1986

A Unifying Framework for Systolic Designs

Concettina Guerra

Report Number:
86-595

A UNIFYING FRAMEWORK FOR SYSTOLIC DESIGN

Concettina Guerra

CSD-TR-595
April 1986

A UNIFYING FRAMEWORK FOR SYSTOLIC DESIGNS

Concettina Guerra
Department of Computer Sciences, Purdue University
West lafayette, Indiana 47907

ABSTRACT

A systematic methodology to synthesize systolic designs is described and used to derive a new design for dynamic programming. This latter design uses fewer processing elements than previously considered ones. The synthesis method consists of two parts: 1) deriving from the high-level problem specification a form more suitable to VLSI implementation; 2) mapping the new specification into physical hardware. The method also provides a unifying framework for existing systolic algorithms.

1. INTRODUCTION

In recent years, a large number of VLSI algorithms have been designed for such problems as matrix multiplication, linear systems, convolution, etc. The mapping of an algorithm into an array of processing elements has been usually done in an ad-hoc manner. This is a difficult and error-prone task, especially for complex nonnumerical problems. A synthesis procedure can save a considerable amount of design effort and, at the same time, guarantee the correctness of the result. Recently, there have been various attempts to formalize the procedure for mapping algorithms into physical hardware. Their appraisal is in [2].

Among all the existing approaches, the transformational approach based on data dependencies proved to be very powerful [1, 7-13]. However, such an approach can be applied only to a restricted class of algorithms, namely, algorithms with highly regular communication patterns. Such algorithms are generally expressed by a uniform recurrence relation, or a nested loop with constant data dependencies. Many algorithms involve non-uniform recurrences or many nested loops with additional statements between the loops. These algorithms are sometimes implemented on VLSI networks with variable-speed data flow. An example is dynamic programming as applied to the optimal parenthesization problem.

In [5], a systolic algorithm for dynamic programming was proposed which can be cast in a triangular array of processing elements. The systolic design is quite complex: the data flow is non constant throughout the array and the action of a processing element varies at different clock cycles. Attempts have been made to synthesize non-uniform designs, concentrating on various aspects of the design process [1,4,9]. The crucial step towards the automatization of the design is in the ability to rewrite the algorithm in a form better suited to VLSI implementation. The synthesis procedure described here attempts to solve this problem. It operates both at the algorithmic level and implementation level. Thus, it consists of two parts: 1) transforming the given problem

specification to adapt it to VLSI implementation 2) mapping the new specification of the problem into a systolic array. The high-level problem specification is rewritten as a system of mutually dependent recurrence relations. This is accomplished by a two-step procedure . The procedure first determines a coarse timing function for the computations in the high-level specification and then uses it to identify chains of linearly dependent computations. Each such chain can be converted into a recurrence with constant data dependencies. Statements between recurrences are introduced to correlate variables in distinct recurrences. Next the mapping of the new form into VLSI hardware is obtained by applying linear time and space transformations to the individual recurrences, subject to global constraints. Such transformations are again based on data dependencies.

Using this procedure we transform the dynamic programming algorithm into a system of two dependent recurrence relations. In addition to rederiving the design presented in [5], we are able to automatically generate another design which obtains a better utilization of the processing elements and therefore requires fewer cells in the array; precisely, $3/8n^2$ instead of $n^2/2$.

This paper is organized as follows. In section 1 the synthesis method is illustrated for the derivation of a systolic algorithm for dynamic programming. Next, time and space transformations to map the new form into hardware are described in sections 2. The new design for dynamic programming is presented in section 3. It is also shown that the application of the proposed methodology to obtain other systolic designs such as convolution and palindrome recognition exploits interesting similarities among such problems.

1. DERIVATION OF A SYSTOLIC ALGORITHM FOR DYNAMIC PROGRAMMING

The dynamic programming can be expressed by a recurrence of the form:

$$1 \le i \le n;\ i < j \le n$$

$$c_{i,j} = \min_{i<k<j} f(c_{i,k}, c_{k,j}) \tag{1}$$

with initial conditions:

$$c_{i,i+1} = c_i \quad 1 \le i \le n$$

for some function f. The recurrence above defines an index set $I^3=\{(i,j,k) \mid 1 \le i < j \le n; i<k<j\}$ for all the computations and also a set $I^2=\{(i,j) \mid 1 \le i < j \le n\}$ of index pairs associated to variable c. Variable c appearing on both sides of (1) introduces non-constant data dependencies. A dependence vector is defined as the difference of the index vectors of the variable on the left and right side of any assignment statement [8]. Data dependence vectors in (1) are different for different vectors of the index set I^2. Dependencies relative to the vector (i,j) can be represented by the columns of the following dependence matrix:

$$D_{(i,j)} = \begin{vmatrix} 0 & i-k \\ j-k & 0 \end{vmatrix}$$

which can be expanded in the matrix below, where each column corresponds to a different value of the index k.

