(0,a[0])
    fp.float(0,x[0])
    fp.float(0,y[0])
    -- while running do
    seq i =[ 1 for total.time]
      seq
        -- i/o
        par
          xin?x[0]
          yin?y[0]
          ain?a[0]
          xout!xtmp
          yout!ytmp
        -- computation
        seq
          fp.mult(x[0],a[0],tmp)
          fp.add(y[0], tmp, ytmp)
          xtmp := x[0] :

proc delay( chan in, out ) =
  --
  -- simple one cycle delay cell
  --
  var tmp[1], t:
  seq
    fp.float(0,t)
    seq i =[1 for total.time]
      seq
        par
          in?tmp[0]
          out!t
        t  := tmp[0] :

proc adder ( chan in1, in2, res ) =
  --
  -- two input adder
  --
  var opd1[1], opd2[1], c :
  seq
    fp.float(0,c)
    seq i =[1 for total.time]
      seq
        par
          in1?opd1[0]
          in2?opd2[0]
          res!c
        fp.add(opd1[0], opd2[0], c) :

proc store(chan res, var result[], value delay ) =
  --
  -- Collection and Storage of Data
  --
  var j , tmp :
  seq
    j := 1
    seq i =[1 for total.time]
      if
        (i <= delay) or ( j > n)
          seq
            res?tmp
        true
          seq
            res?result[j-1]
            j := j + 1 :

proc source ( value type, mem[], i1, i2, delay, chan out ) =
  --
  -- Generic Source : pump data into array
  --
  var d1, d2 , delay2 :
  seq
    d1 := i1
    d2 := i2
    delay2 := delay
    if
      d2 > d1
        delay2 := (d2 - 1) + delay2
    seq i =[1 for total.time]
      seq
        if
          (d1 > n) or (d2 > n) or ( i <= delay2 )
            -- shift
            out!0
          type = 1
            -- Matrix Source
            seq
              out!mem[(((d1-1)*n)+d2)-1]
              d1 := d1 + 1
              d2 := d2 + 1
          type = 2
            -- Vector source
            seq
              out!mem[d1-1]
              d1 := d1 + 1 :
```

```
proc sink ( chan in ) =
  -- garbage signal collector
  seq i =[1 for total.time]
    in?any :

proc alloc.sources(  value pipe, size, mem.mat[], chan a[] ) =
  --
  -- allocation of Sources to grid points in
  -- Virtual processing space
  --
  var k, delay :
  seq
    delay := delay.x
    if
      pipe = 1
        k := p
      pipe = 2
        k := p - 1
    -- make enough sources for
    -- all cells in pipe
    par i =[ 1 for size]
      var delay2, k1:
      seq
        k1 := k - (2*(i-1))
        delay2:= delay + (i - 1)
        if
          k1 <= 0
            source( 1, mem.mat, 2-k1, 1, delay2, a[i-1] )
          true
            source( 1, mem.mat, 1, k1, delay2 , a[i-1] ) :


proc bandvec ( var mem.mat[], mem.vec[], mem.null[], value size,
                              delay.1, delay.2, pipe, chan yout ) =
  --
  -- Abstract definition of the Single pipe band vector array
  -- recall that two such vectors are required for Double pipe
  --
  chan x[max.cells+1], y[max.cells+1], a[max.cells] :

  par
    -- pipe
    ips( x[0], y[1], x[1], yout, a[0] )
    par i=[ 2 for size -1 ]
      ips( x[i-1], y[i], x[i], y[i-1], a[i-1] )
    -- matrix and vector sources
    alloc.sources( pipe, size, mem.mat, a )
    source(2, mem.vec, 1, 0, delay.x, x[0] )
    source(2, mem.null, 1,0, delay.y, y[size] )
    -- collection of garbage falling off the pipe ends
    sink( x[size] ) :

proc getdata( var mat[], vec[], res[], null[] ) =
  --
  -- read matrix and vector data from terminal/Host
  --
  var tmp :
  seq
    str.to.screen("*n band matrix multiplier *n")
    str.to.screen("*n*n using double systolic pipe *n")
    str.to.screen("*n*nband width w = p + q - 1")
    str.to.screen("*np = ")
    str.to.screen("*nq = ")
    str.to.screen("*n enter matrix ")
    seq i =[ 1 for n]
      seq
        str.to.screen("*n[")
        seq j =[1 for n]
          seq
            fp.num.from.keyboard(tmp)
            str.to.screen("*s")
            fp.num.to.screen(tmp)
            mat[(((i-1)*n)+j)-1] := tmp
        str.to.screen("]*n")
    str.to.screen("*n*n enter vector *n[*s")
    seq i =[1 for n]
      seq
        -- clear auxiliary vectors here
        res[i-1] := 0
        null[i-1] := 0
        fp.num.from.keyboard(tmp)
        str.to.screen("*s")
        fp.num.to.screen(tmp)
        vec[i-1] := tmp
    str.to.screen("]*n") :

proc putdata( value vec[] ) =
  --
  -- read out results to Host/Terminal
  --
  seq
    str.to.screen("*n*n*n result vector *n[")
    seq i=[1 for n]
      seq
        str.to.screen("*s")
        fp.num.to.screen(vec[i-1])
    str.to.screen("*s]*n") :

-- main
chan yout1, yout2, d.yout2, res :

seq
  setup
  getdata( a.mem, x.vec, y.vec, null.vec)
  -- Double pipe
  par
    bandvec( a.mem, x.vec, null.vec, no.cells.1, delay.x,
                        delay.y, 1, yout1 )
```

```
  bandvec( a.mem, x.vec, null.vec, no.cells.2, delay.x,
                   delay.y, 2, yout2 )
  delay( yout2, d.yout2)
  adder( yout1, d.yout2, res )
  store( res, y.vec,( no.cells.1 + 1) + delay.y )
putdata( y.vec )
```

```
-- program 6.

-- systolic Array : for Quadrant Interlocking Iterative (XWZ) scheme
--                  for the soluiton of Linear Systems.
--
-- Notes: The method is a parallel implementation using a Double Pipe Array
--        no Over relaxation is used, as we cannot gurantee that every
--        instance of the array will be able to use it.

-- problem dependent constants
def n=4, n2=n/2,  p=n2, q=n2, w=(p+q)-1, max.cells = (w/2)+1 :
def m = 12 ,total.time = (n2+(w/2)+5) :

-- Vector and Matrix storage
var delay.x, delay.y, no.cells.1, no.cells.2 :
var x1[n2], x2[n2], null.vec[n2], xc1[n2], xc2[n2], xc3[n2], xc4[n2] :
var p11[n2*n2], p12[n2*n2], p21[n2*n2], p22[n2*n2] :
var b1[n2], b2[n2] :

-- Host Interface and Floating point Library routines
EXTERNAL proc fp.float( value int, var float ) :
EXTERNAL proc fp.add( value f1, f2, var f3 ) :
EXTERNAL proc fp.sub( value f1, f2, var f3 ) :
EXTERNAL proc fp.div( value f1, f2, var f3 ) :
EXTERNAL proc fp.mult(value f1, f2, var f3 ) :
EXTERNAL proc fp.num.to.screen( value f ) :
EXTERNAL proc fp.num.from.keyboard(var f) :
EXTERNAL proc str.to.screen( value s[] ) :

proc setup =
  --
  -- calculation of synchronisation and
  -- array pipeline positioning parameters
  --
  seq
    if
      (p - (2*(p/2))) = 0
        delay.x := (p/2) -1
      true
        delay.x := p/2
    if
      (q-(2*(q/2))) = 0
        delay.y := (q/2) - 1
      true
        delay.y := q/2
    if
      (w -(2*(w/2))) = 0
        seq
          no.cells.1 := w/2
          no.cells.2 := w/2
      true
        seq
          no.cells.1 := (w/2) + 1
          no.cells.2 := w/2
```

```
    if
      delay.x > delay.y
        seq
          delay.y := delay.x - delay.y
          delay.x := 0
      true
        seq
          delay.x := delay.y - delay.x
          delay.y := 0 :

proc ips ( chan xin, yin, xout, yout ,ain ) =
  --
  -- basic Inner product Cell
  --
  var a[1], x[1], y[1], xtmp, ytmp, tmp :
  seq
    -- intialisation
    fp.float(0,xtmp)
    fp.float(0,ytmp)
    fp.float(0,a[0])
    fp.float(0,x[0])
    fp.float(0,y[0])
    -- while running do
    seq i =[ 1 for total.time]
      seq
        -- i/o
        par
          xin?x[0]
          yin?y[0]
          ain?a[0]
          xout!xtmp
          yout!ytmp
        -- computation
        seq
          fp.mult(x[0],a[0],tmp)
          fp.sub(y[0], tmp, ytmp)
          xtmp := x[0] :

proc delay( chan in, out ) =
  --
  -- single delay cycle
  --
  var tmp[1], t:
  seq
    fp.float(0,t)
    seq i =[1 for total.time]
      seq
        par
          in?tmp[0]
          out!t
        t  := tmp[0] :

proc ddivide( chan in1, in2, in3, out1, out2 )=
  --
  -- Double Division cell
  --  out1 = in1/in3, out2 = in2/in3
  --
  var te[3], e1, e2 :
  seq
    --  intialise
    fp.float(0, e1)
    fp.float(0, e2)
    -- while running do
    seq i =[ 1 for total.time]
      seq
        -- i/o
        par
          in1?te[0]
          in2?te[1]
          in3?te[2]
          out1!e1
          out2!e2
        -- divide by zero test
        if
          te[2] = 0
            -- default result
            par
              e1 := 1
              e2 := 1
          true
            par
              fp.div( te[0], te[2], e1 )
              fp.div( te[1], te[2], e2 ) :

proc determinant( chan in1, in2, in3, in4, out1, out2, out3,
                        out4, out5, out6, value type ) =
  --
  -- determinant cell : calculates determinant of 2*2 system
  --                    outputs two copies of result and operands
  --
  var te[4], e[5], tmp1, tmp2 :
  seq
    -- intialise
    seq i =[ 0 for 4]
      fp.float(0, e[i] )
    fp.float(1, e[4] )
    -- while running do
    seq i =[1 for total.time]
      seq
        -- i/o
        par
          in1?te[0]
          in2?te[1]
          in3?te[2]
          in4?te[3]
          if
            type = 0
              -- pass operands & copy result
```

```
            par
              out1!e[0]
              out2!e[1]
              out3!e[2]
              out4!e[3]
              out5!e[4]
        out6!e[4]
      -- computation
      par
        par i =[ 0 for 4]
          e[i] := te[i]
        fp.mult( te[0], te[1], tmp1 )
        fp.mult( te[2], te[3], tmp2 )
      fp.sub( tmp1, tmp2, e[4] ) :

proc adder ( chan in1, in2, in3, in4, out1, out2 ) =
  --
  -- 4-input adder: delivers two copies of result
  --
  var te[5], e1 :
  seq
    fp.float(0,e1)
    seq i =[1 for total.time]
      seq
        par
          in1?te[0]
          in2?te[1]
          in3?te[2]
          in4?te[3]
          out1!e1
          out2!e1
        par
          fp.add(te[0], te[1], e1)
          fp.add(te[2], te[3], te[4])
        fp.add(e1, te[4], e1) :

proc store(chan res, var result[], value delay ) =
  --
  -- store result vector
  --
  var j , tmp :
  seq
    j := 1
    seq i =[1 for total.time]
      if
        (i <= delay) or ( j > n2)
          seq
            res?tmp
        true
          seq
            res?result[j-1]
            j := j + 1 :

proc source ( value type, mem[], i1, i2, delay, chan out ) =
  --
  -- Generic Source : pumps data into array
  --
  var d1, d2 , delay2 :
  seq
    d1 := i1
    d2 := i2
    delay2 := delay
    -- compute overall delay(shift)
    if
      d2 > d1
        delay2 := (d2 - 1) + delay2
    seq i =[1 for total.time]
      seq
        if
          (d1 > n2) or (d2 > n2) or ( i <= delay2 )
            -- shift
            out!0
          type = 1
            -- Matrix Source
            seq
              out!mem[(((d1-1)*n2)+d2)-1]
              d1 := d1 + 1
              d2 := d2 + 1
          type = 2
            -- Vector Source
            seq
              out!mem[d1-1]
              d1 := d1 + 1 :

proc sink ( chan in ) =
  -- garbage signal collector
  seq i =[1 for total.time]
    in?any :

proc alloc.sources(  value pipe, size, mem.mat[], chan a[] ) =
  --
  -- allocation of Sources to grid points in
  -- Virtual processing space
  --
  var k, delay :
  seq
    delay := delay.x
    if
      pipe = 1
        k := p
      pipe = 2
        k := p - 1
    -- make enough sources for all
    -- cells of pipe
    par i =[ 1 for size]
      var delay2, k1:
      seq
        k1 := k - (2*(i-1))
```

```
        delay2:= delay + (i - 1)
        if
          k1 <= 0
            source( 1, mem.mat, 2-k1, 1, delay2, a[i-1] )
          true
            source( 1, mem.mat, 1, k1, delay2 , a[i-1] ) :


proc bandvec ( var mem.mat[], mem.vec[], mem.null[], value size,
                            delay.1, delay.2, pipe, chan yout ) =
  --
  -- Abstract definition of the single pipe band vector array
  -- recall that two such vectors  are required for a Double pipe
  --
  chan x[max.cells+1], y[max.cells+1], a[max.cells] :

  par
    -- pipe
    ips( x[0], y[1], x[1], yout, a[0] )
    if
      size <> 1
        par i=[ 2 for size -1 ]
          ips( x[i-1], y[i], x[i], y[i-1], a[i-1] )
    -- matrix and vector sources
    alloc.sources( pipe, size, mem.mat, a )
    source(2, mem.vec, 1, 0, delay.x, x[0] )
    source(2, mem.null, 1,0, delay.y, y[size] )
    -- collection of garbage falling off the pipe
    sink( x[size] ) :

proc x2.solver( chan lin[], rin[], x[], out1, out2 ) =
  --
  -- 2*2 system solver : By Cramers rule without pivoting
  --
  chan e[18] , dum[10]  :
  par
    determinant( x[0], x[1], x[2], x[3], e[0], e[1],
                          e[2], e[3], e[4], e[17], 0    )
    ddivide(e[0], e[2], e[4], e[11], e[12] )
    ddivide(e[1], e[3], e[17], e[13], e[14] )
    delay( lin[1], e[5])
    delay( lin[3], e[6])
    delay( rin[1], e[8])
    delay( rin[3], e[7])
    adder(lin[0], e[5], rin[0], e[8], e[10], e[16] )
    adder(lin[2], e[6], rin[2], e[7], e[9], e[15] )
    determinant(e[10], e[11], e[12], e[9], dum[0], dum[1],
                          dum[2], dum[3], dum[4], out1, 1  )
    determinant(e[15], e[13], e[14], e[16], dum[5], dum[6],
                          dum[7], dum[8], dum[9], out2, 1  ) :

proc system =
  --
  -- Single Iteration of the XWZ-QI scheme
  --
  chan yout.l[4], yout.r[4], res1, res2, x[4] :
  par
    -- right p11 pipes
    bandvec( p11, x1, b1, no.cells.1, delay.x, delay.y, 1, yout.l[0])
    bandvec( p11, x1, null.vec, no.cells.2, delay.x, delay.y, 2, yout.l[1])

    -- right p21 pipes
    bandvec( p21, x1, null.vec, no.cells.1, delay.x, delay.y, 1, yout.l[2])
    bandvec( p21, x1, null.vec, no.cells.2, delay.x, delay.y, 2, yout.l[3])

    -- left p12 pipes
    bandvec( p12, x2, null.vec, no.cells.1, delay.x, delay.y, 1, yout.r[0])
    bandvec( p12, x2, null.vec, no.cells.2, delay.x, delay.y, 2, yout.r[1])

    -- left p22 pipes
    bandvec( p22, x2, b2, no.cells.1, delay.x, delay.y, 1, yout.r[2])
    bandvec( p22, x2, null.vec, no.cells.1, delay.x, delay.y, 2, yout.r[3])

    -- 2*2 system solver
    x2.solver( yout.l, yout.r, x, res1, res2 )
    source( 2, xc1, 1, 0, no.cells.1 + (delay.y - 1), x[0] )
    source( 2, xc2, 1, 0, no.cells.1 + (delay.y - 1), x[1] )
    source( 2, xc3, 1, 0, no.cells.1 + (delay.y - 1), x[2] )
    source( 2, xc4, 1, 0, no.cells.1 + (delay.y - 1), x[3] )

    -- data collection
    store( res1, x1, (no.cells.1 + delay.y) + 2)
    store( res2, x2, (no.cells.1 + delay.y) + 2) :

proc getdata( var null[] ) =
  --
  -- read in matrix and vector data from terminal/Host
  --
  var tmp :
  seq
    str.to.screen("*n xwz- iterative system solver *n")
    str.to.screen("*n*n using double systolic pipe*n")
    str.to.screen("*n*nband width w = p + q - 1")
    str.to.screen("*np = ")
    str.to.screen("*nq = ")
    str.to.screen("*n enter matrix ")
    seq i =[ 1 for n]
      seq
        str.to.screen("*n[")
        seq j =[1 for n]
          seq
            fp.num.from.keyboard(tmp)
            str.to.screen("*s")
            fp.num.to.screen(tmp)
            -- construct permuted data matrices
            if
              (i<=n2) and (j<=n2)
```

```
            p11[(((i-1)*n2)+j)-1] := tmp
          (i>n2) and (j<=n2)
            p21[(((n-i)*n2)+j)-1] := tmp
          (i<=n2) and (j>n2)
            p12[((i-1)*n2)+(n-j)] := tmp
          (i>n2) and (j>n2)
            p22[((n-i)*n2)+(n-j)] := tmp
      str.to.screen("]*n")
  -- intialise starting approximation vector
  -- and auxiliaries vectors
  seq i=[ 1 for n2]
    var addr :
    seq
      addr := (((i-1)*n2)+i)-1
      xc2[i-1] := p11[addr]
      p11[addr] := 0
      xc4[i-1] := p21[addr]
      p21[addr] := 0
      xc3[i-1] := p12[addr]
      p12[addr] :=  0
      xc1[i-1] := p22[addr]
      p22[addr] := 0
  str.to.screen("*n*n enter vector *n[*s")
  seq i =[1 for n]
    seq
      fp.num.from.keyboard(tmp)
      str.to.screen("*s")
      fp.num.to.screen(tmp)
      if
        i <= n2
          b1[i-1] := tmp
        true
          b2[n-i] := tmp
  str.to.screen("]*n intial solution vector = *n[")
  seq i =[ 1 for n]
    seq
      fp.num.from.keyboard(tmp)
      str.to.screen("*s")
      fp.num.to.screen(tmp)
      if
        i <= n2
          seq
            null[i-1] := 0
            x1[i-1] := tmp
        true
          x2[n-i] := tmp
  str.to.screen("]*n") :

proc putdata( value x1[], x2[] ) =
  --
  -- read out the result vector to the Host /terminal
  --
  seq
    str.to.screen("result vector *n[")
    seq i=[1 for n]
      seq
        str.to.screen("*s")
        if
          i > n2
            fp.num.to.screen(x2[n-i])
          true
            fp.num.to.screen(x1[i-1])
    str.to.screen("*s]*n") :

-- main

seq
  setup
  getdata( null.vec )
  -- run the system for m iterations
  -- n.b sequential here in real scheme should be
  -- a parallel loop , current execution very slow for
  -- full parallel execution
  seq i =[ 1 for m ]
    seq
      system
      putdata( x1, x2 )
```

```
-- program 7

-- Systolic Array : For BATS Pipeline using O(n) cell
--                  ( using Pickering's Algorithm. )
--
-- Notes : The Cell described uses (L/F)IFO storage
--         operation is controlled by hardwired control bit
--         c1 ( in chip definition ). The Chip itself is a
--         sequential process; it simulates the parallel chip
--         design using extra variables, recall each assignment
--         in OCCAM can be interpreted as a communication. Using
--         this method allows easy termination of the soft-systolic
--         pipe, using existing pipeline controls .

-- problem dependent constants
-- maxsize = upper bound on chip memory
def maxsize = 50, r = 2 :
var n, type :

-- library routines
EXTERNAL proc str.to.screen(value s[] ) :
EXTERNAL proc fp.float(value int, var float) :
EXTERNAL proc fp.num.from.keyboard( var f) :
EXTERNAL proc fp.num.to.screen(value f) :
EXTERNAL proc num.to.screen(value n) :
EXTERNAL proc num.from.keyboard( var n) :
EXTERNAL proc fp.mult( value f1 ,f2, var f3 ) :
EXTERNAL proc fp.sub( value f1, f2, var f3 ) :
EXTERNAL proc fp.div( value f1, f2 ,var f3) :
EXTERNAL proc fp.add( value f1, f2 , var f3) :


proc chip ( chan in, cntrl, tag.in, out, cntrl.out, tag.out , value c1 ) =
  --
  -- Definition of the 0(n) BATS Cell :
  --      l.f.1 and l.f.2 : L/Fifo stores controlled by c1, and tag
  --      c.fifo    : pipelining of control signals
  --      tag       : bits associated with data as End Of Data etc
  --      switch.1 : L/F storage and tag control set by c1 and tag
  --      switch.2 : switches cell out of pipeline and forces close-down
  --      others    : simulation of parallel communication internal to cell
  --
  var c.fifo[maxsize], l.f.1[maxsize], l.f.2[maxsize], tag[maxsize] :
  var u, v, v2, a, yn, tag.a, tag.c, c.out, c , d :
  var usave, vsave, vsave.2, ysave, csave, asave, un, alpha :
  var running, switch.1, switch.2 :
  var one :
  seq
    -- switch on
    fp.float(1,one)
    running := true
    switch.1 := true
    switch.2 := true
```

```
-- cell
while running
  seq
    if
      switch.2
        cntrl?csave
    if
      ( csave = 6) and switch.2
        -- switch off cell input
        switch.2 := false
      c.out = 6
        -- close down cell
        running := false
    -- i/o
    cntrl.out!c.out
    par
      if
        switch.2
          -- input
          par
            in?d
            tag.in?tag.a
      if
        running
          -- output
          par
            out!u
            tag.out!tag.c
    -- forward recursive cell
    if
      csave = 3
        -- clear cell and load alpha
        -- L/F = FIFO
        seq
          alpha := d
          fp.float(0,vsave)
          vsave.2 := d
          switch.1 := true
          fp.float(1,asave)
          fp.sub(vsave,asave,asave)
      csave = 2
        -- simple reset
        seq
          fp.float(0,vsave)
          fp.float(1,asave)
          fp.sub(vsave,asave,asave)
      true
        -- normal forward substitution
        seq
          fp.mult(alpha,vsave,vsave)
          fp.sub(d,vsave,vsave)
          vsave.2 := vsave
          fp.mult(a,alpha, asave)
          fp.sub(0,asave, asave)
```

