A RECONFIGURABLE APPROACH TO A SYSTOLIC SORTING ARCHITECTURE

Thomas Alexander and Mani Soma

Department of Electrical Engineering
University of Washington, Seattle, Washington

ABSTRACT

Systolic architectures are characterized by a simple and regular structure with high computational throughput. To simplify the task of generating a systolic array, researchers are investigating the possibility of using reconfigurable arrays, which can be modified to match many applications. In this paper, we describe a prototype reconfigurable array, and demonstrate that it can be used to realize at least two different classes of problems. The specific applications selected are drawn from the sorting problem and the Warshall algorithm for transitive closure. Some performance measures in terms of the VLSI complexity of the realizations are also made.

INTRODUCTION

Conventional Von Neumann computers suffer from the 'Von Neumann bottleneck' between processing units and memories, which limits their realizable computation rates. Overcoming this bandwidth limitation can require the use of complex and expensive architectures.

Systolic arrays, introduced by Kung [1], have gained prominence by offering an alternative solution for many types of signal and array processing computations. In general, a systolic array is a collection of a number of processing elements, possibly of more than one type, interconnected in some regular fashion. Only the edges of the array participate in data input and output; data fed into the array is generally operated upon many times before being returned to memory. This characteristic leads to a dramatic reduction in the required memory bandwidth, hence obviating the Von Neumann bottleneck. In addition, due to their simple and regular structure, systolic arrays exploit concurrency rather than complexity and are highly amenable to VLSI implementation. Generally, systolic arrays are used as attached processors to some host computer, and perform special-purpose tasks at high throughput. The areas of application include signal processing (convolutions [2], IIR and FIR filtering [3], etc.), matrix arithmetic (matrix multiplication [4], inversion [5] and solutions of linear systems) and other arithmetic and non-numeric applications (searching and sorting [6], graph algorithms [7], parsing [8], etc.).

The regularity and simplicity of systolic arrays have given rise to research into their automatic generation for a particular algorithm or algorithm class. Numerous approaches, involving the functional, temporal, spatial or data-flow mappings of algorithms on to processor arrays, followed by the generation of the actual array from the constraints imposed by locality of interconnection and processor complexity, have been described [9,10].

In general, however, systolic arrays are used to perform special-purpose computations, and are dedicated to their application. This may not prove cost- effective for those applications requiring both a high throughput and considerable flexibility. Additionally, due to their intensive use of VLSI and parallelism, some measure of fault tolerance is desirable [11]. In response to these needs, therefore, reconfigurable systolic arrays are being developed. Such arrays can be dynamically modified to exclude faulty elements as well as to match the task at hand. Considerable advantage is derived from the fact that the architecture exhibits a close match to the algorithm; also, many different systems are not needed for the solution of all the problems encountered in a given application.

Much current work is being undertaken in this area ([9],[12],[13],[14]). Most of the work focuses on the use of an array of processors having a fixed, relatively high-level instruction set and a fixed ordering, with reconfiguration being carried out by means of a variable interconnection scheme. This approach is exemplified by CHiP and TRAC [15]. Control of such systems is often complex, except for algorithms that involve very simple operations that are carried out concurrently over many data items (i.e. SIMD schemes) or data-flow realizations. The use of conventional instruction sets for the processors in the array necessitates local instruction memories and decoding.

This paper seeks to present a different approach to reconfigurable systolic arrays. In addition to making the interconnection network reconfigurable, (to match the topology of the algorithm flow graph) we make the individual processing elements themselves configurable, in order to provide the best match to the computational requirements of individual nodes. Rather than having a fixed instruction set, we hope to tailor the architecture of the different processing elements in such a way as to carry out their functions without the use of instruction streams. This will reduce the problems of control and coordination as well as providing a closer match to the algorithm being implemented. The prototype architecture being developed, and an analysis of its application to a particular class of problems (numeric sorting), are described below.

DESCRIPTION OF ARCHITECTURE

The general topology of the architecture does not differ significantly from those already described in the literature. It consists of a set of processing elements arranged in a rectangular tesselation interconnected with a regular, modifiable network (Fig. 1).

The interconnection network may be separated into two parts: a local interconnect, serving to link neighbouring processors, and a global network, used for transferring data between functional blocks. After analysis of the communication requirements of various configurations, a simple 4-neighbour connection scheme was selected for the local network; additionally, a 4-next-nearest-neighbour interconnect was provided to facilitate the implementation of some commonly used functions (for instance, multi-bit multiplications). Each link in the network is a single bit wide. However, the array supports bit-parallel arithmetic by clustering multiple processing elements and

ISCAS '89

multiple links.

