Systolic solution of linear systems over GF(p) with partial pivoting

Bertrand Hochet, Patrice Quinton, Yves Robert

▶ **To cite this version:**

Bertrand Hochet, Patrice Quinton, Yves Robert. Systolic solution of linear systems over GF(p) with partial pivoting. [Research Report] RR-0567, INRIA. 1986. inria-00075987

HAL Id: inria-00075987
https://inria.hal.science/inria-00075987

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INRIA

CENTRE DE RENNES
IRISA

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
BP105
78153 Le Chesnay Cedex
France
Tél. (1) 39 63 55 11

Rapports de Recherche

N° 567

SYSTOLIC SOLUTION OF LINEAR SYSTEMS OVER GF(p) WITH PARTIAL PIVOTING

Bertrand HOCHET
Patrice QUINTON
Yves ROBERT

Septembre 1986

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTÈMES ALÉATOIRES

Campus Universitaire de Beaulieu
Avenue du Général Leclerc
35042 - RENNES CÉDEX
FRANCE
Tél. : (99) 36.20.00
Télex : UNIRISA 95 0473 F

PUBLICATION INTERNE N° 308

SEPTEMBRE 1986

28 PAGES

Systolic solution of linear systems over GF(p) with partial pivoting

RESOLUTION SYSTOLIQUE DE SYSTEMES LINEAIRES SUR GP(p) AVEC PIVOTAGE PARTIEL

Bertrand HOCHET[+], Patrice QUINTON[++] and Yves ROBERT[+]

[+] CNRS, Laboratoire TIM3, BP 68, 38402 St Martin d'Hères Cedex, France
[++] CNRS, IRISA, 35042 Rennes Cedex, France

Abstract

We propose two systolic architectures for the Gaussian triangularization and the Gauss-Jordan diagonalization of large dense nxn matrices over GF(p), where p is a prime number. The solution of large dense linear systems over GF(p) is the major computational step in various algorithms issued from arithmetic number theory and computer algebra. The two proposed architectures implement the elimination with partial pivoting, although the operation of the array remains purely systolic. The last section is devoted to the design and layout of a CMOS 8 by 8 Gauss-Jordan diagonalization systolic chip over GF(2).

Index terms

Systolic arrays, Gaussian triangularization, Gauss-Jordan diagonalization, Solution of linear systems, Partial pivoting, Finite fields

.../...

This work has been supported by the Coordinated Research Program C3 of CNRS and Ministère de la Recherche et de la Technologie

RESOLUTION SYSTOLIQUE DE SYSTEMES LINEAIRES SUR CG(p) AVEC PIVOTAGE PARTIEL

Bertrand HOCHET[+], Patrice QUINTON[++] et Yves ROBERT[+]

\+ CNRS, Laboratoire TIM3, B.P. 68, 38402 ST MARTIN D'HERES Cédex, France

++ CNRS, IRISA, 35042 RENNES Cédex, France

Résumé

Nous proposons deux architectures systoliques pour la triangularisation de Gauss et la diagonalisation de Gauss-Jordan de grandes matrices denses nxn sur CG(p), où p est un nombre premier. La résolution de grands systèmes linéaires denses sur CG(p) est le principal pas de calcul de nombreux algorithmes issus de théorie des nombres et de l'algèbre. Les deux architectures proposées réalisent une élimination avec pivotage partiel, avec un schéma de fonctionnement totalement systolique. Le dernier paragraphe présente la conception d'un circuit CMOS pour la diagonalisation de Gauss-Jordan, dans le cas où n=8.

PAPIER RÉCUPÉRÉ ET RECYCLÉ

1. Introduction

The solution of large dense linear systems of algebraic equations in finite fields appears to be the computational kernel of many important algorithms. Let us mention three representative applications:
• the large integer factoring routines, where the final stage necessitates performing Gaussian elimination on a large dense matrix over GF(2): [MB] [PW] [Poe].
• the factorization of polynomials over GF(p), where p is a prime number, via Berlekamp's algorithm: [Knu].
• linear prediction in the analysis of discrete signals: [Mak].

In some of these applications, matrices of several thousands of rows and columns are required, and there are a large number of matrices to triangularize. However, Gaussian elimination requires an amount of arithmetic operations proportional to the cube of the order of the matrix, and this leads to serious limitations both on the size of the problems that can be dealt with, and the speed of the solution, when implemented on sequential computers.

Parallel processing therefore offers an interesting perspective for such computations. Indeed, Parkinson and Wunderlich report in [PW] the implementation of Gaussian elimination over GF(2) on the ICL DAP (which has 4096 processors), and Poet [Poe] considers the use of special-pupose hardware to factor 140 digit numbers using the same algorithm.

In this paper, we introduce systolic architectures for the Gauss triangularization algorithm and the Gauss-Jordan diagonalization algorithm of general dense matrices over a finite field GF(p), where p is prime. Systolic arrays have been introduced by Kung and Leiserson [KLe] and consists of a large number of elementary processors (or cells) which are mesh-interconnected in a regular and modular way, and achieves high performance through extensive concurrent and pipeline use of the cells. Such architectures have now proven to be efficient for the solution of compute-bound problems, and we refer the reader to [AFQ] [Kun] for a general presentation of the systolic model.

Systolic arrays for Gaussian triangularization of dense real matrices are well known (see [AMD], [GK] among others). However, partial pivoting has never been implemented in such arrays, due to the fact that the search for a pivot in a whole row or column would break down the locality and the regularity of the systolic design. Hence systolic arrays for Gaussian elimination over $\mathbb{R}$ only apply to symmetric positive definite or diagonally dominant matrices. To triangularize general matrices, one must use an orthogonal factorization via Givens rotations (see the systolic implementations of [AMD], [BBK], [GK], [SK] among others).

In a finite field GF(p), partial pivoting can not be avoided, since in average every other p element is zero. But it turns out that it can be solved much more efficently than in $\mathbb{R}$, since any non-zero element in a given row or column can be chosen as the pivot: there is no need for an entire scanning, the search can stop as soon as a non-zero element has been found out. This simple consideration will be very important in our implementations.

The paper is organized as follows. First we concentrate on GF(2), and we differ the implementation over GF(p) for any prime p up to section 4. Section 2 is devoted to the design of a systolic array for Gaussian elimination over GF(2) of an n by (n+1) matrix (A,b). The geometry of this array is similar to that of the array of Gentleman and Kung [GK], but the operation of the processors is different to manage the additional pivoting features.

When there are q linear systems $Ax_i = b_i$ to be solved, the previous array requires $n^2/2+qn$ cells and 3n+q time-steps. There still remains q triangular systems to solve. Another srategy is to directly implement the Gauss-Jordan diagonalization algorithm with partial pivoting. We show in section 3 that the solution matrix $A^{-1}B$ (where $B = (b_1, b_2, ..., b_q)$ can be computed within 4n+q time-steps on a systolic array of n^2 cells. In particular, the inverse of A can be computed within 5n time-steps.

We show in section 4 how to modify the operation of the elementary processors to deal with an implementation over GF(p), p prime number. Finally, we detail in section 5 the layout of a CMOS 8 by 8 Gauss-Jordan diagonalization systolic chip over GF(2) which has been designed using the system LUCIE [Pai] and which is currently fabricated through the French Multi-Project Chip [ABD].

2. Gaussian elimination over GF(2)

Let A be a dense nxn matrix and b a n-vector. To solve the system

(1) Ax=b

we transform it into an equivalent triangular system

(2) Tx = b'

The transformation of the system (1) into the system (2) is done by triangularizing the matrix A, using Gaussian elimination with partial pivoting.

