Interconnection Networks And Parallel Memory Organizations
For Array Processing

Abhiram G. Ranade

YALEU/DCS/RR#422
September 1985

The author was supported in part by the National Science foundation (NSF) under grant number DCR-8106181, and in part by the Office of Naval Research (ONR) under grant number N00014-84-K-0043, while this work was done.

Interconnection Networks And Parallel Memory Organizations For Array Processing[1]

Abhiram G. Ranade
Department of Computer Science
Yale University
New Haven, Connecticut

Abstract

This paper presents two results on using banked memories for accessing arrays in parallel.

A class of memory organizations involving the use of a prime number of memory banks is presented. Inexpensive and fast special hardware is proposed for performing address related computation for some of the organizations. It is shown that these memory organizations have various advantages over the Prime memory system which was used in the Burroughs Scientific Processor (BSP) and which also uses a prime number of memory banks.

The second result is on interconnection networks needed to effectively use a prime number of memory banks. An interconnection network named the *Linear Permutation Network* is presented which allows most array sections to be fetched without blocking. The network has a prime number M of inputs-outputs, needs $O(M \log M)$ hardware, and has a latency of $O(\log M)$ where M is the number of memory banks. This again is an improvement over the alignment networks used in the BSP and the Burroughs Flow Model Processor (FMP).

1 Introduction

Memories have traditionally been the slowest elements in a computer system. This problem is especially severe for pipelined array processors or multiprocessors. One technique used to alleviate this is to have parallel (or interleaved) memory banks. With these, it is necessary to ensure that the data involved in a parallel operation all reside in different memory banks. Secondly, it is also necessary to provide an interconnection network that is capable of routing the data from the memory banks to the appropriate processors where they are needed. This paper presents solutions to both these problems, under the assumption that the data involved in a parallel operation are linear sections (e.g. rows, columns, diagonals etc.) of an array.

The obvious way of mapping a *physical* address onto parallel memory banks is to divide the physical address by the number of banks, use the remainder to select a bank, and the quotient as the address within the bank. The number of memory banks is chosen to be a power of two to perform the division easily and fast. With this choice, however, a linear section of an array can be accessed at the maximum

[1]The author was supported in part by the National Science foundation (NSF) under grant number DCR-8106181, and in part by the Office of Naval Research (ONR) under grant number N00014-84-K-0043, while this work was done.

memory bandwidth only if the difference of the addresses of successive elements of the section is odd [4, 9]. Notice that it is also necessary to map the *logical* address space onto the physical, and this gives rise to various *skewing schemes* for mapping arrays. A given skewing scheme may allow parallel access to a given linear section of an array. The same scheme may, however, cause another linear section to be placed entirely in a single memory bank, forcing completely sequential access. These issues have been extensively discussed in the literature [3, 4, 8, 9, 13, 14]. Budnik and Kuck [4] point out that in spite of the various skewing schemes, the choice of a power of two as the number of memory banks severely constrains the kinds of linear sections accessible at maximum memory bandwidth, and recommend the use of a prime number of memory banks. This idea was used in the the Prime memory system [10] for the Burroughs Scientific Processor (BSP). A prime number of memories was also proposed for the Burroughs Flow Model Processor (FMP) [11, 2].

The memory organizations presented in this paper also use a prime number of banks. Two general approaches are studied. In the first, various number representation schemes are suggested for representing physical addresses. Because of these schemes, the process of mapping a physical address onto a prime number of memory banks becomes trivial. The second approach is less radical and uses physical addresses represented in the conventional binary representation. This is somewhat similar to the Prime memory system, but uses an address mapping scheme that is much faster and simpler, and allows full memory utilization.

When a multiprocessor configuration is used to process arrays, it is not sufficient to just ensure that array sections required in a parallel operation are distributed over the memory banks. It is also necessary to provide an alignment network that routes the elements of the array section to the appropriate processors. The BSP used a crossbar switch to do the alignment. This allows conflict free routing but needs $O(M^2)$ hardware where M is the number of memory banks. The proposed switch for the FMP was the Omega-network [9]. This has $O(M \log M)$ hardware, but cannot route some linear sections because of conflicts in the switch. Lawrie [9] mentions the lack of a suitable interconnection network to be a major obstacle in using a prime number of memories.

