On the Design of an Integrated Systolic Array for Solving Simultaneous Linear Equations

F.-C. LIN* AND K. CHEN
Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan, R.O.C.

This paper introduces, in a stepwise refinement manner, a new systolic array for solving simultaneous linear equations $AX = B$, where A is an $n \times n$ matrix and B is an $n \times m$ matrix. Instead of using conventional matrix triangularisation, our design is based on a generalised Gauss–Jordan elimination procedure. First A is transformed into a permutation matrix P and B is transformed into a matrix Q. Then X is computed by multiplying P^T and Q. These two stages of computation are tightly pipelined to form an integrated systolic array which is capable of solving the simultaneous linear equations in just $6n+m-2$ time steps. This systolic array achieves maximum data pipelining rate and its computation time, in a sense, is optimal.

Received July 1987, revised September 1988

1. INTRODUCTION

The evolution of very large scale integration (VLSI) technology has promoted a great deal of effort in designing special-purpose systems. Kung[1,2,3] advocated the *systolic* concept for designing specialised computing arrays to handle compute-bound problems. These arrays generally consist of a regular array of simple and identical processing elements (PEs) in which data are transmitted locally and operated on rhythmically. The simplicity and regularity of the systolic arrays render them suitable for VLSI implementation. High performance is achieved by the concurrent use of a large amount of PEs in the arrays.

The purpose of this paper is to introduce a very efficient systolic array for *solving simultaneous linear equations*. Kant and Kimura[4] showed that the solution of a system of n linear equations $Ax = b$ can be obtained in $O(n)$ steps using an array of mesh-connected PEs. But their algorithm requires that the coefficient matrix A be strongly non-singular, namely every square block of A must be non-singular. Kung and Leiserson[1] presented an array of hexagonally connected PEs to do the LU-decomposition of A. The two resulting triangular systems $Ly = b$ and $Ux = y$ then are solved on an array of linearly connected PEs. These three stages of computation use $10n$ time steps to produce the solution. Since many non-singular matrices are not LU-decomposable, the use of their design is very limited. Bojanczyk, Brent and Kung[5] later presented a systolic array to triangularise non-singular matrix A via Givens transformations, and also used a linearly connected array to solve the resulting triangular system. Although the total number of time steps is only $6n$, each of the first $3n$ time steps must include a square root operation which is not only time consuming but also hardware demanding. Moreover, due to the different data pipelining rates and I/O patterns among the stages of computation, considerable extra time is needed for the inter-module communications. Several other recent designs for solving system of linear equations also incur these problems.[6]

Lin and Wu,[7] based on a generalisation of the well-known *Gauss–Jordan elimination procedure*, designed an integrated systolic array to solve the simultaneous linear equations $AX = B$ in $8n+m-4$ time steps, where m is the column number of B. There is no condition imposed on A and the existence and uniqueness of solution can be detected during the computation. We noticed that the data in this systolic array are not tightly pipelined, or more precisely, every two consecutive data in the data streams are separated by an empty step. This observation motivated us to search for a better or even optimal design.

In this paper, we shall present a new systolic array to reduce the number of computation steps to $6n+m-2$ and show that this is actually optimal when the generalised Gauss-Jordan elimination procedure is used. The way of designing this new systolic array can be briefly outlined as follows. First, we write down a sequential algorithm for the generalised Gauss–Jordan elimination procedure and model it as computation activities on an abstract index set.[8,9] From the indexed computations, we identify the data dependencies and represent them as a three-dimensional data dependence graph. In order to project this graph into a regularly operated two-dimensional array in more directions for more design alternatives, the graph is modified according to a *normalisation principle* introduced in Ref. 7. Then a proper direction is chosen for projection. The resulting design turns out to be a less constrained semisystolic array which allows zero-delay interconnections.[10] Finally a *bisection retiming rule*[7] is applied to make all interconnection delays positive, i.e. temporally local, thereby transforming the semisystolic array into the desired systolic array.

An important property of the array is its 'unidirectional' data flow, i.e. no looping of data flow. Because of this, both fault-tolerance and two-level pipelining techniques can be naturally applied to the array to increase its dependability and system throughput.[12] When the problem size exceeds the array size, partitioning methods with either off-array queues[9] or on-array local memories[13] can be adopted. Otherwise, the array still can be extended (rescaled) to fit the problem size due to the uniformity of the PEs.

* To whom correspondence should be addressed.

2. GENERALISED GAUSS–JORDAN ELIMINATION AND DATA DEPENDENCE GRAPH

Consider the solving of the simultaneous linear equations $AX = B$, where A is an $n \times n$ matrix and B is an $n \times m$ matrix. According to a generalisation of the Gauss–Jordan elimination procedure, which will be formally described shortly, A can be transformed by a sequence of elementary row operations into a matrix P in which the first non-zero element of each row is 1 and the column of that element contains no other non-zero element. In the meantime, B is also transformed into a matrix Q by the same sequence of row operations. If A is non-singular, then P turns out to be a *permutation matrix*, i.e. in every row and every column of P there is exactly one element with value 1 and all the others are 0. Since $PX = Q$ and $P^{-1} = P^T$ (the transpose of P), $X = P^TQ$. Following is the generalised Gauss–Jordan elimination procedure for computing P and Q:

for $k := 1$ **to** n **do**
begin (* *search for the first non-zero* a_{kp} *in row k of A and divide row k of* $[A \mid B]$ *by* a_{kp} *)
  find$:=$ *false;*
  for $j := 1$ **to** $n+m$ **do**
  begin
    if *find* **then** $[A \mid B]_{kj} := [A \mid B]_{kj}/a_{kp}$
    else if $j \leqslant n$ **and** $a_{kj} \neq 0$ **then**
      begin $p := j$; $a_{kp} := 1$; *find*$:=$ *true* **end**
  end;
  for $i := 1$ **to** n **except** k **do**
    (* *subtract row k times* a_{ip} *from row i of* $[A \mid B]$ *)
    for $j := 1$ **to** $n+m$ **do**
    begin $(A \mid B)_{ij} := [A \mid B]_{ij} - a_{ip} * [A \mid B]_{kj}$ **end**
end. (* *k-loop* *)

We can model the computation of this procedure as activities on the index set $\{(i,j,k) \mid 1 \leqslant i \leqslant n,\ 1 \leqslant j \leqslant n+m,\ 1 \leqslant k \leqslant n\}$. In order to conceive the 'data movements', we need some variables to carry the data objects all over the index points:

- PV: first non-zero element carried inside row k.
- p: position of first non-zero element carried to other rows.
- AB: elements of $[A \mid B]$.
- RK: row k carried to other rows.
- AL: multiplication factor a_{ip} carried inside row i ($\neq k$).