We briefly recall the operation of Gentleman and Kung's two-dimensional triangular array of orthogonally connected processors to triangularize a real dense nx(n+1) matrix (A,b) without pivoting. This array is depicted in figure 1. The total number of cells in the array is n[(n+1)/2+1]. The array is composed of n rows, each row k including n+2-k processors numbered from left to right $P_{k,1}$, ... , $P_{k,n+2-k}$. The matrix (A,b) is fed into the array column by column. More specifically, column j of the matrix is input to processor P_{1j}, one new element each time-step, beginning at time t=j. This input format is depicted in figure 1.

The first row of (A,b) is stored in the upper row of the array, and, as any row numbered i≥1 is read by the array, it is combined with the first one in order to zero out element a_{i1}; this combination corresponds to a transformation of the form

$$\begin{bmatrix} a_{11}, \dots, a_{1n}, b_1 \\ a_{i1}, \dots, a_{in}, b_i \end{bmatrix} := M_{i1} \begin{bmatrix} a_{11}, \dots, a_{1n}, b_1 \\ a_{i1}, \dots, a_{in}, b_i \end{bmatrix}, \quad M_{i1} = \begin{bmatrix} 1 & 0 \\ -a_{i1}/a_{11} & 1 \end{bmatrix}$$

The 2x2 matrix M_{i1} is computed by P_{11} at time t = i, and sent to the other cells of the first row; more precisely, cell $P_{1,k}$ (k≥2) performs the transformation

$$(a_{1k}, a_{ik})^t := M_{i1} \cdot (a_{1k}, a_{ik})^t$$

at time t = i+k-1.

Let

$$(A^{(k)}, b^{(k)}) = \begin{bmatrix} a_{11}^{(k)} & \dots & & a_{1n}^{(k)} & b_1^{(k)} \\ & & a_{kk}^{(k)} & \dots a_{kn}^{(k)} & b_k^{(k)} \\ 0 & & a_{nk}^{(k)} & \dots a_{nn}^{(k)} & b_n^{(k)} \end{bmatrix}$$

denote the matrix obtained from (A,b) after the elimination of the elements at positions (i,j) such that i>j, j = 1, ... ,k-1. Then ($a_{kk}^{(k)}$, ..., $a_{kn}^{(k)}$, $b_k^{(k)}$) is stored in the k-th row of the array.

When ($a_{ik}^{(k)}$, ..., $a_{in}^{(k)}$, $b_i^{(k)}$), i>k, is read by this row of cells, it is combined with ($a_{kk}^{(k)}$, ..., $a_{kn}^{(k)}$, $b_k^{(k)}$) using a 2x2 matrix M_{ik}, in order to set $a_{ik}^{(k)}$ to zero.

2.1. General description of the array

The key-idea to include partial pivoting in the algorithm is to generate matrices M_{ik} whose structure depends on the values of $a_{kk}^{(k)}$ and $a_{ik}^{(k)}$. For sake of simplicity, let us consider the case k=1, i=2, that is, when the first matrix M_{21} is generated by processor P_{11}. Element $a_{11} = a_{11}^{(1)}$ is stored in P_{11}, and $a_{12} = a_{12}^{(1)}$ is being input to P_{11}. Two cases must be considered:

- (a) if $a_{11} = 1$, row 1 will be chosen as the pivoting row, and the elimination of $a_{21}, a_{31}, \ldots, a_{n1}$ will be done using usual Gaussian elimination. In our example:
  - (a1) if $a_{21} = 0$, there is nothing to do, M_{21} is equal to I_2, the identity matrix of order 2.
  - (a2) if $a_{21} = 1$, add row 1 to row 2 to zero out a_{21}, that is

$$M_{21} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$$

Note that addition denotes here the addition in GF(2), i.e. the exclusive or operation XOR.

- (b) if $a_{11} = 0$, we have to find out a pivoting row. Two cases occur:
  - (b1) if $a_{21} = 0$, we do nothing, and choose $M_{21} = I_2$
  - (b2) if $a_{21} = 1$, we exchange rows 1 and 2, and M_{21} is the permutation matrix

$$M_{21} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

Row 2 is then stored in the first row of the array and acts as the pivoting row for phase 1, which consists in zeroing out all the elements but one of the first column of the matrix (A,b).

Note that in the case (b1), row 2 will not be modified during the first phase of the algorithm. Assume that a_{i1}, i>2, is the first non-zero element that we find out: row i will be used as a pivoting row for phase 1, but since we already know that $a_{11}, a_{21}, \ldots, a_{i-1,1}$ are all zero, we do not have to combine row i with row 1, 2, ..., i-1.

We can give an informal description of the algorithm as follows:

for k := 1 **to** n-1 **do**
{phase k of the algorithm }
begin
1. find out the first $j \geq k$ such that $a_{jk}^{(k)} = 1$
2. exchange row k and row j (if $j \neq k$), that is, choose row j as pivoting row
3. zero out the elements $a_{ik}^{(k)}$ such that i>j and $a_{ik}^{(k)}=1$ by adding row j to row i (here addition denotes the addition in GF(2), that is, the XOR operation)

end

For a given k, note that the steps 2 and 3 are not executed if we can not find a j such that $j \geq k$ and $a_{jk}=1$: in this case, the matrix A is rank deficient.

2.2. Operation of the processors

There are two types of processors in the array, respectively represented as circular and square processors in the figure 1. In the k-th row of the array:
- the processor P_{k1} is a circular processor, which generates the matrices M_{ik}, i>k.
- all the other processors are square processors, which apply the transformation relative to M_{ik}.

The operation of these two types of processors is detailed in the figure 2. The operation of a given processor depends on whether it is the first time it operates. In the description of the figure 2, we assume for simplicity that there is a boolean called init (for "initialization") stored in every processor which is set to "true" at the beginning of the computation. This boolean controls the operation of the processors. In an actual implementation, the instruction to start the computation would be input to processor P_{11} together with the first coefficient a_{11} and systolically propagated through the whole array (it can be easily checked that processor P_{kj} operates for the first time at step 3k+j-3).

Circular processors

The first time they operate, circular processors store their input in their internal register. Afterwards, they compute the matrices M_{ik} and send them rightwards to the square processors.

Rather than directly generating the matrices M_{ik}, the circular processors transmit rightwards one of the following three instructions:
- *id*, which stands for the identity matrix
- *perm*, which requires a permutation of rows
- *add*, which requires an addition of rows.

This allows the instructions generated by the circular processors to be encoded using only two bits.

Square processors

Again, square processors store their input in their internal register the first time they operate. Afterwards, they update their vertical input and possibly their internal register, according to the instruction they receive from the left. See figure 2 for a complete description.

2.3. A detailed example (n=4)

Let n=4 and A x = b be the following system over GF(2):

$$\begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

We want to describe the operation of the first row of the array, which is responsible for the first phase of the algorithm. For sake of clarity, assume that the rows of the matrix (A,b) are input sequentially (rather than pipelined) to the array.

First row 1 is stored in the internal registers of P_{11}, ..., P_{15}. When P_{11} receives a_{21}=1, it generates the instruction *perm*, so that the last four elements of row 2 take place in the internal registers, whereas the corresponding elements of row 1 are output downwards. When P_{11} receives a_{31}=0, it generates *id*, and the last four elements of row 3 are transmitted downwards without modification. Finally, when P_{11} receives a_{41}= 1, it generates *add*, and P_{12}, ..., P_{15} perform a XOR operation.