This paper presents the design of such an interconnection network. This network, named Linear Permutation Network *guarantees* conflict free access to arrays stored in a memory system having a prime number of memory banks, and requires only $O(M \log M)$ hardware. In fact, the order statistics for latency, longest wire length and layout area for VLSI are the same for the Omega-network and the Linear Permutation Network.

2 Address Representation Schemes

This section describes number representation schemes that will be seen to be useful for representing addresses. These schemes have the important feature that if A is an address being represented, the representation explicitly encodes the remainder obtained when A is divided by the number of memory banks. This component of the representation is used to select the memory bank. The other components of the representation are used as the address within the selected bank. There is a one-to-one correspondence between every possible representation in the scheme and the numbers the scheme seeks to represent. Hence every memory location in the system corresponds to an address, and no memory is wasted.

Let M represent the number of memory banks. Let 2^n be the approximate size of the physical address space. Let $M' = 2^m$ be the smallest power of 2 greater than M.

Let a linear section of length l, separation k and start address v_0 be defined as the ordered set of memory locations having addresses v_0, $v_0 + k$, $v_0 + 2k$, ..., $v_0 + (l-1)k$. It can be easily seen that rows, columns or diagonals of two dimensional arrays are linear sections with the usual mapping of arrays onto memory.[2] Let the first *superword* of a linear section be the first M elements of the linear section. The s^{th} superword would be the s^{th} set of M elements of the linear section. The last superword may have fewer than M elements. If all the elements of a superword lie in distinct memory banks, then the superword is said to be *parallely accessible*. A linear section is said to be parallely accessible if all its superwords are parallely accessible.

To determine if a particular memory system allows parallel access to linear sections, it is only necessary to consider the memory bank selection procedure. All the memory organizations to be considered have the same bank selection procedure: an address A lies in memory bank A *modulo* M. Hence if a linear section has separation k, and k is relatively prime to M, then the linear section can be accessed in parallel [4, 9]. Thus M is chosen to be a suitably large prime.

2.1 Radix-M/mixed radix representation

If the address is represented in a radix M system, then the least significant digit is the remainder modulo M, and may be used to select a memory bank. The remaining digits would constitute the address within the selected bank.

For economy of representation, it is useful to have M only slightly less than a power of two.

This may be generalized to mixed radix representations [7]. For example, the radices used could be M and 2^{n-m}.

2.2 Residue Number Systems

A simple set of divisors is M and 2^{n-m}. The residue with respect to M may be used to select the memory bank; the residue with respect to 2^{n-m} would form the address within the selected bank.

Residue number systems seem to be quite well suited for representing addresses. This is because the "difficult" operations like division, magnitude comparison and overflow detection are not important in address arithmetic. The operations that are most frequently needed are addition, subtraction and perhaps comparison to zero; these are easy to implement.

Chosing M to be of the form $2^m - 1$ is very useful in this case. First, representing the residue is space efficient. Secondly, it facilitates easy conversion from an ordinary binary representation as will be discussed in section 3.2.

---

[2] Let X be a d-dimensional array and x an element of X with subscripts s_i, $i = 1 \ldots d$. Then "usual mapping" refers to all mappings in which the address of x is $a_0 + (\sum_{i=1}^{i=d} s_i a_i)$ for some a_i, $i = 0 \ldots d$.

In general, if b_i, $i = 1 \ldots d$ is a a set of constants, then the set of elements having subscripts $s'_i = s_i + (j-1)b_i$ for $j = 1 \ldots l$ forms a linear section of length l.

3 Implementation

The above ideas can be implemented in two ways. One way is to support a hardware data type for numbers (in particular, addresses) in the representations mentioned above. Thus there would have to be hardware support for arithmetic operations on addresses and perhaps some mechanisms for type conversions. If the number of memory banks M is of a certain form, then computing remainders modulo M can be done very easily with some special hardware. This allows easy type conversion from the usual binary representation to residue representation. These ideas may be used in the implementation of a memory system similar to the Prime memory system, but with certain advantages.