$$D_{(i,j)} = \begin{vmatrix} 0 & \cdots & 0 & 0 & -1 & -2 & \cdots & i-j+1 \\ j-i-1 & \cdots & 2 & 1 & 0 & 0 & \cdots & 0 \end{vmatrix}$$

Our aim is to transform expression (1) into a new form which consists of possibly many recurrences each characterized by constant data dependencies; non-constant data dependencies may only occur at the boundaries between the recurrences. For each such recurrence it will be possible to determine a linear time-space transformation into a systolic array, by applying the transformational method described in [11, 13]. We briefly review the tranformational method for uniform recurrences.

Mapping a uniform recurrence into a VLSI array

Consider a recurrence with index set I^n defined by:

$$c(\bar{i}) = f\ (c(\bar{i}-\bar{d}_1),\ldots,c(\bar{i}-\bar{d}_s))$$

where f is a given function and $\bar{i}$ is a vector of I^n. The recurrence is said to be uniform if the data dependence vectors $\bar{d}_1, \ldots, \bar{d}_s$ are constant. Let $D=[\bar{d}_1 \cdots \bar{d}_s]$. It is possible, under certain conditions, to automatically map a uniform recurrence into a VLSI array.

We consider the following simplified model of a VLSI array. Each processor of the array is assigned a label $l \in L^{n-1} \subset Z^{n-1}$. The connection pattern of the array is described by the matrix $\Delta=[\delta_1,\delta_2,\ldots,\delta_s]$ which specifies the links among the processors. Precisely, δ_i is the difference vector of the integer labels of adjacent cells in the network.

A linear time-space transformation of the indexed computations into the VLSI array is defined by:

$$\Pi = \begin{bmatrix} T \\ S \end{bmatrix}$$

where T is a mapping form $I^n \longrightarrow Z$ and S is a mapping from $I^n \longrightarrow L^{n-1}$. The time function T transforms D into TD. Thus, to ensure a correct execution ordering, T must satisfy the following condition:

$$T(\bar{d}_i) > 0 \quad \text{for } each\ \bar{d}_i \in D \tag{2}$$

System (2) may have no solution or several solutions. In this latter case, the one which minimizes the total execution time (defined as the difference between the maximum and minimum value of T) is chosen.

The space mapping S is a mapping from the set of computations to the set of processing elements

$$S : I^n \rightarrow L^{n-1}$$

such that for each $\bar{i},\bar{j} \in I^n$

$$S(\bar{i})=S(\bar{j}) \quad implies \quad T(\bar{i})\neq T(\bar{j}) \tag{3}$$

i.e. concurrent computations cannot be mapped into the same processor. Determining a solution for S which satisfies constraint (3) is equivalent to solve diophantine equations:

$$SD = \Delta K \tag{4}$$

for which the matrix $\begin{bmatrix} T \\ S \end{bmatrix}$ is non-singular. K is an integer matrix with positive elements. The equations may have no solution or several solutions. If no feasible solution is found, the design procedure is repeated by starting with a different timing function or else a different interconnection network. If several solutions are possible, the one which is optimal according to some criterion is chosen.

A two-step procedure for dynamic programming

To adapt expression (1) to systolic implementation, we first eliminate broadcasting by 1) adding missing indeces to variables on both sides of the expression, 2) renaming variables, 3) introducing new variables. However, as is well known, there are many ways to perform such transformations some of which may not lead to any feasible systolic design. The selection of a good transformation is crucial to the entire design process; for complex problems such as dynamic programming this is not a straightforward task. In order to add the index k to variable $c_{i,j}$ on the left side of (1), we need to specify an appropriate ordering for the computations $c_{i,j,k}$ for given i, j and for $i < k < j$, in such a way to introduce as much parallelism as possible. If we choose, say, the lexicographical ordering relative to index k, we cannot overlap computations of $c_{i,j,k}$ for different values of