The operation of the first row of processors can be summarized as follows:

	P_{11}	P_{12}	P_{13}	P_{14}	P_{15}
Internal registers	1	0	1	1	1
		↓	↓	↓	↓
		-	-	-	0
Outputs		-	-	1	1
		-	0	0	1
		1	0	0	-
		1	1	-	-
		0	-	-	-

During phase 2 and phase 3, the following instructions are generated:

Phase 2

time-step	t=5	t=6
Processor P_{22}	*perm*	*add*

Phase 3

time-step	t=8
Processor P_{33}	*id*

At the end of the computation, the following matrix (T,b') is stored in the array:

1	0	1	1	1
	1	0	0	1
		1	0	1
			1	1

and solving the triangular system leads to $x = (1,1,1,1)^t$, which can be checked from the original matrix (A,b) to be the solution of the linear system.

2.4. Unloading the array

At the end, the matrix (T,b') is stored in the array as follows (assuming n=4):

t_{11}	t_{12}	t_{13}	t_{14}	b'_1
	t_{22}	t_{23}	t_{24}	b'_2
		t_{33}	t_{34}	b'_3
			t_{44}	b'_4

There remains to unload the array. According to the general philosophy of the model, this must be done systolically, by propagating special boolean control instructions. Moreover, we would like to pipeline the unloading of the array with the computation phase, so that n additional steps are not necessary. Various schemes are possible. We outline a scheme where each processor

sends the content of its register to the right when it has finished to operate. Let $t_{i,n+1} = b'_i$ for convenience.

At time t=4 in our exemple, P_{11} operates for the last time. It remains idle for one step and then sends t_{11} rightwards. P_{12} finishes to work at time t=5. It transmits t_{11} to the right at t=6, and then sends t_{12} at time t=7. The process goes on, and t_{1i} is output by P_{15} at time t=9+i , for $1 \le i \le n+1$. Similarly, we start unloading the second row at time t=8. We see that the first rows of the array are unloaded, while the last rows still operate, thus achieving the pipelining that is sought.

Indeed, in the general case, the last computation occurs in $P_{n,n+1}$ at time 3n-1, and $t_{i,i+k}$ ($1 \le i \le n$, $0 \le k \le n+1-i$) is output from the array at time 2n+i+k+1. The total computation time is 3n+2.

We point out that the unloading of the array can not be simplified as in the case of Gaussian elimination over $\mathbb{R}$ without pivoting: in the real case, the coefficients $t_{i,i+k}$ are not modified after being stored in the array, hence they can be sent downwards immediately. Here on the contrary, the $t_{i,i+k}$ can be modified by any following permutation of rows.

2.5. General remarks

We have described here an array for matrix triangularization using Gaussian elimination. There still remains a triangular system to be solved. Assuming the matrix A is non-singular, we can use an other systolic array, the triangular system solver of Kung and Leiserson [KLe], which requires 2n additional steps to solve the system T x = b'. However, this would require the triangular matrix to be stored in the host and reordered by diagonals before being fed in Kung and Leiserson's array. Rather, it is more efficient to use the Jordan elimination scheme depicted in [CRT] [RTc] which can be implemented on the same array and requires only n additional steps to provide the solution x, leading to a very efficient scheme of only $n^2/2$ cells and 4n time steps to completely solve the system A x = b.

We point out that checking for the non-singularity of the matrix can be done on the fly very easily, simply by testing the content of the registers of the circular processors at the end of the triangularization phase.

3. The Gauss-Jordan diagonalization algorithm with partial pivoting over GF(2)

The systolic array presented in section 2 allows general dense systems of equations $Ax = b$ to be solved: first triangularize the matrix (A,b), then solve the triangular system.

Assume now that there are q systems $Ax_i = b_i$ to solve, $1 \le i \le q$, and let B be the nxq matrix $B=(b_1, b_2, ..., b_q)$. To triangularize (A,B), we can simply extend the triangularization array by adding q-1 columns on the right, leading to a total number of n[(n+1)/2+q] cells. If PA= LT, where P is a permutation matrix, L is lower triangular with unit diagonal, and T is upper triangular, we obtain the matrix $(T,L^{-1}PB)$ within 3n+q steps. There remains q triangular systems to be solved. This can be done within 2n+q steps using an array of qn cells (simply concatenate q triangular systems solvers of [KLe]). Hence we need a total number of $n^2/2 + 2qn + o(n)$ cells and of 5n+2q time-steps to solve the q linear systems. However, this evaluation does not take into account the fact that the matrix $(T,L^{-1}PB)$ has to be stored in the host and reordered by diagonals before entering the second systolic array.

We concentrate in this section on the design of a systolic array which directly computes the product $A^{-1}B$, where A is a dense (nonsingular) nxn matrix and B is a dense nxq matrix. The inverse of A is computed via the Gauss-Jordan diagonalization algorithm with partial pivoting. The performance of the new array is much better for large q: only n^2 cells and 4n+q time-steps are required by the architecture that we are going to describe. First we briefly recall the well-known Gauss-Jordan diagonalization algorithm.

3.1. The Gauss-Jordan diagonalization algorithm

Let C denote the n by n+q matrix C = (A,B). We want to reduce C to the matrix $(I,A^{-1}B)$. When no pivoting is needed, the usual way to proceed is to pre-multiply C by n elementary nxn matrices $J_1, J_2, ..., J_n$ in order to obtain after n steps

$$J_n \dots J_2 J_1 C = (I,A^{-1}B)$$

Therefore, define $C_k = J_k . C_{k-1}$, starting with $C_0 = C$. We choose J_k so that the first k columns of C_k are those of I_n, the identity matrix of order n. Thus C_k has the following structure:

$$C_k = [\ \text{the first k columns of } I_n \mid C_k^{\circ}\ ]$$

where C_k° denotes the nx(n+q-k) matrix built up with the last n+q-k columns of C_k.

In particular, C_n° is the desired matrix $A^{-1}B$. The matrix J_k only differs from I_n by its k-th column, which we denote for convenience as

$$[\ c_{1k}^{(k)} \ \dots \ c_{nk}^{(k)}\ ]^t$$

The coefficients of C_k° are denoted $(c_{ij}^{(k)})$, $1 \le i \le n$, $k+1 \le j \le n+q$. Therefore $c_{ik}^{(k)}$ refers to the k-th column of J_k, while $c_{ij}^{(k)}$ with $k<j$ refers to the matrix C_k°. The values of the $c_{ij}^{(k)}$, $1 \le i \le n$, $k \le j \le n+q$, $1 \le k \le n$, are computed recursively by the following algorithm, starting from $c_{ij}^{(0)} = c_{ij}$:

```
{ Gauss-Jordan diagonalization algorithm }
for k := 1 to n
begin
        { compute J_k }
        c_kk^(k) := 1 / c_kk^(k-1)
        for i := 1 to n, i ≠ k
                c_ik^(k) := - c_ik^(k-1) * c_kk^(k)
        { compute C_k° }
        for j := k+1 to n+q
        begin
                for i := 1 to n, i≠k
                        c_ij^(k) := c_ij^(k-1) + c_ik^(k) * c_kj^(k-1) ;
                c_kj^(k) := c_kk^(k) * c_kj^(k-1) ;
        end ;
end ;
```

The matrix J_k can be viewed as the (commutative) product of n elementary matrices. The first n-1 ones apply the transformations M_{ik} of section 2, and are chosen so as to annihilate the elements in position (i,k), i≠k. The n-th matrix only differs from I_n by its element in position (k,k) chosen so as to divide row k by the pivot $c_{kk}^{(k-1)}$.