Data transfer between various configured functional blocks is supported by a global interconnect, organized as a rectangular mesh of 4-wide links (Fig 1). Switching elements at the nodes of the rectangles are used to reconfigure the interconnection network by connecting various links together in any desired pattern. This network may also be used to provide inter-element communication within a given functional block (to propagate control signals, for example). A wired-OR capability is available for communications using the global interconnect. This is quite useful, as shall be shown below.

It is intended that the processing elements used in the array should be architecturally configurable to match any reasonable computational requirements imposed upon the individual nodes. At the same time, the overhead and control problems incurred by the use of sequential instruction streams should be avoided. Hence the computational logic used is fully combinatorial, with sequential elements serving local storage functions. A completely general programmable combinatorial element capable of forming any Boolean function of a number of inputs (essentially a look-up table implemented as a random-access memory) would have occupied too much space; a 16-input, 1-output element would require 65,536 bits of memory if implemented in this way. A tradeoff has therefore been made between generality and realizability, and programmable logic arrays are used to create the processors. While a programmable logic array is not as general as a look-up table, it permits a number of Boolean functions to be implemented with a low hardware overhead.

Each processing element, or cell, consists of a set of 4 PLAs, coupled with 2 bits of storage, as illustrated in Fig. 2. Two of the PLAs generate outputs from the cell; the remaining two arrays are used to provide inputs to the storage flip-flops. A common clock is used for all the flip-flops in the array, hence providing for a synchronous architecture. This also simplifies control and analysis at the expense of speed. The cells used in the array are intended to be extremely simple in order to permit the integration of large numbers of them on a single VLSI chip.

Reconfiguration and cell programming are accomplished by means of programmable state registers distributed throughout the array. The individual 1-bit PLAs, for instance, are programmed using 8 16-bit registers (Fig. 3a); the cell inputs are selected at the same time. Cell input and output signal distribution is programmed via 4-bit registers controlling switching logic. The topology of the global interconnection network is also alterable via state registers located at the nodes, as shown in Fig. 3b. A single 16-bit programming bus runs from cell to cell throughout the array; local decoders serve to enable and disable state registers from a global address bus.

PARALLEL SORTER IMPLEMENTATION

Hardware sorting has many applications in various computational processes [16]. Much attention has been paid to special-purpose sorting networks [6], parallel sorting architectures [17] and systolic sorting arrays. Due to the simple, regular and highly parallel nature of many sorting algorithms, systolic sorter implementations are quite attractive for VLSI implementation. We discuss here the configuration of the architecture described above to implement a simple numeric sorting algorithm.

The particular sorting algorithm selected is called the counter-current compare-exchange sort, and is described in [18]. A block diagram illustrating the data flow paths and the functional blocks involved is shown in Fig. 4; it will readily be seen that the architecture is highly regular, and is extensible to the sorting of arbitrary-sized arrays of numbers having any bit width. A brief description of the sorting algorithm is as follows:

1) All the internal registers are first initialized to zero.
2) Data are clocked in at the left-hand end of the linear array.
3) At any compare-exchange stage i, the new data in register R_a^i is compared with the data in register R_b^i.
   - If greater, the contents of R_a^i are loaded into R_b^i, and the data in R_b^i is sent on to the next stage.
   - Otherwise, the data in R_a^i are sent on to the next stage, and new data are clocked in.
4) The process continues until a total of $2N$ clocks (N being the number of data words) have been supplied; the contents of registers R_b^1 through R_b^N will now contain the sorted series in descending order.

If N data items are to be sorted, the hardware complexity is $O(N)$, and the time required to sort them is $O(2N)$. For a proof of the algorithm, see [18].

It is apparent from the above that a systolic implementation of the compare-exchange sorting mechanism is fairly simple. The primary functional blocks required for each stage are a comparator, two registers and a multiplexer. Figure 5a shows the configuration of a cell in the reconfigurable architecture to implement one bit of a sorter stage. For simplicity, the bit width is assumed to be 4 bits; extension to wider words is straightforward and will not be dealt with here. A distributed comparator is used, with vertical global interconnect being used to distribute comparison results between stages, horizontal global interconnect used for array initialization and interstage communication being carried out by the local interconnect. The upper and lower register stages are implemented in each cell using the two bits of local storage supplied. It is readily shown that the Boolean expressions required to implement the control and communication occupies three or fewer product terms in any PLA. The complete sorter configuration is shown in Figure 5b. The operation of the array closely follows the description already given.

