論 文

Systolic Design with Asynchronous Controls for Digital-Signal Processings

Moon Seog Jun* *Regular Member*

디지탈 신호처리를 위한 비동기 제어 시스톨릭 설계

正會員 全 文 錫*

Abstract

In this paper[1], we present new techniques for designing systolic arraya and asynchronous arrays for digital-signal processings. More specifically, we propose a systolic array with simple local interconnections which achieves optimal performance without having undesirable features such as preloading input data or global broadcasting. As asynchronous array for digital-signal processings which can speed up the total computation time significantly is also which can speed up the total computation time significantly is also presented. The key component of the asynchronous array is a presented. The key component of the asynchronous array is a comunication protocol which controls input data flow properly and efficiently. Finally, performance of the arrays is analyzed and a simulation using Occam programmed in a Transputer network is reported.

Key words : digital-signal processings, systolic arrays, asynchronous arrays, VLSI.

要 約

본 논문에서 디지탈 신호처리를 위한 시스톨릭 배열과 비동기 제어 설계를 새로운 기법에 대하여 작성하였다. 특히, 배열안에 사전에 데이터를 입력하는 방법이나, 전체적인 제어와 같은 예전 방법과 다르게 최적의 성능을 성취할 수 있는 간단하고 내부적인 상호 연결을 갖는 시스톨릭 배열을 제안하고 있다. 이와 같은 디지탈 신호처리에 대한 비동기 시스톨릭 배열은 일정하게 설계된 배열에서 비교할 경우 전체 계산 시간을 상당히 줄일 수 있음을 비교를 통해서 다루었다. 비동기 시스톨릭 배열의 중요한 요소는 입력 데이터를 적절하고 효율적으로 제어할 수 있는 통신 프로토콜 설계이다. 마지막으로, Transputer의 병렬 컴퓨터 언어인 OCCAM를 이용하여 배열의 효율성을 비교 분석하고 시뮬레이션의 결과를 보였다.

I. Introduction

Until recently, computer-intensive tasks were handled by high performance supercomputers, including pipelined computers, array processors, and multiprocessor systems. However, the general-purpose nature of these machines results in a complicated system organization and heavy system operation overheads. It turns out that these

*崇實大學校 電算科
論文番號 : 93-43

machines are not suitable for real-time signal processing where a very high throughput rate is absolutely essential. A more promising solution to the real-time requirement of signal processing is to use special purpose VLSI architectures such as systolic arrays, which attain tremendously massive concurrency. In other words, one of the most suitable architectures for VLSI implementation is a systolic array. Its simple communication structure, the use of simple and uniform processing elements, and low I/O requirements are features that make it very attractive from the viewpoint of current technology. The architectures of systolic arrays depend on algorithms. The algorithms under consideration are recursive algorithms including matrix multiplication, FIR filtering, deconvolution, triangular matrix inversion, and discrete Fourier transformation.

In this paper, we propose a new idea for designing systolic arrays to implement recursive algorithms efficiently. More specifically, we design a systolic array to compute matrix multiplication in $2n$-1 time-units using n^2 processors for input matrices of size $n \times n$, no preloading or global broadcasting of input data assumed. For previous fast systolic designs of matrix multiplication, see [1],[4],[5],[11],[12],[16],[21], and [23]. In [11], L.Melkemi and M.Tchuente show that computing an $n \times n$ matrix product in $3n$-1 time-units is optimal for any orthogonally connected systolic array. They proposed a non-orthogonal systolic design in [12] which can compute a $n \times n$ matrix product in $2n$-1 time-units. Unfortunately, their design needs $n(2n\text{-}1)$ processors and cannot be fitted into a rectangle with a uniform pattern. Another $2n$-1 time systolic design in [16] requires preloading of one input matrix, and the pipelining feature will be blocked if there are multiple matrix multiplications. systematic approaches for design of systolic arrays have been studied by many researchers (see [9],[13],[14],[23], and [24]), and research on the subjects such as mapping recursive algorithms to arrays with varying interconnections still remains very active.

The main problem in a systolic approach is that, to assure proper timing and synchronization, extra delays are needed. this slows down the computation, thereby decreasing throughput rate. Moreover, for a large scale array, this synchronization would become very tedious. to overcome this problem, an asynchronous implementation which is a hybrid of systolic and data flow approaches is introduced. It has the advantage of eliminating or reducing waiting time by making the data streams independent of computations executed in each processor. For other designs of asynchronous arrays, see[2],[8],[18], and [20].

This paper is organized into five sections. In section 2, we introudce the idea of transformaition of a recursive algorithm into a locally recursive algorithm. Then, we describe a new technique for designing an interconnection scheme which implements a locally recursive algorithm of matrix multiplication efficiently. With the proposed interconnection scheme, we obtain a systolic algorithm which is the best among all known designs, i.e., minimizing (computing time)×(number of processors). Systolic computing for matrix multiplication of arbitrary size in a proposed array of fixed size is described in section 3. In section 4, an asynchronous design for matrix multiplication which can speed up total computation time significantly is proposed. The structure of processing elements (PEs) and its communication protocols, which control asynchronous computations, are presented. a performance analysis and a simulation using *transputer*[2] networks programmed in Occam are also given. Finally, in section 5, some concluding remarks are made.

Ⅱ. Systolic design for digital-signal processings

First, we define some basic terms that will be used for investigating recursive algorithms. Based on these terms, we will introduce a design

[2]Transputer and Occam are trademarks of the INMOS Group of Companies.

procedure which creates a systolic array and a systolic algorithm for implementing a recursive algorithm efficiently. We illustrate our technique by showing the process of designing a systolic array for matrix multiplication. It is very interesting to see whether the design technique can be applied to other recursive algorithms that share a similar structures.

2.1 Digital-signal processings

Before starting to describe the proposed technique, we introduce some notation that will be used through out this paper.

(Definition 1) A recursive algorithm is an algorithm presented by a set of recursive equations.

The derivation of recursive equations is often straightforward and can be found widely in the literature. We list some recursive equations which appear in many application areas.