Contrarily to the case of Gaussian elimination, every row of the matrix has to be combined with the pivoting row at each step. Provided that the organization of the array allows to do so, partial pivoting can be introduced just as before. During step k, column k is scanned out until a non-zero element in position (i,k), say, is found. M_{ik} is still a permutation matrix, and row i is used as pivoting row and combined with all the rows of the matrix whose k-th element is non-zero.

We can give an informal description of the algorithm as follows:

```
for k := 1 to n do
{phase k of the algorithm }
begin
    1.  find out the first j ∈ {k, k+1, ..., n, 1, 2, ..., k-1} such that a_jk^(k) = 1
    2.  if j≠k, exchange row k and row j, that is, choose row j as pivoting row
    3.  zero out the elements a_ik^(k) such that i≠j and a_ik^(k)=1 by adding row j to row i
        (here addition denotes the addition in GF(2), that is, the XOR operation)
end
```

For a given k, note that statements 2. and 3. are not executed if we can not find a non-zero element in column k, that is, statement 1. does not succeed. In this case, the matrix A is singular. This leads to the systolic implementation which is now presented.

3.2. Systolic implementation

As we already mentioned, every row of the matrix must be combined with the pivoting row at each phase of the algorithm. That is the reason why we choose an implementation which is "dual" of that of the previous section. Rather than storing a coefficient of the matrix in the internal register of each cell, we store (an encoding of) the transformation matrix M_{ik}. Moreover, the matrix is input in a transposed fashion, so that each row can meet all the other rows while moving through the array.

Figure 3 depicts the Gauss-Jordan diagonalization array. It is a two-dimensional array of orthogonally connected processors with n rows. Each row k comprises n processors numbered from left to right $P_{k,1}$, ... , $P_{k,n}$. There is a one-step delay cell at the left of each row, named P_{k0} for convenience. There are also two delay cells at the right of each row, as shown in figure 3. The operation of each processor is detailed in figure 4. There are two types of processors, represented as square cells (type 1) and double square cells (type 2). In the k-th row of the array, the rightmost processor $P_{k,n}$ is of type 2. All the other processors $P_{k,1}$, ..., $P_{k,n-1}$ are of type 1. The cells operate as follows:

Square cells

Square cells first initialize their internal register by storing the operation to be performed on all the subsequent coefficients of the two rows. Just as in the previous section, the operation is to be chosen in the set { *id, perm, add* } according to the value of the first inputs. Once initialized, the cells update their inputs as indicated in figure 4.

Double square cells

Double-square cells first test for the non-singularity of the matrix: if the first input of cell $P_{k,n}$ is zero, then all the elements of column k after the first k-1 phases of the algorithm are zero, and the matrix is singular. To test this condition, a boolean variable, called *test* , is initialized to false and moves through all the double square cells. When it is output from $P_{n,n}$ at the end of the computation, its value is true if and only if the algorithm has failed, that is, if the matrix is singular.

Once initialized, double square cells simply transmit their input (see figure 4).

The operation of a given processor in the array depends on whether it is the first data item it receives. As shown in figure 4, each processor has a control boolean *init* initialized to true. This boolean specifies the operation to be performed and the line along which the data is to be sent out. In an actual implementation, the instruction to start the computation would be input to the top-left corner of the array and systolically propagated to all the processors. It can be easily checked that processor P_{kj} operates for the first time at step 3k+j-2, so that the start signal would move at full speed to the right and at one-third speed downwards.

The matrix A, followed by the matrix B, is fed into the array row by row. More specifically, row k of the matrix C = (A,B) is input to processor $P_{1,k-1}$, one new element each time-step, beginning at time t=k. This input scheme is depicted in figure 3.

The k-th row of the array is devoted to the computation

$$C_k^{\circ} = J_k C_{k-1}^{\circ}$$

Recall that J_k is the product of n elementary matrices, some of which possibly being permutation matrices.The leftmost delay cell P_{k0} transmits the input data arriving from the top to the right. Square cells of the row use the first data value arriving from the top to compute the instruction which they store in their internal register. Then they pass downwards all the following data after modification. Similarly the rightmost processor $P_{k,n}$ modifies and stores the first input data arriving from the left and passes downwards all the following data after modification. Thus after a row of n+q input data flows through the whole array, its length is shortened by n to become a row

of q output data. The n by n+q matrix $C_0 = C_0^\circ = C$ is input to the first row of the array, the n by n+q-1 matrix C_1° is input to the second, ..., and the n by q+1 matrix C_{n-1}° is input to the n-th row; finally the array outputs the n by q matrix $C_n^\circ = A^{-1}B$.

It is important to note that the rows of the matrix are "reordered" in some sense when moving through the array: the input of the processsors $P_{k0}, P_{k1}, ..., P_{k,n}$ in the k-th row of the array are respectively the rows k, k+1, ..., n, 1, ..., k-1 of C_{k-1}°. The matrices M_{ik}, $i \neq k$, are computed and stored in the processors $P_{k,1}, ..., P_{k,n-1}$. More precisely, M_{ik} is stored in $P_{k,(i-k) \bmod n}$. Note that M_{kk} is always the identity in GF(2): either the matrix is singular, or there is already a 1 on the diagonal. After the processors are initialized, they perform the multiplication $C_k^\circ = J_k C_{k-1}^\circ$. The matrix C_k° is output from the k-th row of the array in row order, the leftmost row of the matrix being now row k+1.

We can state the following: given a dense nonsingular n x n matrix A and a dense n x q matrix B, the orthogonal systolic array of n^2 processors can compute $A^{-1}B$ within 4n+q-2 time-steps. In particular, we can compute the inverse A^{-1} of a dense nxn matrix within 5n-2 steps.

3.3. A detailed example (n=4, q=3)

Let n=4, q = 3, and A, B be the following matrices over GF(2) :

$$A = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{bmatrix} \qquad B = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

We want to describe the operation of the first row of the array, which is responsible for the first phase of the algorithm. For sake of clarity, assume that the columns of the matrix C = (A,B) are input sequentially (rather than pipelined) to the array.

First $a_{11}=0$ is transmitted to the right by the delay cell P_{10}. When P_{11} receives $a_{21}=1$ and $a_{11}=0$, it generates the instruction *perm*, so that a_{21} replaces a_{11} and moves rightwards: row 2 will be chosen as the pivoting row for phase 1 of the algorithm. When P_{12} receives $a_{21}=1$ and $a_{31}=0$, it generates the instruction *id*. When P_{13} receives $a_{21}=1$ and $a_{41}=1$, it generates the instruction *add*. When $a_{21}=1$ is input to processor P_{14}, the control boolean test remains equal to false. Once initialized, the processors P_{11}, P_{12} and P_{13} perform the instruction sored in their internal registers.