Our strategy to select the appropriate transformation consists of identifying among the computations indexed by the set $J^3 = \{(i, j, k) \mid i, j \text{ are given and } i < k < j\}$ chains of linearly dependent computations, i.e. computations which have to performed in a certain order. To accomplish that, we first determine a coarse timing function for the computations indexed by I^2. The transformational method described above applies only when constant data dependencies are present. Thus we extract from recurrence (1) a subset of constant data dependencies and derive a linear time transformation T based only on such subset. It obvious that if τ is an actual timing function for I^2 then it must be $\tau(i,j) \geq T(i,j)$ for each $(i,j) \in I^2$. The set $D_{i,j}$ contains non-constant dependencies; however, the intersection of $D_{(i,j)}$ for all the index vectors (i,j) contains only dependencies which are valid in any point of I^2. Let us denote such intersection by D.

$$D = \begin{vmatrix} 0 & -1 \\ 1 & 0 \end{vmatrix}$$

A linear time transformation $T : I^2 \rightarrow Z$ compatible with the subset D can be obtained as described above. T must satisfy (2), that is

$$T(\bar{d}) > 0 \quad \text{for any } \bar{d} \in D$$

Thus the coefficients of $T = [T_1\ T_2]$ must satisfy

$$T_1 > 0 \quad \text{and} \quad T_2 \leq -1$$

The least integer values that satisfy the above equations are:

$$T_1 = 0 \quad \text{and} \quad T_2 = -1.$$

Thus, the optimal time transformation is:

$$T(i,j) = j - i.$$

The function T will guide the search for a schedule of computations indexed by J^3 according to the availability of the variables $c_{i,k}$ and $c_{k,j}$ on the right side of (1). Because of the monotonocity of data dependencies in $D_{(i,j)}$, it must be $\tau(i,j) > \tau(i',j')$ if $T(i,j) > T(i',j')$. According to function T, we introduce a partial ordering $>_T$ in J^3 defined by:

$$(i,j,k') >_T (i,j,k'') \Longleftrightarrow Max\{T(i,k'), T(k',j)\} > Max\{T(i,k''), T(k'',j)\}$$

Notice that the minimal elements with respect to $>_T$ are:

$(i, j, (i+j)/2)$ if $i+j$ is even or

$(i, j, (i+j-1)/2)$ and $(i, j, (i+j+1)/2)$ if $i+j$ is odd.

A partial ordering produces a decomposition of the set into chains of linearly ordered computations. Among all the possible chain decompositions of J^3, we select the one in which the index vectors of a chain are also sorted (either in increasing or decreasing order) according to the third index k.

To obtain such a decomposition we repeatedly find minimal elements after removing the previous minimal elements from the ordered set. For the set J^3 we obtain a decomposition in two chains (here we only write the third component of the index vectors):

{if (i+j) is even}

$(i+j)/2, (i+j)/2-1, \ldots, i+1$; and $(i+j)/2+1, (i+j)/2+2, \ldots, j-1$.

{if (i+j) is odd}

$(i+j-1)/2, (i+j-1)/2-1, \ldots, i+1$; and $(i+j+1)/2, (i+j+1)/2+1, \ldots, j-1$;

We are now able to restructure (1) into a system of two recurrences or modules, each corresponding to a chain. The execution ordering of computations in each recurrence is specified according to the ordering in the chain. Thus, the first recurrence is a forward recurrence where the index k varies from $(i+j)/2$ to $i+1$ (or from $(i+j-1)/2$ to $i+1$ if $i+j$ is odd); and the second is a backward recurrence where k varies from $(i+j)/2$ to $j-1$ (or from $(i+j+1)/2$ to $j-1$ if $i+j$ is odd) . The two recurrences have different sets of variables; boundary conditions relate variables in the two recurrences. Now equations (1) can be converted into the following form.

for i:=1 to $n-1$ do $a''_{i,i+1,i+1} := c_{i,i+1}$; $c_{i,i+1,i+1} := c_{i,i+1}$;
for l:= 2 to $n-1$ do
for i:= 1 to n do begin
  j:= $i+l$;
  if ($i+j$)=*even*) then begin
    k:=($i+j$)/2;
A1: $a'_{i,j,k} = a''_{i,j-1,k}$;
A2: if k= $i+1$ then $b'_{i,j,k} := c_{i+1,j,j}$ else $b'_{i,j,k} := b'_{i+1,j,k}$;
    $c'_{i,j,k} := f(a'_{i,j,k}, b'_{i,j,k})$; $c''_{i,j,k} := c'_{i,j,k}$
    end
  else begin {i+j=odd};
    k:=($i+j-1$)/2;
    $a'_{i,j,k} := a'_{i,j-1,k}$;
    if k= $i+1$ then $b'_{i,j,k} := c_{i+1,j,j}$ else $b'_{i,j,k} := b'_{i+1,j,k}$;
    $c'_{i,j,k} := f(a'_{i,j,k}, b'_{i,j,k})$
    k:= ($i+j+1$)/2;
A3: if k:= $j-1$ then $a''_{i,j,k} := c_{i,j-1,j-1}$ else $a''_{i,j,k} := a''_{i,j-1,k}$;
A4: $b''_{i,j,k} := b'_{i+1,j,k}$;
    $c''_{i,j,k} := f(a''_{i,j,k}, b''_{i,j,k})$;
    end