(*a*) *Matrix Multiplication* :

$$\begin{aligned} & w_{i,j}^{0}=0 && 1\le i,j\le n \\ (2.1)\quad & w_{i,j}^{k}=w_{i,j}^{k-1}+a_{i,k}b_{k,j} && 1\le i,j,k\le n \end{aligned}$$

(*b*) *Finite Impulse Response (FIR) Filtering* :

$$\begin{aligned} & w_{i}^{0}=0 && 1\le i\le n \\ (2.2)\quad & w_{i}^{k}=w_{i}^{k-1}+a_{k}b_{i+j-1} && 1\le j\le n, \\ & && 1\le k\le m,\ b_j=0 \text{ for } j>n \end{aligned}$$

(*c*) *Deconvolution* : {This is the inverse of FIR filtering that solves for vector X, given vector Y and the toplitz matrix A. z_i^k are temporary variables.}

$$\begin{aligned} & z_{i}^{0}=y_i && 1\le i\le n \\ & z_{i}^{k}=z_{i}^{k-1}-a_{m-k+1}x_{i+m-k} && 1\le k\le m-1, \\ & && 1\le i\le n,\ x=0 \text{ for } j\ge n \\ (2.3)\quad & x_i=(z_{i}^{m-1}/a_1) && 1\le i\le n \end{aligned}$$

(*d*) *Triangular Matrix Inversion* : {Given a matrix U such that $u_{i,j}=0$ for $i>j$, the following equations compute $V=U^{-1}$. $w_{i,j}$ are temporary variables.}

$$\begin{aligned} & w_{i,j}^{j+1}=0 && 1\le i<j\le n \\ & w_{i,j}^{k}=w_{i,j}^{k+1}-u_{i,k}v_{k,j} && 1\le i<k\le j\le n \\ & v_{i,j}=0 && 1\le i<j\le n \\ (2.4)\quad & v_{i,i}=(1/u_{ii}) && 1\le i\le n, \end{aligned}$$

(*e*) *Discrete Fourier Transform (DFT)* : {$w=e^{2\pi\sqrt{-1}/n}$ is the nth root of unity.}

$$\begin{aligned} & y_{i}^{0}=0 && 1\le i\le n-1 \\ (2.5)\quad & y_{i}^{k}=y_{i}^{k-1}w^{i}+x_{n-k} && 1\le k\le n,\ 0\le i\le n-1. \end{aligned}$$

The above algorithms are defined over a finite index space. The activities of an algorithm consist of computing the values for a set of indexed variables over this index space. Each of these variables is similar to a multidimensional array, with a separate occurrence at each index point. The single value is computed for each occurrence of each indexed variable. The computation of any indexed variable in the algorithm may depend on the values of other indexed variables. The outputs of an algorithm can be any variables at the boundary of the index space. In the design of VLSI architecture, a need has been recognized for algorithm reindexing in order to avoid broadcasts. The goal is to obtain broadcast-free or broadcast-reduced schedules.

The design of systolic arrays to implement the above recursive algorithms efficiently is a difficult but extremely important problem. In this paper we have explored the process of design for matrix multiplication as described in the following section.

Some more notation that will be used through out this paper are defined below :

(Definition 2) A data dependency is a dependency that dictates the sequence of computation.

(Definition 3) A dependency graph is a graph that shows the dependency of the computations that occur in an algorithm. An algorithm is computable if and only if its dependency graph

contains no loops or cycles.

(Definition 4) A localized dependency graph is a dependency graph which has only local dependencies, i.e., the length of each dependency are is independent of the problem size.

(Definition 5) A computational graph is a localized dependency graph with each node in the graph labeled by the indices of the terms it computes.

(Definition 6) A projective graph is a computational graph after projection along a specific line.

Conceptually the computational graph contains all information about the algorithm that is to be systolized. The purpose of introduction of the projective graph is to improve PE utilization during systolization process.

2.2 A systolic design for recursive algorithms

To achieve maximal parallelism of an algorithm, we must try to find data dependencies in a course of computations. Observe that, in the equation of matrix multiplication, the dependency of $w^{k}_{i,j}$ is local, while the dependencies of $a_{i,k}$ and

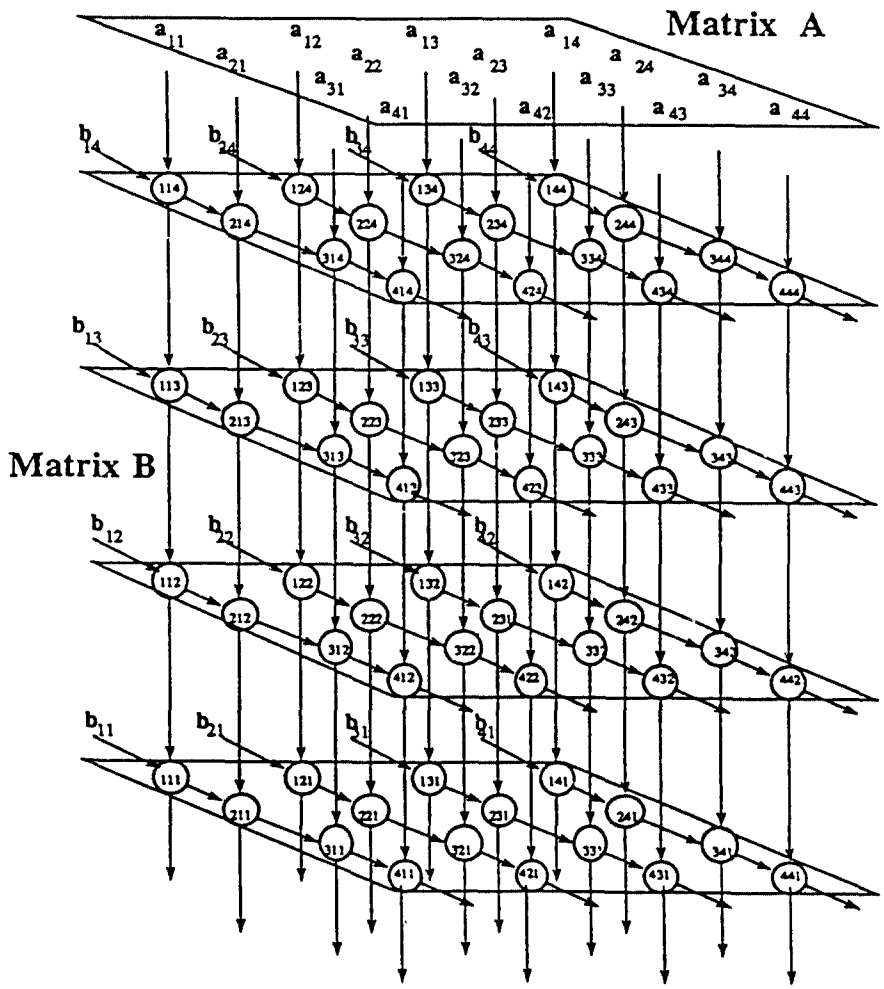

Figure 1. The computational graph of the matrix multiplication algorithm.