With the help of these data object carriers, we now rewrite the procedure as the following recursion equations:

For $1 \leqslant i \leqslant n$, $1 \leqslant j \leqslant n+m$, $1 \leqslant k \leqslant n$,
(* *Propagation of first non-zero element inside row k* *)
$PV(i,j,k) = i = k \rightarrow j = 0 \rightarrow 0$
$\qquad j \leqslant n$ *and* $PV(i,j-1,k) = 0 \rightarrow AB(i,j,k-1)$
$\qquad$ *otherwise* $\rightarrow PV(i,j-1,k)$
$\qquad i \neq k \rightarrow 0$

(* *Propagation of first non-zero element to other rows* *)
$p(i,j,k) = i = k \rightarrow PV(i,j-1,k) = 0 \rightarrow PV(i,j,k)$
$\qquad PV(i,j-1,k) \neq 0 \rightarrow 0$
$\qquad i < k \rightarrow p(i+1,j,k)$
$\qquad i > k \rightarrow p(i-1,j,k)$

(* *Updating of elements of* $[A \mid B]$ *)
$AB(i,j,k) = k = 0 \rightarrow j \leqslant n \rightarrow a_{ij}$
$\qquad j > n \rightarrow b_{i,j-n}$
$\qquad i = k \rightarrow PV(i,j-1,k) = 0$ *and*
$\qquad\qquad AB(i,j,k-1) \neq 0 \rightarrow 1$
$\qquad\qquad PV(i,j-1,k) \neq 0 \rightarrow$
$\qquad\qquad AB(i,j,k-1)/PV(i,j-1,k)$
$\qquad\qquad$ *otherwise* $\rightarrow 0$
$\qquad i \neq k \rightarrow p(i,j,k) \neq 0 \rightarrow$
$\qquad\qquad AB(i,j,k-1) - AL(i,j,k) * RK(i,j,k)$
$\qquad\qquad$ *otherwise* $\rightarrow AB(i,j,k-1)$

(* *Propagation of row k to other rows* *)
$RK(i,j,k) = i = k \rightarrow AB(i,j,k)$
$\qquad i < k \rightarrow RK(i+1,j,k)$
$\qquad i > k \rightarrow RK(i-1,j,k)$

(* *Propagation of multiplication factor inside row* $i \neq k$ *)
$AL(i,j,k) = i = k$ *or* $j = 0$ *or* $p(i,j,k) = 0 \rightarrow 0$
$\qquad AL(i,j-1,k) = 0$ *and* $p(i,j,k) \neq 0 \rightarrow$
$\qquad AB(i,j,k-1)$
$\qquad$ *otherwise* $\rightarrow AL(i,j-1,k)$

If the computation at index point (i,j,k) depends on the outcome of computation at index point (i',j',k'), we use the vector $(i-i', j-j', k-k')^T$ to represent the *data dependency*. From the above recursion equations, all the data dependencies can easily be identified and represented as a three-dimensional *data dependence graph* as shown in Fig. 1 with different i–j planes drawn separately. Not drawn are the dependence vectors in the k direction that run from $(i,j,k-1)$ *to* (i,j,k). This graph can be projected into a regularly operated $(n+m) \times (2n-1)$ array of PEs in the direction of $(1,0,1)^T$, but the resulting design is not what we want because of its poor utilisation of the PEs, only half of them are used at any time step.

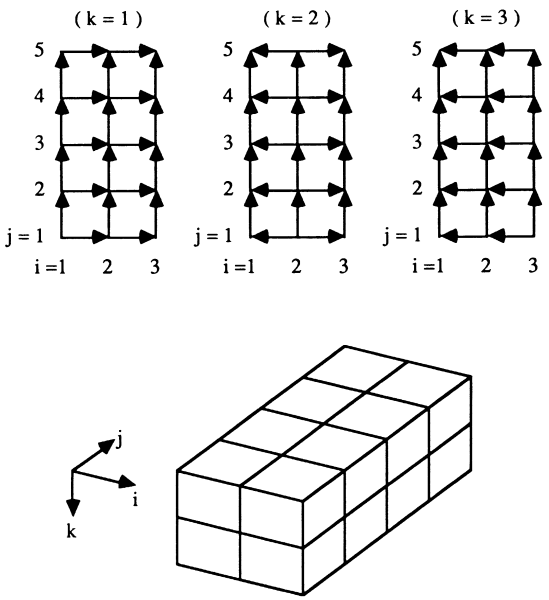


Figure 1. Data dependence graph of the generalised Gauss–Jordan elimination procedure for $n = 3$, $m = 2$.