for k:= $\lceil (i+j-1)/2 - 1 \rceil$ downto $i+1$ do begin
  $a'_{i,j,k} := a'_{i,j-1,k}$;
  if k= $i+1$ then $b'_{i,j,i+1} := c_{i+1,j,j}$ else $b'_{i,j,k} := b'_{i+1,j,k}$;
  $c'_{i,j,k} := h(c'_{i,j,k+1}, f(a'_{i,j,k}, b'_{i,j,k}))$;
  end;

for k:= $\lfloor (i+j+1)/2 + 1 \rfloor$ to $j-1$ do begin
  if k= $j-1$ then $a''_{i,j,k} := c_{i,j-1,j-1}$ else $a''_{i,j,k} := a''_{i,j-1,k}$;
  $b''_{i,j,k} := b''_{i+1,j,k}$;
  $c''_{i,j,k} := h(c''_{i,j,k-1}, f(a''_{i,j,k}, b''_{i,j,k}))$:
  end;

module 1

module 2

A5: $c_{i,j,j} := h(c'_{i,j,i+1}, c''_{i,j,j-1})$;
end;

The two-step mapping procedure can be generalized to any recurrence with index set I^n $I^n=\{(i_1,\ldots,i_n) \mid l_1^1 \le i_1 \le l_1^2,\ldots,l_n^1 \le i_n \le l_n^2\}$ of the form:

$$c(\bar{i}^s) = f(c(\bar{i}^s - \bar{d}_1^s), \ldots, c(\bar{i}^s - \bar{d}_m^s))$$

where:

$s = n-1$;

$\bar{i}^s=(i_1, \ldots, i_s)$ is an s-tuple of indeces of the loop;

$\bar{d}_j^s=(a_1, \ldots, a_{l-1}, i_l - i_n, a_{l+1}, \ldots, a_s)$, $j=1,\ldots,m$;

and where a_l, $1 \le l \le t-1$, $t+1 \le l \le s$, are integer constants and i_n is the loop index missing on the left side of the statement, i.e. $i_n \neq i_l$ for each $l=1,\ldots,s$. Each vector $\bar{d}_j^s$, $(j=1,\ldots,m)$, represents a non constant data dependence for variable c, since its t-th component is a function of the two indeces i_t and i_n.

2. MAPPING THE ALGORITHM INTO HARDWARE

Timing function

Given the new specification of dynamic programming, we extract from each module distinct sets D_1 and D_2 of constant data dependencies.

$$D_1 = \begin{array}{c} \begin{matrix} c' & a' & b' \end{matrix} \\ \begin{vmatrix} 0 & 0 & -1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{vmatrix} \end{array} \qquad D_2 = \begin{array}{c} \begin{matrix} c'' & a'' & b'' \end{matrix} \\ \begin{vmatrix} 0 & 0 & -1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{vmatrix} \end{array}$$

Dependencies between variables in distinct modules, referred to as global dependencies, are defined by statements A1 to A5 in the algorithm.

We seek a linear time transformation for each separate set of local data dependencies which satisfies global constraints. Let $\lambda = [\lambda_1\ \lambda_2\ \lambda_3]$ and $\sigma = [\sigma_1\ \sigma_2\ \sigma_3]$ denote the linear time transformations for module 1 and 2, respectively. Furthermore, let $\tau = [\tau_1\ \tau_2\ \tau_3]$ be the timing function for the computation in A5. Such transformations must satisfy condition (2), that is

$$\lambda \bar{d} > 0 \quad \text{for } \bar{d} \in D_1; \qquad \sigma \bar{d} > 0 \quad \text{for } \bar{d} \in D_2;$$

from which we obtain the system of equations:

$$\lambda_1 \le -1 \qquad \lambda_2 \ge 1 \qquad \lambda_3 \le -1$$

$$\sigma_1 \le -1 \qquad \sigma_2 \ge 1 \qquad \sigma_3 \ge 1.$$

Global dependencies specified by A1-A5 lead to the additional equations:

$$\lambda(i, j, (i+j)/2) > \sigma(i, j-1, (i+j)/2)$$
$$\lambda(i, j, i+1) > \tau(i+1, j, j)$$
$$\sigma(i, j, j-1) > \tau(i, j-1, j-1)$$
$$\sigma(i, j, (i+j+1)/2) > \lambda(i+1, j, (i+j+1)/2)$$
$$\tau(i, j, j) \ge \max[\lambda(i, j, i+1), \sigma(i, j, j-1)]$$