$b_{k,j}$ are global. The global dependencies can be loclized by distributing $a_{i,k}$ and $b_{k,j}$ values without global broadcasting in the way described below. It results in an algorithm with local dependencies. The techniques for the localization of data dependencies have been studied in many lectures, (for example, see [24]). The computational graph of the algorithm which expresses data dependencies precisely is drawn as Figure 1. For $1 \le i,j,k \le n$

$$(2.8)\quad a(i,k,j) = \begin{cases} a_{i,k} & \text{if } j = n+1 \\ a(i,k,j+1) & \text{if } 1 \le i,j,k \le n \end{cases}$$

$$(2.9)\quad b(i,k,j) = \begin{cases} b_{k,j} & \text{if } i = 0 \\ b(i-1,k,j) & \text{if } 1 \le i,j,k \le n \end{cases}$$

$$(2.10)\quad w(i,k,j) = \begin{cases} 0 & \text{if } k = 0 \\ w(i,k-1,j) + a(i,k,j) & \text{if } 1 \le i,j,k \le n \end{cases}$$

$$(2.11)\quad w_{i,j} = w(i,n,j)\ ;$$

The next step of the designing procedure is to group a set of vertices together to produce a two-dimensional computational graph. The direction of grouping is along the second component of the three-dimensional index space ; see Figure 2. The purpose of the grouping is to maximize the utilization of the processors. There are many different ways to group a set of vertices(for example, see[9], and [24]). It seems that none of them can achieve better time complexity. Note, from Figure 2, that the total time complexity is

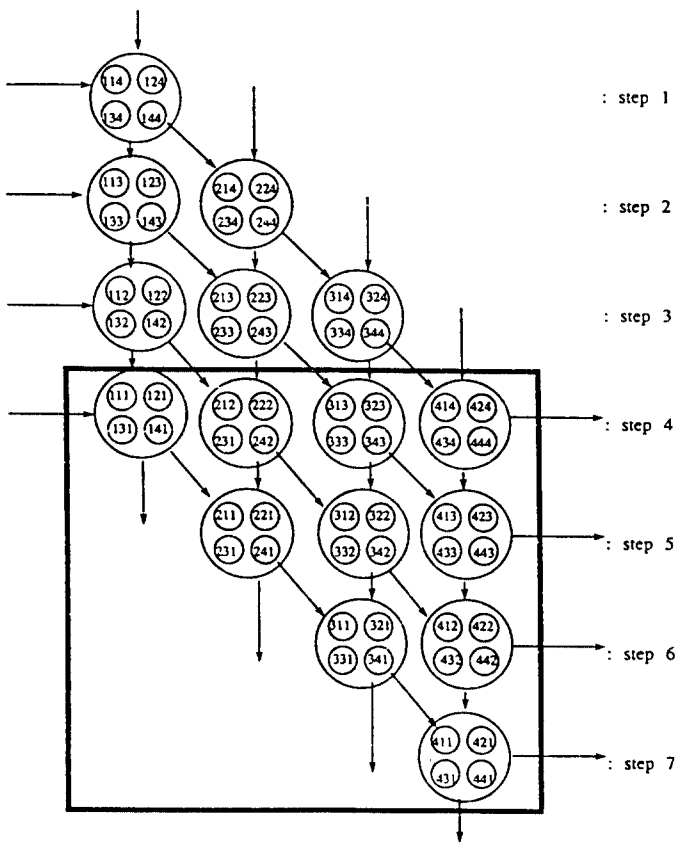

Figure 2. The Computational Graph after grouping.

$3n-1$ as the longest path in the graph is $2n-1$ and the input length is n. In [11], L. Melkemi and M. Tchuente show that time $3n-1$ is the best one can do for mesh-connected arrays.

In order to achieve better design, we must reconfigure the above pattern. The strategy is to rearrange the order of computation in such a way that all PEs at the first row can start their computations immediately. The new order of computation is arranged as follows : embed the top triangle of the figure, i.e., steps 1-3, into the left lower part of the square shown in the figure. To ensure the proper data streams, the wraparound connection of the array is required. The new systolic computing pattern is shown in Figure 3. In this new configuration, the computations occur at step 4, then steps 1 and 5, then steps 2 and 6, and finally steps 3 and 7 simultaneously. Although the computation order is different, it is not difficult to prove that the new configuration results in a correct answer for matrix multiplication. The interconnection of this new systolic array is shown in Figure 4. Each PE has two communication lines : one is vertical and another is diagonal. As we can easily observe, the computation time is reduced to $2n-1$. Note that the product of input matrices, $w_{i,j}$, remains in each PE. $PE_{i,j}$ will