Since the control and coordination are entirely combinatorial and takes place between clock periods, the entire array behaves as a synchronous machine. In every clock period N compares and transfers are carried out in parallel, requiring $2N$ clocks, as discussed above, for a sort. If the word width is m, a total of Nm cells are needed for the array, together with the global interconnect associated with each cell. Assuming unit area per cell, the total area A is Nm; if m is assumed to be $(1+\epsilon)\log N$, ($\epsilon > 0$) [17], then the VLSI complexity of the array is $O(N^2 \log N)$. This compares favorably with the lower bounds on the complexity of parallel sorting proved by Leighton [19].

A DIFFERENT IMPLEMENTATION

In order to illustrate the versatility of the array, we will show how a different sorting algorithm - the odd-even transposition sort - may be implemented. The compare-exchange sort is generally used for sorting 'on-the-fly' - i.e. as the data are generated and fed into the array. In contrast, the odd-even transposition sort is applicable to in-place sorting, as for instance when it is necessary to sort the results of a previous operation. Such requirements occur, for example, in image processing, where a median filter is to be applied to the results of filtering an image region [20].

The odd-even transposition sort may be described as follows:

1) The array is preloaded with $w_1, w_2 \cdots w_N$, the N words to be sorted.

(217) 333-4789

Further information can be obtained from Dr. W. Kenneth Jenkins.

2) In any time step i the odd words w_k, $k = 1,3,5,7 \cdots$ are compared to their successors w_{k+1}, and exchanged if $w_k > w_{k+1}$.

3) In time step $i+1$, however the even words w_k, $k = 2,4,6,8 \cdots$ are compared with their successors and exchanged according to the same criteria.

4) The result after N time steps is a set of words sorted in ascending order.

For a simple proof of this algorithm, see [21].

The configuration of the array for an odd-even transposition sort is carried out in a similar fashion to that for the compare-exchange sort, and is subject to the same considerations for word size and cell distribution. The arrangement of a single cell, used to implement a single bit of the sorter as before, is shown in Fig. 6a. The alternate phases of odd- and even-word comparisons are controlled by a single global interconnect running horizontally along each row of cells (Fig 6b). A multiplexer function programmed into the comparison and register loading PLAs is controlled by this signal to select either the predecessor or successor of a cell in any phase for both magnitude comparison and data transfer. The registers are assumed to have been preloaded with the data to be sorted. The sorting phase control can be driven by a single cell (configured as a toggle) or supplied externally.

As before, the control is entirely combinatorial between clock edges, and the array is synchronous. As the register load time is not taken into account, the sorting time is $O(N)$; since again one cell is used per sorter bit, the area is Nm, and the VLSI complexity calculations are identical to those given above. It is seen that no change in the array was necessary to implement an entirely different sorting algorithm, demonstrating the architectural flexibility of reconfigurable arrays.

EXTENSION TO THE TRANSITIVE CLOSURE PROBLEM

To demonstrate the versatility of the array, it is necessary to prove that it is applicable to a variety of problem classes. Here we will attempt to show how the architecture being considered can be used to implement the Warshall algorithm for transitive closure as well as the sorting architectures described above.

The sequential Warshall algorithm for computing the transitive closure of a graph given its adjacency matrix A is quite well known [22], and is described as follows:

Given a directed graph $G = \{ V,E \}$ with vertex set V, $|V| = N$, and edge set E, the adjacency matrix A of G is an $N*N$ matrix in which any element a_{ij} of the matrix is 1 if there is an edge from vertex v_i to vertex v_j and 0 otherwise. The transitive closure problem is to compute the transitive closure matrix A^+, which is also an $N*N$ matrix where any element a^+_{ij} of the matrix is 1 if there exists a path of length zero or more from vertex v_i to vertex v_j. According to the Warshall algorithm, we may compute this matrix A^+ from A by the following:

For $k = 1$ to N
  For $j = 1$ to N
    For $i = 1$ to N
      $x_{ij}(k) = x_{ij}(k-1)$ *OR* $(x_{ik}(k-1)$ *AND* $x_{kj}(k-1))$

where $X(0) = A$, the adjacency matrix, and $X(N) = A^+$, the transitive closure matrix.