It easy to check that an optimal solution to the above system, i.e. one which minimizes the execution time, is given by:

$$\lambda_1 = -1 \quad \lambda_2 = 2 \quad \lambda_3 = -1$$
$$\sigma_1 = -2 \quad \sigma_2 = 1 \quad \sigma_3 = 1$$
$$\tau_1 = -2 \quad \tau_2 = 1 \quad \tau_3 = 1.$$

Hence, we obtain the timing functions:

$$\lambda(i,j,k) = -i+2j-k$$
$$\sigma(i,j,k) = -2i+j+k$$
$$\tau(i,j,j) = -2i+2j.$$

Space function

The automatic procedure for determining the mapping of computations into the cells of a systolic array is analogous to the one for the timing function. Again, we look for separate solutions to the different modules in the algorithm subject to global constraints. We consider a 2-D array of processing elements modelled by the pair $[L^2,\Delta]$, where L^2 is the set of labels (x,y) assigned to processing elements and Δ is a matrix describing the interconnection network between processing elements. Different interconnection patterns may result in different classes of designs. In the following, we generate the optimal design when Δ is chosen to be:

$$\Delta = \begin{vmatrix} 0 & 1 & 0 \\ 0 & 0 & -1 \end{vmatrix}$$

Δ corresponds to a network with unidirectional links, as shown in fig. 1.

Let S', S'', and S be the space functions for module 1, module 2, and statement A5, respectively.

$$S' = \begin{vmatrix} S_{11}' & S_{12}' & S_{13}' \\ S_{21}' & S_{22}' & S_{23}' \end{vmatrix} \quad S'' = \begin{vmatrix} S_{11}'' & S_{12}'' & S_{13}'' \\ S_{21}'' & S_{22}'' & S_{23}'' \end{vmatrix} \quad S = \begin{vmatrix} S_{11} & S_{12} & S_{13} \\ S_{21} & S_{22} & S_{23} \end{vmatrix}$$

In addition to satisfy condition (4), the coefficients of S', S'', and S must satisfy the constraints imposed by global dependencies. Precisely, if a global dependence involves two variables belonging to different modules which are computed at times t and t' with $t-t'=d$ then the distance of the cells where the two variables will be mapped cannot be more than d. By distance we mean the length of a path consisting of interconnection links between the two cells. From A1 we have:

$$S' \,|i \;\; j \;\; (i+j)/2|' = S'' \,|i \;\; j-1 \;\; (i+j)/2\,|' + \bar{d}_1; \qquad \bar{d}_1 \in \Delta,$$

since $\lambda(i,j,(i+j)/2) - \sigma(i,j-1,(i+j)/2) = 1$ and, consequently, the two computations must occur either in the same cell or in adjacent cells. Similarly, from A2-A4 we obtain:

$$S' \,|i \;\; j \;\; i+1\,|' = S \;\; |i+1 \;\; j \;\; j\,|' + \bar{d}_2; \quad \bar{d}_2 \in \Delta;$$

$$S'' \,|i \;\; j \;\; j-1\,|' = S \;\; |i \;\; j-1 \;\; j-1|' + \bar{d}_3; \quad \bar{d}_3 \in \Delta;$$

$$S'' \,|i \;\; j \;\; (i+j+1)/2\,|' = S' \,|i+1 \;\; j \;\; (i+j+1)/2|' + \bar{d}_4; \quad \bar{d}_4 = \delta_i + \delta_j \;\; \delta_i, \delta_j \in \Delta;$$

$$S \;\; |i \;\; j \;\; j \;\; |' = S \;\; |i \;\; j \;\; i+1|' + \bar{d}_5; \quad \bar{d}_5 \in \Delta;$$

One solution to above system of equations is:

$$S'_{11} = S'_{13} = 0;\ S'_{12} = 1 \qquad S'_{22} = S'_{23} = 0;\ S'_{21} = 1$$

for the first recurrence and:

$$S''_{11} = S''_{13} = 0;\ S''_{12} = 1 \qquad S''_{22} = S''_{23} = 0;\ S''_{21} = 1$$

for the second recurrence. Thus

$$S'(i,j,k) = S''(i,j,k) = S(i,j,j) = (j,i).$$

The resulting design is identical to the one first introduced in [5]. The corresponding systolic array and the action of a cell at different times is depicted in figure 1.