3. MODIFIED DATA DEPENDENCE GRAPH

In the plane $k = 1$ of the data dependence graph, variables $p(1,j,1)$ and $RK(1,j,1)$ are propagated all in one direction to update variables $AB(1,j,1)$. But when

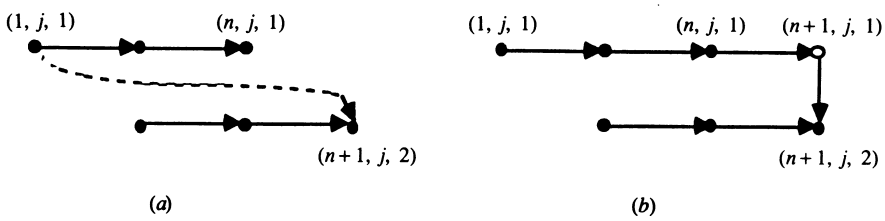
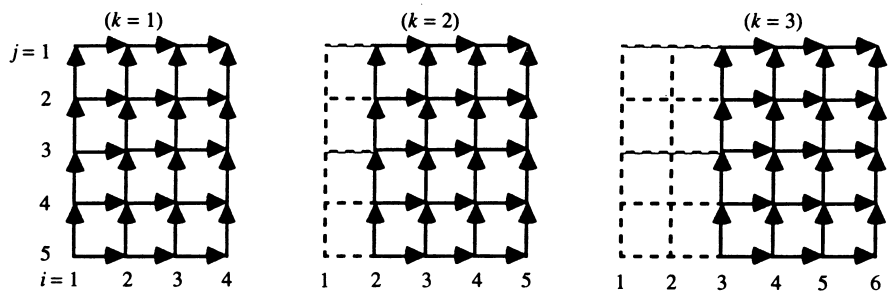


Figure 2. Revising dependencies.



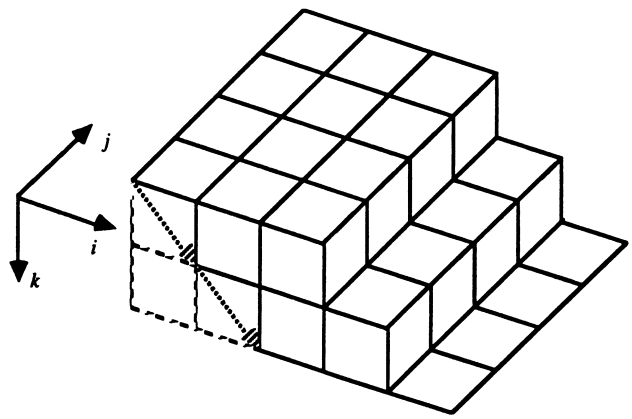


Figure 3. Modified dependence graph.

we go to the next plane $k = 2$, the variables $p(2,j,2)$ and $RK(2,j,2)$ are now propagated in two directions. Suppose we 'globally' move the values $AB(1,j,1)$ to the index points $(n+1,j,2)$, update $AB(1,j,2)$ there, as illustrated in Fig. 2(*a*), unidirectional propagations of $p(2,j,2)$ and $RK(2,j,2)$ will be sufficient and the index points $(2,j,2)$, $1 \leqslant j \leqslant n+m$, can be erased. In fact, such a 'global' move can be 'localised' as follows. In plane $k = 1$, let $AB(1,j,1)$ move along with $p(1,j,1)$ and $RK(1,j,1)$, move further to index point $(n+1,j,1)$ and then move down to $(n+1,j,2)$, as illustrated in Fig. 2(*b*). This brings new local dependencies from $(n,j,1)$ to $(n+1,j,1)$ and from $(n+1,j,1)$ to $(n+1,j,2)$, however the global dependencies are eliminated. The same modification process should be repeated for the planes $k = 2,3,\ldots,n$.

The above modification of the data dependence graph leads to the following recursion equations:

For $1 \leqslant k \leqslant n$, $k \leqslant i \leqslant n+k$, $1 \leqslant j \leqslant n+m$,

(* *Propagation of first non-zero element inside row k* *)

$$\begin{aligned} PV(i,j,k) = i = k &\rightarrow j = 0 \rightarrow 0 \\ &\quad j \leqslant n \text{ and } PV(i,j-1,k) = 0 \rightarrow \\ &\quad AB(i,j,k-1) \\ &\quad \text{otherwise} \rightarrow PV(i,J-1,k) \\ i \neq k &\rightarrow 0 \end{aligned}$$

(* *Propagation of first non-zero element to other rows* *)

$$\begin{aligned} p(i,j,k) = i = k &\rightarrow PV(i,j-1,k) = 0 \rightarrow PV(i,j,k) \\ &\quad PV(i,j-1,k) \neq 0 \rightarrow 0 \\ i > k &\rightarrow p(i-1,j,k) \end{aligned}$$

(* *Updating of elements in* $[A \mid B]$ *)

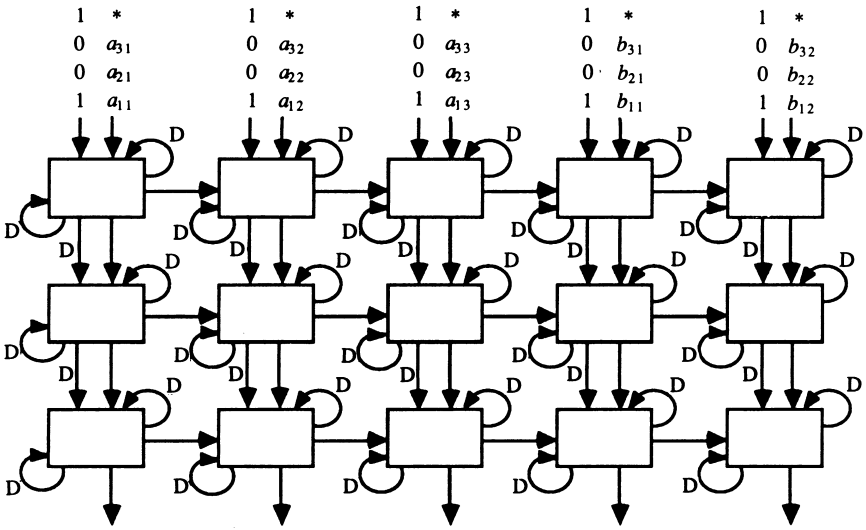
$$\begin{aligned} AB(i,j,k) = k = 0 &\rightarrow j \leqslant n \rightarrow a_{ij} \\ &\quad j > n \rightarrow b_{i,j-n} \\ i = k &\rightarrow PV(i,j-1,k) = 0 \text{ and} \\ &\quad AB(i,j,k-1) = 0 \rightarrow 1 \\ &\quad PV(i,j-1,k) = 0 \rightarrow \\ &\quad AB(i,j,k-1)/PV(i,j-1,k) \\ &\quad \text{otherwise} \rightarrow 0 \\ i = k+n &\rightarrow RK(i,j,k) \\ i \neq k &\rightarrow p(i,j,k) \neq 0 \rightarrow AB(i,j,k-1) \\ &\quad -AL(i,j,k) * RK(i,j,k) \\ &\quad \text{otherwise} \rightarrow AB(i,j,k-1) \end{aligned}$$