3.1 A hardware data type for addresses

In the particular context of array processing, it may be seen that a fairly restricted set of operations is adequate for address arithmetic. A linear section is specified by 3 parameters: the start address, the separation, and the length. It is conceivable that in most cases some or all of these would be known at compile time. If a parameter is known at compile time, it can be converted by the compiler to the selected address representation. Other parameters would have to be transformed to the address representation at run time. Thus, at most three conversions need to be performed in processing a linear section. Addresses of subsequent elements of the linear section are generated by additions, which are easily implemented.

The above scheme may be implemented in a multiprocessor or vector machine, because in either case each individual processing element operates on a linear section.[3] Thus each processor in a system that uses the above scheme must support conversion from integer to address form, and should support addition on addresses.

3.2 Dynamically converting addresses to residue number system

This section presents simple hardware for converting a number to a residue number system. The two divisors chosen are M and 2^{n-m}. The second residue is easy to compute: it is simply the least significant $n-m$ bits of the number. The first residue can be computed easily if M has a particular form.

Let M be of the form $2^m - 1$. Let $M' = 2^m$. Let A be the address to be converted. Let a_i be the digits of A in radix M'. Thus $A = \sum_i a_i M'^i$.

[3] If a linear section of separation k is operated upon by the entire configuration, each processor in a m-processor configuration operates on a linear section of separation k·m.

$$A \text{ modulo } M$$

$$= (\sum_i a_i M^i) \text{ modulo } M$$

$$= (\sum_i a_i (M+1)^i) \text{ modulo } M$$

$$= (\sum_i a_i (1 + \text{multiples of } M)) \text{ modulo } M$$

$$= (\sum_i a_i) \text{ modulo } M$$

This is easy to compute. Modulo M addition is simply 1's complement addition, i.e., an end around carry needs to be used. Thus the digits may be summed using a simple tree of m-bit adders. For example, for a 40 bit address and 31 memory banks, there are 8 radix-32 digits, each of length 5 bits. A straightforward implementation using one's complement adders (OCAs) is shown in figure 1. If simple ripple-through adders were used, this would require 15 ripple-throughs to produce a result. It may be noted that this is very similar to the problem of building a 5 bit by 8 bit multiplier, with similar cost-speed tradeoffs. For example, carry save adders may be used.

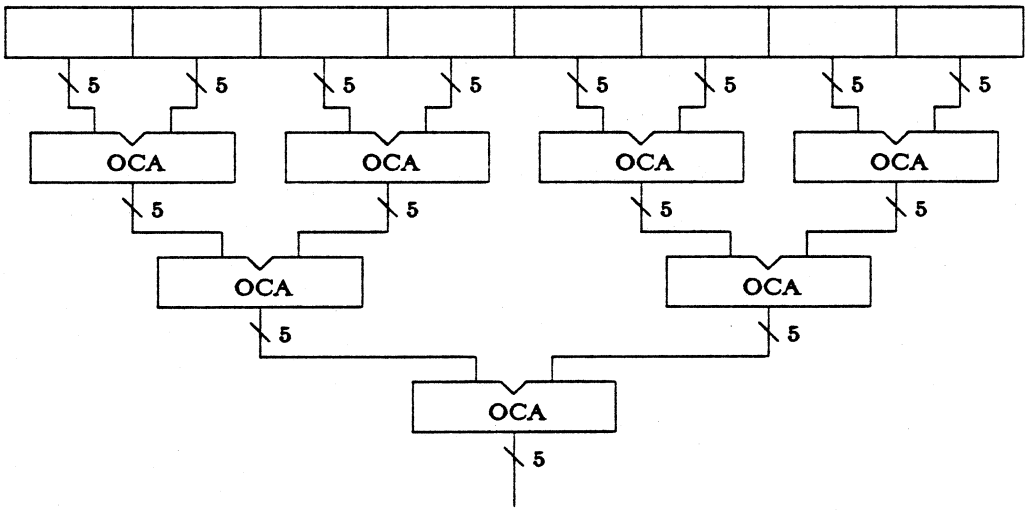

Figure 1: Computing A modulo M

Note that $A \text{ modulo } M$ computed above must be interpreted correctly, i.e., the value M should be interpreted as zero. The other residue is in the range 0 to $2^{n-m}-1$. Thus the actual size of the address space is $M \cdot 2^{n-m} = 2^n (1 - 2^{-m})$ rather than $M' \cdot 2^{n-m} = 2^n$. All the address computation is transparent to the memory, and M memory banks each of size 2^{n-m} need to be used.