```
-- D-cell
fp.add(one,a,ysave)
if
  ysave <> 0
    fp.div(v,ysave,ysave)
-- backward recursive cell
if
  c.fifo[n+1] = 3
    -- pass alpha, load u(n)
    seq
      usave := l.f.1[n+1]
      un := yn
  tag[n+1] = 1
    -- pass u(n) unknown result
    usave := un
  true
    -- normal backward substitution
    seq
      fp.mult(un, l.f.2[n+1], usave)
      fp.sub(l.f.1[n+1], usave, usave)
-- next output control and tag signals
tag.c := tag[n+1]
c.out := c.fifo[n+1]
-- simulated internal cell communication
tag[n+1] := tag[n]
c.fifo[n+1] := c.fifo[n]
l.f.1[n+1] := l.f.1[n]
l.f.2[n+1] := l.f.2[n]
u := usave
v := vsave
a := asave
c := csave
v2 := vsave.2
yn := ysave
-- L/F storage
if
  switch.1
    -- FIFO input/ouptut
    -- input enters l.f.1[0]
    -- output in l.f.1[n]
    seq
      seq i =[ 0 for n]
        seq
          l.f.1[n-i] := l.f.1[(n-i)-1]
          l.f.2[n-i] := l.f.2[(n-i)-1]
          tag[n-i] := tag[(n-i)-1]
      l.f.1[0] := v2
      l.f.2[0] := a
      tag[0] := tag.a
  true
    -- LIFO output
    -- output value is left in l.f.1[n]
    seq
      l.f.1[n] := l.f.1[0]
      l.f.2[n] := l.f.2[0]
      tag[n] := tag[0]
      seq i =[ 0 for n]
        seq
          l.f.1[i] := l.f.1[i+1]
          l.f.2[i] := l.f.2[i+1]
          tag[i] := tag[i+1]
-- control FIFO
seq i =[0 for n]
  c.fifo[n-i] := c.fifo[(n-i)-1]
c.fifo[0] := c
-- until end of data L/F 's act as FIFO
-- use tag and c1 to choose output mode
-- LIFO or FIFO
if
  c1 and (tag.a = 1)
    switch.1 := false
-- Multiplexor swap of tag bits
if
  (c1 = 1) and (tag.c = 2)
    tag.c := 1
  (c1 = 1) and (tag.c = 1)
    tag.c := 2 :


proc getdata( chan out, cntrl.out , tag.out) =
  --
  -- Host input routine :
  --      Reads RHS data and Control from Host, automatically
  --      constructs tag bits associated with RHS, and pumps
  --      signals input Array.
  --
  var j,tag[maxsize], d,c, running :
  seq
    running := true
    while running
      seq
        -- control
        str.to.screen("*ncontrol =")
        num.from.keyboard(c)
        num.to.screen(c)
        if
          c = 6
            -- close down array
            seq
              running := false
          c = 5
            -- size of system and tag setup
            seq
              str.to.screen("*n order of system = ")
              num.from.keyboard(n)
              num.to.screen(n)
              seq k =[ 0 for n]
```

```
            tag[k] := 0
          tag[0] := 2
          tag[n-1] := 1
      c = 3
        -- reset tag outputs
        seq
          j := 0
    -- get RHS value
    str.to.screen("*nd-value = ")
    fp.num.from.keyboard(d)
    fp.num.to.screen(d)
    -- Pump into array
    cntrl.out!c
    if
      running
        par
          out!d
          if
            c <> 0
              tag.out!0
            true
              seq
                tag.out!tag[j]
                j := j + 1
                if
                  j = n
                    j := 0 :


proc putdata(chan in, cntrl.in, tag.in ) =
  --
  -- Host Output Interface :
  --      collect array output, seperate control and tag signals
  --      output the result .
  --
  var c, res, t1, switch, running :
  seq
    running := true
    while running
      seq
        -- control
        cntrl.in?c
        if
          c = 6
            -- array has stopped
            seq
              running :=false
              str.to.screen("*nQUIT*n")
          true
            -- collect result
            par
              in?res
              tag.in?t1
        -- output
          if
            switch and running
              seq
                str.to.screen("*nresult = ")
                fp.num.to.screen(res)
          -- use tag bits and setup control to
          -- remove garbage between pipelined problem
          -- instances
          if
            c = 3
              switch := true
            t1 = 1
              switch := false :

-- main

-- input and output vector are in same order
seq
  -- choose type of array
  str.to.screen("type of array ")
  num.from.keyboard(type)
  num.to.screen(type)
  if
    type = 1
      -- single chip/Cell test
      chan cntrl[2], tag[2], data[2] :
      par
        getdata(data[0], cntrl[0], tag[0] )
        chip(data[0], cntrl[0], tag[0],
             data[1], cntrl[1], tag[1], 1 )
        putdata(data[1],cntrl[1], tag[1])
    type = 2
      -- Tri-diagonal case
      chan cntrl[3], tag[3], data[3] :
      par
        getdata( data[0], cntrl[0], tag[0] )
        chip( data[0], cntrl[0], tag[0],
              data[1], cntrl[1], tag[1], 1)
        chip( data[1], cntrl[1], tag[1],
              data[2], cntrl[2], tag[2], 1)
        putdata(data[2], cntrl[2], tag[2])
    type = 3
      -- general case with bandwidth = 2r + 1
      chan cntrl[(2*r)+1], tag[(2*r)+1], data[(2*r)+1] :
      par
        getdata( data[0], cntrl[0], tag[0])
        par i =[ 1 for r]
          var c1 :
          seq
            -- set hardwired L/F control
            c1 := 0
            if
              i = r
                c1 := 1
```

```
    chip( data[i-1], cntrl[i-1], tag[i-1],
          data[i], cntrl[i], tag[i], c1    )
par i =[ 1 for r]
  var c1 :
  seq
    -- set hardwired control
    c1 := 0
    if
      i = r
        c1 := 1
    chip( data[(r+i)-1], cntrl[(r+i)-1], tag[(r+i)-1] ,
          data[r+i], cntrl[r+i], tag[r+i], c1          )
putdata( data[2*r], cntrl[2*r], tag[2*r] )
```

```
-- program 8

-- Systolic Array : Alternative implementation of the BATS pipeline
--                    using the P-Cyclic O(V) cell.

--  Notes         : In this method we compute the nth unknown of the
--                  particular factor the cell represents by evaluating
--                  the Vth-order polynomial.
--                    The internal cell communication is simulated by
--                   assignment to facilitate easy pipeline close-down

-- problem dependent constants
def v = 40, r = 2 :
var type :

-- library routines
EXTERNAL proc str.to.screen( value s[] ) :
EXTERNAL proc num.to.screen( value n) :
EXTERNAL proc num.from.keyboard( var n) :
EXTERNAL proc fp.num.to.screen( value f) :
EXTERNAL proc fp.num.from.keyboard( var f ) :
EXTERNAL proc fp.mult(value f1, f2 , var f3 ) :
EXTERNAL proc fp.sub( value f1, f2 , var f3 ) :
EXTERNAL proc fp.add( value f1, f2, var f3 ) :
EXTERNAL proc fp.div( value f1, f2, var f3) :
EXTERNAL proc fp.float(value int, var float) :



proc chip( chan din2, din1, control,
                dout2, dout1, control.out ) =
  --
  -- Cell Definition :
  --                  d1.fifo = delay for first n/2 terms of RHS
  --                  d2.fifo = delay for last  n/2 terms of RHS
  --                   c.fifo = pipelining of control signals
  --
  var t1, t2, switch, x1, x2, running :
  var d1.fifo[v+1], d2.fifo[v+1], c.fifo[v+2] :
  var a, alpha[3], xn :
  seq
    -- switch on
    switch := true
    running := true
    fp.float(1,alpha[1])
    -- cell
    while running
      seq
        -- perform FIFO operation
        c.fifo[v+1] := c.fifo[v]
        seq i =[0 for v]
          seq
            c.fifo[v - i] := c.fifo[(v - i)-1]
```

```
    d1.fifo[v - i] := d1.fifo[(v - i) - 1]
    d2.fifo[v - i] := d2.fifo[(v - i)-1]
-- input/output
if
  switch
    control?c.fifo[0]
if
  (c.fifo[0] = 6) and switch
    -- switch of cell inputs
    switch := false
  c.fifo[v+1] = 6
    -- close down cell
    running := false
control.out!c.fifo[v+1]
par
  if
    switch
      -- input
      par
        din1?d1.fifo[0]
        din2?d2.fifo[0]
  if
    running
      -- output
      par
        dout1!x1
        dout2!x2
-- Pre-FIFO control
if
  c.fifo[0] = 1
    -- reset and load alpha
    seq
      fp.sub(0,d1.fifo[0],alpha[0])
      a := alpha[0]
      fp.float(1,t2)
      -- intial value of iterative sequence
      fp.div(t2,d1.fifo[0],alpha[2])
      xn := 0
  (c.fifo[0] = 0) or (c.fifo[0] = 6)
    -- normal computation
    seq
      -- post FIFO control
      if
        c.fifo[v] = 1
          -- pass alpha and setup
          -- ips(2) and d-cell
          seq
            x1 := d1.fifo[v]
            x2 := d1.fifo[v]
            alpha[1] := -alpha[0]
        c.fifo[v] = 2
          -- load polynomial result
          -- reset polynomial cell
          seq
            fp.mult(alpha[1], xn, t1)
            fp.sub(d1.fifo[v],t1, x1)
            x2 := xn
            fp.float(0,xn)
        true
          -- ips(2) and D-cell
          seq
            fp.mult(alpha[1], x1, t1)
            fp.sub(d1.fifo[v], t1, x1)
            fp.sub(d2.fifo[v], x2, t1)
            fp.mult(t1,alpha[2],x2)
      -- normal polynomial cell computation
      -- next polynomial term
      fp.mult(a, d2.fifo[0], t1)
      fp.add(t1 ,xn, xn)
      -- next power of alpha
      fp.mult(a,alpha[0], a)
      fp.mult(alpha[2],alpha[0],t1)
      -- iterative evalutaion of 1/alpha
      -- for stability
      fp.add(t2,t1,t1)
      fp.mult(t1,alpha[2],t1)
      fp.add(alpha[2],t1,alpha[2])
      fp.num.to.screen(alpha[2])
    c.fifo[0] = 2
      -- load first polynomial term
      seq
        xn := d2.fifo[0] :


proc getdata( chan din1, din2, cntrl ) =
  --
  -- Host Input Interface :
  --     Reads control and Rhs values
  --
  var running, d1, d2, c1, alpha :
  seq
    running := true
    while running
      seq
        str.to.screen("*ncontrol = ")
        num.from.keyboard(c1)
        num.to.screen(c1)
        if
          c1 = 6
            -- close down array
            running := false
          c1 = 1
            -- alpha value
            seq
              str.to.screen("*nalpha =")
              fp.num.from.keyboard(alpha)
              fp.num.to.screen(alpha)
              d1 := alpha
```

```
            d2 := alpha
        true
          -- RHS 2-values
          seq
            str.to.screen("*nd1- value = ")
            fp.num.from.keyboard(d1)
            fp.num.to.screen(d1)
            str.to.screen("*nd2- value = ")
            fp.num.from.keyboard(d2)
            fp.num.to.screen(d2)
      -- output
      cntrl!c1
      if
        c1 <> 6
          par
            din1!d1
            din2!d2 :

proc putdata( chan u1, u2, cntrl ) =
  --
  -- Host Output Interface :
  --      Collects Array results and outputs solution to
  --      Host.
  --
  var running, res1, res2, c1 :
  seq
    running := true
    while running
      seq
        cntrl?c1
        if
          c1 = 6
            -- array has stopped
            running := false
          true
            -- next results
            par
              u1?res1
              u2?res2
        -- output to Host
        str.to.screen("*n u1-value =  ")
        fp.num.to.screen(res1)
        str.to.screen("*n u2-value =  ")
        fp.num.to.screen(res2)
        str.to.screen("*n")
    str.to.screen("bye !") :

-- main
seq
  str.to.screen("*n type of array = ")
  num.from.keyboard(type)
  num.to.screen(type)
  if
    type = 1
      -- single chip/cell test
      chan d1[2], d2[2], cntrl[2] :
      par
        getdata(d1[0], d2[0], cntrl[0] )
        chip( d2[0], d1[0], cntrl[0],
              d2[1], d1[1], cntrl[1] )
        putdata(d1[1], d2[1], cntrl[1] )
    type = 2
      -- Tri-diagonal case
      chan d1[3], d2[3], cntrl[3] :
      par
        getdata(d1[0], d2[0], cntrl[0] )
        chip( d2[0], d1[0], cntrl[0],
              d2[1], d1[1], cntrl[1] )
        chip( d1[1], d2[1], cntrl[1],
              d1[2], d2[2], cntrl[2] )
        putdata(d1[2], d2[2], cntrl[2] )
    type = 3
      -- general case with Bandwidth = 2r + 1
      chan d1[(2*r)+1], d2[(2*r)+1], cntrl[(2*r)+1] :
      par
        getdata(d1[0], d2[0], cntrl[0] )
        par i =[ 1 for r]
          chip( d2[i-1], d1[i-1], cntrl[i-1],
                d2[i], d1[i],cntrl[i]        )
        par i =[ 1 for r]
          chip( d1[(r+i)-1], d2[(r+i)-1], cntrl[(r+i)-1],
                d1[(r+i)], d2[(r+i)], cntrl[(r+i)]        )
        putdata(d1[2*r], d2[2*r], cntrl[2*r] )
```

```
-- program 9

-- Systolic Array : To construct the extrapolation table in
--                  Romberg's Integration algorithm.

-- The basic cell is the REP or Richardson's Extrapolation cell
-- which computes the extrapolation values

-- Table size
def n = 5:

-- library routines
EXTERNAL Proc str.to.screen(value s[]) :
EXTERNAL Proc num.to.screen(value n) :
EXTERNAL Proc fp.num.to.screen(Value float f):
EXTERNAL Proc fp.num.from.keyboard(Var float f) :

Proc REP(Chan in1, out1, in2, out2, out3, cntrlin, cntrlout ) =
  --
  -- Richardsons extrapolation cell
  --
  var float t1, t2, p.4, p.res, rnew, rold, res :
  var switch, running, toggle, c.fifo[4] :
  seq
    -- intialisation
    p.res := 0.0
    rold := 0.0
    res := 0.0
    t1 := 0.0
    t2 := 1.0
    seq i =[0 for 3 ]
      c.fifo[i] := 0
    switch := true
    running := true
    toggle  := true
    -- cell
    while running
      seq
        -- control input
        if
          switch
            cntrlin?c.fifo[0]
        -- decide on input/output
        if
          (c.fifo[0] = 6 ) and switch
            -- close input
            switch := false
          c.fifo[3] = 6
            -- destroy cell
            running := false
        cntrlout!c.fifo[3]
        -- i/0
        if
          switch
            par
              in1?rnew
              in2?p.4
          if
            running
              par
                out1!res
                out2!(p.res)
                -- output to fanin network
                if
                  c.fifo[3] = 1
                    seq
                      toggle := false
                      out3!res
                  toggle
                    out3!0
          -- extrapolation formula
          res := t1/t2
          t1 := (p.4*rnew)- rold
          t2 := p.4- 1.0
          rold := rnew
          p.res := p.4 * 4.0
          -- shift control fifo
          seq i =[ 0 for 3]
            c.fifo[3-i] := c.fifo[(3-i)-1]
          if
            c.fifo[0] = 6
              c.fifo[0] := 0 :

proc fnet(chan gather[], var float vec[], var k ) =
  --
  --  fanin network: primitive routine to collect values
  --
  seq
    par j =[0 for n]
      -- check all processes
      seq
        if
          j > k
            -- those still to output
            gather[j]?any
          j = k
            var float tmp :
            -- current output
            seq
              gather[k]?tmp
              if
                tmp <> 0.0
                  seq
                    vec[k] := tmp
                    k := k + 1 :

proc getdata( chan out1,out2, cntrl ) =
  --
```

```
  -- read starting values , and pump into
  -- array then close down array systolically
  --
  var float four, vec[n+1] :
  seq
    four := 4.0
    str.to.screen("*nEnter Romberg Starting values")
    seq i=[0 for (n+1)]
      seq
        str.to.screen("*nR[")
        num.to.screen(i)
        str.to.screen("] =")
        fp.num.from.keyboard(vec[i])
        fp.num.to.screen(vec[i])
    str.to.screen("*n*n*n")
    -- start pumping
    seq i=[0 for (n+1)]
      par
        if
          i = 0
            cntrl!1
          true
            cntrl!0
        out1!vec[i]
        out2!four
    -- close down
    cntrl!6 :

proc putdata( chan in1, in2, fanin[], cntrl ) =
  --
  -- collect garbage falling off array, and
  -- call fanin to collect next result and print out
  -- diagonal entries
  --
  var float vec[n] :
  var running, c1, k :
  seq
    k := 0
    running := true
    -- collect results until stopped
    while running
      seq
        cntrl?c1
        if
          c1 = 6
            running := false
          true
            par
              in1?any
              in2?any
              fnet(fanin, vec, k)
    -- output diagonal approximations.
    str.to.screen("*n*n Diagonal Table Entries")
    seq i =[0 for n]
      seq
        str.to.screen("*n")
        fp.num.to.screen(vec[i]) :

-- main
--
-- The Romberg array
chan in1[n+1], in2[n+1], fanin[n], cntrl[n+1] :

par
  getdata(in1[0], in2[0], cntrl[0])
  par i =[1 for n]
    REP(in1[i-1], in1[i], in2[i-1], in2[i], fanin[i-1], cntrl[i-1], cntrl[i])
  putdata(in2[n], in1[n], fanin, cntrl[n])
```

```
-- program 10

-- Systolic Array : A Systolic Ring implementation of the Romberg
--                  table construction.
--

-- Ring size and Table size respectively
def n = 2, m= 6 :

-- library routines
EXTERNAL Proc str.to.screen(value s[]) :
EXTERNAL Proc num.to.screen(value n) :
EXTERNAL Proc fp.num.to.screen(Value float f):
EXTERNAL Proc fp.num.from.keyboard(Var float f) :

Proc REP(Chan in1, out1, in2, out2, out3, cntrlin, cntrlout ) =
  --
  -- Modified Extrapolation cell (see report)
  --
  var float t1, t2, p.4, p.res, rnew, rold, res :
  var switch, running, toggle, c.fifo[4] :
  seq
    -- intialisation
    p.res := 0.0
    rold := 0.0
    res := 0.0
    t1 := 0.0
    t2 := 1.0
    seq i =[0 for 3 ]
      c.fifo[i] := 0
    switch := true
    running := true
    toggle  := false
    -- cell
    while running
      seq
        -- control i/o
        par
          if
            switch
              cntrlin?c.fifo[0]
          cntrlout!c.fifo[3]
        -- decide on data i/o
        if
          (c.fifo[0] = 6 ) and switch
             switch := false
          c.fifo[3] = 6
             running := false
        -- switch on fanin output line
        if
          c.fifo[0] = 1
            toggle := true
        -- i/0
        par
          if
            switch
              par
                in1?rnew
                if
                  c.fifo[0] = 1
                    in2?p.4
          if
            running
              par
                out1!res
                if
                  c.fifo[3] = 1
                    seq
                      toggle := false
                      par
                        out3!res
                        out2!p.res
                  toggle
                    out3!0
        -- computation
        res := t1/t2
        t1 := (p.4*rnew)- rold
        t2 := p.4- 1.0
        rold := rnew
        p.res := p.4 * 4.0
        -- shift control fifo
        seq i =[ 0 for 3]
          c.fifo[3-i] := c.fifo[(3-i)-1]
        if
          c.fifo[0] = 6
            c.fifo[0] := 0 :

proc fnet(chan gather[], var float vec[], var k,z ) =
  --
  -- Modified fanin simulator.
  --      Sequentially poll ring cells looking for outputs
  --      and accept them if non-zero
  --
  -- Note : z = index of diagonal entry next output;
  --        k = index of next ring cell expected to output diagonal.
  --
  seq j =[0 for n]
    seq
      if
        j = k
          var float tmp :
          seq
            gather[k]?tmp
            if
              tmp <> 0.0
                seq
                  vec[z] := tmp
                  z := z + 1
```

```
                k := k + 1 :

proc host( chan out1,out2, cntrlin, in1,in2, fanin[], cntrlout ) =
  --
  -- Combined getdata and putdata to act as ring arbiter, to switch
  -- from Host input to ring input and collect fanin results
  --
  chan link :
  par
    -- equivalent process to getdata, uses link to
    -- create switch from Host to ring input
    var float four, vec[m] :
    seq
      four := 4.0
      str.to.screen("*nEnter Romberg Starting values")
      seq i=[0 for m]
        seq
          str.to.screen("*nR[")
          num.to.screen(i)
          str.to.screen("] =")
          fp.num.from.keyboard(vec[i])
          fp.num.to.screen(vec[i])
      str.to.screen("*n*n*n")
      -- pump host inputs into ring
      seq i=[0 for m]
        par
          link!0
          if
            i = 0
              par
                cntrlin!1
                out2!four
            true
              cntrlin!0
          out1!vec[i]
      -- switch to ring
      link!1

    -- equivalent to putdata, but augmented with
    -- control to wrap around ends of ring
    -- when link = 1
    var float vec[m] :
    var running, switch, c1, k, z, l1, r1, r2 :
    var rs1, rs2, cs1 :
    seq
      -- intilaise
      z := 0
      k := 0
      cs1 := 0
      running := true
      switch := false
      -- collect and pump till all values
      -- received
      while running
        seq
          -- switch ?
          if
            not switch
              link?l1
          cntrlout?c1
          -- ring wrap around
          if
            l1 = 1
              seq
                -- first value
                l1 := 0
                switch := true
                cntrlin!cs1
            switch and (z <= (m-1))
              seq
                -- rest
                cntrlin!cs1
            switch and (l1 = 0)
              seq
                -- close down ring
                l1 :=  2
                cntrlin!6
          if
            c1 = 6
              -- kill this process
              running := false
            true
              seq
                -- collect garbage and results
                seq
                  in1?r1
                  if
                    c1 = 1
                      in2?r2
                  fnet(fanin, vec, k, z)
                -- ring i/o
                if
                  switch and (l1 <> 2)
                    seq
                      out1!rs1
                      if
                        cs1 = 1
                          out2!rs2
          -- re-sync ring data and control
          rs1 := r1
          rs2 := r2
          cs1 := c1
          if
            (cs1 = 1) and (z <=  (m-1))
              k := 0
      -- print results for user, vec = memory in Host
      str.to.screen("*n*n Diagonal Table Entries")
      seq i =[0 for (m-1)]
```

```
      seq
        str.to.screen("*n")
        fp.num.to.screen(vec[i]) :

-- main

-- Systolic Ring definition
chan in1[n+1], in2[n+1], fanin[n], cntrl[n+1] :

par
  host(in1[0], in2[0], cntrl[0], in1[n], in2[n], fanin, cntrl[n])
  par i =[1 for n]
    REP(in1[i-1], in1[i], in2[i-1], in2[i], fanin[i-1], cntrl[i-1], cntrl[i] )
```

```
-- program 11

-- Systolic Array : An Array for the Generic Group Explicit
--                  Methods (GER, GEL, GEC, GEU ) for
--                  Parabolic Differential Equations.
--

-- Number of groups (cells) and starting values
def m = 5 , n = 4 :
chan ptr :

-- library routines
EXTERNAL proc num.to.screen(value n) :
EXTERNAL proc num.from.keyboard(var n) :
EXTERNAL proc fp.num.to.screen(value float f) :
EXTERNAL proc fp.num.from.keyboard(var float f) :
EXTERNAL proc str.to.screen( value s[]) :
EXTERNAL proc open.file(value path.name[], access[], chan io.chan) :
EXTERNAL proc close.file(chan io.chan) :
EXTERNAL proc str.to.chan(chan c, value s[]) :
EXTERNAL proc fp.num.to.chan(chan c, value float f) :

proc boundary( var float t1, value float a, r, u1, u2 ) =
  --
  -- Boundary cell computations using Asymmetric
  -- approximations.
  --
  seq
    t1 := t1 - (r*t1)
    t1 := t1 + (r*u2)
    t1 := t1 + (r*u1)
    t1 := t1/(a-r) :

proc group(var float t1, value float a, r, u, chan link0, link1 ) =
  --
  -- solution for half of 2*2 system of
  -- internal group points. Note communication/parallelism
  var float t3 :
  seq
    t1 := t1 - (r*t1)
    t1 := t1 + (r*u)
    par
      link0!t1
      link1?t3
    t1 := t1 + (r*t1)
    t1 := t1 + (r*t3)
    t1 := t1/a :

proc memory( chan memin[], memout, cntrl ) =
  --
  -- Simulation of the memory Buffer
  -- general case with g(x,t) not included for
  -- simplicity
  --
```