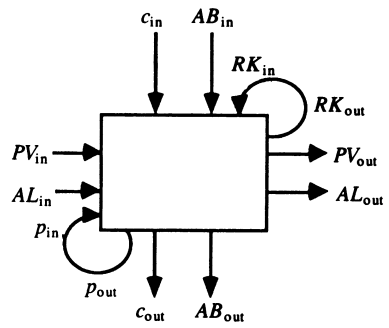
(* *Propagation of row k to other rows* *)

$$\begin{aligned} RK(i,j,k) = i = k &\rightarrow AB(i,j,k) \\ i > k &\rightarrow RK(i-1,j,k) \end{aligned}$$

(* *Propagation of multiplication factor inside row* $i \neq k$ *)

$$\begin{aligned} AL(i,j,k) = i = k \text{ or } j = 0 \text{ or } p(i,j,k) = 0 &\rightarrow 0 \\ AL(i,j-1,k) = 0 \text{ and } p(i,j,k) \neq 0 &\\ \rightarrow AB(i,j,k-1) &\\ \text{otherwise} \rightarrow AL(i,j-1,k) & \end{aligned}$$






```
c_out ← c_in;
if c_in = 1 (*i = k or i = k+n*)
then [AB_out ← RK_in :
        if PV_in ≠ 0 then
        [RK_out ← AB_in /PV_in;
         PV_out ← PV_in; l_out ← 0]
     else [if AB_in ≠ 0 then
             [RK_out ← 1; PV_out ← AB_in]
             else [RK_out ← 0; PV_out ← 0]
             p_out ← PV_out
             ]
     ]
else (*i ≠ k *)
     [if (AL_in = 0) and (p_in ≠ 0)
         then AL_out ← AB_in
         else AL_out ← AL_in;
         if (p_in ≠ 0)
         then AB_out ← AB_in − AL_out * RK_in
         else AB_out ← AB_in;
         RK_out ← RK_in;
         p_out ← p_in
         ]
```

Figure 4. Semisystolic array for computing P and Q.

The new data dependence graph derived from the recursion equations is shown in Fig. 3. Since '$i = k$' and '$i = k+n$' are the conditions to control various functions applied on the data, we add an additional variable c to carry *control signals* from index point (k,j,k) to index point $(k+1,j,k+1)$, as partially indicated by the dotted arrows in the graph. The effect of c will become clear when we map this *data/control dependence graph* into an array.

4. SEMISYSTOLIC ARRAY FOR SOLVING SIMULTANEOUS LINEAR EQUATIONS

Now we have a dependence graph which can be projected into a two-dimensional regular array in many directions. For example, if $(1,0,1)^T$ is selected as the projection direction, we will get the design presented in Ref. 7. To achieve maximum data pipelining rate, we project the graph in the direction of $(1,0,0)^T$ and obtain a semisystolic array as shown in Fig. 4. The function of PE, which can be naturally deduced from the details of recursion equations, is also given there. Note that the delays on the interconnections are so assigned due to the default time-schedule on the dependence graph, in other words, all computations in the vertical plane $i = t$ are executed at the same time step t.

The effect of control variable c is essentially the key to understand how the whole procedure is executed on the semisystolic array. Each row of c-values moves down through the array one row of PEs per time step. The kth row of $[A \mid B]$ (maybe partially updated already) will meet the row of 1's in the kth row of PEs at kth time step due to the input sequencing and the delays on the c-values. This kth row will stay there for n steps until another row of 1's comes in. During this period of time, all the other rows of $[A \mid B]$ will pass by and do the elementary row operations with respect to the kth row. Let us illustrate the data flows of the semisystolic array by a sequence of snapshots as shown in Fig. 5. The superscripts indicate how many times of row operations have been applied to the matrix elements. The first row of $[P \mid Q]$ comes out first, then the second row, and then the third row. This is the right order we need to multiply P^T and Q in the next stage of computation.