3. A NEW DESIGN FOR DYNAMIC PROGRAMMING

Consider an array of processing elements whose communication pattern is described by:

$$\Delta = \begin{vmatrix} 0 & 1 & 0 & -1 & -1 \\ 0 & 0 & -1 & 0 & -1 \end{vmatrix}$$

Cells in the array are connected by bidirectional horizontal links as well as by diagonal and vertical links, as shown in fig. 2. An optimal design for dynamic programming is generated for such an array using the same mapping procedure. Again we solve equations (4) subject to global constraints. We derive:

$$S'_{11} = S'_{12} = 0; S'_{13} = 1 \qquad S'_{22} = S'_{23} = 0; S'_{21} = 1$$

for the first recurrence and:

$$S''_{11} = S''_{12} = 1; S''_{13} = -1 \qquad S''_{22} = S''_{23} = 0; S''_{21} = 1$$

for the second recurrence. Thus we have:

$$S'(i,j,k) = (k,i) \text{ and } S''(i,j,k) = (i+j-k,i).$$

These transformations lead to the systolic design of figure 2. The array consists of $3/8n^2$ cells. All cells are identical. However, the action of a cell varies from time to time. It does computation relative to module 1 or module 2 depending on the values of indeces i, j, and k. Also, the direction of data streams varies for the two modules. The transformation of data dependence vectors D_1 into communication vectors Δ is derived from S'. From:

$$\begin{vmatrix} S_{11}' & S_{12}' & S_{13}' \\ S_{21}' & S_{22}' & S_{23}' \end{vmatrix} \begin{matrix} c' \\ \\ \end{matrix} \begin{matrix} a' \\ \\ \end{matrix} \begin{matrix} b' \\ \\ \end{matrix} \begin{vmatrix} 0 & 0 & -1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{vmatrix} = \begin{vmatrix} -1 & 0 & 0 \\ 0 & 0 & -1 \end{vmatrix}$$

we derive that variables $a'_{i,j,k}$ do not move along the array but stay inside the cells, where they are updated. Furthermore, variables $b'_{i,j,k}$ move up, except at the boundary, where they move along the diagonal. The direction of variables in module 2 is derived from the mapping S''. Variables $a''_{i,j,k}$ move to the right along the horizontal links. The other variables move with the same pattern as in the other module. The action of a cell at each time is illustrated in figure 2.

The new design is interesting not only because of the reduced size of the corresponding array, but also because of its similarity with the design of a systolic palindrome recognizer [6]. A linear array of processing elements with two-way pipelining was described in [6] for this latter problem. The array determines for each character in a string whether the string input up to that character is a palindrome. A string of $l+1$ characters $s_0,s_1,...,s_l$ is a *palindrome* if the k-th character is the same as the $(l-k)$-th character for $k = 0,1,...,l$. Let c_i denote a variable whose value is 1 if the string $s_0....s_i$ is a palindrome, 0 otherwise. Thus:

$$c_i = f(s_k, s_{l-k}) \quad 0 \leqslant k \leqslant l$$

for some function f.

The data flow of input variables s and output variables c along the linear array is the same as the data flow of variables c and a, respectively, along any row of the two-dimensional array of fig.2 for dynamic programming. Indeed, both problems have similar data dependence patterns, as it can be easily observed if we rewrite recurrence (1) in the following form:

$$1 \leq i \leq n;\ 1 \leq l \leq i$$

$$c_{i,l} = \min_{1<k<l} f(c_{i,k}, c_{k,l-k}) \tag{6}$$

In (6) index j in (1) has been replaced by the index $l=j-i$. From the above expression it is apparent that, as for the palindrome recognizer, variables with indeces l and $l-k$, $1<k<l$, have to collide in the same cell at the same time.

It is also easy to show that the automatic mapping procedure described here transforms the palindrome recognition problem into a form similar to the one for convolution and therefore applies the same time-space transformations to both.

REFERENCES

[1] Chen, M. "Synthesizing Systolic Designs " In Proc. Int. Symp. on VLSI Technology, Systems and Applications, Taipei-Taiwan, May 85.

[2] Fortes, J.A.B., Fu, K.S., and Wah, B.W. "Systematic Approaches to the Design of Algorithmically Specified Systolic Arrays", Tech. Rep., Purdue Univ.

[3] Fortes, J.A.B. and Moldovan, D.I. "Parallelism Detection and Algorithm Transformation Techniques Useful for VLSI Architecture Design" J. Parallel Distrib. Comput., May 1985.