```
  var float mem[(2*m)*n] :
  var running, c1 :
  seq
    running := true
    while running
    -- start memory
      seq
        cntrl?c1
        if
          c1 = 6
            running := false
            -- shut down
          c1 = 5
            seq i=[ 1 for n]
            -- Freeze/Empty Buffer
              seq j=[1 for (2*m)]
                memout!mem[(((i-1)*(2*m))+j)-1]
          true
            seq
            -- normal array operation
            -- collect result
              seq i=[2 for (n-1)]
                par j=[ 1 for (2*m)]
                  mem[(((i-2)*(2*m))+j)-1] := mem[(((i-1)*(2*m))+j)-1]
              par j=[1 for (2*m)]
                memin[j-1]?mem[(((n-1)*(2*m))+j)-1] :

proc ge( chan in1,out1,in2,out2, mem1, mem2, cntrlin, value float v1,v2,r,
         value type ) =
  --
  -- Generic Group Explicit cell
  --
  var float u0, u1, t1, t2, a :
  var running, c1 :
  chan link[2] :
  seq
    -- preload
    t1 := v1
    t2 := v2
    a := 1.0 + (2.0*r)
    running := true
    -- start up
    while running
      seq
        cntrlin?c1
        if
          c1 = 6
            -- close down
            running := false
          c1 <> 5
            seq
              -- i/0
              par
                out1!t2
                out2!t1
                in1?u0
                in2?u1
              -- run correct cell
              if
                type = 0
                  -- A group
                  par
                    group(t1,a,r,u0,link[0],link[1])
                    group(t2,a,r,u1,link[1],link[0])
                type = 1
                  -- right Boundary
                  seq
                    t2 := 0.0
                    boundary(t1,a,r,u0,u1)
                type = 2
                  -- left boundary
                  seq
                    t1 := 0.0
                    boundary(t2,a,r,u0,u1)
              -- buffer result
              par
                mem1!t1
                mem2!t2 :

proc getdata(chan out1, out2, in1, in2, cntrl[], memin ) =
  --
  -- Host interface : also generates starting values
  --                  for test Boundary conditions.
  var float bvalue :
  var tmp :
  seq
    bvalue := 0.0
    str.to.screen("*nnext ")
    num.to.screen(n)
    str.to.screen(" values")
    num.from.keyboard(tmp)
    -- Fill and output Buffer
    while tmp = 0
      seq
        -- x=0 and x=n conditions
        seq i =[1 for n]
          par
            par j=[0 for (m+1)]
              cntrl[j]!0
            in1?any
            in2?any
            out1!bvalue
            out2!bvalue
        -- Freeze
        par j=[0 for (m+1)]
          cntrl[j]!5
        -- File dump of Buffer
        seq i=[1 for n]
```

```
        seq
          str.to.screen("*n")
          str.to.chan(ptr,"*n")
          seq j=[ 1 for (2*m)]
            var float res :
            seq
              memin?res
              fp.num.to.screen(res)
              fp.num.to.chan(ptr,res)
              str.to.screen("  ")
              str.to.chan(ptr,"  ")
      str.to.chan(ptr,"*n*n*n")
      str.to.screen("*nnext")
      num.to.screen(n)
      str.to.screen(" values ")
      num.from.keyboard(tmp)
    -- closedown
    par j=[0 for (m+1)]
      cntrl[j]!6 :

-- main

-- Array channel and memory details
chan memin[2*m], ul[m+1], u2[m+1], cntrl[m+1], mout :
var tp, j :
var float x,r,h, sv[(2*m)+2]  :

seq
  -- get array characteristics from Host
  open.file("result","w",ptr)
  str.to.screen("*n 0=ger, 1=gel, 2=geu, 3=gec")
  str.to.screen("*ntype of array = ")
  num.from.keyboard(tp)
  num.to.screen(tp)
  if
    tp = 0
      str.to.chan(ptr,"ger ")
    tp = 1
      str.to.chan(ptr,"gel ")
    tp = 2
      str.to.chan(ptr,"geu ")
    tp = 3
      str.to.chan(ptr,"gec ")
  -- Problem parameters
  str.to.screen("*n r = ")
  fp.num.from.keyboard(r)
  fp.num.to.screen(r)
  str.to.chan(ptr," r = ")
  fp.num.to.chan(ptr,r)
  str.to.chan(ptr,"*n*n")
  h := 0.1
  x := 0.0
  -- Test Boundary conditions
  seq i=[1 for ((2*m)-1)]
    seq
      x := x + h
      sv[i] := (4.0*x)*(1.0-x)
  sv[0] := 0.0
  sv[2*m] := 0.0
  -- shuffle trick for easy specification
  --  of array inputs
  if
    (tp=0) or (tp=3)
      j := 1
    (tp=1) or (tp=2)
      j := 0
  -- The Generic Array.
  --
  -- tp forces a creation of a specific
  -- instance of the an array
  par
    getdata(ul[0], u2[m], u2[0], ul[m], cntrl, mout )
    memory(memin, mout, cntrl[m])
    par i=[1 for m]
      var float  t1,t2  :
      var set :
      seq
        if
          ((tp=0) or (tp=2)) and (i=m)
            set := 1
          ((tp=1) or (tp=2)) and (i=1)
            set := 2
          true
            set := 0
        t1 := sv[((i-1)*2)+j]
        t2 := sv[(((i-1)*2)+j)+1]
        ge(ul[i-1], ul[i], u2[i], u2[i-1], memin[((i-1)*2], memin[(((i-1)*2)+1]
           cntrl[i-1], t1,t2,r, set )
  close.file(ptr)
```

```
-- program 12
--
-- Systolic Array : Implementation of the SAGE algorithm for
--                  Parabolic Equations.
--

-- Group and Boundary value points
def m = 5 , n = 4 :
chan ptr :

-- library Routines
EXTERNAL proc num.to.screen(value n) :
EXTERNAL proc num.from.keyboard(var n) :
EXTERNAL proc fp.num.to.screen(value float f) :
EXTERNAL proc fp.num.from.keyboard(var float f) :
EXTERNAL proc str.to.screen( value s[]) :
EXTERNAL proc open.file(value path.name[], access[], chan io.chan ) :
EXTERNAL proc close.file(chan io.chan) :
EXTERNAL proc str.to.chan(chan c, value s[]) :
EXTERNAL proc fp.num.to.chan(chan c, value float f):

proc boundary( var float t1, value float a, r, u1, u2 ) =
  -- Boundary Equations (Asymmetric)
  seq
    t1 := t1 - (r*t1)
    t1 := t1 + (r*u2)
    t1 := t1 + (r*u1)
    t1 := t1/(a-r) :

proc group(var float t1, value float a, r, u, chan link0, link1 ) =
  -- Group computation
  var float t3 :
  seq
    t1 := t1 - (r*t1)
    t1 := t1 + (r*u)
    par
      link0!t1
      link1?t3
    t1 := t1 + (r*t1)
    t1 := t1 + (r*t3)
    t1 := t1/a :

proc memory( chan memin[], memout, cntrl ) =
  -- memory Buffer
  var float mem[(2*m)*n] :
  var running, c1 :
  seq
    running := true
    while running
      seq
        cntrl?c1
        if
          c1 = 6
            running := false
          c1 = 5
            seq i=[ 1 for n]
              seq j=[1 for (2*m)]
                memout!mem[(((i-1)*(2*m))+j)-1]
          true
            seq
              seq i=[2 for (n-1)]
                par j=[ 1 for (2*m)]
                  mem[(((i-2)*(2*m))+j)-1] := mem[(((i-1)*(2*m))+j)-1]
              par j=[1 for (2*m)]
                memin[j-1]?mem[(((n-1)*(2*m))+j)-1] :

proc ge( chan in1,out1,in2,out2, mem1, mem2, cntrlin, value float v1,v2,r,
         value type ) =
  --
  -- Generic SAGE cell : Using Toggle to control computation
  --
  var float u0, u1, t1, t2, a, tmp :
  var running, toggle, c1 :
  chan link[2] :
  seq
    -- preload
    t1 := v1
    t2 := v2
    a := 1.0 + (2.0*r)
    toggle := true
    running := true
    -- start up
    while running
      seq
        cntrlin?c1
        if
          c1 = 6
            -- close down
            running := false
          c1 <> 5
            seq
              -- select GER or GEL
              if
                toggle
                  seq
                    -- shift data
                    par
                      out2!t1;t2
                      in2?tmp;u1
                    -- align
                    u0 := t1
                    t1 := t2
                    t2 := tmp
                    -- use correct computation
                    if
                      type = 1
                        seq
                          in1?u0
```

```
                  par
                    group(t1,a,r,u0,link[0],link[1] )
                    group(t2,a,r,u1,link[1],link[0] )
                type = 0
                  par
                    group(t1,a,r,u0,link[0],link[1]  )
                    group(t2,a,r,u1,link[1],link[0]  )
                type = 2
                  boundary(t1,a,r,u0,u1 )
          true
            seq
              -- shift data
              par
                out1!t2;t1
                in1?tmp;u0
              -- Align
              u1 := t2
              t2 := t1
              t1 := tmp
              -- correct computation
              if
                type = 1
                  boundary(t2,a,r,u0,u1 )
                type = 0
                  par
                    group(t1,a,r,u0,link[0],link[1] )
                    group(t2,a,r,u1,link[1],link[0] )
                type = 2
                  seq
                    in2?u1
                    par
                      group(t1,a,r,u0,link[0],link[1] )
                      group(t2,a,r,u1,link[1],link[0] )
        -- switch array type
        toggle := not toggle
        -- buffer data
        par
          mem1!t1
          mem2!t2 :

proc getdata(chan out1, out2, in1, in2, cntrl[], memin )=
  --
  -- Host array Interface and communication
  --
  var float bvalue :
  var tmp,toggle :
  seq
    bvalue := 0.0
    str.to.screen("*nnext ")
    num.to.screen(n)
    str.to.screen(" values")
    num.from.keyboard(tmp)
    toggle := true
    while tmp = 0
      var float junk :
      seq
        seq i =[1 for n]
          seq
            par j=[0 for (m+1)]
              cntrl[j]!0
            -- decide type of Array inputs
            if
              toggle
                par
                  in1?junk;junk
                  out1!bvalue
                  out2!0.0;bvalue
              true
                par
                  in2?junk;junk
                  out2!bvalue
                  out1!0.0;bvalue
            -- switch array type
            toggle := not toggle
        -- Freeze/empty buffer
        par j=[0 for (m+1)]
          cntrl[j]!5
        -- dump buffer output (file/screen)
        seq i=[1 for n]
          seq
            str.to.screen("*n")
            str.to.chan(ptr,"*n")
            seq j=[ 1 for (2*m)]
              var float res :
              seq
                memin?res
                fp.num.to.screen(res)
                fp.num.to.chan(ptr,res)
                str.to.screen("  ")
                str.to.chan(ptr,"   ")
        -- unfreeze array
        str.to.chan(ptr,"*n*n*n")
        str.to.screen("*nnext")
        num.to.screen(n)
        str.to.screen(" values ")
        num.from.keyboard(tmp)
      -- closedown
      par j=[0 for (m+1)]
        cntrl[j]!6 :

-- main

-- Array channels
chan memin[2*m], u1[m+1], u2[m+1], cntrl[m+1], mout :
var float x,r,h, sv[(2*m)+2]  :

seq
  -- problem setup
```

```
open.file("result","w",ptr)
str.to.screen("*n r = ")
fp.num.from.keyboard(r)
fp.num.to.screen(r)
str.to.chan(ptr,"sage : r = ")
fp.num.to.chan(ptr,r)
str.to.chan(ptr,"*n*n")
h := 0.1
x := 0.0
-- Test case
seq i=[1 for ((2*m)-1)]
  seq
    x := x + h
    sv[i] := (4.0*x)*(1.0-x)
sv[0] := 0.0
sv[2*m] := 0.0
-- The array
par
  getdata(u1[0], u2[m], u2[0], u1[m], cntrl, mout )
  memory(memin, mout, cntrl[m])
  par i=[1 for m]
    var float  t1,t2  :
    var set :
    seq
      if
        i = m
          set := 2
        i=1
          set := 1
        true
          set := 0
      t1 := sv[(i-1)*2]
      t2 := sv[((i-1)*2)+1]
      ge(u1[i-1], u1[i], u2[i], u2[i-1], memin[(i-1)*2], memin[((i-1)*2)+1],
         cntrl[i-1], t1,t2,r, set )
close.file(ptr)
```

```
-- program 13

-- Systolic Array : To compute the DAGE method for Parabolic
--                  Equations.
--
def m = 5 , n = 4 :
chan ptr :

-- library routines
EXTERNAL proc num.to.screen(value n) :
EXTERNAL proc num.from.keyboard(var n) :
EXTERNAL proc fp.num.to.screen(value float f) :
EXTERNAL proc fp.num.from.keyboard(var float f) :
EXTERNAL proc str.to.screen( value s[]) :
EXTERNAL proc open.file(value path.name[], access[], chan io.chan) :
EXTERNAL proc close.file(chan io.chan) :
EXTERNAL proc str.to.chan(chan c, value s[]) :
EXTERNAL proc fp.num.to.chan(chan c, value float f) :

proc boundary( var float t1, value float a, r, u1, u2 ) =
  -- boundary computation (Asymmetric)
  seq
    t1 := t1 - (r*t1)
    t1 := t1 + (r*u2)
    t1 := t1 + (r*u1)
    t1 := t1/(a-r) :

proc group(var float t1, value float a, r, u, chan link0, link1 ) =
  -- Group computation
  var float t3 :
  seq
    t1 := t1 - (r*t1)
    t1 := t1 + (r*u)
    par
      link0!t1
      link1?t3
    t1 := t1 + (r*t1)
    t1 := t1 + (r*t3)
    t1 := t1/a :

proc memory( chan memin[], memout, cntrl ) =
  -- Memory Buffer
  var float mem[(2*m)*n] :
  var running, c1 :
  seq
    running := true
    while running
      seq
        cntrl?c1
        if
          c1 = 6
            running := false
          c1 = 5
            seq i=[ 1 for n]
```

```
            seq j=[1 for (2*m)]
              memout!mem[(((i-1)*(2*m))+j)-1]
        true
          seq
            seq i=[2 for (n-1)]
              par j=[ 1 for (2*m)]
                mem[(((i-2)*(2*m))+j)-1] := mem[(((i-1)*(2*m))+j)-1]
            par j=[1 for (2*m)]
              memin[j-1]?mem[(((n-1)*(2*m))+j)-1] :

proc ge( chan in1,out1,in2,out2, mem1, mem2, cntrlin, value float v1,v2,r,
        value  type ) =
  --
  -- Group Explicit Cell for DAGE computation
  --    Toggles and steps used to change between GER and GEL type
  --    computations.
  --
  var float u0, u1, t1, t2, a, l[4] :
  var running, toggle, step, step.no,  c1 :
  chan link[2] :
  seq
    -- preload
    t1 := v1
    t2 := v2
    a := 1.0 + (2.0*r)
    -- setup start position
    step.no := 4
    toggle := false
    step := 0
    running := true
    -- start
    while running
      seq
        cntrlin?c1
        if
          c1 = 6
            -- close down
            running := false
          c1 <> 5
            seq
              -- i/0
              par
                in1?l[0];l[1]
                in2?l[2];l[3]
                out1!t2;t1
                out2!t1;t2
              if
                toggle
                  seq
                    u1 := l[2]
                    u0 := l[0]
              if
                (step = 0) or (step = 3)
                  -- act as GER
                  seq
                    -- align data
                    if
                      not toggle
                        seq
                          u0 := t1
                          t1 := t2
                          t2 := l[2]
                          u1 := l[3]
                    -- select correct cell computation
                    if
                      type = 1
                        seq
                          u0 := l[1]
                          par
                            group(t1,a,r,u0,link[0],link[1] )
                            group(t2,a,r,u1,link[1],link[0] )
                      type = 0
                        par
                          group(t1,a,r,u0,link[0],link[1]  )
                          group(t2,a,r,u1,link[1],link[0]  )
                      type = 2
                        boundary(t1,a,r,u0,u1 )
                true
                  -- act as GEL
                  seq
                    -- align data
                    if
                      not toggle
                        seq
                          u1 := t2
                          t2 := t1
                          t1 := l[0]
                          u0 := l[1]
                    -- choose correct computation
                    if
                      type = 1
                        boundary(t2,a,r,u0,u1 )
                      type = 0
                        par
                          group(t1,a,r,u0,link[0],link[1] )
                          group(t2,a,r,u1,link[1],link[0] )
                      type = 2
                        seq
                          u1 := l[3]
                          par
                            group(t1,a,r,u0,link[0],link[1] )
                            group(t2,a,r,u1,link[1],link[0] )
        -- next step modulo 4
        step := (step + 1) \ step.no
        -- set toggles
        if
          ((step=2) or (step=0))
            toggle := true
```

```
            ((step=1) or (step=3))
              toggle := false
          -- output to buffer
          par
            mem1!t1
            mem2!t2 :

proc getdata(chan out1, out2, in1, in2, cntrl[], memin )=
  -- Host Interface
  var float bvalue :
  var tmp :
  seq
    bvalue := 0.0
    str.to.screen("*nnext ")
    num.to.screen(n)
    str.to.screen(" values")
    num.from.keyboard(tmp)
    while tmp = 0
      var float junk, junk1 :
      seq
        seq i =[1 for n]
          seq
            par j=[0 for (m+1)]
              cntrl[j]!0
            par
              in1?junk;junk
              in2?junk1;junk1
              out1!0.0;bvalue
              out2!0.0;bvalue
        -- Freeze
        par j=[0 for (m+1)]
          cntrl[j]!5
        -- Dump Buffer (file/screen)
        seq i=[1 for n]
          seq
            str.to.screen("*n")
            str.to.chan(ptr,"*n")
            seq j=[ 1 for (2*m)]
              var float res :
              seq
                memin?res
                fp.num.to.chan(ptr,res)
                fp.num.to.screen(res)
                str.to.chan(ptr,"  ")
                str.to.screen("  ")
        str.to.chan(ptr,"*n*n*n")
        str.to.screen("*nnext")
        num.to.screen(n)
        str.to.screen(" values ")
        num.from.keyboard(tmp)
    -- close down
    par j=[0 for (m+1)]
      cntrl[j]!6 :
```

```
-- main

chan memin[2*m], u1[m+1], u2[m+1], cntrl[m+1], mout :
var float x,r,h, sv[(2*m)+2]  :

seq
  -- setup details
  open.file("result1","w", ptr)
  str.to.screen("*n r = ")
  fp.num.from.keyboard(r)
  fp.num.to.screen(r)
  str.to.chan(ptr,"dage : r = ")
  fp.num.to.chan(ptr,r)
  str.to.chan(ptr,"*n*n")
  -- test case
  h := 0.1
  x := 0.0
  seq i=[1 for ((2*m)-1)]
    seq
      x := x + h
      sv[i] := (4.0*x)*(1.0-x)
  sv[0] := 0.0
  sv[2*m] := 0.0
  -- The array
  par
    getdata(u1[0], u2[m], u2[0], u1[m], cntrl, mout )
    memory(memin, mout, cntrl[m])
    par i=[1 for m]
      var float  t1,t2  :
      var set :
      seq
        if
          i = m
            set := 2
          i=1
            set := 1
          true
            set := 0
        t1 := sv[(i-1)*2]
        t2 := sv[((i-1)*2)+1]
        ge(u1[i-1], u1[i], u2[i], u2[i-1], memin[(i-1)*2], memin[((i-1)*2)+1],
           cntrl[i-1], t1,t2,r, set )
  close.file(ptr)
```

```
-- program 14

-- Systolic Array : To find all the roots of a  polynomial
--                  using the QD algorithm.
-- NOTES          : Fails for non-distinct roots and indicates
--                  existance of complex roots.
--                  With 1 cell is equivalent to Bernoulli's method
--                  for dominant root

-- The basic cell is the QD or Quotient Difference cell
-- which computes the Rhombus rules values

-- Table size
def n = 16:
chan ptr :

-- library routines
EXTERNAL Proc str.to.screen(value s[]) :
EXTERNAL Proc num.to.screen(value n) :
EXTERNAL Proc fp.num.to.screen(Value float f):
EXTERNAL Proc fp.num.from.keyboard(Var float f) :
EXTERNAL Proc open.file(value pathname[], access[],chan io.chan) :
EXTERNAL Proc close.file(chan io.chan ):
EXTERNAL Proc str.to.chan(chan c, value s[] ) :
EXTERNAL Proc fp.num.to.chan( chan c, value float f) :
EXTERNAL Proc num.to.chan(chan c, value n) :

Proc QD(Chan in1, out1, in2, out2, cntrlin, cntrlout ) =
  --
  --
  var float a, b, r[3], s[2], t[7] :
  var switch, running, toggle, c.fifo[5] :
  seq
    -- intialisation
    seq i=[0 for 3]
      r[i] := 0.0
    seq i=[0 for 7]
      t[i] := 1.0
    s[0] := 0.0
    s[1] := 0.0
    seq i =[0 for 5 ]
      c.fifo[i] := 0
    switch := true
    running := true
    -- cell
    while running
      seq
        -- control input
        if
          switch
            cntrlin?c.fifo[0]
        -- decide on input/output
        if
          (c.fifo[0] = 6 ) and switch
            -- close input
            switch := false
          c.fifo[4] = 6
            -- destroy cell
            running := false
        cntrlout!c.fifo[4]
        -- i/0
        if
          switch
            par
              in1?a
              in2?b
        if
          running
            par
              out1!t[6]
              out2!s[1]
        -- qd formula
        r[2] := r[1]
        r[1] := r[0]
        s[1] := s[0]
        t[1] := t[0]
        t[3] := t[2]
        t[5] := t[4]
        par
          r[0] := a
          s[0] := t[3]
          t[0] := a + b
          t[2] := t[1] - r[2]
          if
            t[3] = 0.0
              t[4] := 0.0
            true
              t[4] := r[2]/t[3]
          t[6] := t[5]*t[3]
        -- shift control fifo
        seq i =[ 0 for 4]
          c.fifo[4-i] := c.fifo[(4-i)-1]
        if
          c.fifo[0] = 6
            c.fifo[0] := 0 :


proc getdata( chan out1,out2, cntrl ) =
  --
  -- read starting values , and pump into
  -- array then close down array systolically
  --
  var float vec2[n], vec1[n] :
  seq
    str.to.screen("*nEnter Qd Starting values")
    seq i=[0 for n]
      seq
        str.to.screen("*nQD[")
```

```
      num.to.screen(i)
      str.to.screen("] =")
      fp.num.from.keyboard(vec1[i])
      fp.num.to.screen(vec1[i])
      str.to.screen("==")
      fp.num.from.keyboard(vec2[i])
      fp.num.to.screen(vec2[i])
  str.to.screen("*n*n*n")
  -- start pumping
  seq i=[0 for n]
    par
      cntrl!1
      out1!vec1[i]
      out2!vec2[i]
  -- close down
  cntrl!6 :

proc putdata( chan in1, in2, cntrl) =
  --
  -- collect garbage falling off array, and
  -- collects next result and print out
  -- root and d  entries
  --
  var float vec1[n], vec2[n] :
  var running, c1, k :
  seq
    k := 0
    running := true
    -- collect results until stopped
    while running
      seq
        cntrl?c1
        if
          c1 = 6
            running := false
          c1 = 1
            seq
              par
                in1?vec1[k]
                in2?vec2[k]
              k := k + 1
          true
            par
              in1?any
              in2?any
  -- output root approximations.
  str.to.screen("*n*n root Table Entries")
  str.to.chan(ptr,"*n*n root Table Entries")
  seq i =[0 for n]
    seq
      str.to.screen("*n")
      str.to.chan(ptr,"*n")
      fp.num.to.screen(vec1[i])
      fp.num.to.chan(ptr,vec1[i])
      str.to.screen("  ")
      str.to.chan(ptr,"  ")
      fp.num.to.chan(ptr,vec2[i])
      fp.num.to.screen(vec2[i]) :

-- main
--
-- The QD array
chan in1[n+1], in2[n+1], cntrl[n+1] :

seq
  open.file("proot","w", ptr )
  str.to.screen("*nQD with pipeline length n =")
  num.to.chan(ptr,n)
  str.to.chan(ptr,"*n*n")
  par
    getdata(in1[0], in2[0], cntrl[0])
    par i =[1 for n]
      QD(in1[i-1], in1[i], in2[i-1], in2[i], cntrl[i-1], cntrl[i])
    putdata(in2[n], in1[n], cntrl[n])
  close.file(ptr)
```