The above technique of computing remainders may be generalized for divisors of the form $2^n \pm 2^m$. Suppose M is a prime of the form $M' - k$ where M' is a power of two. Then,

$$A \textit{ modulo } M$$

$$= (\sum_i a_i M^i) \textit{ modulo } M$$

$$= (\sum_i a_i (M+k)^i) \textit{ modulo } M$$

$$= (\sum_i a_i k^i) \textit{ modulo } M$$

Only $k = \pm 1$ could possibly yield a prime. For $k = -1$, the above means that odd digits need to be subtracted rather than added. This causes only a small increase in the circuit complexity. Using this, remainders with respect to 17, 257 etc. can be easily computed.

It must be mentioned that the techniques of this section can be used to compute remainders with respect to only a few numbers in the feasible range, viz. 7, 17, 31, 127, 257. For other numbers, the ideas of section 3.1 should prove to be useful.

3.3 Comparison to the Prime memory system

The ideas in this paper differ from those in the Prime Memory system also on the subject of interconnection networks, but that will be discussed later.

For the present, the most relevant aspect of the Prime memory system [10] is the address mapping mechanism. For an M memory bank system and an implementation parameter D where $1 \leq D \leq M$, an address A is mapped onto the memory as follows. The memory bank in which A lies is $A \textit{ mod } M$, while the address within the bank is $\lfloor A/D \rfloor$. While this causes at most one address to be mapped onto every memory location, unless $D = M$ certain memory locations may have no addresses mapped onto them, resulting in about $1 - D/M$ of the memory to be wasted. However, to speedup the division in computing the address within the bank, it is useful to chose D to be a power of two. Thus there is a tradeoff between speeding up division and efficiently utilising memory. For the BSP, D was chosen to be a power of two (16), and M to be a very close larger prime (17). This trivializes division but wastes 1/17th of the memory. Note, however, that it is still necessary to compute the remainder modulo M for bank selection.

If, on the other hand, the residue number system is used to represent addresses as discussed in section 2.2 the address within the bank is very simply the least significant $n - m$ bits (Recall that the divisors for the residue number system are $M = 2^m - k$ and 2^{n-m}). The size of the address space is $2^n (1 - k \cdot 2^{-m})$, instead of 2^n if addresses were represented in the usual binary system. Thus to have efficiency in address encoding it is useful to have M less than but close to a power of two. Notice, however that this is a much less important tradeoff than that concerned with memory utilization. Finally, if M is of the form $2^p \pm 1$ then the hardware of section 3.2 allows fast computation of remainder modulo M for bank selection.

To summarize, the memory organizations presented in this paper are faster than the Prime memory system as far as address related computation is concerned. Further, they allow full memory utilization. It should be obvious that the other features of the Prime memory system like remote address generation are of an orthogonal nature, and they can be implemented for the memory organizations presented in this paper as easily.

4 Linear Permutation Network

This section presents an interconnection network called the Linear Permutation Network. It has M inputs and M outputs and allows conflict free routing of messages[4] between input i and output $a \cdot i + b \; mod \; M$ if M is a prime and a is not a multiple of M. For this, the following result from number theory is required, which is stated here without proof from [12]:

> The non zero integers $1 \ldots M-1$ where M is a prime number form a cyclic group under the operation of multiplication modulo M.

Let G represent the above group. Then G has a generator. A Linear Permutation Network can be constructed for each different generator of G.

The network is formed of 3 subnetworks. The first subnetwork routes input i to output $a \cdot i \; mod \; M$, and the third maps input i to output $i + b \; mod \; M$. The second subnetwork serves to connect the other subnetworks together. Let g be the generator of G chosen for constructing the network. g will be called the *distinguished generator* of the network.

4.1 Subnetwork 1

In the first subnetwork, input 0 is directly connected to output 0. The other inputs are arranged by ascending powers of g, and form the input to a $M-1$ input circular barrel shifter, i.e., g^i is the i^{th} barrel shifter input. The non zero outputs are likewise taken from the barrel shifter, i.e., output g^i is taken from the i^{th} barrel shifter output.