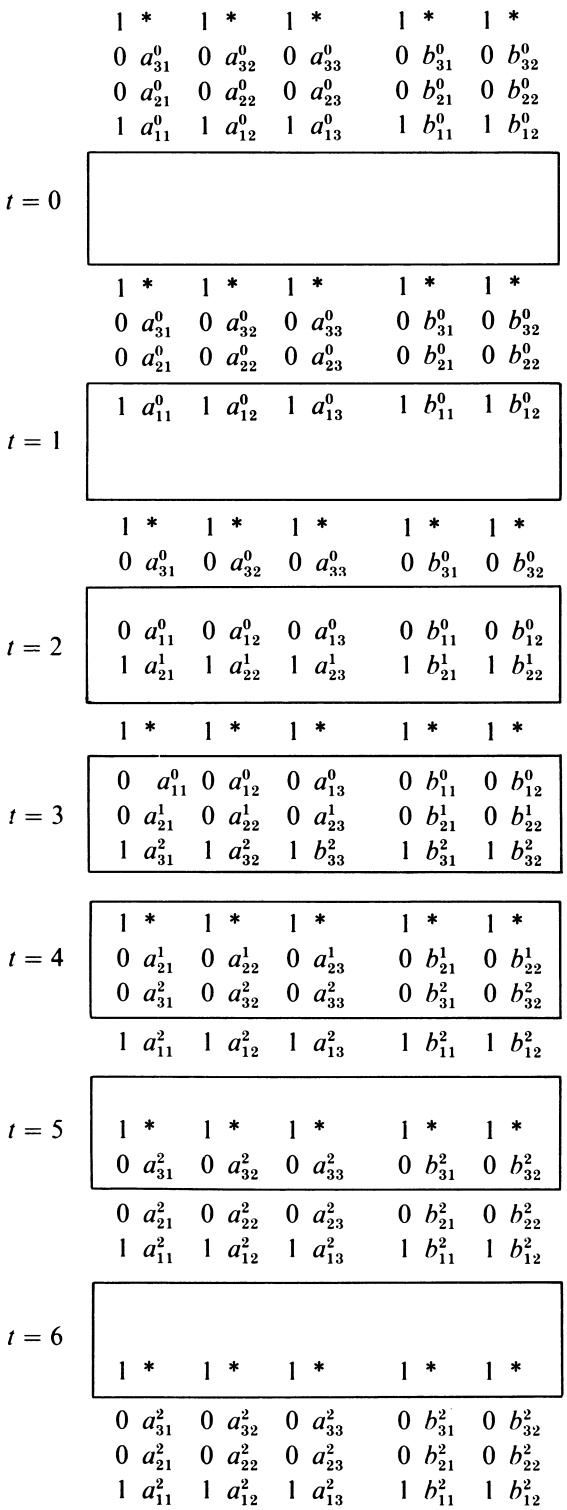


Figure 5. Snapshots of the semisystolic array for computing P and Q.

Because of the special feature of permutation matrix P, the following simple procedure can be used to multiply P^T and Q:

```
for k := 1 to n do
  for i := 1 to n do
    for j := 1 to m do
      if p_ki = 1 then x_ij := q_kj;
```

We locate another $n \times m$ rectangular array of PEs below the right part of the previously designed array to hold the values of X. In step $n+k+1$, $1 \leqslant k \leqslant n$, row k of P moves in parallel through the array from the left and row k of Q moves in parallel through the array from the top. If a value of Q meets a value 1 of P in a PE, it stays there as the final result of X. The entire structure of semisystolic array for solving simultaneous linear equations is now completed and depicted in Fig. 6.

5. THE PROPOSED SYSTOLIC ARRAY

Since a temporally local systolic array does not allow zero-delays on the interconnections, we have to make all delays positive. Here we introduce a very simple rule to serve the purpose.

Bisection retiming rule

- Divide the array in two parts P_1 and P_2 by a bisection line which intersects interconnections only.
- Choose a proper number d as the bisection delay.
- Add d to the delays on the intersected interconnections directing from P_1 to P_2.
- Add d to the delays on the input connections in P_2.
- Subtract d from the delays on the intersected interconnections directing from P_2 to P_1.

The idea behind this retiming rule is simply that we want all activities in part P_2 to be delayed by d steps and all the PEs in the system still work correctly. It has been proved that any semisystolic array can be converted into a functionally equivalent systolic array by repeatedly applying this bisection retiming rule.[7]

Now look at Fig. 6 again. Between every two consecutive rows (columns) we make a horizontal (vertical) bisection and do one retiming with delay 1. After a total of $3n+m-2$ retimings, the semisystolic array is systolized as shown in Fig. 7. In the systolic array, we add an $n \times n$ array in the lower left part just for transmitting data and make the whole array really spatially local. The wavefronts of input data streams are bent, because different numbers of delays are added to the input connections during the retiming process.

There are only three kinds of PEs in Fig. 7 and the functions of them are the same as those specified in Fig. 4 and Fig. 6 because the only difference between the semisystolic array and the systolic array is the timing of computation. One way of outputing the result data is to shift a column of X downward after the column is computed. This can be accomplished by supplying another set of input data right after the original set. In the second set, let A be the identity matrix and B be the zero matrix. Clearly the second set will not be altered by the elimination procedure. The diagonal 1's of the second A will eventually reach the lower-right PEs at the right time steps to trigger down the computed results of X. A sequence of snapshots is put in the appendix to illustrate the data flows of the systolic array.

By tracing the movements of b_{nm}, one can conclude that the systolic array uses $4n+m-2$ time steps to compute P and Q, and uses $6n+m-2$ time steps to solve the simultaneous linear equations. Since there is no empty step separating the consecutive data in the data streams, this systolic array achieves maximum data pipelining rate.