Now consider the configuration of the array shown in Fig. 7, with $N+1$ rows and N columns of cells. The global interconnect links are configured to form signal lines running from one end of the array to the other, in both vertical and horizontal directions. The wired-OR capability of the global interconnect (described earlier) is used here such that a given horizontal or vertical line is driven to a 1 if one or more of the cells along the particular row or column respectively outputs a 1; otherwise, the line remains at a 0 state. Further, the lower N rows of cells are configured to implement the following:

1) if the internal storage element is loaded with a 1, then it will continue to hold a 1.

2) otherwise, if both the vertical and horizontal lines connected to a cell are a 1, then the internal state register will be loaded with a 1 on the next clock.

3) if either the vertical or the horizontal line connected to a cell is driven to a 1, and the internal state is a 1, then the cell will output a 1 along both its vertical and horizontal lines.

4) Finally, the uppermost row of cells is configured as a simple shift register, initially loaded as in Fig. 7; on every clock, the initial 1 is shifted 1 place to the right and a 0 is clocked into the left-hand side.

To carry out the Warshall algorithm computation, the array is preloaded with the adjacency matrix A and clocked N times. After N clocks, the contents of the array will be the transitive closure matrix.

Theorem 1. The arrangement described above computes the transitive closure matrix when the array is preloaded with the adjacency matrix.

Proof. All the diagonal elements x_{ii}, $1 \le i \le N$, of the adjacency matrix are 1's since every vertex v_i may be considered as being connected to itself by an edge of length zero. Now consider the state of the array at time step k, $1 < k \le N$. According to condition (4) above, the k^{th} vertical line will be driven to a 1 by the topmost row of the array. However, as the diagonal elements are also 1's, by condition (3) the k^{th} horizontal line will also be driven to a 1.

Consider now the i^{th} cell along the k^{th} column. If the internal contents of the cell at time step k, $x_{ik}(k-1)$, are a 1, then by condition (3) the i^{th} horizontal line will be driven to a 1. Again, consider the j^{th} cell along the k^{th} row; if the contents of the cell, $x_{kj}(k-1)$, are a 1, then by condition (3) the j^{th} vertical line is driven to a 1. If both the i^{th} horizontal line and the j^{th} vertical line are driven to a 1, then by condition (2) cell x_{ij} will be loaded with a 1 on the next clock. Otherwise, by condition (1), cell x_{ij} will continue to retain its value. Hence the contents of cell x_{ij}, $1 \le i \le N$, $1 \le j \le N$, in the next state k may be described by $x_{ij}(k) = x_{ij}(k-1)$ *OR* $(x_{ik}(k-1)$ *AND* $x_{kj}(k-1))$

But this is precisely the expression given above for the Warshall algorithm. Hence the proof.

CONCLUSIONS

We have described a reconfigurable systolic processor array, constructed with processors that are themselves configurable, and explored applications of such a system. We have shown that it is applicable not only to the general class of systolic sorting problems, but is also capable of implementing a solution for a problem from an entirely different domain, i.e. the transitive closure problem. This serves to illustrate the flexibility and wide applicability of such arrays. Currently, we are in the process of investigating the development of a generalized mapping strategy for the transformation and implementation of algorithms from a large number of problem classes on to reconfigurable arrays such as the one described.

REFERENCES

[1] H.T.Kung, "Why Systolic Architectures?", *IEEE Computer*, Vol 15, No. 1, pp. 37-46, Jan. 1982.

[2] K. Doshi and P.Varman, "A Modular Systolic Architecture for Image Convolutions", *ACM Comput. Arch. News*, Vol 15, No.2, June 1987, pp. 56-63.

[3] H.T.Kung, "Special Purpose Devices for Signal and Image Processing: An Opportunity in VLSI", in *Proc. Society of Photo-Optical Engineers*, July 1980, Vol 241, pp. 76-84.

[4] L.Melkemi and M.Tchuente, "Complexity of Matrix Product on a Class of Orthogonally Connected Systolic Arrays", *IEEE Trans. Comput.*, Vol C-36, No. 5, pp. 615-619, May 1987.

[5] J.Bu and E.F.Deprettere, "A Parallel VLSI Algorithm for Fast Sparse Matrix Inversion by Gauss-Seidel Iteration", in *Proc. IEEE Intl. Symp. Circuits and Systems*, May 1987, pp. 1052-55.

[6] H.W.Lang et al, "Systolic Sorting on a Mesh-Connected Network", *IEEE Trans Comput.*, Vol C-34, No. 7, pp. 652-658, July 1985.