Multiplication by a modulo M is equivalent to a circular shift of j where $g^j = a \; modulo \; M$. Thus the network needs to be supplied with j in order to have input i routed to output $a \cdot i \; mod \; M$. Notice that computing j given a is easy, it only requires the computation of $a \; modulo \; M$ followed by a table lookup.

4.2 Subnetwork 2

This subnetwork does not contain any active logic, but just rearranges the inputs statically. It receives inputs from subnetwork 1 geometrically ordered as 0, g^0, g^1, ..., g^{M-2}, while the outputs are produced in the order 0, 1, 2, ..., $M-1$.

4.3 Subnetwork 3

This subnetwork is simply a circular barrel shifter of M elements.

Input i is routed to output $i + b \; mod \; M$ for all i simply by doing a circular shift of b.

4.4 Barrel Shifter Implementation

Barrel shifters have been exhaustively studied e.g. [1, 15, 16]. For M inputs and outputs well known implementations have $O(log \, M)$ stages and a component count of $O(M \, log \, M)$. The basic idea is to use the i^{th} stage ($i = 0 \ldots (log \, M) - 1$) to acheive a shift of 2^i if required; whether or not shifting takes place in the i^{th} stage is decided by the i^{th} bit in the binary representation of the shift amount.

---

[4] Or alternatively, establishing non-blocking connections

The time required is O($log\ M$), but the operation can be pipelined so that new inputs may be introduced every cycle. For VLSI, [16] gives a layout that requires O(M^2) area. It is easy to show that the component count, delay and layout area are optimal to within a constant factor if pipelined operation (i.e. allowing new inputs to be introduced every cycle) is a requirement.

The issue of barrel shifter control needs some attention since most of the discussion in literature assumes that barrel shifters have centralized control. While this is perfectly acceptable and may even be preferable for SIMD [5] machines, for MIMD machines this may be an undue constraint, making distributed control schemes more desirable. A simple distributed control scheme is suggested here - the amount of shift is used as a tag to control the shifter. As mentioned above, the i^{th} bit of the tag would decide whether the particular input would be shifted in the i^{th} stage. Because there is a unique path from any input to any output, using the distributed control with all tags equal is equivalent to using centralized control, i.e., there would be no conflicts. The technique of using a single tag bit to encode one routing decision (or equivalently, one switch setting) is similar to the one used in connection with the Omega-network [6].

4.5 Area, component and time complexity

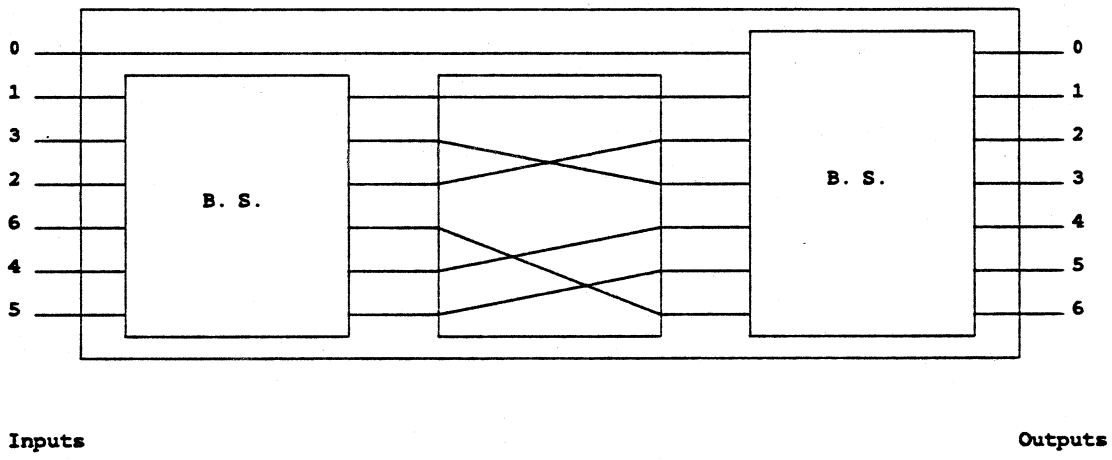

Figure 2: Linear Permutation Network for $M = 7$, using $g = 3$

Subnetwork 2 serves to rearrange outputs of subnetwork 1 so that they are in the right order for entry into subnetwork 3. Again [16] shows that this can be done in O(M^2) area.