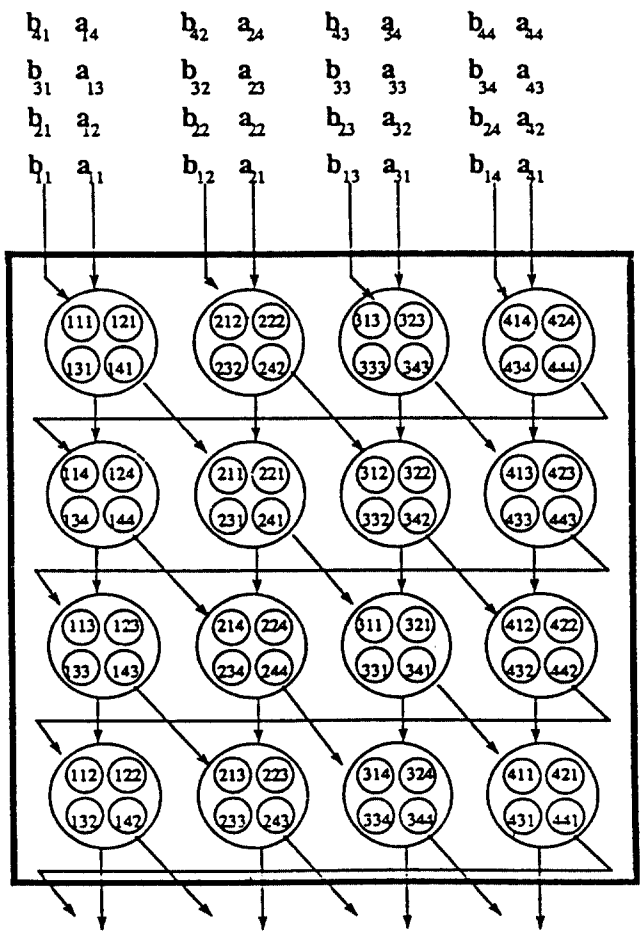

Figure 3. The computational graph after reconfiguration.

hold $w_{j,(j-i+1)mod(n)}$, where we assume $1 \le (j-i+1) mod(n) \le n$. Based on the above discussion, we can state the new systolic algorithm as follows :

Algorithm 1. Matrix Multiplication (systolic version)

{$a(i,j)$, $b(i,j)$, $w(i,j)$ are all local variables of $PE_{i,j}$.}

begin

$$a(i,j) = \begin{cases} a_{i,x} & \text{if } i=1 \\ a(i-1,j) & \text{if } 1<i) \end{cases}$$

$$b(i,j) = \begin{cases} b_{x,j} & \text{if } i=1 \\ b(i-1,(j-1)mod(n)) & \text{if } 1<i) \end{cases}$$

$$w(i,j) = w(i,j) + a(i,j) * b(i,j)$$

end

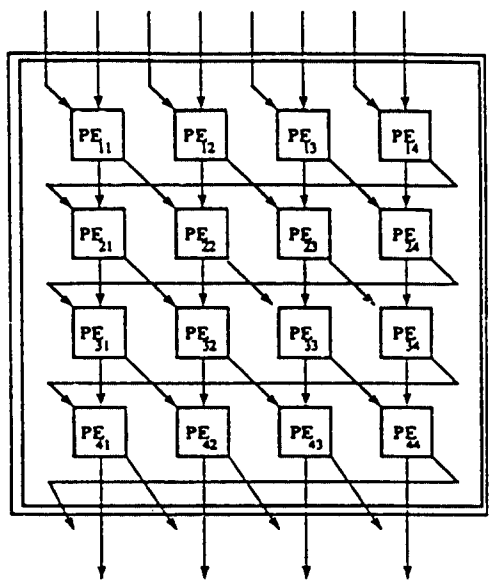

Figure 4. The interconnection scheme of the proposed systolic array.

Note that the index x is a time variable. It varies from 1 to n indicating that two input matrices are loaded in a pipelined fashion into the array in the first n steps. The computations are completed after $2n-1$ steps.

The above systolic design for matrix multiplication is the best among all known designs in terms of the computing time and array size, i.e., (computing time) × (array size). In the next section, we describe how the matrix multiplication of arbitrary size can be computed efficiently in a proposed array of fixed size.

Ⅲ. Digital-signal processings for arbitrarily large matrices

The rigidity of VLSI arrays, caused by their fixed size and fixed interconnection, is a serious problem which must be overcome before these devices become widely used. a natural solution to this problem is to divide the computations into small pieces to match the array size. We describe below how to apply this partitioning approach to compute the matrix product of arbitrary size in a fixed size array with the proposed architecture.

Suppose that the input matrices are of size $n \times n$ and the array is of size $m \times m$, where $m \ll n$. Partition the product W into k^2 submatrices each of size $m \times m$, where $k = n/m$. Each of these submatrices can be computed in $n+m-1$ steps. The input data needed for computing submatrix $W_{i,j}$, where $1 \le i,j \le k$, are m rows of matrix A, from $(m(j)+1)$th colum to (im)th row, and m columns of matrix B, from $(m(j-1)+1)$th column to (jm)th column. The configuration of these inputs for computing submatrix $W_{1,1}$, assuming $m=4$, is depicted in Figure 5.

The total computing time is $k^2(n+m-1) \approx n^2/m^2$. This result is nearly optimal in the following sense :

(total computing time) × (# of processors) $\approx n^3$.

The performance of the proposed array can be improved further if an asynchronous approach is adopted. In the following section, we will explore the potential of this approach. The idea used here is to design self-timed processors and communication protocols to get control of data streams such that each computation can start as soon as all of its data are available. The proposed array is a hybrid of systolic arrays and data flow machines.

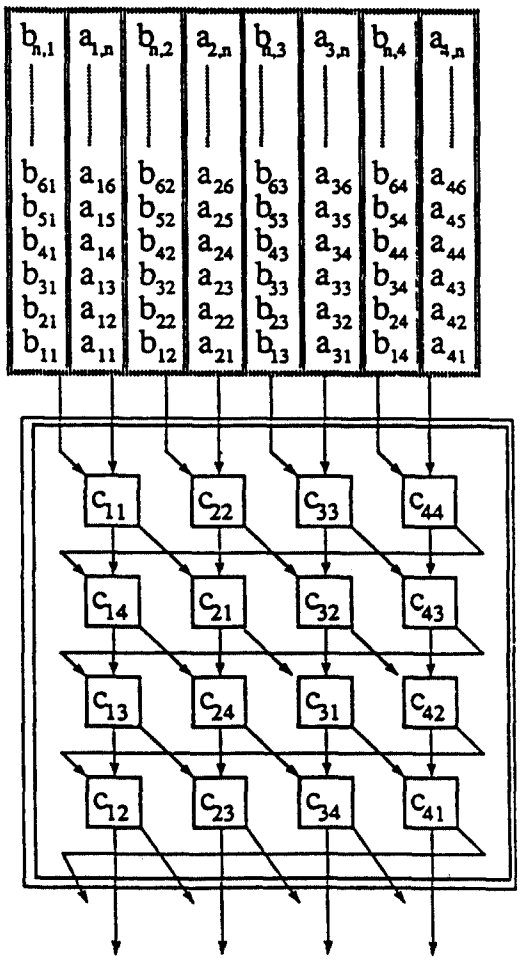

Figure 5. The computational scheme for computing submatrix $W_{1,1}$.