```
--  program 15 a

-- Instruction Systolic Array (ISA)
--
-- Notes  : implements an orthogonally connected grid of processors
--          each processor can be plugged into the system or a group
--          of processors can be plugged into the same grid point
--          programs and data are read from files and buffered into the array
--          Results are read from any of the four boundaries as dictated by the
--          program. The grid cannot be closed down systolically the program
--          termination is performed by an abort at the end of the user program.
--
-- dimensions of array and interface routines
DEF n = 4 :

EXTERNAL proc abort.program :
EXTERNAL proc open.file(value path.name[], access[], chan io.chan):
EXTERNAL proc close.file(chan io.chan) :
EXTERNAL proc str.to.chan(chan c, value s[]) :
EXTERNAL proc fp.num.to.chan(chan c, value float f) :
EXTERNAL proc fp.num.from.chan(chan c, var float f) :
EXTERNAL proc num.to.chan(chan c, value n ) :
EXTERNAL proc num.from.chan(chan c, var n) :
EXTERNAL proc str.to.screen(value s[]) :
EXTERNAL proc fp.num.to.screen(value float f) :
EXTERNAL proc num.to.screen(value n ) :
EXTERNAL proc fp.num.from.keyboard(var float f) :
EXTERNAL proc num.from.keyboard(var n) :

-- plug to expand system -each plug point can be an m*m isa grid

EXTERNAL proc plug(chan wn,we,ws,ww,rn,re,rs,rw,
                   in,is,sw,se             ) :

-- plug/processor grid allocation function

PROC loc(VALUE i,j, VAR r) =
  SEQ
    r := (((i-1)*(n+1))+j)-1 :

-- sequential to parallel program bus expander

PROC source(CHAN out[], link, VALUE t)=
  VAR k,i,j,buffer[n] :
  CHAN ptr :
  SEQ
    IF
      t = 0
        open.file("selector","r",ptr)
      TRUE
        open.file("instruct","r",ptr)
    num.from.chan(ptr,k)
    link!k
    SEQ i=[1 for k]
      SEQ
        IF
          i > k
            PAR j=[1 for n]
              VAR t1 :
              SEQ
                loc(j,1,t1)
                out[t1]!0
          TRUE
            SEQ
              SEQ j=[1 for n]
                num.from.chan(ptr,buffer[j-1])
              PAR j=[ 1 for n]
                VAR t1 :
                SEQ
                  loc(j,1,t1)
                  out[t1]!buffer[j-1]
    close.file(ptr)
    str.to.screen("*n Source closed")
    link!0 :


-- Garbage collector

PROC sink( CHAN in[], link ) =
  VAR i,j, k :
  SEQ
    link?k
    SEQ i=[1 for k]
      PAR j =[1 for n]
        VAR t1 :
        SEQ
          loc(j,n,t1)
          in[t1+1]?any
    str.to.screen("*nSink closed")
    link?any :

-- data bus expander

PROC data.source( CHAN ans[],bns[],awe[],bwe[],link ) =
  DEF n2=2*n,n3=3*n :
  VAR k,i,j,t :
  VAR FLOAT buffer[4*n] :
  CHAN ptr :
  SEQ
    open.file("datain","r",ptr)
    num.from.chan(ptr,k)
    link!k
    str.to.screen("*nk = ")
    num.to.screen(k)
    SEQ i=[1 for k ]
      SEQ
        str.to.screen("*ni = ")
        num.to.screen(i)
```

```
      SEQ j=[ 0 for 4]
        IF
          i <= k
          SEQ
            num.from.chan(ptr,t)
            IF
              t < 0
              SEQ z =[0 for n]
                buffer[(j*n)+ z] := 0.0
              TRUE
                SEQ z =[ 0 for n]
                  fp.num.from.chan(ptr,buffer[(j*n)+z])
          TRUE
            SEQ z=[0 for n]
              buffer[(j*n)+z] := 0.0
      PAR j=[1 for n]
        VAR t1,t2 :
        SEQ
          loc(j,1,t1)
          loc(j,n,t2)
          t2 := t2 + 1
          PAR
            bns[t1]!buffer[j-1]
            bwe[t2]!buffer[n+(j-1)]
            awe[t1]!buffer[n3+(j-1)]
            ans[t2]!buffer[n2+(j-1)]
    close.file(ptr)
    str.to.screen("*n Data Source closed")
    link!0 :

-- parallel to sequential bus condenser

PROC data.sink( CHAN ans[],bns[],awe[],bwe[], link) =
  DEF n2=2*n, n3=3*n :
  VAR k,i,j :
  VAR FLOAT buffer[4*n] :
  CHAN ptr :
  SEQ
    open.file("dataout","w",ptr)
    num.from.chan(ptr,k)
    link?k
    SEQ i=[1 for k]
      SEQ
        PAR j=[1 for n]
          VAR t1,t2 :
          SEQ
            loc(j,1,t1)
            loc(j,n,t2)
            t2 := t2 + 1
            PAR
              ans[t1]?buffer[j-1]
              awe[t2]?buffer[n+(j-1)]
              bns[t2]?buffer[n2+(j-1)]
              bwe[t1]?buffer[n3+(j-1)]
        SEQ
          SEQ j=[0 for 4]
            SEQ
              str.to.chan(ptr,"*n")
              SEQ z=[0 for n]
                SEQ
                  fp.num.to.chan(ptr,buffer[(j*n)+z])
                  str.to.chan(ptr," ")
          str.to.chan(ptr,"*n")
    close.file(ptr)
    str.to.screen("*n Data sink closed")
    link?any
    abort.program :

-- main
-- setups and starts the isa grid

DEF size = n*(n+1) :
CHAN  ans[size],bns[size], awe[size],bwe[size],sel[size], ins[size] :
CHAN link[3] :
VAR i,j :
PAR
  -- The grid
  PAR i=[1 for n]
    PAR j=[1 for n]
      VAR t1,t2,t3,t4 :
      SEQ
        loc(i,j,t1)
        loc(j,i,t2)
        t3 :=t1+1
        t4 := t2 + 1
        plug(ans[t2],awe[t3],bns[t4],bwe[t1], bns[t2],bwe[t3],
           ans[t4],awe[t1], ins[t2],ins[t4],sel[t1],sel[t3] )
  -- program interface
  source(sel,link[0], 0)
  sink(sel,link[0])

  source(ins,link[1],1)
  sink(ins,link[1])

  -- data input/output
  data.source(ans,bns,awe,bwe,link[2])
  data.sink(ans,bns,awe,bwe,link[2])
```

```
-- program 15 b
--
-- single processor plug
--
EXTERNAL proc PE(CHAN wn,we,ws,ww,rn,re,rs,rw,in,is,sw,se) :

LIBRARY PROC plug(CHAN wn,we,ws,ww,rn,re,rs,rw,in,is,sw,se) =
  SEQ
    PE(wn,we,ws,ww,rn,re,rs,rw,in,is,sw,se) :
```

```
-- program 15 c
--
-- plug for a grid of processors
--
EXTERNAL proc str.to.screen(value s[] ) :
EXTERNAL proc PE(CHAN wn,we,ws,ww, rn,re,rs,rw, in,is,sw,se ) :

PROC i.o.port(CHAN in,out, VALUE type ) =
  -- plug bus expander
  VAR float tmp1 :
  VAR tmp2 :
  SEQ
    IF
      type = 0
        SEQ
          in?tmp2
          out!tmp2
      type = 1
        SEQ
          in?tmp1
          out!tmp1 :

LIBRARY PROC plug(CHAN wn,we,ws,ww,rn,re,rs,rw,
                  in,is,sw,se                ) =
  -- sqr(p.size) to 1 plug
  DEF p.size = 2, size = p.size * (p.size + 1):
  CHAN ans[size], bns[size], awe[size], bwe[size], sel[size], ins[size] :
  SEQ
    -- virtual processor grid
    PAR
      PAR i =[ 1 for p.size]
        PAR j = [ 1 for p.size]
          VAR t1,t2,t3,t4 :
          SEQ
            t1 := (((i-1)*(p.size+1))+j)-1
            t2 := (((j-1)*(p.size+1))+i)-1
            t3 := t1 + 1
            t4 := t2 + 1
            PE( ans[t2], awe[t3], bns[t4], bwe[t1], bns[t2], bwe[t3],
                ans[t4], awe[t1], ins[t2], ins[t4], sel[t1], sel[t3] )
    -- plug spoolers
      WHILE true
        PAR
          SEQ j =[ 1 for p.size]
            VAR t1,t2 :
            SEQ
              t1 := (j-1)*(p.size+1)
              t2 := (((j-1)*(p.size+1))+p.size)-1
              t2 := t2 + 1
              PAR
                i.o.port(in,ins[t1],0)
                i.o.port(sw,sel[t1],0)
                i.o.port(rn,bns[t1],1)
                i.o.port(re,bwe[t2],1)
```

```
      i.o.port(rs,ans[t2],1)
      i.o.port(rw,awe[t1],1)
SEQ j = [ 1 for p.size]
  VAR t1,t2 :
  SEQ
    t1 := (j-1)*(p.size+1)
    t2 := (((j-1)*(p.size+1))+p.size)-1
    t2 := t2 + 1
    PAR
      i.o.port(ins[t2],is,0)
      i.o.port(sel[t2],se,0)
      i.o.port(ans[t1],wn,1)
      i.o.port(awe[t2],we,1)
      i.o.port(bns[t2],ws,1)
      i.o.port(bwe[t1],ww,1) :
```

```
-- program 15 d

-- general processor to illustrate the development of a PE for
-- the ISA grid, it is placed in the grid by a plug procedure
-- which allows the same defintion to implement a grid of processors
-- and is control by a Assembler program generated by the risal.p compiler
--
LIBRARY PROC PE(CHAN wn,we,ws,ww,rn,re,rs,rw,
                in,is,sw,se           )=
  DEF msize = 10:
  VAR FLOAT a,b, mem[msize],c, i.o.buf[4] :
  VAR i,j,s,port,p[4],fd[4],op,old.i,old.s :
  VAR running :
  SEQ
    running := true
    mem[1] := 0.0
    mem[0] := 0.0
    old.i := 0
    old.s := 0
    WHILE running
      SEQ
        -- Fetch Instruction
        c := mem[1]
        PAR
          in?i
          is!old.i
          sw?s
          se!old.s
          wn!c
          we!c
          ws!c
          ww!c
          rn?i.o.buf[0]
          re?i.o.buf[1]
          rs?i.o.buf[2]
          rw?i.o.buf[3]
        old.s := s
        old.i := i
        -- Decode intstruction
        SEQ
          SEQ j =[0 for 4]
            SEQ
              fd[j] := i\100
              i := i/100
          port := fd[2]
          op := fd[3]
        -- Communication enable
        SEQ
          SEQ i=[0 for 4]
            SEQ
              p[i] := port\2
              port := port/2
          SEQ i=[0 for 4]
            IF
```

```
      p[i] = 1
        mem[i+3] := i.o.buf[i]
-- Execute instruction
a := mem[fd[1]]
b := mem[fd[0]]
IF
  (s<>0) AND (op <> 0)
    IF
      op = 1
        mem[1] := mem[0]
      op = 2
        mem[0] := a + b
      op = 3
        mem[0] := a - b
      op = 4
        mem[0] := a * b
      op = 5
        mem[0] := a / b
      op = 6
        SEQ
          IF
            a < b
              mem[0] := a
            TRUE
              mem[0] := b
      op = 7
        SEQ
          IF
            a > b
              mem[0] := a
            TRUE
              mem[0] := b
      op = 8
        mem[1] := mem[fd[1]]
      op = 9
        mem[fd[0]]:= a    :
```

```
-- program 15 e

-- special processor for the simulation of the gentleman & kung
-- gaussian elimination algorithm used in Least squares approximation
--
LIBRARY PROC PE(CHAN wn,we,ws,ww,rn,re,rs,rw,
                in,is,sw,se             )=
  DEF msize = 10:
  VAR FLOAT a,b, mem[msize],c, i.o.buf[4] :
  VAR i,j,s,port,p[4],fd[4],op,opd1,opd2,old.i,old.s :
  VAR type ,toggle :
  SEQ
    type := 0
    mem[1] := 0.0
    mem[0] := 0.0
    old.i := 0
    old.s := 0
    c := 0.0
    toggle := false
    WHILE true
      SEQ
        -- Fetch Instruction
        c := mem[7]
        PAR
          in?i
          is!old.i
          sw?s
          se!old.s
          wn!c
          we!mem[4]
          ws!mem[5]
          ww!c
          rn?i.o.buf[0]
          re?i.o.buf[1]
          rs?i.o.buf[2]
          rw?i.o.buf[3]
        old.s := s
        old.i := i
        -- Decode intstruction
        SEQ
          SEQ j =[0 for 4]
            SEQ
              fd[j] := i\100
              i := i/100
          port := fd[2]
          op := fd[3]
          opd1 := fd[1]
          opd2 := fd[0]
        -- communication enable
        SEQ
          SEQ i=[0 for 4]
            SEQ
              p[i] := port\2
              port := port/2
```

```
  SEQ i=[0 for 4]
    IF
      p[i] = 1
        mem[i+3] := i.o.buf[i]
-- Execute instruction
IF
  (s<>0) AND (op <> 0)
    IF
      op = 8
        SEQ
          IF
            opd2 = 3
              SEQ
                IF
                  opd1 = 1
                    SEQ
                      old.i := old.i + 100
                      toggle := true
                      mem[7]:= mem[3]
                  opd1 = 0
                    SEQ
                      IF
                        type = 0
                          mem[3] := mem[3] - (mem[7]*mem[6])
                        type = 1
                          mem[6] := mem[3]/mem[7]
                      IF
                        toggle
                          SEQ
                            old.i := old.i + 100
                            toggle := false
              mem[5] := mem[3]
              mem[4] := mem[6]
            opd2 = 2
              SEQ
                type := 1
                old.i := old.i + 3
            opd2 = 1
              SEQ
                old.i := old.i + 1
                type := 0
            opd2 = 4
              SEQ
                mem[7] := mem[5]
            opd2 = 5
              type := 3 :
```

```
-- program 15 f
--
-- generic PE for testing ISA grid
--
-- NOTES : specialized processsor for performing LU-Decomposition
--         on an hexagonally connected array of kung & leiserson
--         which is simulated on an orthogonally connected ISA

LIBRARY PROC PE(CHAN wn,we,ws,ww,rn,re,rs,rw,
                in,is,sw,se              )=
  -- small 10 location memory
  DEF msize = 10:
  VAR FLOAT a,b, mem[msize],i.o.buf[4] :
  VAR i,j,s,port,p[4],fd[4],op,opd1,opd2,old.i,old.s :
  VAR type ,toggle :
  SEQ
    -- intialisation
    type := 0
    mem[1] := 0.0
    mem[0] := 0.0
    old.i := 0
    old.s := 0
    toggle := false
    -- start processor
    WHILE true
      SEQ
        -- Fetch Instruction
        PAR
          in?i
          is!old.i
          sw?s
          se!old.s
          wn!mem[3]
          we!mem[4]
          ws!mem[5]
          ww!mem[6]
          rn?i.o.buf[0]
          re?i.o.buf[1]
          rs?i.o.buf[2]
          rw?i.o.buf[3]
        old.s := s
        old.i := i
        -- Decode intstruction
        SEQ
          SEQ j =[0 for 4]
            SEQ
              fd[j] := i\100
              i := i/100
          port := fd[2]
          op := fd[3]
          opd1 := fd[1]
          opd2 := fd[0]
        -- communication enable
        SEQ
```

```
  SEQ i=[0 for 4]
    SEQ
      p[i] := port\2
      port := port/2
  SEQ i=[0 for 4]
    IF
      p[i] = 1
        mem[i+3] := i.o.buf[i]
-- Execute instruction
IF
  (s<>0) AND (op <> 0)
    IF
      op = 8
        SEQ
          IF
            opd2 = 5
              SEQ
                IF
                  toggle
                    SEQ
                      mem[6] := mem[5]
                      mem[4] := mem[7]
                      mem[5] := mem[8]
                  type = 1
                    SEQ
                      mem[7] := mem[6]
                      mem[8] := mem[3]
                      mem[3] := mem[4] + (mem[8]*mem[7])
                  type = 2
                    SEQ
                      mem[7] := mem[6]
                      mem[8] := mem[3] + (mem[4]*mem[7])
                      mem[3] := mem[4]
                  type = 3
                    SEQ
                      mem[7] := mem[6] + (mem[3]*mem[4])
                      mem[8] := mem[3]
                      mem[3] := mem[7]
                  type = 4
                    SEQ
                      IF
                        mem[4] = 0.0
                          mem[8] := 0.0
                        true
                          mem[8] := 1.0/mem[4]
                      mem[3] := mem[4]
                      mem[7] := -1.0
                toggle := not toggle
            opd2 = 4
              SEQ
                old.i := old.i -1
                type := 4
            opd2 = 3
              type := 3
            opd2 = 2
              SEQ
                old.i := old.i - 1
                type := 2
            opd2 = 1
              type := 1
            (opd2 = 0) and (opd1 > 0)
              SEQ
                IF
                  (opd1 -1 ) = 0
                    old.i := (old.i - 100) + 5
                  true
                    old.i := old.i - 100 :
```

```
-- program 16 a

-- Implementation of Systolic Simplex
--
-- Notes : The systolic array implements a tableau method in which
--         basic cells correspond to elements in the Simplex Tableau.
--         The array is expressed in top down fashion with three files
--         the current files sets up Host interfacing and the basic simplex
--         array, the remaining program files define basic cell definitions
--         and represent logical partitions of the design into subarrays.
--         The program partitioning allows the computation and control of
--         indivdual cells to be assessed without recourse to the lengthy
--         array setup procedures.

-- Problem dependent constants - the tableau dimensions
DEF d1 =6, d2 =4, size.1 = (d1+1)*d2, size.2 = (d2+1)*d1 :
CHAN link[5], port[(d1+1)*(d2+1)] :

-- Interfacing routines for input/output

EXTERNAL proc abort.program :
EXTERNAL proc open.file(value path.name[], access[], chan io.chan) :
EXTERNAL proc close.file(chan io.chan) :
EXTERNAL proc fp.num.to.chan(chan c,value float f) :
EXTERNAL proc num.to.chan(chan c, value n) :
EXTERNAL proc str.to.chan(chan c, value s[]) :
EXTERNAL proc str.to.screen( value s[] ) :
EXTERNAL proc num.to.screen(value n):
EXTERNAL proc num.from.keyboard(var n) :
EXTERNAL proc fp.num.to.screen(value float f) :
EXTERNAL proc fp.num.to.screen.f(value float f, value w,d):
EXTERNAL proc fp.num.from.keyboard(var float f) :

-- The required basic cells, and some extra procedures to
-- keep data flowing in Occam

EXTERNAL proc row.sort(chan Nin,Sout,Ncin,Scout,
                           Nout,Sin,Ncout,Scin,
                           Eout,Ein,Ecout,Ecin,port,
                       value type, var j ) :
EXTERNAL proc col.sort(chan Win,Eout,Wcin,Ecout,
                           Wout,Ein,Wcout,Ecin,
                           Nin,Nout,Ncin,Ncout,port,
                       value type,j ) :
EXTERNAL proc dummy.a(chan  Nin,Ncin,Nout,Ncout,
                           Eout,Ein,Ecout,Ecin, link[],port) :
EXTERNAL proc dummy.b(chan  Eout,Ecout,Ein,Ecin,
                           Nin,Nout,Ncin,Ncout, link[],port) :
EXTERNAL proc t.anchor(chan Nout,Nin,Ncin,Ncout) :
EXTERNAL proc r.anchor(chan Eout,Ein,Ecin,Ecout) :
EXTERNAL proc x.cell(CHAN Nin,Sout,Nout,Sin,Win,Eout,Wout,Ein,
                     VAR c[], c.bar[], flag[],
                     VAR FLOAT a,b,d  ) :
EXTERNAL proc pivot.row.1(CHAN Nin,Sout,Nout,Sin,Win,Eout,Wout,Ein,
                          VAR c[], c.bar[], flag[],
                          VAR FLOAT a,b,d  ):
EXTERNAL proc h.cell(CHAN Nin,Sout,Nout,Sin,Win,Eout,Wout,Ein,
                          VAR c[], c.bar[], flag[],
                          VAR FLOAT a,b,d ):
EXTERNAL proc pivot.col.b(CHAN Nin,Sout,Nout,Sin,Win,Eout,Wout,Ein,
                          VAR c[], c.bar[], flag[],
                          VAR FLOAT a,b,d ) :
EXTERNAL proc pivot(CHAN Nin,Sout,Nout,Sin,Win,Eout,Wout,Ein, link,
                    VAR c[], c.bar[],flag[],
                    VAR FLOAT a,b,d ) :
EXTERNAL proc pivot.col.a(CHAN Nin,Sout,Nout,Sin,Win,Eout,Wout,Ein,
                          VAR c[], c.bar[], flag[],
                          VAR FLOAT a,b,d ):
EXTERNAL proc h.j.cell(CHAN Nin,Sout,Nout,Sin,Win,Eout,Wout,Ein,
                       VAR c[], c.bar[], flag[],
                       VAR FLOAT a,b,d  ):
EXTERNAL proc pivot.row.r(CHAN Nin,Sout,Nout,Sin,Win,Eout,Wout,Ein,
                          VAR c[], c.bar[], flag[],
                          VAR FLOAT a,b,d  ) :
EXTERNAL proc mat.cell(CHAN Nin,Sout,Nout,Sin,Win,Eout,Wout,Ein,
                       VAR c[], c.bar[], flag[],
                       VAR FLOAT a,b,d  ):

-- Procedures concerned with setting up the Tableau

PROC PE( CHAN Nin, Sout, Ncin, Scout,
              Nout,Sin , Ncout,Scin ,
              Win ,Eout, Wcin  ,Ecout,
              Wout,Ein , Wcout,Ecin , port,
         VALUE p,q,VAR FLOAT a ) =
  -- the basic grid processing element, the celltype is
  -- fixed by passing the grid position to the PE
  --
  -- Variables :
  --      flag    - defines the state (flag[2]) of computation and
  --                provides auxilary variables for staggered cell i/o
  --      c,c.bar - the input and output control values
  --      test    - as for all other procdeures allows the inclusion
  --                of s.mix for trace purposes
  --
  VAR running,c[4],c.bar[4],flag[3],k,test :
  VAR FLOAT b,d  :
  SEQ
    test := true
    running := true
    SEQ k =[0 FOR 3]
      flag[k] := 0
    SEQ k=[0 FOR 4]
      c.bar[k] := 0
    WHILE running
      SEQ
--      port!a;c[0];c[1];c[2];c[3]
        -- control i/o
```