Thus, in VLSI, all the 3 subnetworks require O(M^2) area. Further, they can be abutted together without any routing, hence the total area required is O(M^2). The component count for subnetworks 1 and 3 is O($M\ log\ M$), and subnetwork 2 only contains wiring. Thus the total component count is also O($M\ log\ M$). Similarly the total delay is O($log\ M$). Since the individual subnetworks can be pipelined, it is possible to pipeline the overall operation simply by inserting $log\ M - 1$ delays on input 0, to compensate for the fact that it does not pass through subnetwork 1.

The network is illustrated for $M = 7$ in figure 2. The generator used is 3. "B. S." stands for barrel shifter.

It may be mentioned here that the order statistics are the same as those for the Omega-network [9] and related networks.

4.6 Remarks

The network receives its inputs geometrically in an unusual order. In an application, since the processors would be connected on the side considered as the input side in the above discussion, this is unimportant if there are no direct interprocessor connections. In any case, a network similar to subnetwork 2 could be used before subnetwork 1 to statically rearrange inputs if necessary.

The network is not symmetrical. In the above discussion data was arbitrarily assumed to flow in one direction. In practice, one may have data flowing in the opposite direction or in both directions. Also the above discussion considered M inputs and M outputs. It is possible to use just a few of the inputs if required. Thus the network can also be used in configurations where there are fewer processors than memories.

5 Using the Linear Permutation Network

The network presented in the previous section can be used to interconnect processors and memories. A bidirectional Linear Permutation Network, or two unidirectional networks with data flowing in opposite directions would be used. The processors would be on the side considered to be the input in the preceding discussion, and the memories on the side considered as output. First, an SIMD kind of operation with the number of memories (M) equal to the number of processors (P) will be discussed.

Let g be the distinguished generator for the network. A single controller is used to set up the shift amounts for the two barrel shifters in the network. The controller has facilities for computing remainder with respect to M and a table which maps integers $0 < x < M$ onto integers $logmod(x)$ where $g^{\mathrm{logmod(x)}} \bmod M = x$. Let a linear section L have start address b and separation a, where a is not a multiple of M. It is easy to see that the network configuration does not vary while accessing different superwords of a given linear section, hence only the first superword need be considered. The i^{th} processor receives the i^{th} element of the superword, and its address is $a \cdot i + b$. Thus processor i needs to be connected to memory bank $a \cdot i + b \bmod M$. But this is precisely the permutation acheived by the Linear Permutation Network. The network can be controlled very simply. The controller computes $a \bmod M$ and then uses the value obtained to lookup the table to determine $j = logmod\,(a \bmod M)$. The shift value for the first subnetwork is j, and that for the third is b. If $a \bmod M = 0$, then sequential access is forced because the entire linear section lies in a single memory bank. This can easily be dealt with as a special case.

If the number of processors P is less than M, then only the P terminals on the input side are used. A linear section is broken up into smaller linear sections of length P, and then the method mentioned in the previous paragraph is used.

The Linear Permutation Network can also be used in a distributed environment. As noted in section 4.4, barrel shifters can be controlled in a distributed manner by attaching routing tags to the messages. Two tags are needed, one for each barrel shifter. For the computation of the first tag, each processor would have to have the hardware for computing remainders with respect to M, and also the ROM to

compute *logmod*. However, notice that conflict free access is possible only if processor i seeks to access memory bank $a \cdot i + b \bmod M$, for all i. This seems to suggest that all processors must operate in lockstep. However if a linear section is several superwords long and if $M = P$, then for the entire linear section every processor accesses the same memory bank. Thus, within a linear section processors need not operate in lockstep, whereas a coarse level of synchronization is necessary, i.e., all processors must operate on the same linear section. This model of execution seems compatible with the *DOALL* construct proposed in FORTRAN extensions, e.g., for the FMP [2].