[7] S.Y.Kung, S.C.Lo and P.S.Lewis, "Optimal Systolic Design for the Transitive Closure and the Shortest Path Problems", *IEEE Trans. Comput.*, Vol C-36, No. 5, pp. 603-614, May 1987.

[8] J.H.Chang et al, "Parallel Parsing on a One-Way Array of Finite-State Machines", *IEEE Trans. Comput.*, Vol C-36, No. 1, pp. 64-75, Jan. 1987.

[9] D.I.Moldovan and J.A.B.Fortes, "Partitioning and Mapping Algorithms into Fixed Size Systolic Arrays", *IEEE Trans Comput.*, Vol C-35, No. 1, pp. 1-12, Jan. 1986.

[10] H.Nelis and E.F.Deprettere, "A Systematic Method for Mapping Algorithms of Arbitrarily Large Dimensions onto Fixed Size Systolic Arrays", in *Proc. IEEE Intl. Symp. Circuits and Systems*, May 1987, pp. 559-563.

[11] M.Sami and R.Stefanelli, "Fault-Tolerance and Functional Reconfiguration in VLSI Arrays", in *Proc. IEEE Intl. Symp. Circuits and Systems*, Aug. 1986, pp. 643-648.

[12] L.Snyder, "Introduction to the Configurable, Highly-Parallel Processor", *IEEE Computer*, Vol 15, No. 1, Jan. 1982.

[13] F.Lombardi et al, "Functional Reconfiguration in Fixed-Size VLSI Arrays", in *Proc. IEEE Intl. Symp. Circuits and Systems*, May 1987, pp. 386-389.

[14] B.Mendelson and G.M.Silberman, "Mapping Data Flow Programs on a VLSI Array of Processors", *Computer Arch. News*, Vol 15, No.2, June 1987, pp. 56-63.

[15] S.R.Deshpande et al, "TRAC: An Experience with a Novel Architectural Prototype", in *Proc. AFIPS National Computer Conf.*, Vol 54, 1985, pp. 247-258.

[16] H.B.Demuth, "Electronic Data Sorting", *IEEE Trans Comput.*, Vol C-34, No. 7, pp. 652-658, July 1985.

[17] G.Bilardi and F.P.Preparata, "A Minimum Area VLSI Network for O(log n) Time Sorting", *IEEE Trans Comput.*, Vol C-34, No. 4, pp. 336-343, April 1985.

[18] M.Vandehey, "Design of a VLSI Sorting Circuit", M.S. Thesis, Dept. of Elect. Engg., Univ. of Wash., Seattle, 1983.

[19] T.Leighton, "Tight Bounds on the Complexity of Parallel Sorting", *IEEE Trans Comput.*, Vol C-34, No. 7, pp. 344-354, July 1985.

[20] I.Song and S.A.Kassam, "Nonlinear Filters based on Generalized Ranks for Edge Preserving Smoothing", in *Proc. IEEE Intl. Symp. Circuits and Systems*, Aug. 1986, pp. 401-404.

[21] J.van Leeuwen, "Distributed Computing", Tech. Rep. RUU-CS-82-8, Dep. Comput. Sci, Univ. of Utrecht, The Netherlands, 1982.

[22] S.Warshall, "A Theorem on Boolean Matrices", *Journal of the ACM*, Vol 9, Jan. 1962.

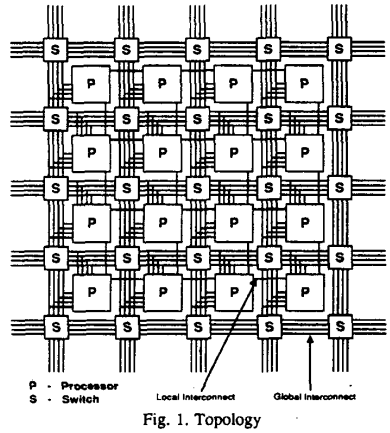

Fig. 1. Topology

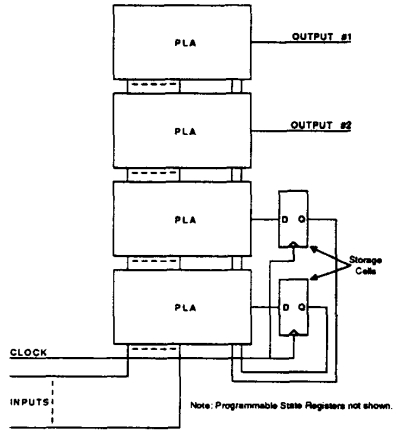

Fig. 2. Cell

(217) 333-4789 Further information can be obtained from Dr. W. Kenneth Jenkins.