```
IF
  test
    PAR
      IF
        c.bar[3] <> 6
          PAR
            Wcin?c[1]
            Wcout!c.bar[1]
      IF
        c.bar[2] <> 6
          PAR
            Scin?c[0]
            Scout!c.bar[0]
      IF
        c.bar[1] <> 6
          PAR
            Ecin?c[3]
            Ecout!c.bar[3]
      IF
        c.bar[0] <> 6
          PAR
            Ncin?c[2]
            Ncout!c.bar[2]
-- decide for closedown
IF
  c.bar[3] = 6
    c[3] := 6
IF
  c.bar[2] = 6
    c[2] := 6
c.bar[2] := c[0]
-- evaluate new output controls
-- and if ready to stop
c.bar[3] := c[1]
c.bar[0] := c[2]
c.bar[1] := c[3]
test := (c.bar[2]=6) AND (c.bar[3]=6)
test := NOT((((c.bar[0]=6)AND(c.bar[1]=6))AND test))
running := test
IF
-- select the correct cell
-- type
  q = 1
    SEQ
      IF
        p < (d2 - 1)
          x.cell(Nin,Sout,Nout,Sin,Win,Eout,Wout,Ein,c,c.bar,flag,a,b,d)
        p = (d2 - 1)
          pivot.row.l(Nin,Sout,Nout,Sin,Win,Eout,Wout,Ein,
                      c,c.bar,flag,a,b,d)
        p = d2
          h.cell(Nin,Sout,Nout,Sin,Win,Eout,Wout,Ein,c,c.bar,flag,a,b,d)
  q = 2
    SEQ
      IF
        p = d2
          pivot.col.b(Nin,Sout,Nout,Sin,Win,Eout,Wout,Ein,
                      c,c.bar,flag,a,b,d)
        (p = (d2 -1)) AND (flag[2] <> 4)
          pivot(Nin,Sout,Nout,Sin,Win,Eout,Wout,Ein,
                link[2],c,c.bar,flag,a,b,d)
        p < (d2 - 1)
          pivot.col.a(Nin,Sout,Nout,Sin,Win,Eout,Wout,Ein,
                      c,c.bar,flag,a,b,d)
  q > 2
    SEQ
      IF
        p = d2
          h.j.cell(Nin,Sout,Nout,Sin,Win,Eout,Wout,Ein,
                   c,c.bar,flag,a,b,d)
        p = (d2 - 1)
          pivot.row.r(Nin,Sout,Nout,Sin,Win,Eout,Wout,Ein,
                      c,c.bar,flag,a,b,d)
        p < (d2 - 1)
          mat.cell(Nin,Sout,Nout,Sin,Win,Eout,Wout,Ein,
                   c,c.bar,flag,a,b,d) :

proc s.mix(CHAN port[]) =
  --
  -- screen mixer for tracing
  --
  -- provides a series of tableaus one for each
  -- cell cycle. when s.mix is not used the port channel
  -- used in other procedures is commented out
  --
  chan fptr :
  var float a:
  var c[4],z,b,running,flag :
  seq
    -- output to screen and file "result"
    -- produces a control wavefront or
    -- a record of the updated table
    open.file("result","w",fptr)
    str.to.screen("*n options")
    str.to.screen("*n1: Table *n2: control wavefront")
    str.to.screen("*n*n Enter option :")
    num.from.keyboard(b)
    num.to.screen(b)
    running := true
    flag := 0
    -- for all cycles
    while running
      seq
        z := 0
        str.to.screen("*nstart cycle*n")
        str.to.chan(fptr,"*nstart cycle *n")
        seq i =[0 for (d1+1)*(d2+1)]
          seq
```

```
          z := z + 1
          port[i]?a;c[0];c[1];c[2];c[3]
          str.to.screen(" ")
          str.to.chan(fptr," ")
          IF
            b = 1
              seq
                fp.num.to.chan(fptr,a)
                fp.num.to.screen.f(a,8,4)
            true
              seq j=[0 for 4]
                seq
                  num.to.screen(c[j])
                  num.to.chan(fptr,c[j])
          str.to.screen(" ")
          str.to.chan(fptr," ")
          if
            z = (d1+1)
              seq
                z := 0
                str.to.chan(fptr,"*n")
                str.to.screen("*n")
          -- termination conditions
          if
            c[0] = 7
              flag := 1
            c[0] = 8
              flag := 2
        -- termination is produced by
        -- the tableau itself
        if
          flag = 1
            seq
              str.to.screen("*n correct termination")
              str.to.chan(fptr,"*n correct termination")
              running := false
          flag = 2
            seq
              str.to.screen("*n all pivot contenders invalid")
              str.to.chan(fptr,"*n all pivot contenders invalid")
              running := false
    -- rather primitive but
    -- effective program stop
    close.file(fptr)
    abort.program :

PROC sink(CHAN Eout,Win,Ecout,Wcin, VALUE type) =
  --
  -- As with all occam programs for systolic arrays
  -- we must collect used signals on the array bounadries.
  --
  VAR running, c, flag :
  SEQ
    flag := 1
    running := true
    -- the type of cell dictates
    -- the signals it will recieve
    -- inturn dependent upon position in the
    -- array.
    WHILE running
      SEQ
        PAR
          Ecout?c
          Wcin!0
        IF
          c = 6
            running := false
          (c=1) AND (flag = 1)
            SEQ
              Eout?c
              IF
                type = 0
                  flag := 2
          (c=3) AND (flag = 2)
            SEQ
              flag := 1 :

PROC source(CHAN Nin,Nout,Ncin,Ncout,VALUE type) =
  --
  -- The source routine also does the job of a sink
  -- but can also be used for the systolic loading
  -- of data in the minimum time. This is not
  -- implement here but is trivial
  --
  VAR running ,c, flag :
  SEQ
    flag := 1
    running := true
    WHILE running
      SEQ
        PAR
          Ncin!0
          Ncout?c
        IF
          c = 6
            running := false
          (type=0) AND (c=1)
            SEQ
              Nout?any
          type = 1
            IF
              (c=1) AND (flag=1)
                SEQ
                  Nout?any
                  flag := 2
              (c=3) AND (flag=2)
                flag := 3
              (c=1) AND (flag=3)
```

```
            SEQ
              Nout?any
              flag := 1
        type = 2
          IF
            (c=1) AND (flag = 1)
              SEQ
                Nout?any
                flag := 2
            (c=3) AND (flag = 2)
              flag := 1 :

-- Simple 2-D to 1-D vector mappings useful
-- for locating data elements , PE's and channels
-- in the design

PROC loc.r(VALUE i,j, VAR r) =
  SEQ
    r := (((i-1)*(d1+1))+j)-1 :

PROC loc.c(VALUE j,i, VAR r) =
  SEQ
    r := (((j-1)*(d2+1))+i)-1 :

PROC loc.m(VALUE i,j, VAR r ) =
  SEQ
    r := (((i-1)*d1)+j)-1 :

PROC setup (VAR FLOAT mem[], VAR cind[],rind[] ) =
  --
  -- Host interface to read the tableau to be used
  -- also collects the rowand column indices
  --
  VAR t1 :
  SEQ
    -- constraint and basis matrices
    str.to.screen("*n Enter Tableau*n")
    SEQ i=[1 FOR d2]
      SEQ
        SEQ j = [ 1 FOR d1]
          SEQ
            loc.m(i,j,t1)
            fp.num.from.keyboard(mem[t1])
            fp.num.to.screen(mem[t1])
            str.to.screen(" ")
        str.to.screen("*n")
    -- column vectors index
    str.to.screen("*n*nEnter Column Indices")
    SEQ i=[0 FOR d1-1]
      SEQ
        str.to.screen("*n[")
        num.to.screen(i)
        str.to.screen("] =")
        num.from.keyboard(cind[i])
        num.to.screen(cind[i])
    -- indexes of unknowns in solution
    str.to.screen("*n*nEnter Row Indices")
    SEQ i=[0 FOR d2-1]
      SEQ
        str.to.screen("*n[")
        num.to.screen(i)
        str.to.screen("] = ")
        num.from.keyboard(rind[i])
        num.to.screen(rind[i])
    str.to.screen("*n") :

PROC putdata(VALUE FLOAT mem[], VALUE rind[]) =
  --
  -- Host output interface
  -- simply writes final value of objective function(H)
  -- and unknowns in solution with optimal values
  -- output to file and screen
  --
  CHAN fptr :
  VAR t1 :
  SEQ
    open.file("ftab","w",fptr)
    str.to.screen("*n results")
    str.to.chan(fptr,"*n  results *n")
    SEQ i=[ 1 for d2]
      SEQ
        str.to.screen("*n  [")
        str.to.chan(fptr,"*n  [")
        IF
          i = d2
            SEQ
              str.to.screen("H ")
              str.to.chan(fptr,"H ")
          TRUE
            SEQ
              num.to.screen(rind[i-1])
              num.to.chan(fptr,rind[i-1])
        str.to.screen("] = ")
        str.to.chan(fptr,"] = ")
        loc.m(i,1,t1)
        fp.num.to.chan(fptr,mem[t1])
        fp.num.to.screen(mem[t1])
    close.file(fptr) :

PROC merge(CHAN link[],port) =
  --
  -- The computation can terminate because of
  --  a) optimal solution
  --  b) failure to find optimal solution
  -- also
  --  c) another iteration
  -- the control signals to the pivot are merged here
  -- from row sorting and column sorting
```

```
  --
  VAR FLOAT d :
  VAR running, c[3],test :
  SEQ
    d := 0.0
    c[2] := 1
    running := true
    test := true
    -- computation
    WHILE running
      SEQ
        str.to.screen("c")
--      port!d;c[0];c[1];c[2];0
        -- i/o
        IF
          test
            PAR
              IF
                (c[0] <> 6) AND (c[1] <> 6)
                  PAR
                    link[0]?c[0]
                    link[1]?c[1]
              link[2]!c[2]
        -- decide pivot signal
        IF
          c[2] = 6
            SEQ
              test := false
              running := false
          (c[0]=1) OR (c[1]=1)
            c[2] := 1
          (c[0]=6) OR (c[1]=6)
            c[2] := 6
          TRUE
            c[2] := 0 :


-- main

-- Specification of the whole Tableau
-- channels are for two communication north-south(ns) and east-west(ew)
-- procedure call s.mix can be uncommented to produce trace

CHAN ns.1[size.2], ns.2[size.2], cns.1[size.2], cns.2[size.2] :
CHAN ew.1[size.1], ew.2[size.1], cew.1[size.1], cew.2[size.1] :
VAR FLOAT mem[d1*d2] :
VAR c.index[d1-1], r.index[d2-1] :
SEQ
  setup(mem,c.index,r.index)
  PAR
    -- tableau
    VAR i,j :
    PAR i= [1 FOR d2]
      PAR j= [1 FOR d1]
        VAR t1,t2,t3,t4,t5,t6 :
        SEQ
          loc.m(i,j,t5)
          loc.r(i,j,t1)
          loc.r(i,j+1,t6)
          loc.c(j,i,t2)
          t3 := t1 + 1
          t4 := t2 + 1
          PE( ns.1[t2], ns.1[t4], cns.1[t2], cns.1[t4],
              ns.2[t2], ns.2[t4], cns.2[t2], cns.2[t4],
              ew.1[t1], ew.1[t3], cew.1[t1], cew.1[t3],
              ew.2[t1], ew.2[t3], cew.2[t1], cew.2[t3], port[t6],
              i, j , mem[t5] )
    merge(link,port[d2*(d1+1)])
    -- s.mix(port)
    -- clear boundary channels
    VAR i :
    PAR i=[1 FOR d2]
      VAR t1,type :
      SEQ
        type := 0
        IF
          i = d2
            type := 1
        loc.r(i,d1+1,t1)
        sink(ew.1[t1], ew.2[t1], cew.1[t1], cew.2[t1], type)
    -- Top Boundary Sources
    VAR j :
    PAR j =[1 FOR d1]
      VAR t1,type :
      SEQ
        type := 2
        IF
          j = 1
            type := 0
          j = 2
            type := 1
        loc.c(j,1,t1)
        source(ns.1[t1], ns.2[t1], cns.1[t1], cns.2[t1],type)
    -- Row Sorter
    VAR i :
    CHAN ns.3[d2], ns.4[d2], cns.3[d2],cns.4[d2] :
    SEQ
      PAR
        PAR i =[ 1 FOR d2]
          VAR t1,type :
          SEQ
            type := 1
            IF
              i = 1
                type := 2
              i = (d2-1)
                type := 0
            loc.r(i,1,t1)
```

```
          IF
            i = d2
              dummy.a(ns.3[d2-1],cns.3[d2-1],
                      ns.4[d2-1],cns.4[d2-1],
                      ew.1[t1], ew.2[t1], cew.1[t1], cew.2[t1], link,port[t1])
            TRUE
              row.sort(ns.3[i-1],ns.3[i],cns.3[i-1],cns.3[i],
                       ns.4[i-1],ns.4[i],cns.4[i-1],cns.4[i],
                       ew.1[t1], ew.2[t1], cew.1[t1],cew.2[t1],port[t1],
                       type,r.index[i-1])
      t.anchor(ns.3[0],ns.4[0],cns.3[0],cns.4[0])
  -- Column sorter
  VAR j :
  CHAN ew.3[d1], ew.4[d1], cew.3[d1], cew.4[d1] :
  SEQ
    PAR
      PAR j=[ 1 FOR d1]
        VAR t1,t2,type :
        SEQ
          type := 1
          IF
            j = 2
              type := 0
            j = d1
              type := 2
          loc.c(j,d2+1,t1)
          loc.r(d2+1,j+1,t2)
          IF
            j = 1
              dummy.b(ew.3[0],cew.3[0],
                      ew.4[0],cew.4[0],
                      ns.1[t1],ns.2[t1],cns.1[t1],cns.2[t1], link,port[t2])
            TRUE
              col.sort(ew.3[j-2],ew.3[j-1],cew.3[j-2],cew.3[j-1],
                       ew.4[j-2],ew.4[j-1],cew.4[j-2],cew.4[j-1],
                       ns.1[t1], ns.2[t1], cns.1[t1], cns.2[t1],port[t2],
                       type,c.index[j-2])
      r.anchor(ew.3[d1-1],ew.4[d1-1],cew.3[d1-1],cew.4[d1-1])
putdata(mem, r.index)
```

```
-- program 16 b

-- Column and Row sorter cell defintions
--
-- Notes : The systolic Simplex design is split into
--         three sub arrays the tableau, column , and row sorters
--         here the cells for the latter two are specified
--
EXTERNAL proc str.to.screen(value s[] ) :

LIBRARY proc col.sort(chan Win,Eout,Wcin,Ecout,
                           Wout,Ein,Wcout,Ecin,
                           Nin,Nout,Ncin,Ncout,port,
                           value type,k ) =
  --
  -- Column sorter cell finds the maximum piece
  -- which will reduce the objective function the most
  -- also generates a control value which will swap
  -- columns of the tableau as neccesary to place the vector
  -- corresponding to the varibale to be introduced into the
  -- solution into the pivot column of the tableau
  --
  VAR float a,b :
  VAR flag,toggle,running,left,right,j,i,test :
  VAR c.bar[3], c[3] :
  SEQ
    -- setup
    j := k
    left := (type=0) OR (type=1)
    right := (type=2)OR (type=1)
    flag := 0
    running := true
    test := true
    toggle := false
    SEQ i =[0 FOR 3]
      c.bar[i] := 0
    -- computation
    WHILE running
      SEQ
--      port!a;c[0];c[1];c[2];j
        -- i/o
        IF
          test
            PAR
              Ncin?c[0]
              Ecin?c[1]
              Ncout!c.bar[0]
              Ecout!c.bar[1]
              IF
                c.bar[1] <> 6
                  PAR
                    Wcin?c[2]
                    Wcout!c.bar[2]
        IF
```

```
  (type = 0) AND (c.bar[1]<>6)
    SEQ
      Wout!a;j
  c.bar[1] = 6
    SEQ
      test := false
      running := false
c.bar[0] := 0
c.bar[1] := 0
c.bar[2] := c[1]
-- cell computation
IF
  (flag=0) AND (c[0]=1)
    -- load cell
    SEQ
      Nin?a
      toggle := true
      c.bar[1] := 1
      flag := 1
  flag = 1
    -- sorting
    IF
      c[1] = 1
        -- stop sorting
        SEQ
          flag := 0
          c.bar[0] := 3
      toggle AND left
        SEQ
          Ein?b;i
          IF
            (b>a)OR((b=a)AND(i<j))
              SEQ
                Eout!a;j
                a := b
                j := i
                c.bar[0] := 1
            TRUE
              Eout!b;i
      (NOT toggle) AND right
        SEQ
          Wout!a;j
          Win?a;i
          IF
            i <> j
              SEQ
                j := i
                c.bar[0] := 2
  c[2] = 6
    -- closedown cell
    SEQ
      c.bar[0] := 6
      c.bar[1] := 6
      c.bar[2] := 6
```

```
        toggle := NOT toggle :

LIBRARY proc row.sort(chan Nin,Sout,Ncin,Scout,
                           Nout,Sin,Ncout,Scin,
                           Eout,Ein,Ecout,Ecin, port,
                           value type, var j ) =
  --
  -- Row sorting cell identifies a minimum p, from
  -- positive values. Ensures that row of tableau
  -- corresponding to index of variable ejected from
  -- solution is pivot row of cells
  VAR FLOAT a,b :
  VAR toggle,running,top,bottom,flag,i,test :
  VAR c.bar[3],c[3] :
  SEQ
    --  setup
    top := (type=2)OR(type=1)
    bottom := (type=1)OR(type=0)
    flag := 0
    running := true
    test := true
    toggle := false
    SEQ i=[0 FOR 3]
      c.bar[i] := 0
    -- computation
    WHILE running
      SEQ
--      port!a;c[0];c[1];c[2];j
        -- i/o
        IF
          test
            PAR
              Ecin?c[0]
              Ncin?c[1]
              Ecout!c.bar[0]
              Ncout!c.bar[1]
              IF
                c.bar[1] <> 6
                  PAR
                    Scin?c[2]
                    Scout!c.bar[2]
        IF
          (type =0) AND (c.bar[1] <> 6)
            SEQ
              Sout!a;j
          c.bar[1] = 6
            SEQ
              test := false
              running := false
        c.bar[0] := 0
        c.bar[1] := 0
        c.bar[2] := c[1]
        -- sorter
        IF
```

```
      (flag = 0) AND (c[0]=1)
        SEQ
          -- load
          Ein?a
          toggle := true
          c.bar[1] := c[0]
          flag := 1
      flag = 1
        IF
          c[1] = 1
            SEQ
              -- stop sort
              flag := 0
              c.bar[0] := 3
          toggle AND bottom
            SEQ
              Nin?b;i
              IF
                ((((b=a)AND(i<j))OR(b<a))OR(a<=0.0))
                  SEQ
                    Nout!a;j
                    a := b
                    j := i
                    c.bar[0] := 1
                TRUE
                  Nout!b;i
          (Not toggle) AND top
            SEQ
              Sout!a;j
              Sin?a;i
              IF
                i <> j
                  SEQ
                    c.bar[0] := 2
                    j := i
      c[2] = 1
        -- new varibale index
        Sin?j
      c[2] = 6
        -- kill cell
        SEQ
          c.bar[0] := 6
          c.bar[1] := 6
          c.bar[2] := 6
    toggle := NOT toggle :


LIBRARY proc dummy.a(chan Nin,Ncin,Nout,Ncout,
                         Eout,Ein,Ecout,Ecin, link[] ,port) =
  --
  -- This procedure recives signals from
  -- column sorter with data on the index of
  -- the new variable to be introduced to the
  -- solution. Also takes data from row sorter
  -- to determine a termination condition which
  -- is sent to pivot cellvia merge
  --
  VAR running,flag, c[7],i,j,j1,k, test :
  VAR FLOAT a,b :
  SEQ
    -- setup
    SEQ z=[0 FOR 7]
      c[z] := 0
    flag := 0
    running := true
    test := true
    -- computation
    WHILE running
      SEQ
--      port!a;c[4];c[0];c[1];j1
        -- i/o
        IF
          test
            PAR
              Ecout!c[5]
              Ecin?c[4]
              Nin?a;i
              IF
                c[3] <> 6
                  PAR
                    link[3]?c[3];j
                    link[4]!c[6]
                    link[0]!c[2]
              Ncin?c[0]
              Ncout!c[1]
        c[1] := 0
        c[2] := 0
        IF
          c[4] = 1
            Ein?b
        -- book keeping
        IF
          c[5] = 6
            SEQ
              running := false
              test := false
          c[3] = 1
            j1 := j
          (c[0]=1) AND (a>0.0)
            -- new solution
            SEQ
              c[2] := 1
              c[1] := 1
              flag := 1
          flag = 1
            -- swap unknowns
            SEQ
              Nout!j1
```

```
            j1 := 0
            flag := 0
          (c[0]=1) AND (a<=0.0)
            -- error
            SEQ
--            str.to.screen("*n Termination on Vij")
              c[1] := 6
              c[5] := 6
              c[6] := 6
              c[2] := 6
              c[4] := 8
          c[3] = 6
            SEQ
              c[5] := 6
              c[1] := 6
              c[2] := 6 :

LIBRARY proc dummy.b(chan Eout,Ecout,Ein,Ecin,
                         Nin,Nout,Ncin,Ncout, link[], port ) =
  --
  -- Similar routine to dummy.a except for
  -- column sorting. Decides if optimal soultion is found
  -- and terminates tableau. the value 6=closedown
  --
  VAR running,c[7], i,j,j1,k,test :
  VAR FLOAT a :
  SEQ
    -- setup
    running :=  true
    test := true
    SEQ z=[0 FOR 7]
      c[z] := 0
    -- computation
    WHILE running
      SEQ
--      port!a;c[0];c[1];c[3];j
        -- i/o
        IF
          test
            PAR
              Ncin?c[4]
              Ncout!c[5]
              IF
                c[6] <> 6
                  SEQ
                    link[4]?c[6]
                    link[3]!c[2];k
                    link[1]!c[3]
              Ein?a;i
              Ecin?c[0]
              Ecout!c[1]
        c[1] := 0
        c[2] := 0
        c[3] := 0
        -- book keeping
        IF
          c[5] = 6
            SEQ
              running := false
              test := false
          (c[0]=1) AND (a>0.0)
            -- best contender found
            SEQ
              k := i
              c[2] := 1
              c[3] := 1
          (c[0]=1) AND (a <=0.0)
            -- optimal soulution
            SEQ
--              str.to.screen("*ncorrect termination")
                c[3] := 6
                c[2] := 6
                c[5] := 6
                c[1] := 6
                c[0] := 7
          c[6] = 6
            -- kill the cell
            SEQ
              c[1] := 6
              c[5] := 6 :

-- additional routines to maintain data flow and
-- easy specification of the array

LIBRARY proc t.anchor(chan Sout,Sin,Scout,Scin) =
  -- associated with row sorter
  VAR running , csave1, csave2 :
  SEQ
    csave2 := 0
    running := true
    WHILE running
      SEQ
        PAR
          Scin?csave1
          Scout!csave2
        IF
          csave1 = 6
            running := false
        csave2 := csave1 :

LIBRARY proc r.anchor(chan Wout,Win,Wcin,Wcout) =
  -- associated with column sorter
  VAR running, csave1, csave2 :
  SEQ
    csave2 := 0
    running := true
    WHILE running
      SEQ
```

```
PAR
  Wcin?csave1
  Wcout!csave2
IF
  csave1 = 6
    running := false
csave2 := csave1 :
```

```
-- program 16 c

-- Basic cell computation defintions:
--
--Notes : these procedures are the basic cell defintions of the
--        array and will not be detailed here see accompanying report
--

-- two auxiliary procedures to swap tableau columns and rows

PROC column.swap( CHAN Ein,Eout, Win,Wout, VAR c,flag, VAR FLOAT a) =
  VAR FLOAT b :
  SEQ
    IF
      c = 1
        SEQ
          PAR
            Ein?b
            Eout!a
          a := b
      c = 2
        SEQ
          PAR
            Win?b
            Wout!a
          a := b
      c = 3
        SEQ
          flag := 2 :

PROC row.swap( CHAN Nin,Nout,Sin,Sout, VAR  c, flag,VAR FLOAT a) =
  VAR FLOAT b :
  SEQ
    IF
      c = 1
        SEQ
          PAR
            Nin?b
            Nout!a
          a := b
      c = 2
        SEQ
          PAR
            Sout!a
            Sin?b
          a := b
      c = 3
        flag := 0 :

-- The main computational procedures

LIBRARY proc mat.cell(CHAN Nin,Sout,Nout,Sin,Win,Eout,Wout,Ein,
                      VAR c[],c.bar[], flag[], VAR FLOAT a,b,d  ) =
  SEQ
```

```
    IF
      (flag[2]=0) AND ((c[0]=1) AND (c[1]=1))
        SEQ
          flag[2] := 1
          PAR
            Sin?b
            Win?d
          a := a - (b*d)
          flag[0] := 1
      flag[0] = 1
        SEQ
          PAR
            Nout!b
            Eout!d
          flag[0] := 0
      flag[2] = 1
        column.swap(Ein,Eout,Win,Wout,c[0],flag[2],a)
      flag[2] = 2
        flag[2] := 3
      flag[2] = 3
        row.swap(Nin,Nout,Sin,Sout, c[1],flag[2],a) :

LIBRARY PROC h.j.cell(CHAN Nin,Sout,Nout,Sin,Win,Eout,Wout,Ein,
                      VAR c[],c.bar[],flag[],VAR FLOAT a,b,d) =
  SEQ
    IF
      (flag[2] = 0) AND ((c[2]=1)AND(c[1]=1))
        SEQ
          flag[2] := 1
          PAR
            Nin?b
            Win?d
          a := a - (b*d)
          flag[0] := 1
      flag[0] = 1
        SEQ
          PAR
            Eout!d
            Sout!a
          flag[0] := 0
      flag[2] = 1
        SEQ
          column.swap(Ein,Eout,Win,Wout,c[0],flag[2],a)
      flag[2] = 2
        flag[2] := 0 :