The operation of the first row of processors can be summarized as follows:

	P_{11}	P_{12}	P_{13}	P_{14}
Internal registers	perm	id	add	-
	↓	↓	↓	↓
	-	-	-	0
Outputs	-	-	0	1
	-	1	1	1
	1	1	0	1
	0	1	1	1
	1	0	0	0
	0	0	1	-
	1	1	-	-
	0	-	-	-

In other words, the first row of the array outputs the matrix

$$C_1^{\circ} = \begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$

Therefore, during phase 2, row 3 is chosen as pivoting row: P_{21}, P_{22} and P_{23} generate respectively the instructions *perm*, *add* and *id*, so that

$$C_2^{\circ} = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 \end{bmatrix}$$

During phase 3, row 3 is the pivoting row; P_{31}, P_{32} and P_{33} generate respectively the instructions *id*, *add*, and *id*, so that

$$C_3^{\circ} = \begin{bmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 \end{bmatrix}$$

Finally, during phase 4, row 4 is the pivoting row; P_{41}, P_{42} and P_{43} generate respectively the instructions *add*, *id*, and *id*, so that at the end of the computation, the matrix $A^{-1}B$ is output from the array:

$$C_4^{\circ} = A^{-1}B = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 0 & 1 \end{bmatrix}$$

Since the first input to all the processors P_{14}, P_{24}, P_{34} and P_{44} was a 1, the final value of the boolean test is false. This guarantees that the algorithm has not failed, that is, that the matrix A is non-singular. The first column of $A^{-1}B$ is $x = (1,1,1,1)^t$, the solution vector of section 2.3.

4. Extension to GF(p), p prime number

The two previous arrays can be easily extended to deal with computations over a finite field GF(p), where p is any prime number. The key idea is the same: the first non-zero element found out during the search is chosen as pivot.

The only modification in the operation of the processors is to replace the *add* instruction by the appropriate combination factor. Hence the processors now generate an instruction

$$op \in \{ id, perm, comb \}$$

Whenever op=comb, an appropriate combination factor is also be generated, and transmitted or stored according to the program of the cell

4.1. Gaussian elimination

The operation of the processors is detailed in figure 5.

Circular cells

Circular cells now compute the inverse of their first non-zero input and store it in their internal register. Together with the instruction generated, they transmit rightwards the combination factor, which is significant only if op = *comb*.

Square cells

Square processors update their inputs according to the instructions that they receive. In particular, when op= *comb*, they perform a multiply-and-add.

4.2. Gauss-Jordan diagonalization

The operation of the processors is detailed in figure 6.

Square cells

In the Gauss-Jordan diagonalization systolic array over $\mathbb{R}$ of [RTr], the leftmost delay cell is replaced by a cell which computes the inverse of its first input (and transmits all the following inputs without modification). This allows square processors to only perform multiply-and-add operations. Due to the permutation instructions in our algorithm, all square cells must be able to perform a division as well as a multiply-and-add. Division occurs when the cell operates for the first time, as indicated in figure 6.

Double square cells

Again, double square processor P_{kn} must be able to compute the inverse of their first input, provided it is non-zero, so as to generate the transformation matrix M_{kk}. If the first input is zero, they set the boolean *test* to true, which means that the algorithm has failed.

4.3. Arithmetic requirements

The arithmetic requirements are more complex in GF(p) for general p than in GF(2). A possible way to implement the inverse computation is a table look-up, whereas the multiplication can be done using the MMA Modular Multiplication Algorithm of [Bla], which consists of reducing each partial summation modulo p before going on: this prevents to perform a division by p at the end. Finally, the cost of an addition is twice that in $\mathbb{R}$ (for the same number of bits) because a+b mod p requires the computation of a+b and a+b-p.

5. A 8 x 8 implementation over GF(2)

We briefly describe the design and layout of a systolic chip which implements the Gauss-Jordan diagonalization algorithm over GF(2). The chip has been designed using the system LUCIE [Pai]. The technology is CMOS double metallisation with a 2 µm grid. The chip is currently being fabricated via the French Multi-Project Chip [ABD]. We have designed a 8 by 8 array, but of course the modularity of systolic arrays would permit to design larger arrays without any difficulty.

The only modification to the description of section 3.2. is the initialization of the array. As we already mentioned, there is a control boolean moving systolically through the whole array together with the coefficients of the matrix. The number of input pads is 13, distributed as follows: 8 pads for the rows of the matrix, 1 port for the initialization boolean, 2 pads for the alimentation and the ground, and 2 pads for the two non-overlapping clocks (the operation of the array is totally synchronous). We have 9 outputs pads, 8 for the rows of the solution matrix, and 1 for the boolean indicating whether the algorithm has failed or not.

More generally, the design of an n x n array will require 2n+6 pads. As usual in two-dimensional systolic arrays implementations, the main limitation to the size of the array is due to the number of input/output pins rather than to area considerations.

The dimensions of a square cell are 380 x 200 μm^2, and those of a double square cell are 170 x 200 μm^2. A global layout (without the internal description of each cell) of the whole 8 by 8 array is given in figure 7. The chip is 3100 x 2800 μm^2.

6. Conclusion

We have introduced two systolic arrays for solving dense linear systems over GF(p), p prime number. These two arrays implement elimination algorithms with partial pivoting, although the operation of the array remains purely systolic.

Moreover, there is no feedback cycles in our arrays. The importance of designing arrays without feedback cycles is emphasized in Kung and Lam [KLa]: acyclic implementations usually exhibit more favorable characteristics with respect to fault-tolerance, two-level pipelining, and problem decomposition (see Hwang and Cheng [HC]) in general.

Using massive parallelism and pipelining, the systolic array concept allows a system implementor to design extremely efficient machines for specific computations. The suitability of the systolic model to chip design is well illustrated by the prototype array that we have designed within only a few weeks.

References

[AMD] H.M. AHMED, J.M. DELOSME, M. MORF, Highly concurrent computing structures for matrix arithmetic and signal processing, IEEE Computer 15, 1, 65-82 (1982)

[AFQ] F. ANDRE, P. FRISON, P. QUINTON, Algorithmes systoliqes: de la théorie à la pratique, IRISA Research Report 214, May 1983

[ABD] C. ANTONINO, B. BOSC, H. DELORI, A. GUYOT, J.F. PAILLOTIN, The French MPC 84-85, Internal Report, TIM3/INPG Grenoble, February 1986

[Bla] G.R. BLAKLEY, A computer algorithm for computing the product AB modulo M, IEEE Trans. Comput. 32, 4 (1983), 497-500

[BBK] A. BOJANCZYK, R.P. BRENT, H.T. KUNG, Numerically stable solution of dense systems of linear equations using mesh-connected processors, Technical Report, Carnegie Mellon University, 1981

[CRT] M. COSNARD, Y. ROBERT, M. TCHUENTE, Matching parallel algorithms with architectures: a case study, IFIP Working Conference on Highly Parallel Computers, Nice, France, 24-26 March 1986

[GK] W.M. GENTLEMAN, H.T. KUNG, Matrix triangularisation by systolic arrays, Proc SPIE 298, Real-time Signal Processing IV, San Diego, California, 1981, 19-26

[Hel] D. HELLER, Partitioning big matrices for small systolic arrays, in VLSI and Modern Signal Processing, S. Y. Kung et al. eds, Prentice Hall, Englewood Cliffs, NJ (1985), 185-199

[HC] K. HWANG, Y.H. CHENG, Partitioned matrix algorithm for VLSI arithmetic systems, IEEE Trans. Computers 31, 12 (1982), 1215-1224

[Kun] H.T. KUNG, Why systolic architectures, IEEE Computer 15, 1 (1982), 37-46

[KLa] H.T. KUNG, M.S. LAM, Fault-tolerance and two-level pipelining in VLSI systolic arrays, J. of Parallel & Distributed Computing 1, 1 (1984), 32-63

[KLe] H.T. KUNG, C.E. LEISERSON, Systolic arrays for (VLSI), Proc. of the Symposium on Sparse Matrices Computations, I.S. Duff et al. eds, Knoxville, Tenn. (1978), 256-282