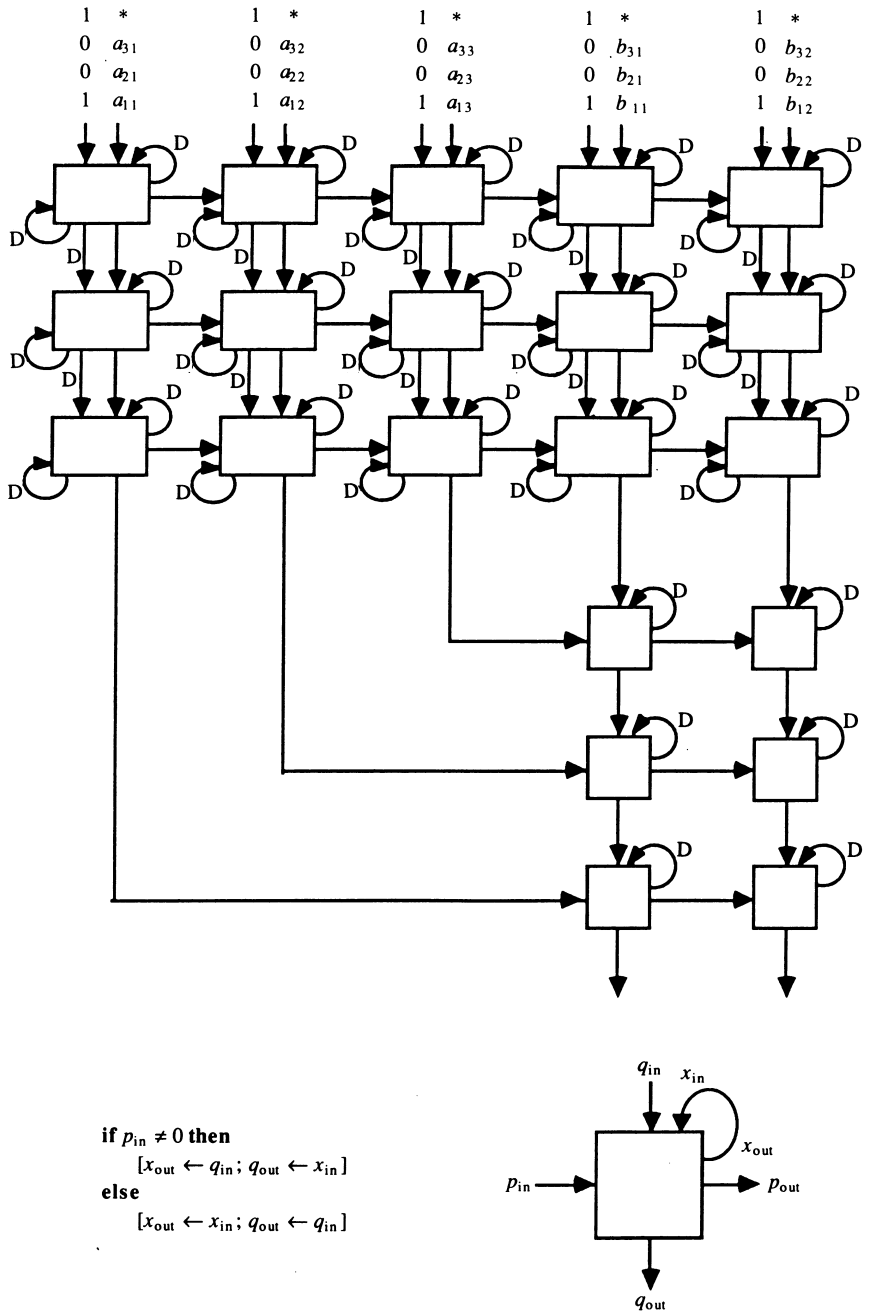


Figure 6. Semisystolic array for solving simultaneous linear equations.

6. CONCLUSION

In the dependence graph of the generalised Gauss–Jordan elimination procedure (Fig. 1), we can find a directed path which consists of $4n+m-3$ index points, for example, the path $(1,1,1) \rightarrow (n,1,1) \rightarrow (n,1,n) \rightarrow (n, n+m, n) \rightarrow (1, n+m, n)$. The time ordering of the computations along this path must be reserved by any parallel implementation of the procedure. This means that for this part of the computation our systolic array is only one time step higher than the lower bound. Although the whole computation is divided into two stages, they are tightly pipelined to form an integrated array and therefore totally avoid the external communications inherent in the matrix triangularisation-based designs.

Our design can easily be extended to detect the existence and uniqueness of solution by adding a right-bound detection line to each row of the upper half PEs. In the first stage of computation, if a detection line finds that all elements in the left n PEs are zero but there is a non-zero element in the right m PEs, the solution does not exist. If a detection line finds that all elements in the row are zero, then the problem has multiple solutions. For this case, we can directly output the matrices P and Q to save the features of the multiple solutions. But if we must get one of the solutions, we should modify the PEs in the lower right part to perform '*if* p_{ki} *is the first non-zero element then* $x_{ij} := q_{kj}$ *else if* p_{ki} *is other non-zero element then* $x_{ij} := 0$'.

For one system of linear equations ($m = 1$), our systolic array produces the solution in $6n-1$ time steps. In some applications,[10] the coefficient matrix A is fixed for many systems of linear equations. If m is sufficiently large, say $m > n$, then the average computation time for

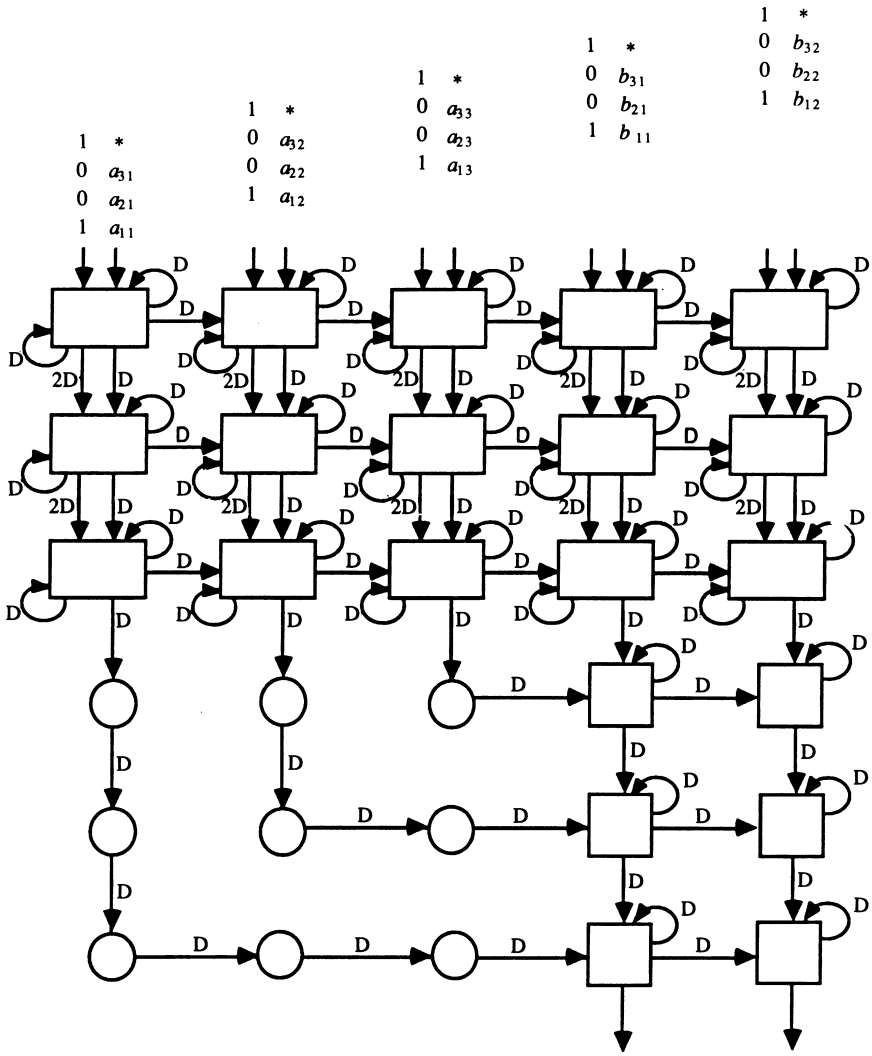


Figure 7. Systolic array for solving simultaneous linear equations.

one system is a small constant $(6n+m-2)/m < 7$. Another particular application is that when $m = n$ and B is the identity matrix our systolic array computes the *inverse* of A in $7n-2$ time steps. None of the previous designs mentioned in Section 1 achieve these performances because in those designs different systems of linear equations cannot be solved simultaneously.