LIBRARY PROC pivot.col.a (CHAN Nin,Sout,Nout,Sin,Win,Eout,Wout,Ein,
                          VAR c[], c.bar[], flag[],
                          VAR FLOAT a,b,d ) =
  SEQ
    IF
      (flag[2] =0) AND (c[0]=1)
        SEQ
          Sin?b
          flag[0] := 1
          flag[2] := 1
          c.bar[3] := 1
          c.bar[1] := 1
      flag[0] = 1
        SEQ
          PAR
            Nout!b
            Eout!a
            Wout!a
          a := 0
          flag[0] := 0
      flag[2] = 1
        column.swap(Ein,Eout,Win,Wout, c[0],flag[2],a)
      (flag[2] = 2) AND (c[0]=1)
        SEQ
          Sin?b
          IF
            a = 0.0
              d := 0.0
            TRUE
              d := 0.0 +(1.0/a)
          flag[1] := 1
          flag[2] := 3
          c.bar[1] := 1
      flag[1] = 1
        SEQ
          Nout!b
          wout!d
          flag[1] := 0
      (flag[2] = 3)
        row.swap(Nin,Nout,Sin,Sout,c[1],flag[2],a) :

LIBRARY PROC pivot.row.1( CHAN Nin,Sout,Nout,Sin,Win,Eout,Wout,Ein,
                          VAR c[], c.bar[] ,flag[],
                          VAR FLOAT a,b,d ) =
  SEQ
    IF
      (flag[2]=0) AND (c[3]=1)
        SEQ
          Ein?b
          a := 0.0 + (a*b)
          flag[0] := 1
          flag[2] := 2
          c.bar[2] := 1
          c.bar[1] := 0
          c.bar[0] := 1
      flag[0] = 1
        SEQ
          PAR
            Nout!a
            Sout!a
          flag[0] := 0
      (flag[2] = 2) AND (c[3] = 1)
```

```
        SEQ
          Ein?b
          d := 0.0 + (a*b)
          flag[1] := 1
          flag[2] := 3
      flag[1]
        SEQ
          Wout!d
          flag[1] := 0
      (flag[2] = 3)
        row.swap(Nin,Nout,Ein,Eout,c[1],flag[2],a) :

LIBRARY PROC pivot.col.b( CHAN Nin,Sout,Nout,Sin,Win,Eout,Wout,Ein,
                          VAR c[], c.bar[], flag[],
                          VAR FLOAT a,b,d ) =
  var zero :
  SEQ
    zero := 0.0
    IF
      (flag[2] = 0) AND (c[2]=1)
        SEQ
          Nin?b
          flag[0] := 1
          flag[2] := 1
          c.bar[1] := 1
          c.bar[3] := 1
      flag[0] = 1
        SEQ
          flag[0] := 0
          PAR
            Wout!a
            Eout!a
            Sout!zero
          a := 0.0
      flag[2] = 1
        column.swap(Ein,Eout,Win,Wout,c[0],flag[2],a)
      (flag[2] = 2)
        SEQ
          flag[2] := 0 :

LIBRARY PROC pivot.row.r(CHAN Nin,Sout,Nout,Sin,Win,Eout,Wout,Ein,
                         VAR c[], c.bar[], flag[],
                         VAR FLOAT a,b,d ) =
  SEQ
    IF
      (flag[2] = 0) AND (c[1] =1 )
        SEQ
          Win?b
          a := 0.0 + (a*b)
          c.bar[2] := 1
          c.bar[0] := 1
          flag[2] := 1
          flag[0] := 1
      flag[0] = 1
        SEQ
          PAR
            Nout!a
            Sout!a
            Eout!b
          flag[0] := 0
      (flag[2] =1)
        column.swap(Ein,Eout,Win,Wout,c[0], flag[2],a)
      flag[2] =2
        flag[2] := 3
      (flag[2] = 3)
        row.swap(Nin,Nout,Sin,Sout,c[1],flag[2],a) :

LIBRARY PROC x.cell(CHAN Nin,Sout,Nout,Sin,Win,Eout,Wout,Ein,
                    VAR c[], c.bar[], flag[],
                    VAR FLOAT a,b,d ) =
  SEQ
    IF
      (flag[2] =0) AND (( c[0]=1) AND (c[3]=1))
        SEQ
          PAR
            Sin?b
            Ein?d
          a := a - (b*d)
          flag[2] := 1
          flag[0] := 1
          c.bar[1] := 0
      flag[0] =1
        SEQ
          Nout!b
          flag[0] := 0
      flag[2] = 1
        flag[2] := 2
      (flag[2] = 2) AND (c[3] = 1)
        SEQ
          Ein?b
          d := 0.0 + (a*b)
          flag[2] := 3
          flag[1] := 1
      flag[1] = 1
        SEQ
          Wout!d
          flag[1] := 0
      (flag[2] = 3)
        row.swap(Nin,Nout,Sin,Sout,c[1],flag[2],a) :

LIBRARY PROC h.cell(CHAN Nin,Sout,Nout,Sin,Win,Eout,Wout,Ein,
                    VAR c[], c.bar[], flag[],
                    VAR FLOAT a,b,d ) =
  SEQ
    IF
      (c[2]=1) AND (c[3] =1)
        SEQ
          PAR
```

```
          Nin?b
          Ein?d
        a := a - (b*d)
        flag[0] := 1
    flag[0] = 1
      SEQ
        Wout!a
        flag[0] := 0 :

LIBRARY PROC pivot( CHAN Nin, Sout, Nout,Sin, Win,Eout, Wout,Ein,link,
            VAR c[],c.bar[], flag[], VAR FLOAT a,b,d  ) =
  VAR con :
  SEQ
    link?con
    IF
      con = 6
        flag[2] := 4
    (flag[2]=0) AND (con=1)
      SEQ
        flag[2] := 1
        c.bar[0] := 1
        c.bar[1] := 1
        c.bar[2] := 1
        c.bar[3] := 1
        IF
          a <> 0.0
            a := 1.0/a
        flag[0] := 1
    flag[0] = 1
      SEQ
        PAR
          Nout!a
          Wout!a
          Eout!a
          Sout!a
        flag[0] := 0
        a := 1.0
    flag[2] = 1
      SEQ
        column.swap(Ein,Eout,Win,Wout,c[0],flag[2],a)
    (flag[2] = 2) AND (con=1)
      SEQ
        IF
          a = 0.0
            d := 0.0
          TRUE
            d := 1.0/a
        c.bar[0] := 0
        c.bar[1] := 1
        c.bar[2] := 1
        c.bar[3] := 0
        flag[2] := 3
        flag[1] := 1
    flag[1] = 1
      SEQ
        PAR
          Nout!d
          Wout!d
        flag[1] := 0
    (flag[2]=3)
      row.swap(Nin,Nout,Sin,Sout,c[1],flag[2],a) :
```

```
-- program 17

-- Implementation of the Systolic Assignment Problem Iteration(API)
--
-- Notes : The program implements the API orthogonally connected mesh of
--         of (n+2)*(n+2) incorporating a n*n tableau mesh embedded in
--         a Systolic Control Ring (SCR). The array computes the final
--         reduced cost matrix from which an answer to the Assignment
--         problem can be produced. A trace can be included producing
--         Snapshots of computational wavefronts or the reduced cost
--         matrix including the row and column weights. The trace can be
--         used by commenting out assignments to running inside the cells
--         main loops. and including s.mix routine and the port commands
--         from each cell. The trace is removed by the reverse procedure.
--         All screen output is duplicated in files result for trace and
--         ftab with trace off.
--           Sorting is by Parallel bubblesort - or ODD-EVEN Trans-
--         position sort which requires O(n) cycle for a list of size
--         n to be sorted into ascending or descending order.

-- Problem dependent constants - the tableau dimensions
DEF n = 3, size = (n+2)*(n+1) :
CHAN port[(n+2)*(n+2)] :

-- Interfacing routines for input/output

EXTERNAL proc abort.program :
EXTERNAL proc open.file(value path.name[], access[], chan io.chan) :
EXTERNAL proc close.file(chan io.chan) :
EXTERNAL proc fp.num.to.chan(chan c,value float f) :
EXTERNAL proc num.to.chan(chan c, value n) :
EXTERNAL proc str.to.chan(chan c, value s[]) :
EXTERNAL proc str.to.screen( value s[] ) :
EXTERNAL proc num.to.screen(value n):
EXTERNAL proc num.from.keyboard(var n) :
EXTERNAL proc fp.num.to.screen(value float f) :
EXTERNAL proc fp.num.to.screen.f(value float f, value w,d):
EXTERNAL proc fp.num.from.keyboard(var float f) :

-- cell definitions

PROC row.sort(CHAN Nin,Nout, Sout,Sin, Win,Wout, port, VAR i ) =
  --
  -- Row sorting cell : Filters SCR signals when not in use
  --                    contains the row zero weight, and maintains a
  --                    sorted list of row weights when not activated.
  --                    When activated generates row swap controls for
  --                    the Tableau and swaps weights accordingly.
  --
  VAR running,toggle,test,flag,type :
  VAR c.bar[4],c[4], w[2],nd[4],s[4],weight :
  SEQ
    -- setup
    type := i
    flag := 0
    running := true
    test := true
    SEQ i=[0 FOR 4]
      c.bar[i] := 0
    -- run cell
    WHILE running
      SEQ
        -- input output and trace
        port!weight;i;c[1];c[2];c[0]
        IF
          test
            PAR
              IF
                c.bar[0] <> 6
                  PAR
                    Win?c[1];w[0];w[1]
                    Wout!c.bar[1];0;0
                    Nin?c[2];nd[0];nd[1]
                    Nout!c.bar[2];nd[2];nd[3]
              IF
                c.bar[2]<>6
                  PAR
                    Sin?c[0];s[0];s[1]
                    Sout!c.bar[0];s[2];s[3]
        -- decide on closedown
        IF
          c.bar[2] = 6
            c[2] := 6
        c.bar[1] := 0
        c.bar[0] := c[2]
        c.bar[2] := c[0]
        test := NOT((c.bar[0]=6) AND (c.bar[2]=6))
        -- running := test
        -- computation
        IF
          (flag =0) AND (c[0]=3)
            SEQ
              -- zero weight during
              -- line drawing
              IF
                type = n
                  SEQ
                    weight := 0
                    c.bar[1] := 3
              c.bar[2] := 3
              flag := 1
              toggle := true
          (flag =0) AND ((c[1] = 2) OR (c[1]=4))
            SEQ
              -- modify row weight
              IF
                c[1] = 2
                  weight := weight - 1
```

```
          toggle := true
          flag := 1
        (flag = 0) AND(c[1] = 1)
          SEQ
            -- load row weight
            weight := w[0]
            toggle := true
            flag := 1
        flag = 1
          -- sort
          IF
            ((c[2]=2) OR (c[2]=3)) OR (c[2]=4)
              -- stop sort
              flag := 0
            toggle AND (weight < nd[0])
              SEQ
                c.bar[1] := 1
                weight := nd[0]
                i := nd[1]
            (NOT toggle) AND (weight > s[0])
              SEQ
                c.bar[1] := 2
                weight := s[0]
                i := s[1]
        c[0] = 6
          -- kill cell
          c.bar[1] := 6
      -- set up next i/o
      nd[2] := weight
      s[2] := weight
      nd[3] := i
      s[3] := i
      -- change sorting state
      toggle := NOT toggle :

PROC col.sort(CHAN Nin,Nout, Eout,Ein, Win,Wout, port, VAR j ) =
  --
  -- Column.sorting cell : Filters SCR controls when not sorting
  --                       Maintains list of column weights in sorted
  --                       order when inactive.
  --                       When active generates column swapping controls
  --                       to tableau cells
  --
  VAR flag,toggle,running,test,type :
  VAR c.bar[4],c[4], nd[2],w[4],e[4], weight :
  SEQ
    -- setup
    type := j
    running := true
    test := true
    SEQ i=[0 FOR 4]
      c.bar[i] := 0
    toggle := false
    flag := 0
```

```
    -- cell
    WHILE running
      SEQ
        -- input/output and trace
        port!weight;j;c[1];c[2];c[3]
        IF
          test
            PAR
              IF
                c.bar[3] <> 6
                  PAR
                    Nin?c[2];nd[0];nd[1]
                    Nout!c.bar[2];0;0
                    Win?c[1];w[0];w[1]
                    Wout!c.bar[1];w[2];w[3]
              IF
                c.bar[1] <> 6
                  PAR
                    Ein?c[3];e[0];e[1]
                    Eout!c.bar[3];e[2];e[3]
        -- test closedown and set next
        -- cycles control output
        IF
          c.bar[1] =6
            SEQ
              c[1] := 6
        c.bar[1] := c[3]
        c.bar[2] := 0
        c.bar[3] := c[1]
        test := NOT((c.bar[3]=6) AND (c.bar[1]=6))
        -- running := test
        -- computation
        IF
          (flag = 0) AND (c[3] = 3)
            SEQ
              -- zero weight for line
              -- drawing
              IF
                type = n
                  SEQ
                    c.bar[2] := 3
                    weight := 0
              c.bar[1] := 3
              toggle := false
              flag := 1
          (flag = 0) AND ((c[2] = 2) OR (c[2]=4))
            SEQ
              -- modify column weight
              IF
                c[2] = 2
                  weight := weight - 1
              toggle := false
              flag := 1
          (flag = 0) AND (c[2] = 1)
```

```
        SEQ
          -- load column weight
          weight := nd[0]
          toggle := true
          flag:= 1
      flag = 1
        -- sort
        IF
          (c[1] = 3) OR ((c[3] = 1) OR (c[1] = 4))
            -- stop sort
            flag := 0
          toggle AND (weight > e[0])
            SEQ
              c.bar[2] := 1
              weight := e[0]
              j := e[1]
          (NOT toggle) AND (weight < w[0])
            SEQ
              c.bar[2] := 2
              weight := w[0]
              j := w[1]
      c[3] = 6
        -- kill cell
        c.bar[2] := 6
      c[3] = 2
        c.bar[2] := 4
    -- set i/o of data
    w[2] := weight
    e[2] := weight
    w[3] := j
    e[3] := j
    -- change stateof sorting
    toggle := NOT toggle :


PROC t.cell( CHAN Nin, Nout, Sout, Sin,
             Win,Wout , Eout,Ein , port, VAR Telement ) =
  --
  -- Tableau cell
  --
  VAR running,c[4],c.bar[4],test :
  VAR n[4],e[4],s[4],w[4],line.state :
  SEQ
    -- setup
    test := true
    running := true
    SEQ j=[0 FOR 4]
      SEQ
        n[j] := 0
        e[j] := 0
        s[j] := 0
        w[j] := 0
        c.bar[j] := 0
    line.state := 0
    -- run cell
    WHILE running
      SEQ
        port!Telement;c[0];c[1];c[2];c[3]
        -- control i/o
        IF
          test
            PAR
              IF
                c.bar[3] <> 6
                  PAR
                    Win?c[1];w[0];w[1]
                    Wout!c.bar[1];w[2];w[3]
              IF
                c.bar[2] <> 6
                  PAR
                    Sin?c[0];s[0];s[1]
                    Sout!c.bar[0];s[2];s[3]
              IF
                c.bar[1] <> 6
                  PAR
                    Ein?c[3];e[0];e[1]
                    Eout!c.bar[3];e[2];e[3]
              IF
                c.bar[0] <> 6
                  PAR
                    Nin?c[2];n[0];n[1]
                    Nout!c.bar[2];n[2];n[3]
        -- decide for closedown
        IF
          c.bar[1] = 6
            c[1] := 6
        IF
          c.bar[2] = 6
            c[2] := 6
        -- evaluate new output controls
        -- and if ready to stop
        c.bar[2] := c[0]
        c.bar[3] := c[1]
        c.bar[0] := c[2]
        c.bar[1] := c[3]
        test := NOT(((((c.bar[0]=6)AND(c.bar[1]=6))
                    AND(c.bar[2]=6))AND(c.bar[3]=6)))
        -- running := test
        -- computation
        IF
          -- column sorting
          c[0] = 1
            SEQ
              Telement := e[0]
              line.state := e[1]
          c[0] = 2
            SEQ
              Telement := w[0]
```

```
      line.state := w[1]
  -- generate weight modify control
  -- and mark line
  (c[0] = 3) OR (c[3] = 3)
    SEQ
      IF
        (Telement = 0) AND ((c[3] = 3) AND (line.state = 0))
          c.bar[0] := 2
        (Telement = 0) AND ((c[0] = 3) AND (line.state = 0))
          c.bar[3] := 2
        c[3] = 3
          c.bar[0] := 4
        c[0] = 3
          c.bar[3] := 4
      line.state := line.state + 1
  -- row sorting
  c[3] = 1
    SEQ
      Telement := n[0]
      line.state := n[1]
  c[3] = 2
    SEQ
      Telement := s[0]
      line.state := s[1]
-- output for next cycle
w[2] := Telement
w[3] := line.state
e[2] := Telement
e[3] := line.state
n[2] := Telement
n[3] := line.state
s[2] := Telement
s[3] := line.state
-- modification of output
IF
  c[2] = 3
    SEQ
      -- update reduced cost matrix
      IF
        line.state = 0
          Telement := Telement - w[0]
        line.state = 2
          Telement := Telement + w[0]
      e[2] := w[0]
  c[0] = 4
    -- shift minimum left
    IF
      (line.state <> 0) OR ((e[0] < Telement) AND (e[0] > 0))
        w[2] := e[0]
      TRUE
        w[2] := Telement
  c[1] = 1
    -- accumulate row weight
    IF
      (line.state = 0) AND (Telement = 0)
        e[2] := w[0] + 1
      TRUE
        e[2] := w[0]
  c[2] = 1
    -- accumulate column weight
    IF
      (line.state = 0) AND (Telement = 0)
        s[2] := n[0] + 1
      TRUE
        s[2] := n[0]
  -- simple trick for
  -- output of line.state and cell data
  IF
    line.state > 0
      Telement := Telement + (line.state*1000) :

PROC controller.1(CHAN Sout,Sin, Eout,Ein, port) =
  VAR running, test, state :
  VAR c[4], c.bar[4],s[2] :
  SEQ
    -- setup
    test := true
    running := true
    state := 1
    -- cell
    WHILE running
      SEQ
        port!0;c[0];c[1];c[2];c[3]
        IF
          test
            PAR
              Sin?c[0];s[0]
              Sout!c.bar[0];s[1]
              Eout!c.bar[3]
              Ein?c[3]
          c.bar[3] := c[0]
          -- str.to.screen("c")
          -- SCR controls
          IF
            c[3] = 6
              SEQ
                test := false
                -- running := test
            (state = 1) OR (c[0] = 2)
              SEQ
                c.bar[0] := 3
                c.bar[3] := 3
                s[1] := s[0]
                state := 2
            state = 2
              SEQ
                c.bar[0] := 0
                c.bar[3] := 1
```

```
            state := 3
          (state = 3) AND (c[0]=1)
            SEQ
              c.bar[3] := 2
          TRUE
            c.bar[3] := 0 :

PROC controller.2(CHAN Sout,Sin, Win, Wout, port ) =
  DEF minus.1 = -1 :
  VAR test,running,c.bar[4],c[4] :
  SEQ
    test := true
    running := true
    WHILE running
      SEQ
        -- i/o
        port!0;c[0];c[1];c[2];c[3]
        IF
          test
            PAR
              IF
                c.bar[1] <> 6
                  PAR
                    Sin?c[0];c[2];c[3]
                    Sout!c.bar[0];minus.1;0
              IF
                c.bar[0] <> 6
                  PAR
                    Wout!c.bar[1]
                    Win?c[1]
        test := NOT( (c.bar[1]=6) AND (c.bar[0]=6))
        -- running := test
        -- filter controls signals recieved
        IF
          c.bar[1] = 6
            c.bar[0] := 6
          c[1] = 2
            c.bar[0] := 2
          c[1] = 1
            c.bar[0] := 1
          c[0] = 6
            c.bar[1] := 6
          (c[0] = 3) OR (c[0] = 4)
            SEQ
              c.bar[1] := 0
              c.bar[0] := c[0]
          TRUE
            c.bar[0] := 0 :

PROC Host.interface(CHAN Sout, Sin, Eout,Ein, Win,Wout, port) =
  --
  -- Host routine : in this design only the SCR properties are
  --                implemented. The routine can be extended for
  --                Host loading and unloading of results
  --
  VAR c[4], c.bar[4],s[2], test, running :
  SEQ
    -- setup
    test := true
    running := true
    -- cell
    WHILE running
      SEQ
        -- i/o
        port!0;c[0];c[1];c[2];c[3]
        IF
          test
            PAR
              IF
                c.bar[1] <> 6
                  PAR
                    Ein?c[3]
                    Eout!c.bar[3]
                    Sout!c.bar[0];0;0
                    Sin?c[0];s[0];s[1]
              IF
                c.bar[3] <> 6
                  PAR
                    Win?c[1]
                    Wout!c.bar[1]
        test := NOT((c.bar[1] = 6)AND(c.bar[3] = 6))
        -- running := test
        -- pass and filter control
        -- signals
        IF
          c.bar[1] = 6
            c[1] := 6
          (c[1] = 1) OR (c[1] = 3)
            c.bar[0] := c[1]
          TRUE
            c.bar[0] := 0
        c.bar[3] := c[1]
        c.bar[1] := c[3] :

PROC controller.4(CHAN Nin,Nout, Eout,Ein, port) =
  DEF minus.1 = -1 :
  VAR test,running,c[4],c.bar[4], e[2],n :
  SEQ
    -- setup
    test := true
    running := true
    -- cell
    WHILE running
      SEQ
        -- i/0
        port!0;c[0];c[1];c[2];c[3]
        IF
          test
```

```
          PAR
            IF
              c.bar[2] <> 6
                PAR
                  Ein?c[3];e[0];e[1]
                  Eout!c.bar[3];minus.1;0
            IF
              c.bar[3] <> 6
                PAR
                  Nout!c.bar[2];minus.1
                  Nin?c[2];n
      test := NOT((c.bar[2]=6)AND(c.bar[3]=6))
      -- running := test
      -- filter SCR signals
      IF
        c.bar[2] = 6
          c.bar[3] := 6
        (c[3] = 3) OR (c[3] = 4)
          SEQ
            c.bar[3] := c[3]
            c[3] := 0
        TRUE
          c.bar[3] := 0
      c.bar[2] := c[3] :

PROC controller.3(CHAN Nin,Nout, Win,Wout, port) =
  --
  -- controller.3 : line drawing controllers. Keeps count of
  --                lines drawn , and issues weight clearance
  --                signals to sorters.
  --                Issuses API closedown signal.
  --                Filters some SCR controls.
  --
  DEF maxval = 1000 :
  VAR test,running, c[4],c.bar[4],out[4],a,b,state,nlines :
  SEQ
    -- setup
    nlines := 0
    running := true
    test := true
    state := 1
    -- cell
    WHILE running
      SEQ
        -- i/0
        port!0;c[0];c[1];c[2];c[3]
        IF
          test
            PAR
              Nin?c[2];a;c[3]
              Nout!c.bar[2];out[0];out[2]
              Win?c[1];b;c[0]
              Wout!c.bar[1];out[1];out[3]
        -- set next data output
        out[0] := maxval
        out[1] := maxval
        out[2] := c[3]
        out[3] := c[0]
        -- decide if cell dead
        IF
          c.bar[1] = 6
            SEQ
              test := false
              -- running := false
        c.bar[1] := 0
        c.bar[2] := 0
        -- line drawing control
        IF
          state = 1
            -- wait for row and column
            -- weights to be loaded and
            -- sorted
            SEQ
              c.bar[2] := 0
              IF
                c[2] = 2
                  state := 2
                TRUE
                  c.bar[1] := c[2]
          state = 2
            SEQ
              -- draw lines
              IF
                nlines = n
                  SEQ
                    -- stop enough lines
                    -- c[0] := 7
                    c.bar[1] := 6
                    c.bar[2] := 6
                (a = 0) AND (b = 0)
                  SEQ
                    -- all zeroes covered and
                    -- < n lines. Modify cost
                    -- Tableau.
                    c.bar[1] := 2
                    c.bar[2] := 0
                    state := 1
                (b < a)
                  SEQ
                    -- draw column line
                    c.bar[2] := 3
                    nlines := nlines + 1
                    out[0] := 0
                    state := 3
                (b >= a)
                  SEQ
                    -- draw row line
                    c.bar[1] := 3
```