[Knu] D.E. KNUTH, The art of computer programming, vol. 2, chap. 4.6.2., Addison WWesley (1969)

[Mak] J. MAKHOUL, Linear prediction: a tutorial review, Proc. IEEE 63, 4 (1975), 561-580

[MB] M.A. MORRISON, J. BRILLHART, A method of factoring and the factorization of F_7, Math. of Comput. 29, 129 (1975), 183-205

[PW] D. PARKINSON, M. WUNDERLICH, A compact algorithm for Gaussian elimination over GF(2) implemented on highly parallel computers, Parallel Computing 1 (1984), 65-73

[Pai] J.F. PAILLOTIN, Le système LUCIE, Internal Report, TIM3/INPG Grenoble, July 1985

[Poe] R. POET, The design ofspecial purpose hardware to factor large integers, Comput. Physics Communications 37 (1985), 337-341

[RTc] Y. ROBERT, M. TCHUENTE, Résolution systolique de systèmes linéaires denses, RAIRO Modélisation et Analyse Numérique 19, 2 (1985), 315-326

[RTr] Y. ROBERT, D. TRYSTRAM, Un réseau systolique orthogonal pour le problème du chemin algébrique, C.R.A.S. Paris, 302, I, 6 (1986), 241-244

[SK] R. SCHREIBER, P. KUEKES, Systolic linear algebra machines in digital signal processing, in VLSI and Modern Signal Processing, S. Y. Kung et al. eds, Prentice Hall, Englewood Cliffs, NJ, 1985

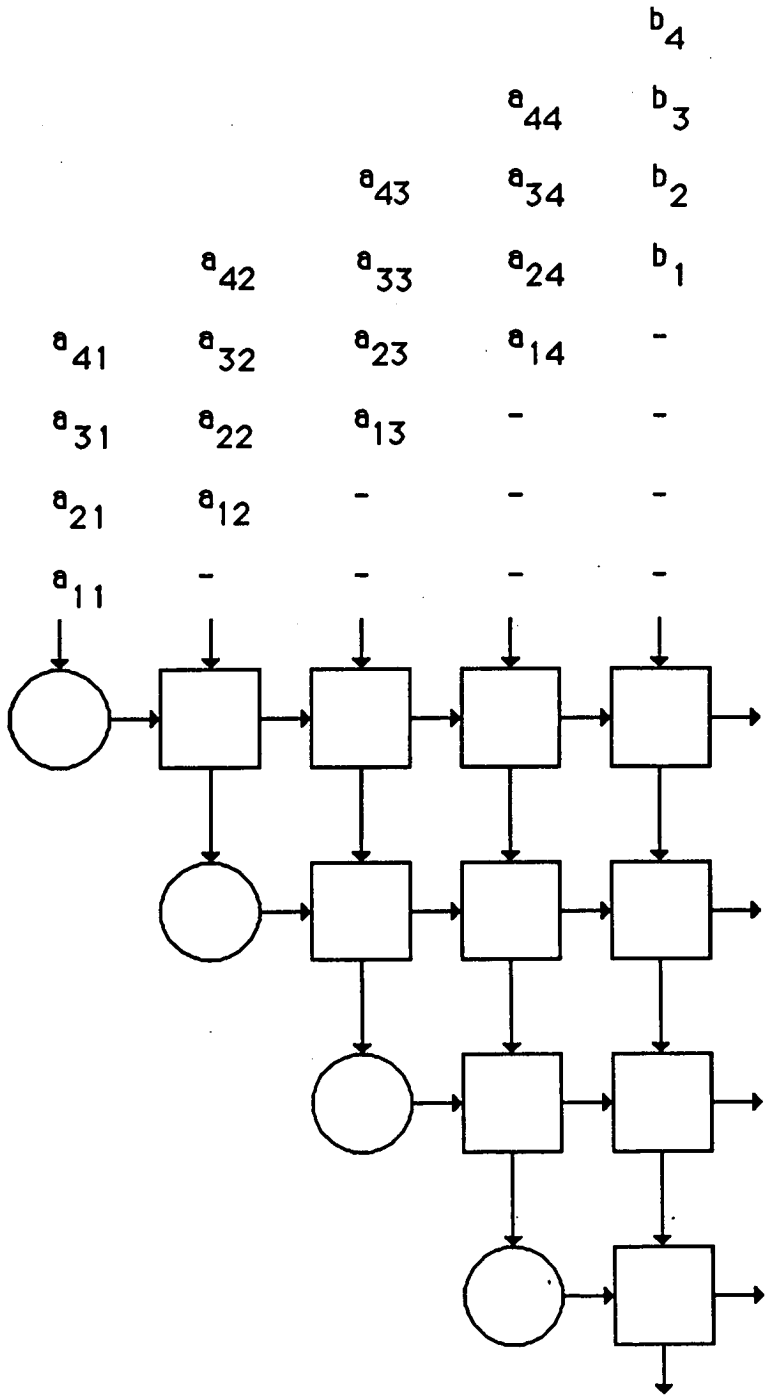
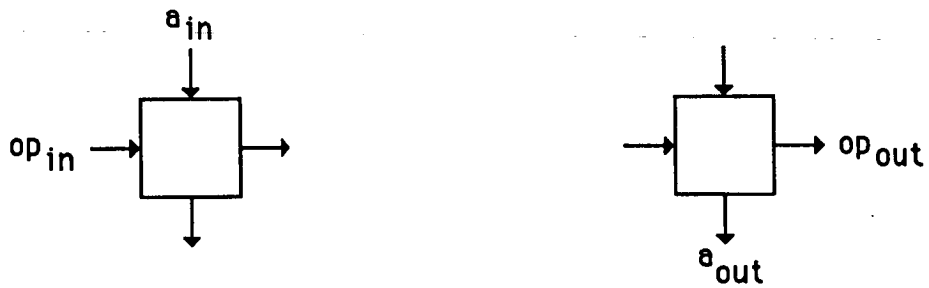


Figure 1 : The systolic array for Gaussian elimination (n=4)

Step t Step t+1

```
if init = true
then { store a_in }
        begin r := a_in ; init := false end
else { generate instruction }
        begin
        case (a_in,r) of
                (0,0) : op_out := id { identity - still looking for pivot } ;
                (0,1) : op_out := id { element already zero } ;
                (1,0) : op_out := perm { exchange rows } ;
                (1,1) : op_out := add { XOR operation } ;
        end
```

Circular cell



Step t Step t+1

```
if init = true
then { store a_in }
        begin r := a_in ; init := false end
else { update a_in (and r if a permutation is required) }
        begin
        case op_in of
                id :    a_out := a_in { r is not modified } ;
                add :   a_out := a_in XOR r { r is not modified } ;
                perm : a_out := r ; r := a_in { exchanging a_in and r } ;
        op_out := op_in ;
        end
```