Ⅳ. An asynchronous design and its time analysis

A majority of the signal processing algorithms require the computation of inner product. In a systolic array each PE receives the data, carries out the multiply-and-add operation, and pumps the results rhythmically to the neighboring PE. Systolic arrays have been extensively used in many signal and image processing problems, and in matrix multiplication.

One problem with previous systolic arrays is the global control of data movement in different PEs. To assure proper timing and synchronization in systolic arrays, extra delays are needed. This slows down the computation, thereby decreasing throughput rate. Moreover, for large scale arrays this synchronization could become very tedious.

To overcome these difficulties and to speed up the computation time, design of asynchronous arrays was explord ; see [2],[8], and [17]. In an asynchronous design, instead of using gloval clock, self-timed PEs and communication protocols are provided. The advantage is that the whole period of a clock unit for multiplication, addition, and routing can be separated into several small steps and some of these steps can be executed simultaneously. The concept of asynchronous computations can be specified as below :

(1 *step*) : { *send an acknowledge signal to previous processors while getting data from them*; *send a request signal to next processors while forwarding data to them* }

(2 *step*) : *transfer data to next prcessors*

(3 *step*) : *multiply input data and accumulate the results ;*

Note that steps 2 and 3 can be executed simultaneously. In this section, we will develop a prococol to implement the above processes. The idea is to use self-timed PEs in which the inner product operation is triggered by the availability of the data. The major difference between the two architectures is the fact that the new array transfers the data to the next cell asynchronously by its local control unit, while systolic arrays require global timing for the control of data flows. Therefore a PE does not have to wait for data until the previous PE completes its computation. It has the basic features of the previous systolic array with the exception that the data routing and computing in each PE can be operated simultaneously. The following algorithm reflects this new feature.

Algorithm 2. Matrix Multiplication (preliminary asynchronous version)

begin
  while there are data entering $PE_{x,y}$ **Do**
    begin
      receive input data $a_{i,y}$ & $b_{x,j}$;
      transfer $a_{i,k}$ to $PE_{x+1,y}$;
      & transfer $b_{i,k}$ to $PE_{x+1,(y+1)mod(n)}$;
      & $mult_{i,j} \leftarrow a_{ik} * b_{kj}$;
      $w_{i,j} \leftarrow w_{ij} + mult_{ij}$;
    end *while* ;
end

4.1 A protocol for implementing Algorithm 2

To make the data flow independent of the multiply-and-add operation in each PE, we need a protocol to control the flow of data such that the values of input variables will not be overwritten during their computing periods. In the proposed protocol, four kinds of signals (REQ, ACK, FIG, and EMP) are intorduced : two external signals and two internal signals. The function of an REQ signal is to report to the next PE that the data in its output port is read
y for transmission. The function of an ACK signal is to report to the previous PE that its input port is ready to receive new data. The functions of FLG and EMP signals are to report the completeness of the multiply-and-add computation and the emptiness of the multiply-and-add computation and the emptiness of the input port. the protocol can be described formally as below. Note that only the protocol for dat stream $a_{i,j}$ is specified. The specification of the protocol for data stream $b_{i,j}$ is similar and is omitted.

1. Each $PE_{i,j}$ receives a request, $<REQ^{a}_{i-1,j}>$, from $PE_{i-1,j}$ when the data in the output port of $PE_{i-1,j}$ are ready to be transmitted.
2. Each $PE_{i,j}$ receives an acknowledge, $<ACK^{a}_{i+1,j}>$, from $PE_{i+1,j}$ when the input port of $PE_{i+1,j}$ is ready to accept new data.
3. Each $PE_{i,j}$ has two internal signals, $<FLG_{ij}>$ and $<EMP_{ij}>$ which report the completeness of the multiply-and-add operation and the emptiness of the input port
4. There are three transition latches in each PE_{ij}.

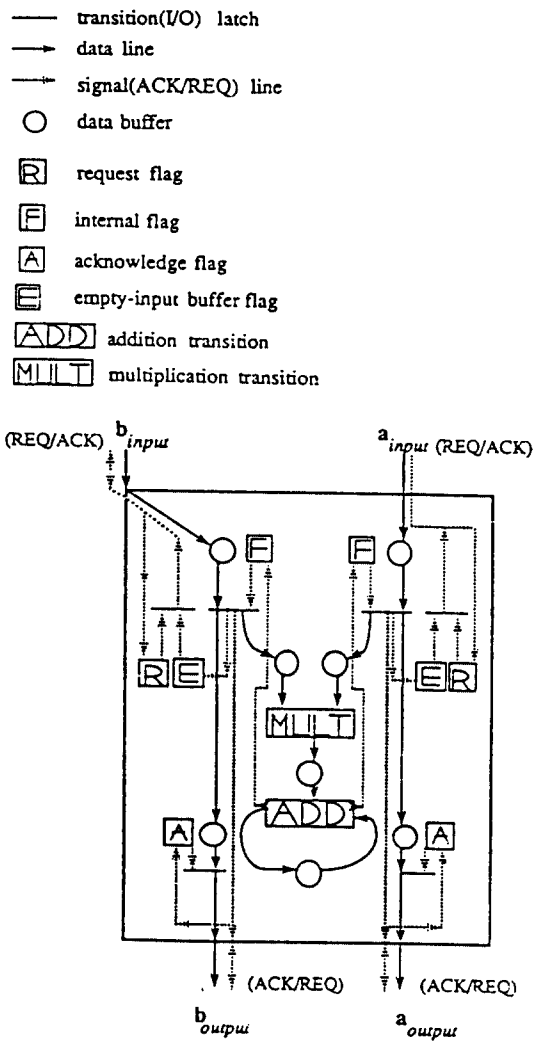

Figure 6. The proposed protocol in each PE.

Latch #1 : to control the data flow from the output port of $PE_{i,j}$ to the input port of $PE_{i+1,j}$

Latch #2 : to control the data flow from input port to buffer-for-mult and to output

port, and to activate the signals, $< REQ^{a}_{i-1,j} >$ and $<EMP_{i,j}>$, it is fired only after the signal, $<FLG_{ij}>$, is received.