```
                nlines := nlines + 1
                out[1] := 0
                state := 5
          -- delay next control for
          -- synchronisation purposes
          state = 3
            state := 4
          state = 4
            SEQ
              state := 7
              c.bar[1] := 4
          state = 5
            state := 6
          state = 6
            SEQ
              c.bar[2] := 4
              state := 7
          -- wait for new row and column weights
          -- to be resorted
          (state = 7) AND ((c[2]=3) OR (c[1] = 3))
            state := 8
          (state = 8) AND ((c[2]=4) OR (c[1] = 4))
            state := 2 :

PROC min.shift(CHAN Nin,Nout, Sout,Sin, Eout,Ein, port) =
  --
  -- Min.shift : collects minimum uncovered element in each tableau
  --             row and pushes overall minimum upto controller.1
  --
  VAR test,running,c[4],c.bar[4],e[4],a,b,d :
  SEQ
    -- setup
    test := true
    running := true
    -- cell
    WHILE running
      SEQ
        -- i/o
        portid;b;c[0];c[2];c[3]
        IF
          test
            PAR
              IF
                c.bar[2] <> 6
                  PAR
                    Sout!c.bar[0];e[3]
                    Sin?c[0];a
                    Eout!c.bar[3];e[2];0
                    Ein?c[3];e[0];e[1]
              IF
                c.bar[3] <> 6
                  PAR
                    Nin?c[2];d
                    Nout!c.bar[2];b
          -- SCR filtering
          IF
            c.bar[2] = 6
              c[2] := 6
          c.bar[2] := c[0]
          c.bar[0] := c[2]
          c.bar[3] := c[2]
          test := NOT((c.bar[2] =6) AND (c.bar[3] =6))
          -- running := test
          -- shuffle minimum available
          e[3] := 0
          IF
            c[2] = 3
              SEQ
                e[2] := d
                e[3] := d
            c[0] = 1
              SEQ
                c.bar[3] := 1
                e[2] := 0
            c[0] = 2
              IF
                (a > 0) AND ((a < e[0])AND (e[0] >0))
                  SEQ
                    b := a
                    e[2] := 0
                TRUE
                  SEQ
                    b := e[0]
                    e[2] := 0 :

-- procedures related to the set up and running of the API

-- mapping functions to locate grid elements and channels

PROC loc.p(VALUE i,j, VAR r )=
  SEQ
    r := (((i-1)*(n+1))+j)-1 :

PROC loc.d(VALUE i,j, VAR r) =
  SEQ
    r := (((i-1)*n)+j)-1 :

PROC loc.t(VALUE i,j , VAR r) =
  SEQ
    r := (((i-1)*(n+2))+j)-1 :

-- input output interface routines
PROC getdata(VAR mem[], r.index[], c.index[]) =
  SEQ
    str.to.screen("*n enter reduced cost matrix")
    SEQ i =[1 FOR n]
      SEQ
        str.to.screen("*n[ ")
```

```
      SEQ j =[1 FOR n]
        VAR t1 :
        SEQ
          loc.d(i,j,t1)
          num.from.keyboard(mem[t1])
          num.to.screen(mem[t1])
          str.to.screen(" ")
      str.to.screen("]")
      r.index[i-1] := i
      c.index[i-1] := i
  str.to.screen("*n*n") :

PROC putdata(VALUE mem[], r.index[], c.index[]) =
  CHAN fptr :
  SEQ
    open.file("ftab","w",fptr)
    str.to.chan(fptr,"*n solution tableau")
    str.to.screen("*n solution tableau")
    SEQ i=[1 FOR n]
      SEQ
        str.to.chan(fptr,"*n[ ")
        str.to.screen("*n[ ")
        num.to.chan(fptr,r.index[i-1])
        num.to.screen(r.index[i-1])
        str.to.chan(fptr,":")
        str.to.screen(":")
        SEQ j =[1 FOR n]
          VAR t1 :
          SEQ
            loc.d(i,j,t1)
            num.to.chan(fptr,mem[t1])
            num.to.screen(mem[t1])
            str.to.chan(fptr,"  ")
            str.to.screen("  ")
        str.to.chan(fptr,"]")
        str.to.screen("]")
    str.to.chan(fptr,"*n[")
    str.to.screen("*n[ ")
    SEQ i =[1 FOR n]
      SEQ
        num.to.chan(fptr,c.index[i-1])
        num.to.screen(c.index[i-1])
        str.to.chan(fptr,"  ")
        str.to.screen("  ")
    str.to.chan(fptr,"]*n")
    str.to.screen("]*n")
    close.file(fptr) :


proc s.mix(CHAN port[]) =
  --
  -- screen mixer for tracing
  --
  -- provides a series of tableaus one for each
  -- cell cycle. when s.mix is not used the port channel
  -- used in other procedures is commented out
  --
  chan fptr :
  var a :
  var c[4],z,b,running,flag :
  seq
    open.file("result","w",fptr)
    str.to.screen("*n options")
    str.to.screen("*n1: Table *n2: control wavefront")
    str.to.screen("*n*n Enter option :")
    num.from.keyboard(b)
    num.to.screen(b)
    running := true
    flag := 0
    -- for all cycles
    while running
      seq
        z := 0
        str.to.screen("*nstart cycle*n")
        str.to.chan(fptr,"*nstart cycle *n")
        seq i =[0 for (n+2)*(n+2)]
          seq
            z := z + 1
            port[i]?a;c[0];c[1];c[2];c[3]
            str.to.screen(" ")
            str.to.chan(fptr," ")
            IF
              b = 1
                seq
                  num.to.chan(fptr,a)
                  num.to.screen(a)
              true
                seq j=[0 for 4]
                  seq
                    num.to.screen(c[j])
                    num.to.chan(fptr,c[j])
            str.to.screen(" ")
            str.to.chan(fptr," ")
            if
              z = (n+2)
                seq
                  z := 0
                  str.to.chan(fptr,"*n")
                  str.to.screen("*n")
            -- termination conditions
            if
              c[0] = 7
                running := false
    -- rather primitive but
    -- effective program stop
    close.file(fptr)
    abort.program :
```

```
-- main

-- Specification of the whole Tableau
-- channels are for two communication north-south(ns) and east-west(ew)
-- procedure call s.mix can be uncommented to produce trace

CHAN ns.1[size], ns.2[size], ew.1[size], ew.2[size] :
VAR mem[n*n], c.index[n], r.index[n]  :
SEQ
  getdata(mem,c.index,r.index)
  PAR
    s.mix(port)
    -- tableau
    VAR i,j :
    PAR i= [1 FOR (n+2)]
      PAR j= [1 FOR (n+2)]
        VAR t1,t2,t3,t4,t5,t6 :
        SEQ
          loc.p(i,j,t1)
          loc.p(j,i,t3)
          loc.t(i,j,t5)
          t2 := t1 - 1
          t4 := t3 - 1
          IF
            i = 1
              IF
                j = 1
                  controller.1(ns.1[t3],ns.2[t3],ew.1[t1],ew.2[t1],port[t5])
                j = (n+2)
                  controller.2(ns.1[t3],ns.2[t3],ew.1[t2],ew.2[t2],port[t5])
                TRUE
                  Host.interface(ns.1[t3],ns.2[t3],ew.1[t1],ew.2[t1],
                                 ew.1[t2],ew.2[t2],port[t5]  )
            i = (n+2)
              IF
                j = 1
                  controller.4(ns.1[t4],ns.2[t4],ew.1[t1],ew.2[t1],port[t5])
                j = (n+2)
                  controller.3(ns.1[t4],ns.2[t4],ew.1[t2],ew.2[t2],port[t5])
                TRUE
                  col.sort(ns.1[t4],ns.2[t4],ew.1[t1],ew.2[t1],
                           ew.1[t2],ew.2[t2],port[t5], c.index[j-2])
            TRUE
              VAR t6 :
              SEQ
                loc.d(i-1,j-1,t6)
                IF
                  j = 1
                    min.shift(ns.1[t4],ns.2[t4],ns.1[t3],ns.2[t3],
                              ew.1[t1],ew.2[t1],port[t5]          )
                  j = (n+2)
                    row.sort(ns.1[t4],ns.2[t4],ns.1[t3],ns.2[t3],
                             ew.1[t2],ew.2[t2],port[t5], r.index[i-2])
                  TRUE
                    t.cell(ns.1[t4],ns.2[t4],ns.1[t3],ns.2[t3],ew.1[t2],
                           ew.2[t2],ew.1[t1],ew.2[t1],port[t5],mem[t6])
  putdata(mem,r.index,c.index)
```

```
-- program 18

-- A Systolic Rank Annihilation Network
--
-- NOTES : The program implements the Sherman-Morrison Rank Annihilation
--         formula for the computation of the inverse of a matrix B, which
--         differs from a matrix A of known inverse by only a single row
--         column or single element. The array can be used as a building
--         block for cascaded matrix schemes for arbitray matrix inversion.
--          The array itself consists of three Linearly connected arrays,
--         seperated by a 2-D grids of delay elements implementing matrix
--         transposition operations. Pipelining occurs within each Linear
--         Array and also globally through the whole system. As the directions
--         of pipelining are at right angles this is called Orthogonal
--         Pipelining.
--          The array expects the known inverse and the modification data
--         in in two vectors input from a file called stream.
--

--  starting parameters, w=2n-1 the bandwidth of a dense n*n matrix.

DEF w = 5,pipe.length=2 :

-- Interfacing routines for input/output

EXTERNAL proc fp.num.to.chan(chan c, value float f) :
EXTERNAL proc num.to.chan(chan c, value n) :
EXTERNAL proc str.to.chan(chan c, value s[]) :
EXTERNAL proc fp.num.from.chan(chan c,var float f):
EXTERNAL proc num.from.chan(chan c,var n) :
EXTERNAL proc open.file(value path.name[],access[],chan io.chan):
EXTERNAL proc close.file(chan io.chan) :
EXTERNAL proc fp.num.to.screen.f(value float f,value w,d) :
EXTERNAL proc num.to.screen(value n) :
EXTERNAL proc str.to.screen(value s[]) :

-- include cells to construct Transposition Network

EXTERNAL proc trans.net(Chan nin[],sout[],port) :
EXTERNAL proc t.cell(chan n,s,win,eout,ein,wout,clk) :
EXTERNAL proc loc(value i,j, var r) :

-- include cells for Outer Product array

EXTERNAL proc m.t.m.b(CHAN Nin[],Sout[],Left[],Right[],port) :
EXTERNAL proc ips.b(CHAN Nin,Sout,Win,Eout,Ein,Wout,clk) :
EXTERNAL proc z.cell(CHAN nin,sin,ein,eout,clk) :
EXTERNAL proc router(CHAN nin,win,wout,clk) :

-- include cells for Update old Inverse

EXTERNAL proc modifier(CHAN Nin[],Sout[],Left[],Right[],port) :
EXTERNAL proc ips.c(CHAN Nin,Sout,Win,Eout,Ein,Wout,clk) :

-- include cells for matrix vector and transposed matrix vector

EXTERNAL proc m.t.m.a(CHAN Nin[],Sout[],Left[],Right[],port) :
EXTERNAL proc ips(CHAN Nin,Sout,Win,Eout,Ein,Wout,clk) :

PROC rank.a(CHAN north[],south[],left.1[],right.1[],left.3[],right.3[],port) =
  --
  -- High level definition of the Rank Annihilator
  --
  CHAN link1[w],link2[w],link3[w],link4[w]:
  CHAN left.2[2],right.2[2],clk[7] :
  PAR
    m.t.m.a(north,link1,left.1,right.1,clk[0])
    Trans.net(link1,link2,clk[1])
    m.t.m.b(link2,link3,left.2,right.2,clk[2])
    z.cell(left.1[1],left.3[0],left.2[1],left.2[0],clk[3])
    router(right.1[0],right.2[0],right.2[1],clk[4])
    Trans.net(link3,link4,clk[5])
    modifier(link4,south,left.3,right.3,clk[6])
    -- local clock distributor
    VAR running :
    SEQ
      running := TRUE
      WHILE running
        SEQ
          port?running
          PAR i=[0 FOR 7]
            clk[i]!running :

-- main

-- Themain section calls the Annihilation procedure
-- and provides a global Source and Sink routine for
-- reading input and outputting the result to the Host.
--
CHAN north[w],south[w]:
CHAN left.1[2],right.1[2] ,left.3[2],right.3[2] :
CHAN link1[w],link2[w],link3[w],link4[w] :
CHAN clk[pipe.length] :
SEQ
  PAR

    rank.a(north,south,left.1,right.1,left.3,right.3,clk[1])

    --
    -- Source
    --
    VAR running,n,k,cvec[w+2] :
    VAR FLOAT vec[w+2],zero :
    CHAN ptr :
    SEQ
      zero := 0.0
      running:= TRUE
      k := 0
```

```
  -- matrix comes from file stream
  open.file("stream","r",ptr)
  num.from.chan(ptr,n)
  WHILE running
    SEQ
      clk[0]?running
      IF
        running
          SEQ
            IF
              k = n
                SEQ
                  -- file empty
                  close.file(ptr)
                  k:= k+1
                  SEQ i=[0 FOR w+2]
                    SEQ
                      vec[i] := 0.0
                      cvec[i] := 0
              k <= n
                SEQ
                  -- get next input line
                  k := k+1
                  SEQ i=[0 FOR w+2]
                    SEQ
                      fp.num.from.chan(ptr,vec[i])
                      num.from.chan(ptr,cvec[i])
            -- send inputs
            PAR
              PAR i=[1 FOR w]
                north[i-1]!vec[i];cvec[i]
              left.1[0]!vec[0]
              right.1[1]!vec[w+1];cvec[w+1]
              right.3[1]!zero;zero
--
-- Sink
--
VAR running,cvec[w+2] :
VAR FLOAT vec[w+2] :
CHAN fptr :
SEQ
  running := TRUE
  -- results in file result
  open.file("result","w",fptr)
  WHILE running
    SEQ
      -- stop when last
      -- result read
      IF
        cvec[(w+1)/2] = 2
          running := FALSE
      -- distribute clock
      PAR i=[0 FOR pipe.length]
        clk[i]!running
      IF
        running
          SEQ
            -- get output
            PAR
              PAR i=[1 FOR w]
                south[i-1]?vec[i];cvec[i]
              left.3[1]?vec[0];cvec[0]
              right.3[0]?any
            -- send to Host
            SEQ i=[0 FOR w+2]
              SEQ
                fp.num.to.screen.f(vec[i],10,3)
                fp.num.to.chan(fptr,vec[i])
                str.to.chan(fptr," ")
            str.to.screen("*n")
            str.to.chan(fptr,"*n")
  close.file(fptr)
```

```
-- Program 18a
--
-- A simple Linear Array interlevaing of two matrix vector problems
-- on the same array.One problem is Au=x andthe second (vt)A=(yt),
-- where vt andyt are transpose vectors v and y. Both problems use
-- the matrix A, which is only input once
--

-- size of the array = Bandwidth of matrix A

DEF w0 = 5 :

LIBRARY PROC ips(CHAN Nin,Sout,Win,Eout,Ein,Wout,clk)=
  --
  -- basic cell :alternates between problems on succesive cycles.
  --
  VAR FLOAT aout,a,t1,t2,out1,out2 :
  VAR c,c1,c2,c3,running :
  SEQ
    -- setup
    running := TRUE
    aout := 0.0
    out1 := 0.0
    out2 := 0.0
    -- cell
    WHILE running
      SEQ
        clk?running
        IF
          running
            SEQ
              -- i/o
              PAR
                Nin?a;c
                Win?t1
                Ein?t2;c1
                Sout!aout;c2
                Eout!out1
                Wout!out2;c3
              -- computation
              c2 := c
              c3 := c1
              IF
                c = 1
                  SEQ
                    out2 := t2 + (a*t1)
                    out1 := t1
                c = 0
                  SEQ
                    out2:= t2
                    out1 := t1 + (aout*t2)
              aout :=  a:

LIBRARY PROC m.t.m.a(CHAN Nin[],Sout[],Left[],Right[],port) =
--
-- The Linear Array Specification
--
CHAN we[w0-1],ew[w0-1],clk[w0] :
PAR
  ips(Nin[0],Sout[0],Left[0],we[0],ew[0],Left[1],clk[0])
  PAR i=[1 FOR w0-2]
    ips(Nin[i],Sout[i],we[i-1],we[i],ew[i],ew[i-1],clk[i])
  ips(Nin[w0-1],Sout[w0-1],we[w0-2],Right[0],Right[1],ew[w0-2],clk[w0-1])
  -- local clock distributer
  VAR running :
  SEQ
    running := TRUE
    WHILE running
      SEQ
        port?running
        PAR i=[0 FOR w0]
          clk[i]!running :
```

```
-- Program 18b
--
-- Linear Array for computing the Outer Product x(yt) with yt a
-- vector which isthe transpose of y
--

-- bandwidth of matrix x(yt).
DEF w2 = 5 :

LIBRARY PROC ips.b(CHAN Nin,Sout,Win,Eout,Ein,Wout,clk)=
  --
  -- Basic cell: reads amatrix from north inputs in typical systolic
  --             format x and y fromleft and right respectively.
  --             computes x(i) * y(j) i,j=1(1)n inserting behind the
  --             matrix input outputting south.
  --
  VAR FLOAT aout,a,t1,t2,out1,out2 :
  VAR c,c1,c2,c3,running :
  SEQ
    -- setup
    running := TRUE
    aout := 0.0
    out1 := 0.0
    out2 := 0.0
    -- cell
    WHILE running
      SEQ
        clk?running
        IF
          running
            SEQ
              -- i/o
              PAR
                Nin?a;c
                Win?t1
                Ein?t2;c1
                Sout!aout;c2
                Eout!out1
                Wout!out2;c3
              -- computation
              c2 := c
              c3 := c1
              IF
                c = 1
                  SEQ
                    aout := a
                c = 0
                  SEQ
                    aout := out1*out2
              out1 := t1
              out2 := t2 :

LIBRARY PROC m.t.m.b(CHAN Nin[],Sout[],Left[],Right[],port) =
  --
  -- Specification of the array
  --
  CHAN we[w2-1],ew[w2-1],clk[w2] :
  PAR
    ips.b(Nin[0],Sout[0],Left[0],we[0],ew[0],Left[1],clk[0])
    PAR i=[1 FOR w2-2]
      ips.b(Nin[i],Sout[i],we[i-1],we[i],ew[i],ew[i-1],clk[i])
    ips.b(Nin[w2-1],Sout[w2-1],we[w2-2],Right[0],Right[1],ew[w2-2],clk[w2-1])
    -- local clock distributer
    VAR running :
    SEQ
      running := TRUE
      WHILE running
        SEQ
          port?running
          PAR i=[0 FOR w2]
            clk[i]!running :

LIBRARY PROC z.cell(CHAN nin,sout,ein,eout,clk) =
  --
  -- Scalar cell : Computes x*v inner product of vectors,and
  --               delays x(i) i=1(1)n to synchronise with
  --               y(j) j=1(1)n in the outer product array
  --
  VAR FLOAT total,result,x,xout,t1 :
  VAR c,c1,running :
  SEQ
    -- setup
    running := TRUE
    total := 1.0
    xout := 0.0
    -- cell
    WHILE running
      SEQ
        clk?running
        IF
          running
            SEQ
              -- i/o
              PAR
                nin?x;c
                eout!xout
                ein?t1;c1
                sout!result
              -- computation
              IF
                c = 1
                  SEQ
                    total := total + (x*xout)
                    xout := 0.0
                c = 2
                  SEQ
                    result := total + (x*xout)
                    total := 1.0
```

```
              xout := 0.0
            TRUE
              xout := x :

LIBRARY PROC router(CHAN nin,win,wout,clk) =
  --
  -- link cell : simplifies specification details
  --             of global network,performs no computation.
  --
  VAR FLOAT y :
  VAR running :
  SEQ
    -- setup
    y := 0.0
    running := TRUE
    -- cell
    WHILE running
      SEQ
        clk?running
        IF
          running
            SEQ
              -- route
              -- data
              PAR
                nin?y
                win?any
              wout!y;0 :
```

```
-- Program 18c
--
-- Linear Array for updating a Known inverse to inverse of target
-- matrix. Requires input of a scalar z+1 from the left, and known
-- inverse interleved with x*(yt) outer product.
--

-- Maximum Bandwidth two interleaved matrices

DEF w3 = 5 :


LIBRARY PROC ips.c(CHAN Nin,Sout,Win,Eout,Ein,Wout,clk)=
  --
  -- basic cell : Performs divide and subtract
  --
  VAR FLOAT aout,a,t1,t2,out1,out2,tmp :
  VAR c[6],running,flag :
  SEQ
    -- setup
    flag := FALSE
    running := TRUE
    SEQ i=[0 FOR 6]
      c[i] := 0
    aout := 0.0
    out1 := 0.0
    out2 := 0.0
    tmp := 0.0
    -- cell
    WHILE running
      SEQ
        clk?running
        IF
          running
            SEQ
              -- i/0
              PAR
                Nin?a;c[0]
                Win?t1
                Ein?t2;c[1]
                Sout!aout;c[4]
                Eout!out1
                Wout!out2;c[5]
              -- computation
              c[4] := c[2]
              c[5] := c[3]
              c[2] := c[0]
              c[3] := c[1]
              IF
                c[0] = 2
                  aout := 0.0
                flag  AND (out1 <> 0.0)
                  -- modify
                  SEQ
```

```
                aout := tmp -(a/out1)
                flag := FALSE
            (c[0] = 1)
              -- prepare
              SEQ
                aout := a
                tmp := a
                flag := TRUE
            (c[0] = 0) OR (out1 = 0.0)
              aout := 0.0
          out1 := t1
          out2 := t2 :

LIBRARY PROC modifier(CHAN Nin[],Sout[],Left[],Right[],port) =
  --
  -- Specification of the Array
  --
  CHAN we[w3-1],ew[w3-1],clk[w3] :
  PAR
    ips.c(Nin[0],Sout[0],Left[0],we[0],ew[0],Left[1],clk[0])
    PAR i=[1 FOR w3-2]
      ips.c(Nin[i],Sout[i],we[i-1],we[i],ew[i],ew[i-1],clk[i])
    ips.c(Nin[w3-1],Sout[w3-1],we[w3-2],Right[0],Right[1],ew[w3-2],clk[w3-1])
    -- local clock distributer.
    VAR running :
    SEQ
      running := TRUE
      WHILE running
        SEQ
          port?running
          PAR i =[0 FOR w3]
            clk[i]!running :
```

```
-- Program 18d
--
-- A 2-D array of delay/swap cells for Transposing a matrix with
-- bandwidth w1. Each cell is either a simplr delay cell or capable
-- of passing and recieving data from the left or right adjacent
-- neighbours.