Finally, a few words about numerical stability must be given. The most interesting part of our design, perhaps, is the possibility to pipeline, which makes crucial use of the assumption that no pivoting is needed. If pivoting is required for the reason of numerical stability, then a 'backward' wave is needed for every cycle and thus prevent the expedience of pipelining. This is also confirmed by the recent work of Roychowdhury and Kailath[14] in that a non-systolic algorithm is derived for Gaussian elimination with partial pivoting.

REFERENCES

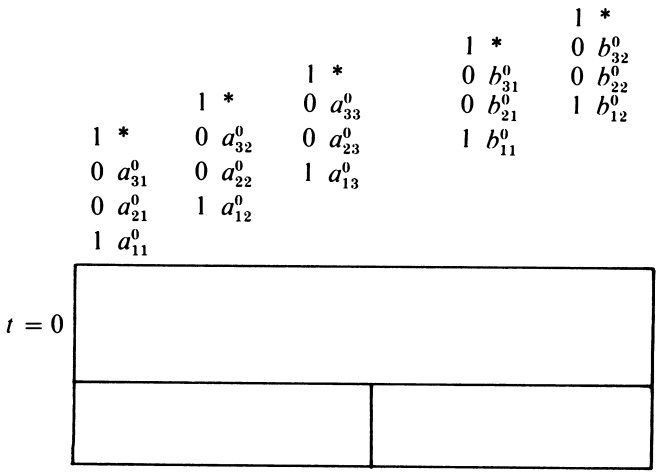
1. H. T. Kung and C. E. Leiserson, Systolic arrays (for VLSI). *Proceedings, SIAM Sparse Matrix Symposium*, 256–282 (1978).
2. H. T. Kung, Let's design algorithms for VLSI systems. *Proceedings, Cal. Tech. Conference on VLSI*, 65–90 (1979).
3. H. T. Kung, Why systolic architectures? *Computer*, **15**, 37–46 (1982).
4. R. M. Kant and T. Kimura, Decentralized parallel algorithms for matrix computation. *Proceedings, Fifth Annual Symposium on Computer Architecture*, 96–100 (1978).
5. A. Bojanczyk, R. P. Brent and H. T. Kung, Numerically stable solution of dense system of linear equations using mesh-connected processors. *SIAM Journal of Scientific Statistical Computing*, **5**, 95–104 (1984).
6. P. S. Lin and T. Y. Young, VLSI array design under constraints of limited I/O bandwidth. *IEEE Transactions on Computers*, **C-32**, 1160–1170 (1983).
7. F. C. Lin and I. C. Wu, On designing systolic algorithms. *Proceedings, Second International Symposium on VLSI Technology, Systems and Applications, Taipei, Taiwan*, 204–208 (1985). A revision of this paper is to appear in *IEEE Transactions on Computers*.
8. D. Moldovan, On the design of algorithms for VLSI systolic arrays. *Proceedings, IEEE*, **71**, 113–120 (1983).
9. D. Moldovan and J. A. B. Fortes, Partitioning and mapping algorithms into fixed-size systolic arrays. *IEEE Transactions on Computers*, **C-35**, 1–12 (1986).
10. C. E. Leiserson and J. B. Saxe, Optimizing synchronous systems. *Journal of VLSI and Computer Systems*, **1**, 41–67 (1983).

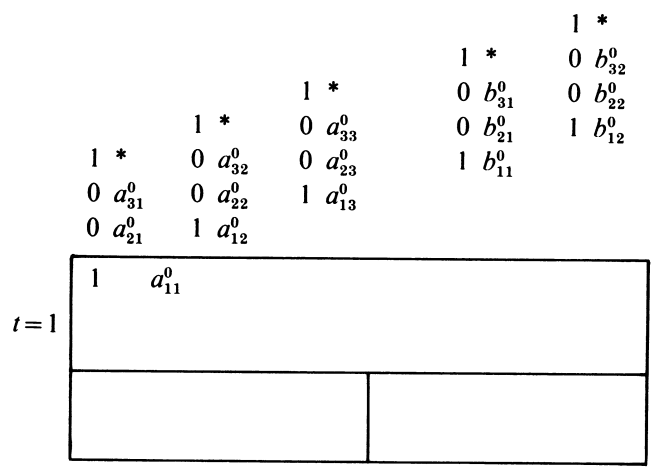
11. K. Hwang and Y. H. Cheng, Partitioned matrix algorithm for VLSI arithmetic systems. *IEEE Transactions on Computers*, **C-31**, 1125–1224 (1982).
12. H. T. Kung and M. S. Lam, Wafer-scale integration and two-level pipelined implementation of systolic arrays. *Journal of Parallel and Distributed Computing*, **1**, 32–63 (1984).
13. H. T. Kung, Memory requirement for balanced computer architectures. *Proceedings, 13th Annual Symposium on Computer Architecture*, 49–54 (1986).
14. V. Roychowdhury and T. Kailath, A non-systolic algorithm implementable on regular processor arrays. *Proceedings, IEEE International Symposium on Circuits and Systems*, 1328–1331 (1986).