Latch #3 : to activate the signal $<ACK^{a}_{i-1,j}>$ which is fired toward $PE_{i-1,j}$ only after both signals, $<REQ^{a}_{i-1,j}>$, is received.

In Figure 6, we depict a detailed configuration of this protocol. The signal buses and the data lines among different PEs in a 2×2 array are shown in Figure 7. Note that for each input matrix we need two lines which cross different PEs : one is a one-way data line and the other is a two-way bus for transmission of bit-signals, REQ and ACK.

The completed version of asynchronous algorithm for matrix multiplication equipped with the proposed protocol is described below.

Algorithm 3. Matrix Multiplication (a completed asynchronous version)

```
Begin
  For all PEs asynchronously do
    Begin
      Wait for REQ_input from the previous PE ;
      Receive REQ_input ;
      While input port is empty do
        Fire     REQ_input & EMP ;
        Send     ACK_input ;
        Receive  data from the previous PEs ;
        Store    data into the input port ;
        Fire     Send REQ_output to the next PE ;
                 Send data to output port &
                 buffer-for-mult ;
        Wait for ACK_output from the next PE ;
            {The instructions below are for
            internal computations.}
            While two input data in buffer-for-
            mult available do
              Read data from buffer-for-mult ;
              Mult (w_xx ← a_xy * b_yx) ;
              Store w_xx into buffer-for-add ;
              Read w_xx from buffer-for-add ;
                   old w_xx from buffer-for-W ;
              Add (old w_xx ← old w_xx + w_xx) ;
              Store old w_xx into buffer-for-W ;
              Send FLG signal ;
            Endwhile
        Endwhile
    Endfor
  Until there are no more data entering PE ;
End
```

It is easy to see that the algorithm described above correctly implements the preliminary asynchronous version of the matrix multiplication algorithm. Since the new input datum is received only after ACK_{input} is fired, indicating the completeness of internal computations, it guarantees that overwriting of input data will never occur.

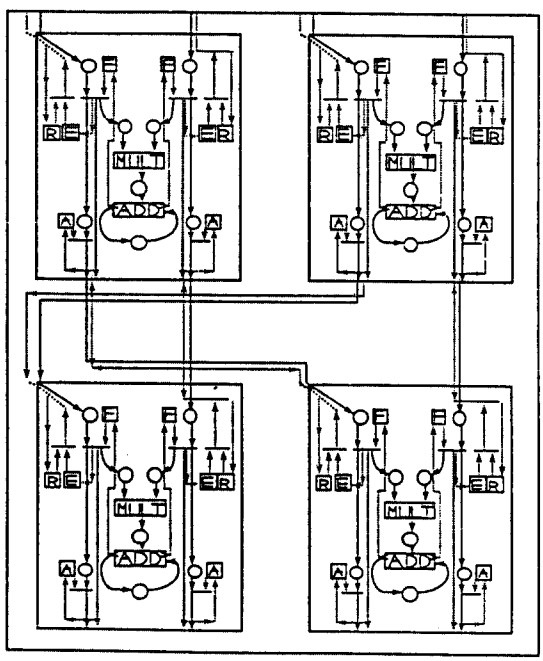

Figure 7. A complete configuration of the protocol for a 2-by-2 array.

In order to confirm that the performance of the asynchronous algorithm equipped with the proposed protocol will be better than that of systolic algorithms, we simulate the proposed asynchronous array using an Occam language. The channel

communications defined in Occam are basically the same as the communication protocol used. Hence a *Transputer* network programmed in Occam may quite naturally simulate the asynchronous arry. We describe a delay model and our simulation results under the described delay model at the next section.

4.2 Simulation models and performance analysis

The simulation model used here is called graphic data flow model. This model is suitable for description of time complexity for both synchronous and asynchronous computations.

First, we give definitions of some notations which will be used for describing the tie models of the arrays.

T_{fire} time for firing data via latch #2
T_{in} time for reading data from input port
T_R propagation time of signal REQ
T_A propagation time of signal ACK
T_{D_i} time for internal data transfer
T_{D_o} time for external data routing
T_E propagation time of signal EMP
T_W time for fetching data from buffer-for-mult
T_M comjputing time for multiplication
T_{Ad} computing time for addition
T_F propagation time of internal signal FLG

From the above definiton, a simulation model for the time complexity of a single PE of the asynchronous array is drawn as Figure 8. Note that all parallel edges can be executed simultaneously.

From this graph, the time delay of a single PE of the asynchronous array, $T_{PE(asyn)}$, can be described by the following formula :

$$T_{PE(asyn)} = T_{in} + T_{fire} + max\{T_R + T_{Ad} + T_F, max\{T_R + T_A, T_{D_i}\} + T_{D_o}\}.$$

The total computing time of the asynchronous array is approximately :

$$T_{total(asyn)} = (n-1)(T_{in} + T_{fire} + T_R + T_A, T_{D_o}) + nT_{PE(asyn)}$$

In order to evaluate the performance of the proposed asynchronous array, we carried out a simulation. The simulation was run in a Transputer network programmed in Occam. The

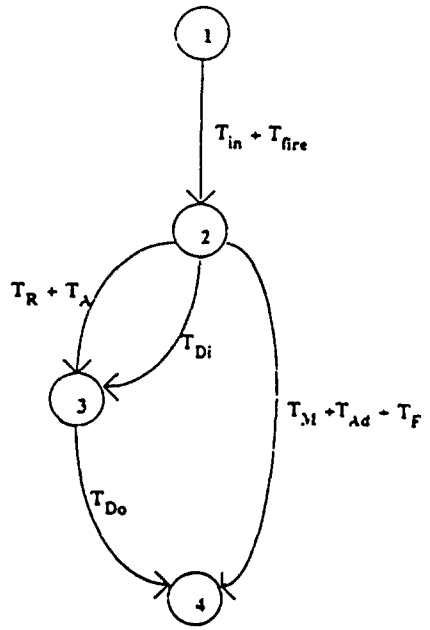

Figure 8. A simulation model for an PE of the asynchronous array.

Occam language is based on the models of communciation and concurrency : therefore it is suitable for programming and simulating either systolic arrays or asynchronous arrays. A configuration which indicates the assignment of channels for the communication protocol of a PE is shown in Figure 9. The main procedure for the simulation is liste in Figure 10, with corresponding time notations indicated.