6 Conclusion and future work

The use of a prime number of memories is not a new idea and it was implemented in the Prime memory system. However, the memory organization schemes introduced in this paper, in particular those based on the residue number systems have certain advantages over the Prime memory system. First, they allow full memory utilization. Secondly, if the number of memories M is of a certain form, then special inexpensive and fast hardware can be built to allow address related computations to be performed much faster than in the Prime memory system.

The above ideas are applicable for array processing in general. Thus, besides multiprocessing systems, the memory organizations described can also be used fruitfully in pipelined vector processors like the Cray X-MP, Cyber-205, or also horizontal microcode machines like the FPS-164. In these cases, just the fact that all the elements of a superword lie in different memory banks is sufficient to ensure that the maximum memory bandwidth is utilized.

The other contribution presented was the Linear Permutation Network which allows conflict free routing of array sections. The Linear Permutation Network requires $O(M \log M)$ hardware, has a latency of $O(\log M)$, and can be pipelined to operate every cycle. These characteristics are similar to those of well-known networks like the Omega-network etc.

For SIMD array processing, the Linear Permutation Network is clearly useful, and allows one to conceive of multiprocessors similar to the BSP but with many more processors.

The Linear Permutation Network can also be controlled in a distributed manner. Thus its use in an MIMD environment needs investigation. In an MIMD environment, even for array processing, the lack of tight coupling might cause problems. Further, experience with vector processors shows that not all code can be 'vectorized', and thus it is also necessary to find how the network functions when array processing is not the only activity. The Linear Permutation Network has redundant paths, and allows distributed control, and therefore its properties also as a general purpose interconnection network should be evaluated.

7 Acknowledgements

I am very grateful to my advisor Prof. Lennart Johnsson for his encouragement and detailed comments, without which this paper could not have been written. I would also like to thank Sarat Chandran for stylistic comments on an earlier version, and Doug Baldwin for help with generating the pictures in this paper.

References

[1] Jean-Loup Baer.
Computer Systems Architecture.
Computer Science Press, 1980.

[2] G. H. Barnes, S. F. Lundstrom.
Design and validation of a connection network for many-processor multiprocessor systems.
Computer :31-41, December, 1981.

[3] K. E. Batcher.
The multidimensional access memory in STARAN.
IEEE Transactions on Computers C-26:174-177, February, 1977.

[4] P. Budnik, D. J. Kuck.
The organization and use of parallel memories.
IEEE Transactions on Computers C-20:1566-1569, Decmeber, 1971.

[5] M. J. Flynn.
Very high speed computing systems.
Proc. IEEE 54:1901-1909, Decmeber, 1966.

[6] A. Gottlieb, R. Grishman, C. Kruskal, K. McAuliffe, L. Rudolph and M. Snir.
The NYU Ultracomputer - Designing an MIMD shared memory parallel computer.
IEEE Transactions on Computers C-32:175-189, Feburary, 1983.

[7] D. E. Knuth.
The art of computer programming.
Addison-Wesley, 1973.

[8] D. J. Kuck.
ILLIAC IV software and application programming.
IEEE Transactions on Computers C-17:758-770, August, 1968.

[9] D. H. Lawrie.
Access and alignment of data in an array processor.
IEEE Transactions on Computers C-24:1145-1155, December, 1975.

[10] D. H. Lawrie, C. R. Vora.
The Prime memory system for array access.
IEEE Transactions on Computers C-31:435-442, May, 1982.

[11] S. F. Lundstrom, G. H. Barnes.
A controllable MIMD architecture.
In *Proc. Int'l Conf. on Parallel Processing*, pages 19-27. August, 1980.

[12] I. Niven, H. S. Zuckerman.
An introduction to the theory of numbers.
John Wiley & Sons Inc., 1972.
page 60, problem 12.

[13] H. D. Shapiro.
Theoretical limitations on the use of parallel memories.
IEEE Transactions on Computers , December, 1975.

[14] R. C. Swanson.
Interconnections for parallel memories to unscramble p-ordered vectors.
IEEE Transactions on Computers C-23:1105-1115, November, 1974.

[15] J. Thornton.
Design of a computer. The Control Data 6600.
Scott, Foresman and Co., Glenview, Ill., 1970.

[16] J. D. Ullman.
Computational aspects of VLSI.
Computer Science Press, 1984.