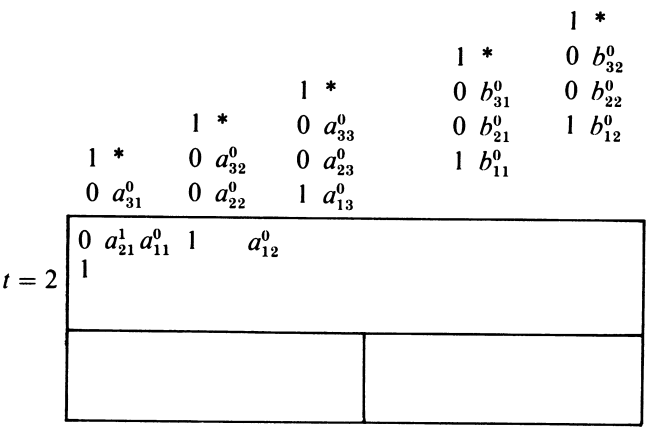
APPENDIX

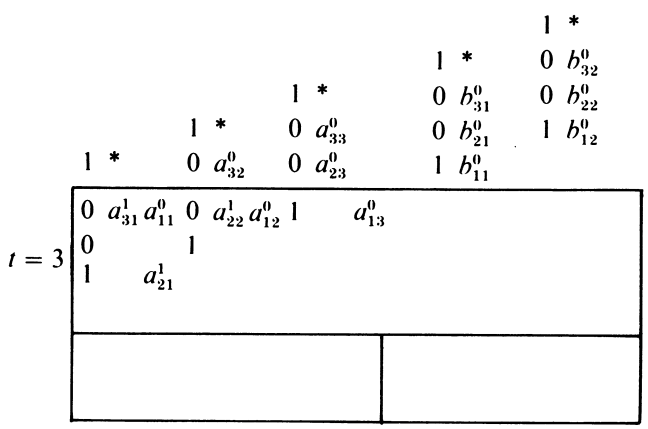
Snapshots of the proposed systolic array for solving simultaneous linear equations with $n = 3$, $m = 2$, *assuming*

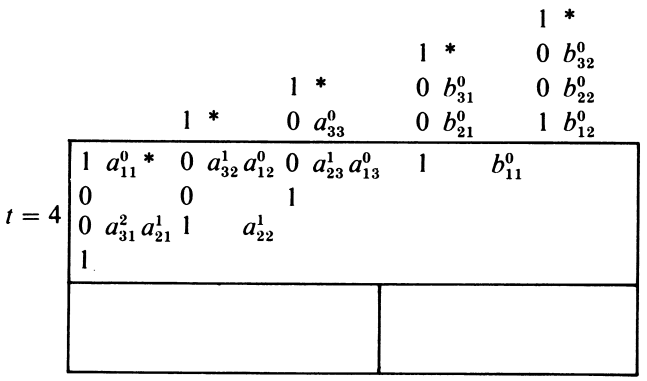
$$P = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}.$$

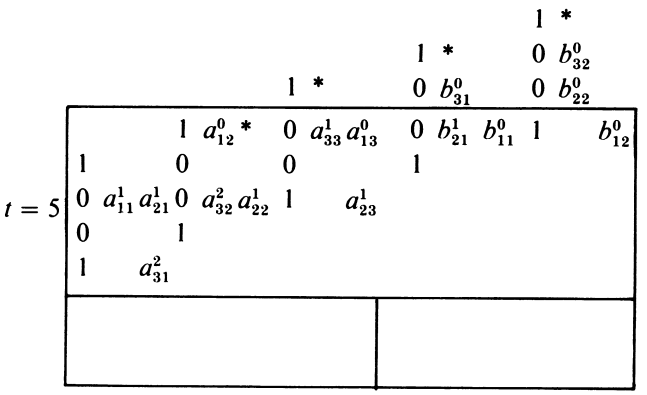












$t = 6$

										1	*
						1	*			0	b_{32}^0
				1	a_{13}^0	*	0	b_{31}^1	b_{11}^0	0	b_{22}^1	b_{12}^0
		1		0			0			1		
1	a_{21}^1	0	a_{12}^1	a_{22}^1	0	a_{33}^2	a_{23}^1	1		b_{21}^1		
0		0										
0	a_{11}^2	a_{31}^2	1		a_{32}^2							

$t = 7$

											1	*	
								1	b_{11}^0	*	0	b_{32}^1	b_{12}^0
					1			0			0		
		1	a_{22}^1		0	a_{13}^2	a_{23}^1	0	b_{31}^2	b_{21}^1	1		b_{22}^1
1		0			0			1					
0	a_{21}^2	a_{31}^2	0	a_{12}^2	a_{32}^2	1	a_{33}^2						
	0												

$t = 8$

												1	b_{12}^0 *
								1			0		
				1	a_{23}^1		0	b_{11}^1	b_{21}^1	0	b_{32}^2	b_{22}^1	
		1		0			0			1			
1	a_{31}^2	0	a_{22}^2	a_{32}^2	0	a_{13}^2	a_{33}^2	1		b_{31}^2			
	1		0										
	0												

$t = 9$

												1	
								1	b_{21}^1		0	b_{12}^1	b_{22}^1
				1			0			0			
		1	a_{32}^2		0	a_{23}^2	a_{33}^2	0	b_{11}^2	b_{31}^2	1		b_{32}^2
0		0		1									
1		0											
0													

$t = 10$

												1	b_{22}^1
								1			0		
					1	a_{33}^2		0	b_{21}^2	b_{31}^2	0	b_{12}^2	b_{32}^2
		1		0			d_{11}						
0		0		0									
1		0											

$t = 11$

												1	
									1	b_{31}^2	0	b_{22}^2	b_{32}^2
						0			d_{21}	d_{11}		d_{12}	
		1				0							
0		1				0							

$t = 12$

												1	b_{32}^2
									d_{31}	d_{11}		d_{22}	d_{12}
						1			d_{21}				
	0					1							

$t = 13$

										d_{11}		d_{32}	d_{12}
										d_{31}		d_{22}	
						0				d_{21}			

$t = 14$

										d_{11}		d_{12}	
										d_{31}		d_{32}	
										d_{21}		d_{22}	

$t = 15$

										d_{11}		d_{12}	
										d_{31}		d_{32}	
										d_{21}		d_{22}	