Square cell

Gaussian elimination over GF(2) - Operation of the processors
Figure 2

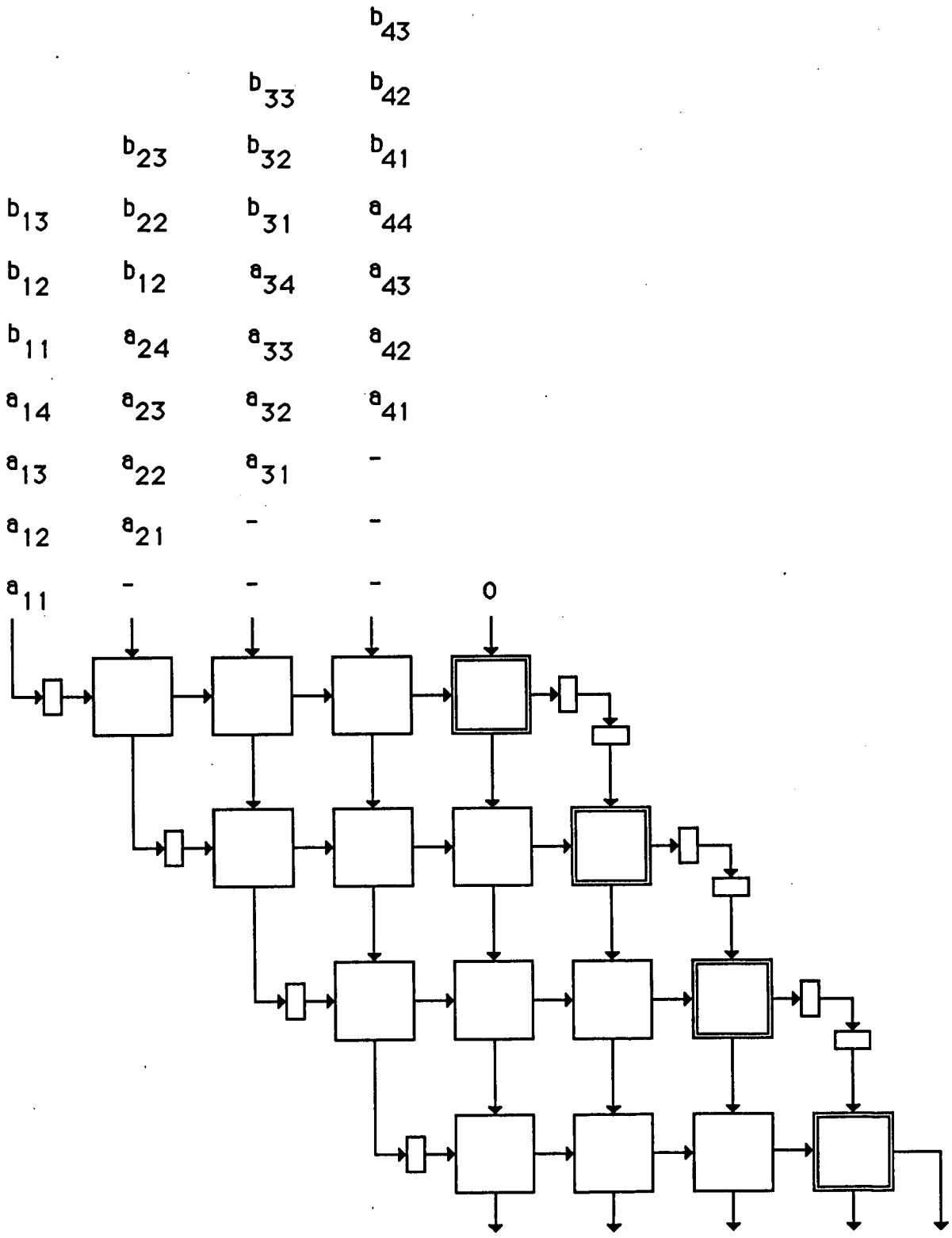


Figure 3 : The systolic array for computing $A^{-1}B$ (n=4, q=3)

Step t Step t+1

{ one-step delay }
$a_{out} := a_{in}$

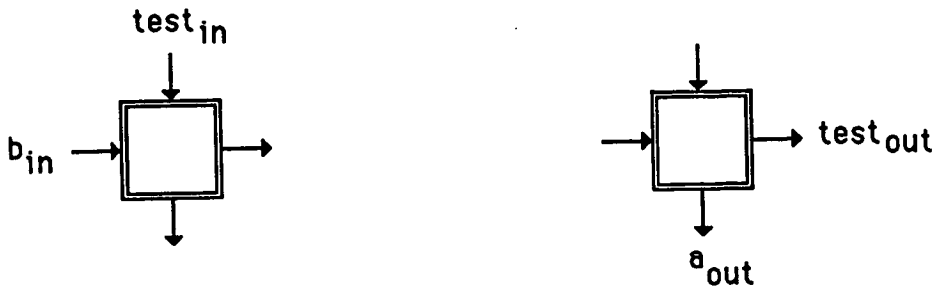
Delay cell

Step t Step t+1

if init = true
then { generate instruction and store it in internal register }
  begin
    init := false ;
    case (a_{in}, b_{in}) **of**
      (0,0) : op := id ; $b_{out} := b_{in}$ { identity - still looking for pivot } ;
      (0,1) : op := id ; $b_{out} := b_{in}$ { element already zero } ;
      (1,0) : op := perm ; $b_{out} := a_{in}$ {exchange rows } ;
      (1,1) : op := add ; $b_{out} := b_{in}$ { XOR operation } ;
  end
else { update a_{in} (and b_{in} if a permutation is required) }
  begin
    case op **of**
      id : $a_{out} := a_{in}$; $b_{out} := b_{in}$;
      add : $a_{out} := a_{in}$ XOR b_{in} ; $b_{out} := b_{in}$;
      perm : $a_{out} := b_{in}$; $b_{out} := a_{in}$;

Square cell

Gauss-Jordan diagonalization over GF(2) - Operation of the processors
Figure 4

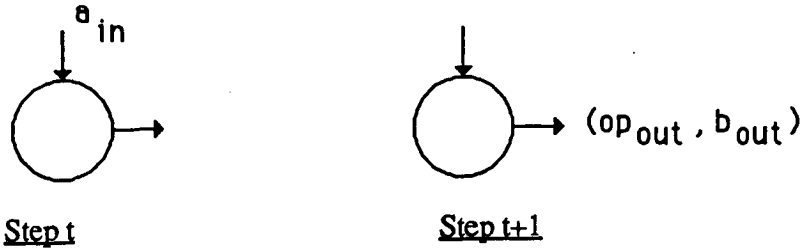


Step t Step t+1

```
if init = true
then
{ test for non singularity: b_in=0 means that all elements in the column are zero }
        begin
                init := false ;
                test_out := test_in OR not(b_in) ;
                { test_out=1 means that the matrix is already known to be singular }
        end
else
{ transmit b_in }
        a_out := b_in
```

Double square cell

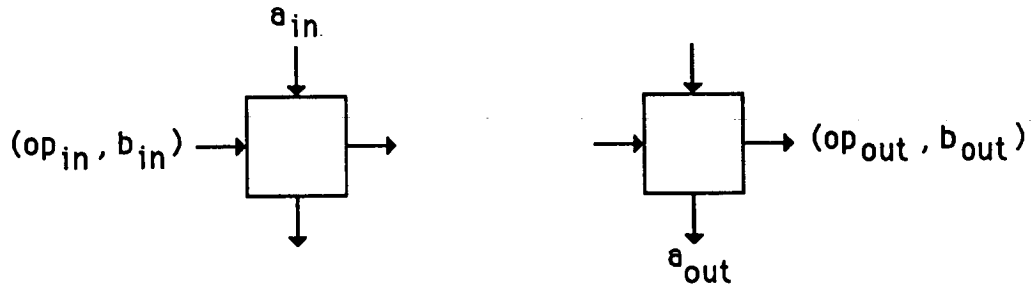
Gauss-Jordan diagonalization over GF(2) - Operation of the processors
Figure 4 (continued)