In a synchronous model, estimating the time delay of single PE is rather simple and can be state as below :

$$T_{PE(syn)} = T_{in} + T_M + T_{Ad} + max\{T_R + T_A, T_{D_i}\} + T_{D_o}.$$

Note that the internal signal FLG is not necessary in a synchronous array. The total computing time of the synchronous array is :

$$T_{total(syn)} = (2n-1)T_{PE(syn)}.$$

The two curves, shown in figure 11, are plotted from the results of our simulation. In this simulation the size of input matrices is equal to the size of the arrya. It turns out that the performance of the asynchronous array is superior to that

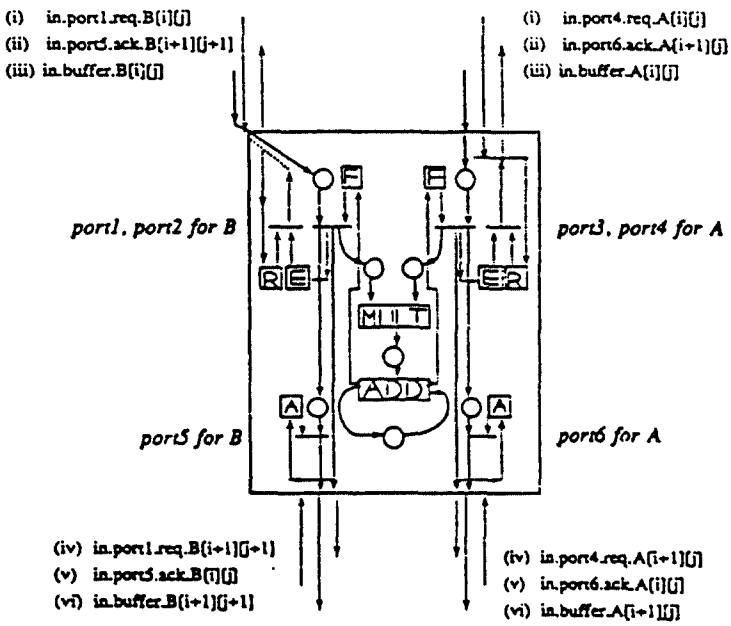

Figure 9. Channel assignments for an asynchronous simulation.

```
SEQ
  PAR                                                                          T_in
    Read.input.A ( in.buffer.A[i][j], in.port2.A[i][j] )
    Read.input.B ( in.buffer.B[i][j], in.port2.B[i][j] )
  PAR                                                                          T_fire
    Fire.port3.for.A ( in.port3.A[i][j], in.port3.Fa[i][j], out.port3.A[i][j],
                       out.port3.req.A[i][j], out.port3.arith.A[i][j], out.port3.Ea[i][j] )
    Fire.port2.for.B ( in.port2.B[i][j], in.port2.Fb[i][j], out.port2.B[i][j],
                       out.port2.req.B[i][j], out.port2.arith.B[i][j], out.port2.Eb[i][j] )
  PAR
    SEQ
      PAR                                                                      T_R
        Send.req.A ( out.port3.req.A[i][j], in.port4.req.A[i+1][j] )
        Send.req.B ( out.port2.req.B[i][j], in.port1.req.B[i+1][j+1] )
      PAR                                                                      T_A
        Wait.for.in.ack.A ( in.port4.req.A[i+1][j], in.port4.Ea[i+1][j],
                            in.port6.ack.A[i][j] )
        Wait.for.in.ack.B ( in.port1.req.B[i+1][j+1], in.port1.Ea[i+1][j+1],
                            in.port5.ack.B[i][j] )
    SEQ
      PAR                                                                      T_Di
        To.out.buffer.A ( out.port3.A[i][j], out.buffer.A[i][j] )
        To.out.buffer.B ( out.port2.B[i][j], out.buffer.B[i][j] )
```

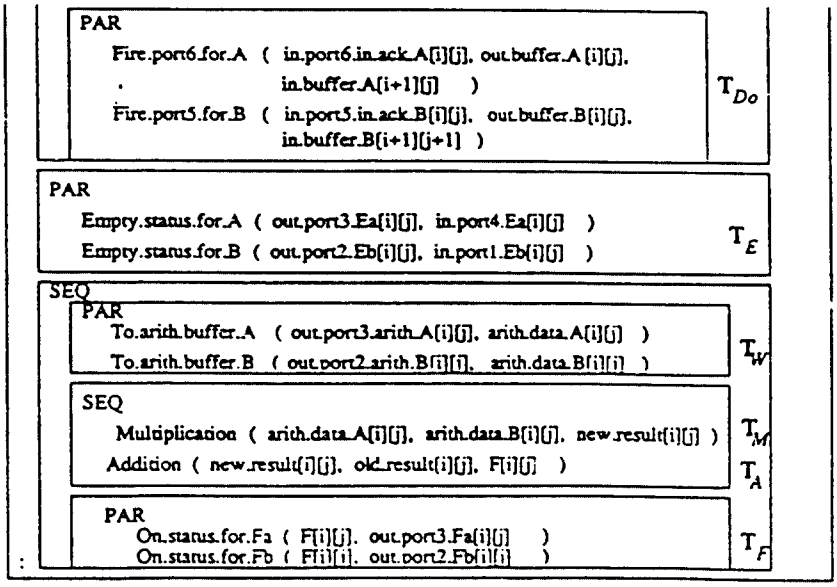

Figure 10. A simulation procedure for asynchronous matrix multiplication.

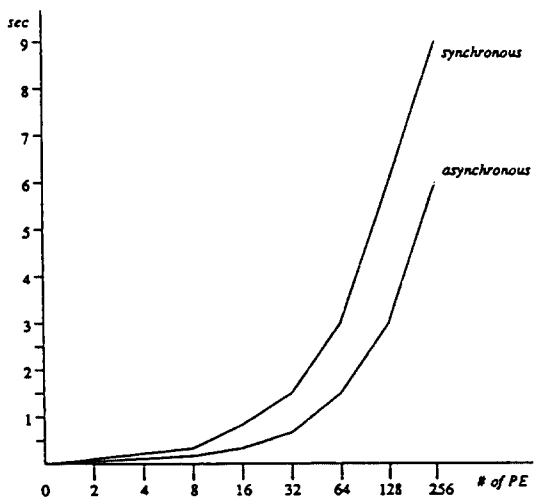

Figure 11. Systolic vs. asynchronous arrsy.

of the systolic array

4.3 Efficiencies and Comparisons

In this approach we have presented an advanced systolic array with asynchronous dataflow. The array has the features such as locality, modularity, self-timed PEs with asynchronous data transfer, and independent dataflow with respect to PEs operations.