-- define number of processors and channels
DEF w1= 5, size = w1*(w1+1) :


proc loc.p(value i,j,var r) =
  -- locate cells
  seq
    r:= (((i-1)*w1)+j)-1 :

LIBRARY PROC loc(VALUE i,j, VAR r) =
  -- locate channels
  SEQ
    r := (((i-1)*(w1+1))+j)-1 :

LIBRARY PROC t.cell(VALUE type, CHAN n,s,win,eout,ein,wout,clk ) =
  --
  -- Basic cell : Type  dictates whether right swap or left swap
  --              cell or just a plain delay cell
  --
  DEF left = 1, right = 0:
  VAR FLOAT tsave,t :
  VAR running,c,c1 :
  SEQ
    -- setup
    running:= true
    tsave := 0.0
    -- cell
    WHILE running
      SEQ
        clk?running
        IF
          running
            SEQ
              -- i/o
              PAR
                n?t;c
                s!tsave;c1
              -- compute
              IF
                type = left
                  PAR
                    wout!t;c
                    win?tsave;c1
                type = right
                  PAR
                    eout!t;c
                    ein?tsave;c1
```

```
            TRUE
              SEQ
                c1 := c
                tsave := t :

LIBRARY PROC trans.net(CHAN nin[], sout[] ,port) =
  --
  -- Define 2-D grid of t.cells
  --
  DEF left=1, right=0, delay=2 :
  CHAN ns[size], ew[size],we[size],clk[w1*w1] :
  SEQ
    PAR
      -- Array inteface
      VAR running :
      SEQ
        running := true
        WHILE running
          SEQ
            -- local clock
            port?running
            PAR i=[0 FOR (w1*w1)]
              clk[i]!running
            -- i/o interface
            IF
              running
                PAR
                  -- input north
                  PAR j=[1 FOR w1]
                    VAR t1,x,c :
                    SEQ
                      loc(j,1,t1)
                      nin[j-1]?x;c
                      ns[t1]!x;c
                  -- output south
                  PAR j =[1 FOR w1]
                    VAR t1,x,c :
                    SEQ
                      loc(j,w1,t1)
                      ns[t1+1]?x;c
                      sout[j-1]!x;c
      -- layout cells
      PAR i=[1 FOR w1]
        PAR j=[1 FOR w1]
          VAR t1,t2,t3,t4,t5 :
          VAR even.i,even.j,type :
          SEQ
            -- locate cell channels
            even.i := ((((i/2)*2)-i)  = 0)
            even.j := ((((j/2)*2)-j)  = 0)
            loc(i,j,t1)
            t2 := t1 + 1
            loc(j,i,t3)
            t4:= t3 + 1
            loc.p(i,j,t5)
            -- decide type
            IF
              NOT even.i
                SEQ
                  IF
                    even.j
                      type := left
                    j = w1
                      type := delay
                    TRUE
                      type := right
              even.i
                SEQ
                  IF
                    j = 1
                      type := delay
                    NOT even.j
                      type := left
                    j = w1
                      type := delay
                    TRUE
                      type := right
            -- create cell
            t.cell(type,ns[t3],ns[t4],we[t1],we[t2],ew[t2],ew[t1],clk[t5]) :
```

```
-- program 19
--
-- Bi-Diagonal Triangular (Toeplitz) Inverter
--
-- Notes : This systolic Array computes the inverse of a triangular
--         matrix with constant elements on the diagonals. It expects
--         loading controls for loading a parameter corresponding to
--         diagonal element and a choice of lower or upper triangular
--         inversion.Parameters are loaded sequentially,and the
--         resulting inverse is output in standard diagonal format
--         for Linear systolic Arrays. A control signal accompanies
--         the output matrix for use with other Systolic Designs.

-- bandwidth of inverse
DEF w = 5 :

-- input/output routines
EXTERNAL proc fp.num.to.screen(value float f) :
EXTERNAL proc num.to.screen(value n) :
EXTERNAL proc str.to.screen(value s[]) :
EXTERNAL proc fp.num.from.keyboard(var float f) :
EXTERNAL proc num.from.keyboard(var n ) :

PROC b.cell(CHAN win,sout,ein,wout,clk) =
  --
  -- Boundary cell
  --
  VAR running,toggle,flag,cout1,cout2,c :
  VAR FLOAT y,x,p,a :
  SEQ
    running := TRUE
    toggle := FALSE
    y := 0.0
    WHILE running
      SEQ
        clk?running
        IF
          running
            SEQ
              -- i/o
              PAR
                win?y
                ein?a;c
                sout!x;cout2
                wout!x;cout1
              cout1 := c
              -- loading and computation
              IF
                c = 2
                  x := a
                c = 1
                  SEQ
                    -- load
                    p := a
                    cout1 := 0
                    flag := TRUE
                    toggle := TRUE
                (c=0) AND flag
                  SEQ
                    -- start computing
                    IF
                      (p = 0.0) OR (NOT toggle)
                        x := 0.0
                      toggle
                        x := (1.0-y)/p
                    toggle := NOT toggle
                    cout2 := 1
                    flag := FALSE
                c = 0
                  SEQ
                    -- simulate systolic
                    -- input
                    IF
                      (p = 0.0) OR (NOT toggle)
                        x := 0.0
                      toggle
                        x := (0.0-y)/p
                    toggle := NOT toggle
                    cout2 := 0 :

PROC i.cell(CHAN win,eout,ein,wout,clk) =
  --
  -- inner product cell
  --
  VAR running,toggle,c,cout :
  VAR FLOAT y,x,xout,yout,p  :
  SEQ
    running := TRUE
    toggle := FALSE
    yout := 0.0
    xout := 0.0
    WHILE running
      SEQ
        clk?running
        IF
          running
            SEQ
              -- i/o
              PAR
                win?y
                ein?x;c
                wout!xout;cout
                eout!yout
              IF
                c = 2
                  -- load
                  SEQ
                    p := x
```

```
                cout := 0
                toggle := FALSE
            c = 0
              SEQ
                -- compute
                IF
                  toggle
                    yout := y + (p*x)
                  TRUE
                    yout := 0.0
                toggle := NOT toggle
                xout := x
                cout := 0 :

PROC g.cell(CHAN win,eout,ein,wout,sout,clk) =
  --
  -- Generating cell : stores implicit form of inverse
  -- and generates true inverse in standard systolic diagonals
  -- format.
  --
  VAR running,cout1,cout2,cout3,c1,c2,on,toggle :
  VAR FLOAT r,tmp[3],xout,x,save :
  SEQ
    -- intialise
    running := TRUE
    r := 0.0
    tmp[0] := 0.0
    tmp[1] := 0.0
    cout1 := 0.0
    cout2 := 0.0
    cout3 := 0.0
    on := FALSE
    -- start cell
    WHILE running
      SEQ
        clk?running
        IF
          running
            SEQ
              -- i/o
              PAR
                win?c2
                ein?x;c1
                eout!cout2
                wout!xout;cout1
                sout!r;cout3
              -- book-keeping
              tmp[2] := tmp[1]
              tmp[1] := tmp[0]
              tmp[0] := x
              cout2 := c2
              cout1 := c1
              xout  := x
              IF
                c1 = 3
                  SEQ
                    -- cell off
                    on := FALSE
                    r := 0.0
                    cout3 := 0.0
                c2 = 2
                  SEQ
                    -- wake cell
                    r := tmp[2]
                    toggle := FALSE
                    on := TRUE
                    cout3 := 1.0
                toggle AND on
                  SEQ
                    -- output dummy element
                    cout3 := 1.0
                    r := save
                    toggle := FALSE
                (NOT toggle) AND on
                  SEQ
                    -- output true element
                    cout3 := 0.0
                    save := r
                    r := 0.0
                    toggle := TRUE :

PROC border.1(CHAN ein1,eout1,ein2,eout2,clk) =
  --
  -- relay cell: keeps data moving in generater
  --             array, times controls
  --
  VAR c1,c2,cout1,cout2,running :
  VAR FLOAT t1,t2,out1 :
  SEQ
    -- intialise
    running := TRUE
    out1 :=  0.0
    cout1:= 0
    cout2 := 0
    -- start cell
    WHILE running
      SEQ
        clk?running
        IF
          running
            SEQ
              -- i/o
              PAR
                ein1?t1;c1
                ein2?t2;c2
                eout1!out1
                eout2!cout2
              -- start inverse output
```

```
            IF
              c2 = 1
                SEQ
                  str.to.screen("here")
                  cout2 := 2
              TRUE
                cout2 := 0 :

PROC border.r(CHAN nin,win,wout,clk) =
  --
  -- connector between substitution array
  -- and generator array, modifies control
  -- values and identities end of inverse output
  --
  VAR running,c1,c2,cout :
  VAR FLOAT out1,t1 :
  SEQ
    -- start cell
    running := TRUE
    WHILE running
      SEQ
        clk?running
        IF
          running
            SEQ
              -- i/o
              PAR
                nin?t1;c1
                win?c2
                wout!out1;cout
              -- move data
              IF
                c2 = 2
                  SEQ
                    out1 := 0.0
                    cout := 3
                TRUE
                  SEQ
                    out1 :=t1
                    cout := c1 :

PROC tri.inv(CHAN south[],input,port) =
  --
  -- The Triangular Inverter
  --
  -- Notes : consists of a substitution array
  --         and a generating array forthe triangular
  --         matrix inverse. substitution array is given
  --         toeplitz parameters andcontrols on input,matrix
  --         output on south.
  --
  CHAN we[w+1],ew[w+1],clk[w+4],l[5] :
  PAR
    -- substitution array
    b.cell(l[0],l[1],input,l[2],clk[0])
    i.cell(l[3],l[0],l[2],l[4],clk[1])
    -- generating array
    border.l(l[4],l[3],ew[0],we[0],clk[2])
    border.r(l[1],we[w],ew[w],clk[3])
    PAR i=[0 FOR w]
      g.cell(we[i],we[i+1],ew[i+1],ew[i],south[i],clk[i+4])
    -- interface spooler
    VAR running :
    SEQ
      running := TRUE
      WHILE running
        SEQ
          port?running
          PAR i=[0 FOR w+4]
            clk[i]!running :


PROC inverter(CHAN input,south[],port) =
  --
  -- Full Inverter : consists oftwo triangular
  --                 inverters and communication
  -- interface. One inverter produces upper triangular
  -- inverse the other Lower triangular. Parameters are
  -- loaded by input channels and split by interface to
  -- correct inverter. Output is via south channel and
  -- retains position in full matrix output
  --
  CHAN south.l[w],south.r[w],in[2],clk[2] :
  PAR
    -- inverters
    tri.inv(south.l,in[0],clk[0])
    tri.inv(south.r,in[1],clk[1])
    -- interface
    VAR running,t1,c,c1,c2,x :
    VAR FLOAT a :
    SEQ
      running := TRUE
      WHILE running
        SEQ
          port?running
          PAR i=[0 FOR 2]
            clk[i]!running
          IF
            running
              SEQ
                -- parameters
                input?a;c1;c2
                PAR
                  -- upper or lower
                  IF
                    c2 = 1
                      PAR
                        in[0]!a;c1
```

```
        in[1]!0.0;0
    TRUE
      PAR
        in[1]!a;c1
        in[0]!0.0;0
  -- distribute input
  PAR i=[1 FOR w-1]
    VAR FLOAT x :
    VAR c :
    SEQ
      south.l[i]?x;c
      south[w-(1+i)]!x;c
  -- collect output
  PAR i=[1 FOR w-1]
    VAR FLOAT x :
    VAR c :
    SEQ
      south.r[i]?x;c
      south[i+(w-1)]!x;c
  -- merge leading diagonal
  IF
    c2 = 1
      VAR FLOAT x :
      VAR c :
      SEQ
        PAR
          south.l[0]?x;c
          south.r[0]?t1;c1
        south[w-1]!x;c
    TRUE
      VAR FLOAT x :
      VAR c :
      SEQ
        PAR
          south.l[0]?t1;c1
          south.r[0]?x;c
        south[w-1]!x;c :

-- main
CHAN south[2*w],clk,in :
VAR FLOAT p1,vec[(2*w)-1] :
VAR p2,p3,cvec[(2*w)-1] :
PAR
  inverter(in,south,clk)
  -- Host Interface
  VAR running :
  SEQ
    running := TRUE
    WHILE running
      SEQ
        str.to.screen("*n parameters >")
        fp.num.from.keyboard(p1)
        fp.num.to.screen(p1)
        str.to.screen(" ")
        num.from.keyboard(p2)
        num.to.screen(p2)
        str.to.screen(" ")
        num.from.keyboard(p3)
        num.to.screen(p3)
        str.to.screen("*n")
        if
          p3 = 6
            running := FALSE
        clk!running
        -- collect result and
        -- output
        if
          running
            SEQ
              in!p1;p2;p3
              PAR i =[0 FOR (2*w)-1]
                south[i]?vec[i];cvec[i]
              SEQ i=[0 FOR (2*w)-1]
                SEQ
                  fp.num.to.screen(vec[i])
                  str.to.screen(" ")
              str.to.screen("*n")
              SEQ i=[0 FOR (2*w)-1]
                SEQ
                  num.to.screen(cvec[i])
                  str.to.screen(" ")
```

```
-- program 20
--
-- Bi-linear array for 2-D Group Explicit methods :
--
-- Notes : The systolic Array implements a marching technique
--         on a 2-D region R=(x,y).Marching can take place in
--         the y-direction (row wise) or in the x-direction
--         (column-wise).
--         The Array consist of a linear array of Macro ips cells
--         which contain two tiers of basic point cells. Point cells
--         are connected into a systolic ring allowing computation
--         of four points (a group) in parallel. Hence minimising
--         cycle time.
--         The program assummes communication with an external host
--         and expects input from two files called "odds" and "evens".
--         "odds" provides co-efficients for bottom tier point cells
--         and known points in R from odd rows(columns). Likewise
--         "evens" supplies co-efficients for top tier cells and even
--         row(column) data from R.
--

-- size= number of grid points along marching axis, size2 number of
-- macro cells required. port used for tracing and debugging only

DEF size=8,size2=4 :
CHAN port[2*size] :

-- necessary communication procedures for i/o

EXTERNAL proc fp.num.to.screen.f(value float f,value w,d) :
EXTERNAL proc fp.num.from.keyboard(Var float f) :
EXTERNAL proc num.to.screen(value n) :
EXTERNAL proc num.from.keyboard(var n) :
EXTERNAL proc str.to.screen(value s[]) :
EXTERNAL proc open.file(VALUE path.name[], access[], CHAN io.chan) :
EXTERNAL proc close.file(CHAN io.chan) :
EXTERNAL proc num.from.chan(CHAN ptr, VAR n) :
EXTERNAL proc fp.num.from.chan(CHAN ptr, VAR FLOAT f) :

PROC point.ips(VALUE pos, CHAN in1,out1,in2,out2,accin,accout,clk,port)=
  --
  -- Point ips cell :
  --   permits loading of grid values, coefficients of molecule
  --   portions covering the point, and operation of internal
  --   macro cell ring. Switching is achieved by control c.
  --
  VAR FLOAT mem1[6],mem2[6],uvalue,t1,t2,r1,r2,a :
  VAR c,running :
  SEQ
    -- setup
    r1 := 0.0
    r2 := 0.0
    uvalue := 0.0
    running := TRUE
    -- start cell
    WHILE running
      SEQ
        -- clock and trace
        clk?running
        port!running;uvalue
        -- compute
        IF
          running
            SEQ
              -- local i/o
              PAR
                in1?t1
                in2?t2;c
                out1!uvalue
                out2!uvalue
              -- decode command
              IF
                c=0
                  -- compute molecules
                  -- using ring
                  SEQ i=[0 FOR 5]
                    SEQ
                      PAR
                        accin?a
                        accout!r1
                      IF
                        i=4
                          -- result
                          SEQ
                            uvalue := a
                        TRUE
                          SEQ
                            -- accumulate
                            PAR
                              r2 := (t1+t2)*mem1[i]
                              r1 := (uvalue*mem2[i])+a
                            r1 := r1 + r2
                c=1
                  SEQ
                    -- load local coefficients
                    SEQ i=[0 FOR 5]
                      mem1[5-i] := mem1[(5-i)-1]
                    mem1[0] := t2
                c=2
                  SEQ
                    -- load neighbour co-effs
                    SEQ i=[0 FOR 5]
                      mem2[5-i] := mem2[(5-i)-1]
                    mem2[0] := t2
                c=3
                  SEQ
                    -- load grid-point
                    r1 := 0.0
```

```
                    uvalue := t2 :

PROC macro.ips(CHAN nin.1,nin.2,sin.1,sin.2,
                    nout.1,nout.2,sout.1,sout.2,
                    ein.1,ein.2,win.1,win.2,
                    wout.1,wout.2,eout.1,eout.2, clk,value i ) =
  --
  -- Macro cell : defines group of point cells connected
  --              by  a systolic Ring
  --
  CHAN ring[4], clks[4] :
  PAR
    point.ips(0,ein.2,eout.2,sin.1,sout.1,ring[3],ring[0],
              clks[0],port[(2*i)+size])
    point.ips(1,ein.1,eout.1,nin.1,nout.1,ring[0],ring[1],
              clks[1],port[2*i])
    point.ips(2,win.1,wout.1,nin.2,nout.2,ring[1],ring[2],
              clks[2],port[(2*i)+1])
    point.ips(3,win.2,wout.2,sin.2,sout.2,ring[2],ring[3],
             clks[3],port[((2*i)+1)+size])

    -- clock distribution process
    VAR running :
    SEQ
      running := TRUE
      WHILE running
        SEQ
          clk?running
          PAR i=[0 FOR 4]
            clks[i]!running :


PROC boundary(CHAN in1,in2,out1,out2,clk) =
  --
  -- neat disposal of comunication on
  -- ends of array .
  --
  VAR running :
  SEQ
    running := TRUE
    WHILE running
      SEQ
        clk?running
        IF
          running
            PAR
              in1?any
              in2?any
              out1!0.0
              out2!0.0 :


PROC bi.linear(CHAN n1[],s1[],n2[],s2[],clk) =
  --
```

```
  -- An Array of Macro.cells
  --
  CHAN east.1[size2+1],east.2[size2+1],west.1[size2+1],west.2[size2+1] :
  CHAN clks[size2+2] :
  SEQ
    PAR
      -- The array
      PAR i=[0 FOR size2]
        VAR i2 :
        SEQ
          i2 := 2*i
          macro.ips(n1[i2],n1[i2+1],s1[i2],s1[i2+1],
                    n2[i2],n2[i2+1],s2[i2],s2[i2+1],
                    west.1[i],west.2[i],east.1[i+1],east.2[i+1],
                    west.1[i+1],west.2[i+1],east.1[i],east.2[i], clks[i],i)
      boundary(east.1[0],east.2[0],west.1[0],west.2[0],clks[size2])
      boundary(west.1[size2],west.2[size2],east.1[size2],east.2[size2],
               clks[size2+1])

      -- clock distribution process
      VAR running :
      SEQ
        running:= TRUE
        WHILE running
          SEQ
            clk?running
            PAR i=[0 FOR size2+2]
              clks[i]!running :


PROC s.mix(Chan port[]) =
  --
  -- screen mixer : for debugging
  --                and trace
  --
  VAR FLOAT p :
  VAR running :
  SEQ
    running := true
    WHILE running
      SEQ
        --prod point cells
        SEQ i=[0 for 2*size]
          SEQ
            port[i]?running;p
            fp.num.to.screen.f(p,8,2)
            if
              i = (size-1)
                str.to.screen("*n")
        -- display contents
        str.to.screen("*n*n") :


-- main
```

```
-- setup and running of the array including
-- Host interface. Procedure s.mix is included
-- for tracing.To switch off trace remove comments
-- on lines below and comment out s.mix, and the port
-- communication in point.ips cells.
--   The routine belowallows computation of only a single
-- level over R.It assummed that the Host presents a
-- modified set of files for subsequent passes using data
-- output by the array, and coefficients for altering cell
-- molecule types.
--   Alternatively a much extended main program can be
-- added to modify inputs as output occurs. This is not
-- presented as a general 2-D problem can involve function
-- evaluations not supported directly by occam.
--
CHAN fptr1,fptr2,north.1[size],north.2[size] :
CHAN south.1[size],south.2[size],clk :
PAR
  s.mix(port)
  bi.linear(north.1,south.1,north.2,south.2,clk)
  VAR c1,c2,running :
  VAR FLOAT buf.1[size],buf.2[size],sbuf.1[size],sbuf.2[size] :
  VAR FLOAT rbuf.1[size], rbuf.2[size] :
  SEQ
    -- open data files
    open.file("odds","r",fptr1)
    open.file("evens","r",fptr2)
    running :=TRUE
    -- until stopped
    WHILE running
      SEQ
        -- read controls
        num.from.chan(fptr1,c1)
        num.from.chan(fptr2,c2)
        IF
          (c1=6) AND (c2=6)
            SEQ
              -- stop
              running := FALSE
              clk!running
          TRUE
            SEQ
              -- synchronise cells
              clk!running
              -- read next input vectors
              SEQ i=[0 FOR size]
                SEQ
                  fp.num.from.chan(fptr1,buf.1[i])
                  fp.num.from.chan(fptr2,buf.2[i])
              -- process
              IF
                (c1=0) AND (c2=0)
                  SEQ
                    -- load and compute
                    PAR i=[0 FOR size]
                      PAR
                        north.1[i]!buf.2[i];3
                        north.2[i]?rbuf.1[i]
                        south.1[i]!sbuf.1[i];3
                        south.2[i]?rbuf.2[i]
                    SEQ i=[0 FOR size]
                      fp.num.to.screen.f(rbuf.1[i],8,2)
                    str.to.screen("*n")
                    SEQ i =[0 FOR size]
                      fp.num.to.screen.f(rbuf.2[i],8,2)
                    str.to.screen("*n*n")
                    clk!running
                    PAR i=[0 FOR size]
                      PAR
                        north.1[i]!buf.1[i];c1
                        north.2[i]?rbuf.1[i]
                        south.1[i]!sbuf.2[i];c2
                        south.2[i]?rbuf.2[i]
                    -- shuffle for easy input to array
                    SEQ i=[0 FOR size]
                      SEQ
                        sbuf.1[i] := buf.1[i]
                        sbuf.2[i] := buf.2[i]
                TRUE
                  -- load coefficients
                  PAR i=[0 FOR size]
                    PAR
                      north.1[i]!buf.2[i];c1
                      north.2[i]?rbuf.2[i]
                      south.1[i]!buf.1[i];c2
                      south.2[i]?rbuf.1[i]
                -- output if no trace
                SEQ i=[0 FOR size]
                  fp.num.to.screen.f(rbuf.1[i],8,2)
                str.to.screen("*n")
                SEQ i=[0 FOR size]
                  fp.num.to.screen.f(rbuf.2[i],8,2)
                str.to.screen("*n*n")
```

LIST OF PUBLISHED PAPERS

1. *"Soft-systolic Pipelined Matrix Algorithms"*,

   G.M. Megson & D.J. Evans, Parallel Computing 85, ed. Feilmeier, Joubert & Schendel, Elsevier Science Publishers, 1986.

   Int.Conf. Parallel Computing 85.

2. *"Romberg Integration using Systolic Arrays"*,

   D.J. Evans & G.M. Megson, Parallel Computing 3(1986), 289-304.

3. *"Construction of Extrapolation Tables by Systolic Arrays for Solving Ordinary Differential Equations"*,

   D.J. Evans & G.M. Megson, Parallel Computing 4(1987), 33-48.

4. *"The Unification of Systolic Differencing Algorithms"*,

   G.M. Megson & D.J. Evans, to appear in the Computer Journal, Vol. 30, No.3, 1987.

5. *"Matrix Inversion by Systolic Rank Annihilation"*,

   D.J. Evans & G.M. Megson, Intern.J.Computer Math. 1987, Vol. 21, pg. 319-358.

6. *"The Solution of Parabolic Partial Differential Equations by Systolic Marching Techniques"*,

   D.J. Evans & G.M. Megson, to appear, 6th IMACS International Symposium on Computer Methods in P.D.E.'s, Lehigh University 1987.

7. *"Matrix Power Generation Using An Optical Reduced Bandwidth Systolic Array"*,

   G.M. Megson & D.J. Evans, to appear, 7th International Congress of Cybernetics & Systems, Sept. 1987.

8. *"LISA: A Parallel Processing Architecture"*,

G.M. Megson & D.J. Evans, "CONPAR 86", Lecture Notes in Computer Science 237, Springer Verlag, eds. G. Goos & J. Hartimanis

9. *"A Systolic Matrix Norm Generator"*,

D.J. Evans & G.M. Megson, to appear in Major Advances in Parallel Processing, Gower Publications 1987, p.291-299.

10. *"The Alternating Group Explicit (AGE) Systolic Array for the Solution of Large Linear Systems"*,

D.J. Evans & G.M. Megson, to appear in 1st International Conf. on Industrial & Applied Mathematics, ICIAM 1987.