Step t Step t+1

if init = true
then { store a_{in} and prepare pivot if not zero }
begin **if** $a_{in} = 0$ **then** r := 0 **else** r := $1/a_{in}$; init := false **end**
else { generate instruction }
begin
if $a_{in} = 0$ **then** op_{out} := id { b_{out} is not significant }
else { $a_{in} \neq 0$ }
if r = 0
then begin op_{out} := perm ; r := $1/a_{in}$ { exchange rows - prepare pivot } **end**
else begin op_{out} := comb ; b_{out} := - a_{in} * r { combination of rows } **end**
end

Circular cell



Step t Step t+1

if init = true
then { store a_{in} }
begin r := a_{in} ; init := false **end**
else { update a_{in} (and r if a permutation is required) }
begin
case op_{in} **of**
id : a_{out} := a_{in} { r si not modified } ;
comb : a_{out} := a_{in} + r * b_{in} mod p { r si not modified } ;
perm : a_{out} := r ; r := a_{in} { exchanging a_{in} and r } ;
op_{out} := op_{in} ; b_{out} := b_{in} ;
end

Square cell

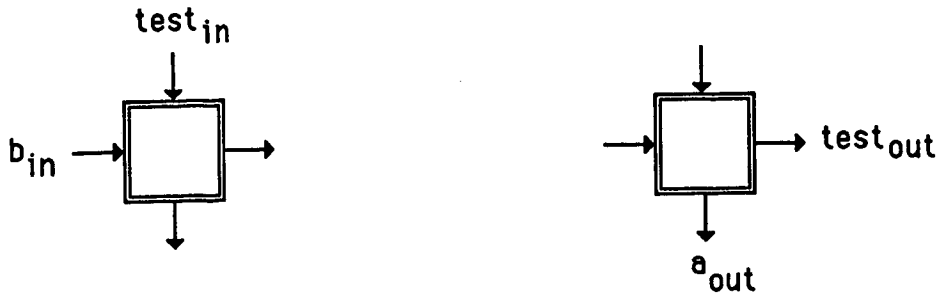
Gaussian elimination over GF(p) - Operation of the processors
Figure 5

Step t Step t+1

```
if init = true
then { generate instruction and store it in internal register - prepare pivot if needed }
        begin
                init := false ;
                if a_in = 0 then begin op := id ; b_out := b_in ; end
                else { a_in≠0 }
                begin
                        if b_in=0 then  { exchange rows }
                                begin op := perm ; b_out := a_in ; end
                        else { combination of rows required }
                                begin
                                        op := comb ;
                                        r := - a_in / b_in mod p ;
                                        b_out := b_in ;
                                end
                end
        end
else { update a_in (and b_in if a permutation is required) }
        begin
        case op of
                id : a_out := a_in ; b_out := b_in ;
                comb : a_out := a_in + r * b_in mod p ; b_out := b_in ;
                perm : a_out := b_in ; b_out := a_in ;
        end
```

Square cell

Gauss-Jordan diagonalization over GF(p) - Operation of the processors
Figure 6



Step t Step t+1

```
if init = true
then { test for non singularity }
     begin
          init := false ;
          if b_in=0 then { singularity: all elements in the column are zero }
          begin
               test_out := true ;
               r := nil { the value of r has no importance }
          else
               test_out := test_in ;
               r := 1 / b_in ;
          end
          { test_out=1 means that the matrix is rank deficient }
     end
else { transmit b_in }
     a_out := b_in * r
```

Double square cell

Gauss-Jordan diagonalization over GF(p) - Operation of the processors
Figure 6 (continued)

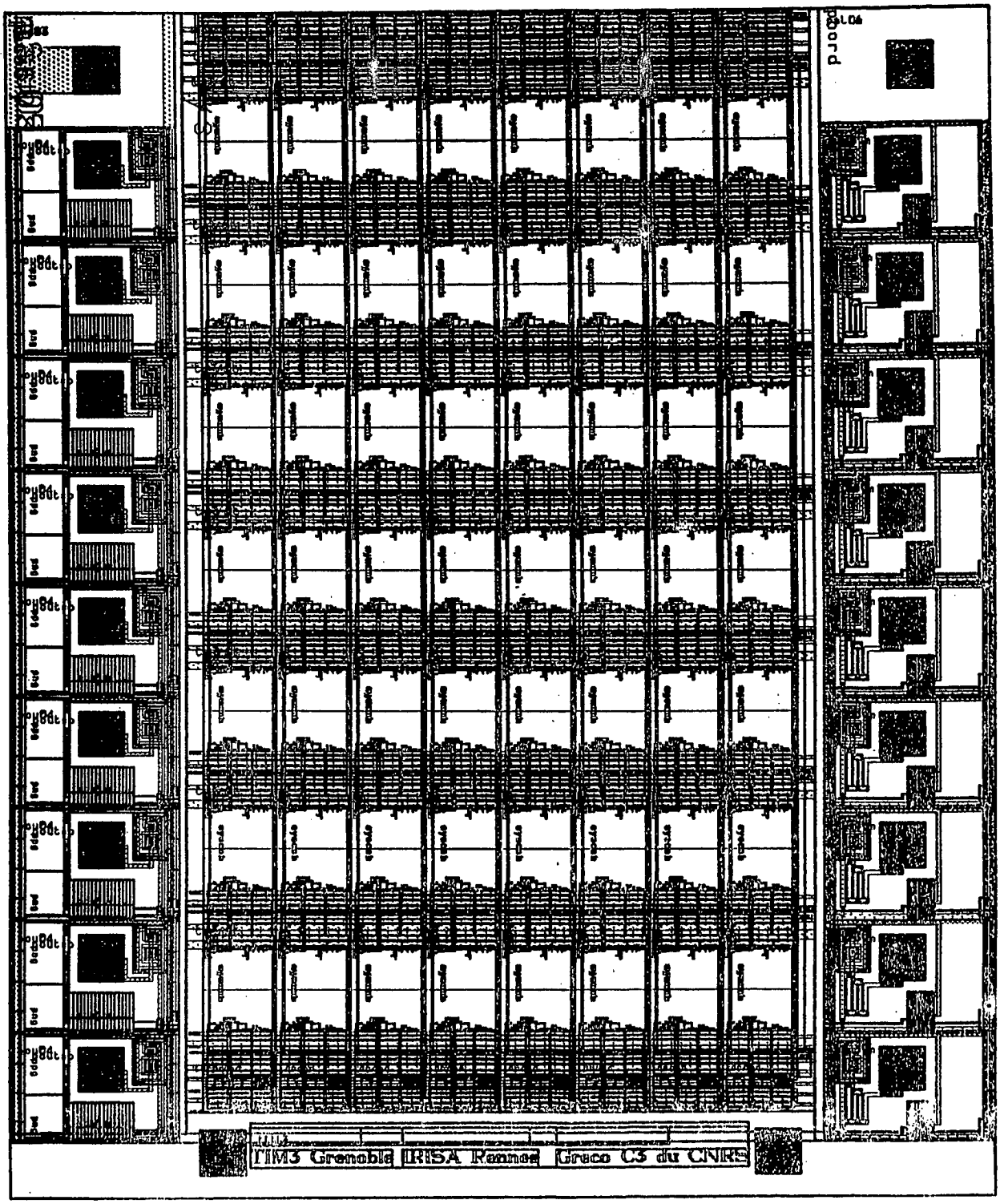


Figure 7 : Layout of the 8 by 8 array