[4] Guerra, C. and Melhem, R. " Synthesizing Non-uniform Systolic Designs" Tech. Rep., Dept. Comp. Sc., Purdue Univ. (submitted for publication)

[5] Guibas, L.J., Kung, H.T. and Thompson, C.D. "Direct VLSI Implementation of Combinatorial Algorithms " Proc. of Caltech Conf. on VLSI, 1979.

[6] Leiserson, C., Saxe, F. "Optimizing synchronous systems" VLSI Algorithms, 1984.

[7] Kung, H.T. and Lin, W. "An Algebra for VLSI Algorithm Design" Proc. Conf. on Elliptic Problem Solvers, 1983.

[8] Kunh, R.H. "Transforming Algorithms for Single-Stage and VLSI Architectures" Workshop on Interconnection Networks for Parallel and Distributed Processing, 1980.

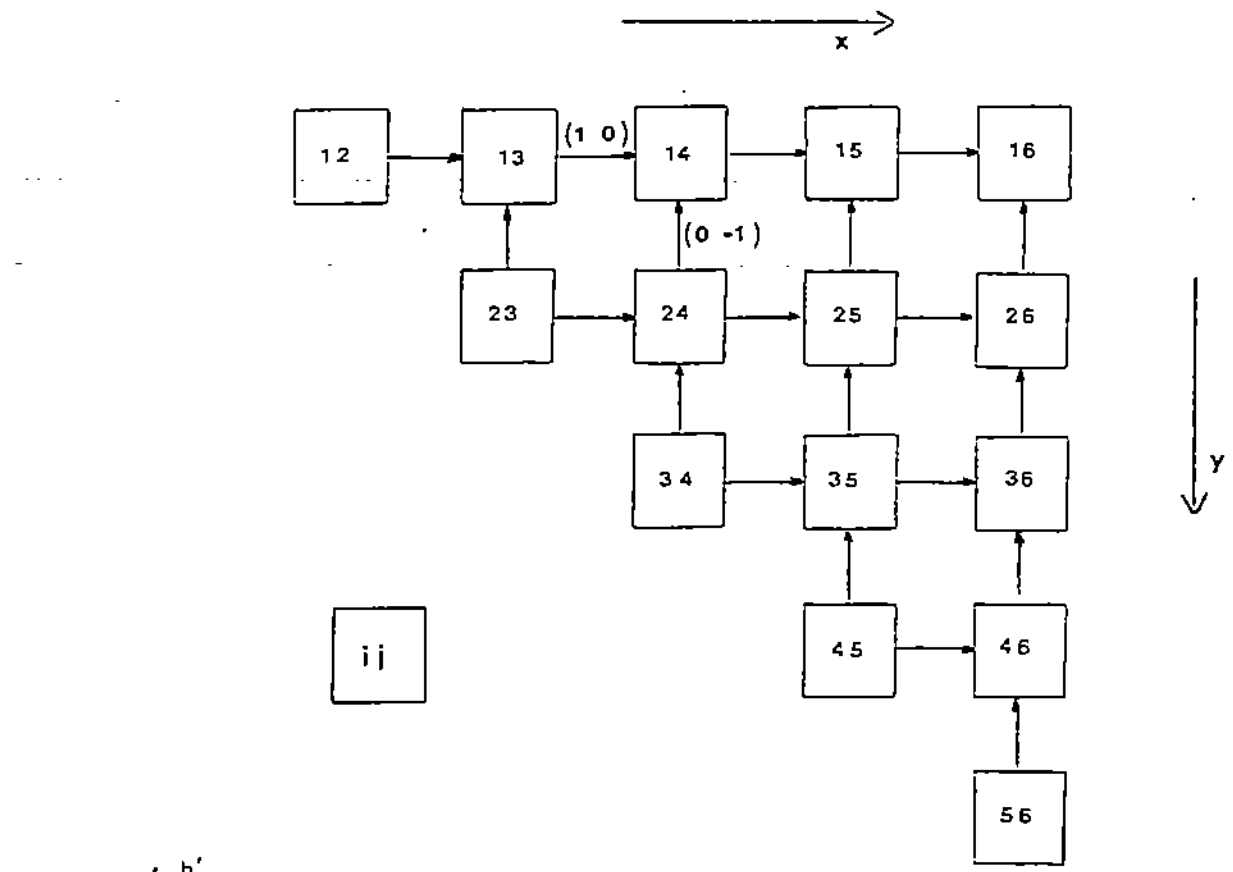
[9] Lam, M. and Mostow, J. "A Transformational Model of VLSI Systolic Design" Computer, pp. 42-52, 1985.