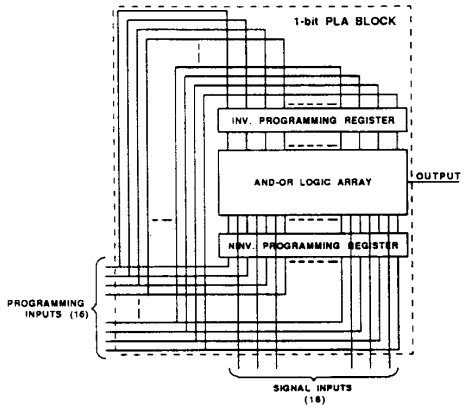

Fig. 3a. Organization of PLA Configuration

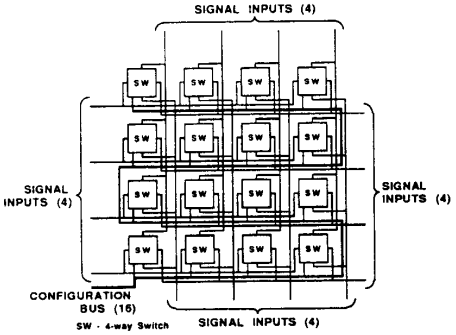

Fig. 3b. Organization of Switch Configuration

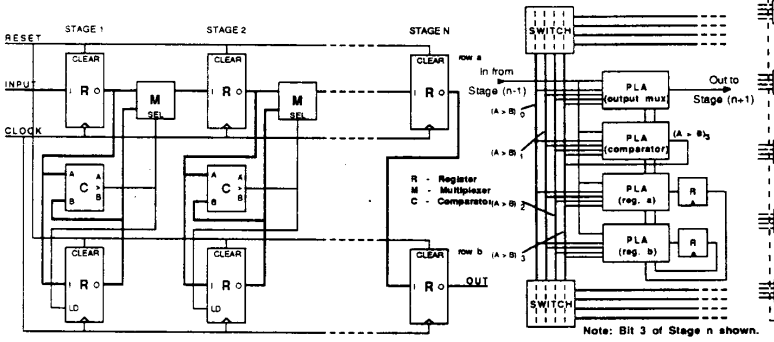

Fig. 4. Compare-Exchange

Fig. 5a. Compare-Exchange Sorter Cell

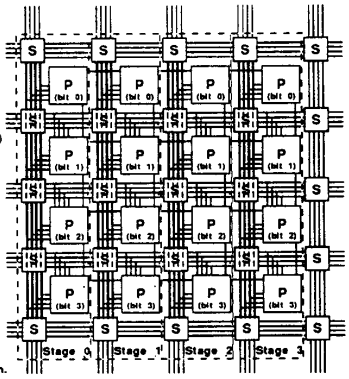

Fig. 5b. Compare-Exchange Array

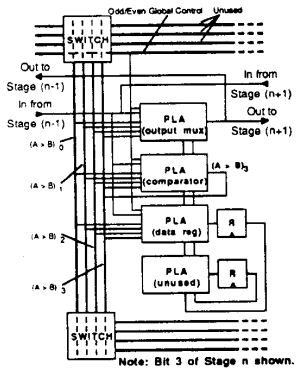

Fig. 6a. Odd-Even Sorter

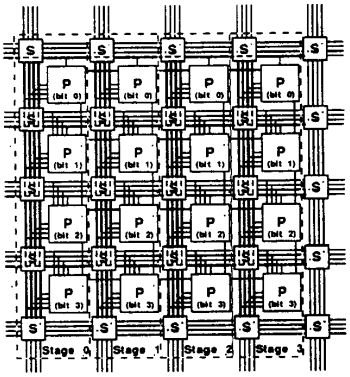

Fig. 6b. Odd-Even Array

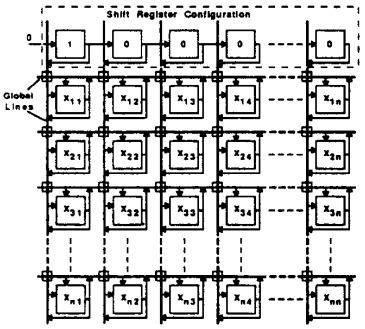

Fig. 7. Configuration For Transitive