The advanced systolic array(ASA) and the previous systolic arrays(PSA) can be analyzed as follows.

(1) Let T_λ be the average time for transfer datum from PEs to PEs. The T_λ is the total time of a input transfer, a output transfer, a acknowledge, and a request time in each processing element.

(2) In most matrix processing, each PE performs both a multiplication and a addition operations. Let T_m be the required time for multiplication and T_a for addition operation. Then T_{m+a} time be the average time for multiply-and-add.

(3) If $T_{m+a} \leq T_\lambda$, then the c omputing time is completely by the data-transfer time ex cept the computing time of the bottom PEs.

In this case,

$T_{total}=(n-1)(T_\lambda)+nT_{m+a}$

(4) If $T_{m+a}>T_\lambda$, then time delay $\alpha=T_{m+a}-T_\lambda$ need to be added for the actual data-transfer time between PEs.

In this case,

$T_{total}=(n-1)(T_\lambda+\alpha)+nT_{m+a}=(2n-1)T_{m+a}$.

The total processing time is

$T_{total}=(n-1)max\{T_\lambda, T_{m+a}\}+nT_{m+a}$.

(5) On the other hand, the total time T_{PSA} for the previous systolic array(PSA) with synchronous controls is given by

$T_{PSA}=(2n-1)T_\lambda+(2n-1)T_{m+a}=(2n-1)(T_\lambda+T_{m+a})$

(6) We can conclude that

(a) $T_{m+a}\leq T_\lambda : T_{PSA}-T_{ASA}=nT_\lambda+(n-1)T_{m+a}$
(b) $T_{m+a}>T_\lambda : T_{PSA}-T_{ASA}=(2n-1)T_\lambda$

Hence our advanced systolic array(ASA) scheme is more effcient than the previous systolic array(PSA), because data-transfer time is possibly masked. For more efficent VLSI implementation, the operations of each PE is controlled locally and handles asynchronously. This approach speeds up the computation time by allowing the individual PE to operate independently to reduce the waiting time.

V. Conclusions

In this paper, we have shown new designs of a systolic array and an asynchronous array for matrix multiplication. The design procedure should be applicable to other recursive algorithms. It will be of great interest to design effcient systolic arrays and asynchronous arrays for thos recursive algorithms described in section 2. The new systolic array achieves better performance by using a wraparound connection. the asynchronous array improves the performance of the systolic array further as indicated in our simulation. Some additional hardware is needed for implementing protocols, but a reduction of computing time is significant for large scale computations. It maight be possible to improve the proposed protocol for data communication. Issues about implementation and evaluation of the asynchronous array deserve more research attention. More research can be conducted in this direction.

Acknowledgement

We would like to thank Dr. Harriett B.Rigas for many useful comments and suggestions.

References

1. H. Y. Chung and G. He, "A versatile systolic array for matrix computations," *12th Int. Symp. on Comp. Arch.*, pp.315-322, 1985.
2. R. G. Cytron, "Compile-time Scheduling and Optimization for Asynchronous Machines," Ph.D. Dissertation, Univ. of Illinois at Urbana-Champaign, 1985.
3. V. Fragnelli, "Architectures for sparse band matrix dense vector multiplication," *Parallel Computing*, pp.507-514,1984.
4. H. T. Kung and C. E. Leiserson, "Systolic arrays (for VLSI)," in *Sparse Matrix Proc.*, Duff et al., PA, pp.256-282, 1979.
5. H. T. Kung and C. E. Leiserson, "Algorithms for VLSI processor Arrays," in C. Mead and L. Conway, *Introduction to VLSI System*. Reading, MA : Addison-Wesley,pp.263-332, 1980.
6. H. T. Kung, "Why systolic architectures?," *Computer*, Vol. 15, No.1,pp. 37-46,Jan.1982.
7. K. Hwang and Y. Cheng, "Partitioned matrix algorithm for VLSi arithmetic systems," *IEEE Trans. Comput.*, Vol. 15, No.1,pp.37-46, Jan. 1982.
8. M. A. Krank and D. F. Wann, "Asynchronous and clocked control structures for VLSI based

interconnection networks," *9th Symp. on comput. arch.*, pp.50-59, 1982.
9. G. J. Li and B. W. Wah, "The design of optimal systolic arrays," *IEEE Trans. Comput.*, Vol. C-34,pp. 66-77,Jan. 1985.
10. F. C. Lin and I. C. Wu, "On designing systolic algorithms," in *Proc. Int'l Symposium on VLSI Tech., System and Appl.*, Taipei, Taiwan, May 1985.
11. L. Melkemi and M. Tchuente, "Complexity of matrix product on a class of orthogonally commected systolic arrays," *IEEE Trans. Comput.*, Vol. C-36,No.5,pp.615-619, May 1987.
12. L. Melkemi and M. Tchuente, "Algebraic and combinatorial aspects of systolic algorithms," *Proc. of Int'l Workshop on Parallel Algorithms and Arch.*, France, pp. 269-279, April 1986.

This paper was suported by NON DIRECTED RESERACH FUND, Korea Research Foundation 1991.

全 文 錫(Moon-Seog Jun) **正會員**

1980년 2월 : 숭실대학교 전자계산학과 졸업
1986년 12월 : University of Maryland 전산과 졸업 (석사)
1989년 1월 : University of Maryland 전산과 졸업 (박사)
1989년 8월 : Morgan State University 전산수학과 조교수
1991년 2월 : New Mexico State University 부설 Physical Science Lab. 책임 연구원
1991년 3월~현재 : 숭실대학교 전산과 조교수
※관심분야 : 병렬 알고리즘, 병렬컴퓨터구조, 암호학 알고리즘, 컴퓨터 이론, VLSI