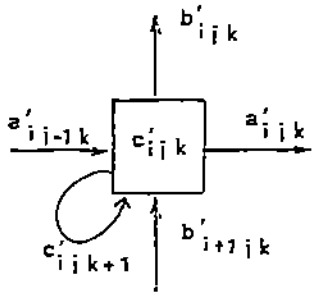
[10] Melhem, R. and Guerra, C. "The Application of a Sequence Notation to the Design of Systolic Computations" Techn. Rep. 568 Dept. Comp. Sc., Purdue University.

[11] Moldovan, D. "On the Analysis and Synthesis of VLSI Algorithms" IEEE Trans. on Computers, C-31, pp. 1121-1126, 1982.

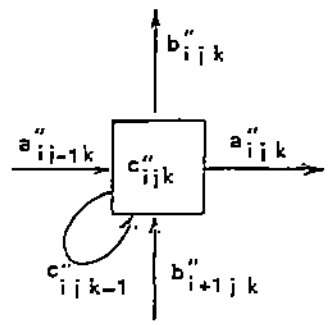
[12] Moldovan, D. "On the Design of Algorithms for VLSI Systolic Arrays" Proc. IEEE, vol. 71, pp. 113-120, Jan 1983.

[13] Quinton, P. "Automatic Synthesis of Systolic Arrays from Uniform Recurrent Equations " Proc. 11-th Annual Symp. on Computer Architecture, pp. 208-214, 1984.



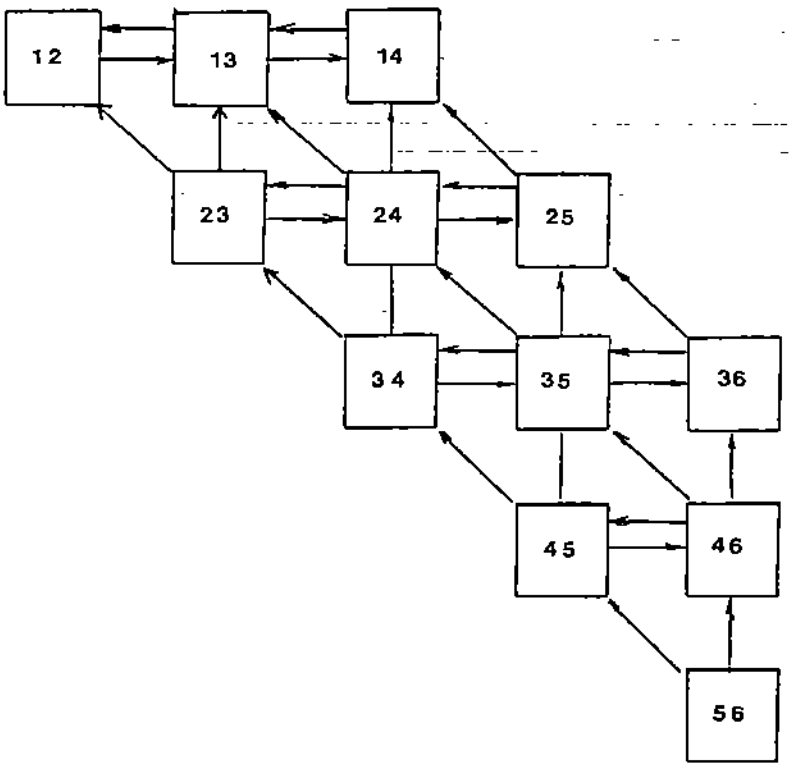


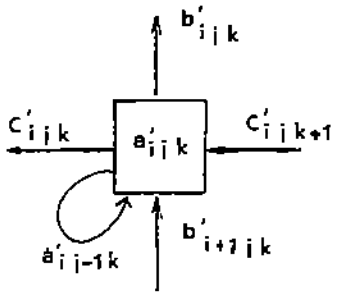
FOR $k < \lceil (i+j)/2 \rceil$



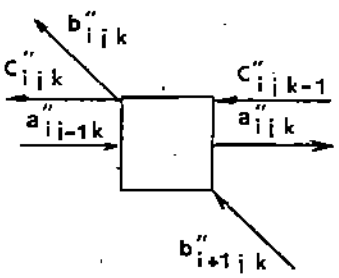
FOR $k > \lfloor (i+j-1)/2 \rfloor$

Fig. 1 A systolic network for dynamic programming





FOR $k < \lceil (i+j)/2 \rceil$



FOR $k > \lfloor (i+j-1)/2 \rfloor$

Fig. 2