A Projective Geometry Architecture for Scientific Computation

Bharadwaj S. Amrutur Rajeev Joshi Narendra K. Karmarkar
AT&T Bell Laboratories, Murray Hill, NJ 07974

Abstract.

A large fraction of scientific and engineering computations involve sparse matrices. While dense matrix computations can be parallelized relatively easily, sparse matrices with arbitrary or irregular structure pose a real challenge to designers of highly parallel machines. A recent paper by Karmarkar [KAR 91] proposed a new parallel architecture for sparse matrix computations based on finite projective geometries. Mathematical structure of these geometries plays an important role in defining the interconnections between the processors and memories in this architecture, and also aids in efficiently solving several difficult problems (such as load balancing, data-routing, memory-access conflicts, etc.) that are encountered in the design of parallel systems. In this paper we discuss some of the key issues in the system design of such a machine, and show how exploiting the structure of the geometry results in an efficient hardware implementation of the machine. We also present circuit designs and simulation results for key elements of the system: a 200 MHz pipelined memory; a pipelined multiplier based on an adder unit with a delay of 2ns; and a 500 Mbit/s CMOS input/output buffer.

1 Introduction

A large fraction of scientific and engineering computations involve sparse matrices. While dense matrix computations can be parallelized relatively easily, sparse matrices with arbitrary or irregular structure pose a real challenge to designers of highly parallel machines. A recent paper by Karmarkar [KAR 91] proposed a new parallel architecture for sparse matrix computations based on mathematical objects called finite projective geometries. The structure of these geometries defines the interconnections between processors and memories, and also helps solve the difficult tasks of load balancing, avoiding conflicts, bandwidth matching, data routing etc. This paper describes the architecture and discusses some of the key issues involved in the implementation of a typical machine with this architecture. We show how exploiting the mathematical structure of the underlying geometry results in an efficient hardware implementation of the machine. We also present circuit designs and simulation results for key elements of the system: a 200 MHz pipelined memory; a pipelined multiplier based on an adder unit with a delay of 2ns; and a 500 Mbit/s CMOS input/output buffer.

2 Description of the architecture

2.1 Computational environment

The proposed architecture is meant to be used as an attached accelerator to a general purpose host processor. The accelerator and the host share a global memory system (see Fig 1). The main program runs on the host, while computationally intensive subroutines are to be executed on the attached accelerator. The shared memory consists of partitioned memory modules, which are shared by the processors in the accelerator over an interconnection network. Since the host and the accelerator share the memory system, it is not necessary to communicate large amounts

0-8186-2967-3/92 $03.00 © 1992 IEEE

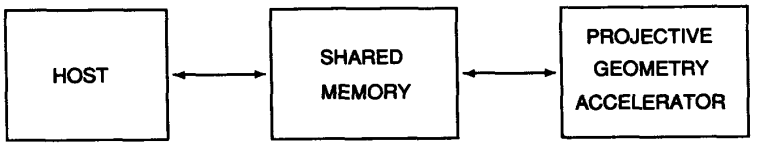

Figure 1: Computational Environment

of data between the two on a separate I/O bus. Only certain structural information such as base addresses of arrays need to be communicated from the host processor to the attached accelerator before invoking a subroutine to be executed on the attached accelerator.

2.2 Finite projective geometries

We now describe mathematical constructs known as finite projective geometries, which play an important role in defining the architecture. In later subsections, we will discuss how the structure of these geometries aids in efficiently solving several difficult problems encountered in the design of parallel systems, such as load balancing, data routing, memory access conflicts, etc.

Consider a finite field, $\mathcal{F}_s = GF(s)$ which has $s = p^k$ elements, where p is a prime, and k a positive integer. A projective geometry of dimension d, denoted by $\mathcal{P}^d(\mathcal{F}_s)$, is the set of all one dimensional subspaces of the $(d+1)$-dimensional vector space $\mathcal{F}_s^{d+1}$ over the field $\mathcal{F}_s$. A one dimensional subspace of $\mathcal{F}_s^{d+1}$ generated by $\mathbf{x}$, $\mathbf{x} \in \mathcal{F}_s^{d+1}$, $\mathbf{x} \neq 0$, is the set of all nonzero elements of the form $\lambda\mathbf{x}$, $\lambda \in \mathcal{F}_s$. These subspaces are the points of the projective geometry. Since there are $(s^{d+1}-1)$ nonzero elements in $\mathcal{F}_s^{d+1}$, and $(s-1)$ nonzero elements in $\mathcal{F}_s$, the number of points in the geometry, n_d, is given by $(s^{d+1}-1)/(s-1)$. Similarly, an m-dimensional subspace of the projective geometry consists of all one dimensional subspaces of an $(m+1)$-dimensional subspace of $\mathcal{F}_s^{d+1}$. If $\{\mathbf{b_0}, \mathbf{b_1}, \ldots \mathbf{b_m}\}$ forms a basis of this vector subspace, then the elements of the subspace are of the form

$$\sum_{i=0}^{m} \alpha_i \mathbf{b_i} \ , \quad where\ \alpha_i \in \mathcal{F}_s$$

The number of elements in the subspace, n_m, is given by $(s^{m+1}-1)/(s-1)$. The set of all m-dimensional subspaces of $\mathcal{P}^d(\mathcal{F}_s)$ is denoted by Ω_m. For $n \geq m$, define the function

$$\phi(n,m,s) = \frac{(s^{n+1}-1)(s^n-1)\cdots(s^{n-m+1}-1)}{(s^{m+1}-1)(s^m-1)\cdots(s-1)}$$

Let $0 \leq l < m \leq d$. Then the number of l-dimensional subspaces of $\mathcal{P}^d(\mathcal{F}_s)$ contained in a given m-dimensional subspace is given by $\phi(m,l,s)$, and the number of m-dimensional subspaces of $\mathcal{P}^d(\mathcal{F}_s)$ containing a given l-dimensional subspace is given by $\phi(d-l-1, m-l-1, s)$.

2.3 Interconnection scheme

Given a finite projective geometry of dimension d, we define the architecture as follows. Choose a pair of dimensions $0 \leq d_m < d_p \leq d$. Put processors in the system in a one-to-one correspondence with subspaces of dimension d_p, and put memory modules in a one-to-one correspondence with subspaces of dimension d_m. Form a connection between a processor and a memory module iff the subspace corresponding to the processor contains the subspace corresponding to the memory module. From the discussion above, the number of processors in the system will be $\phi(d, d_p, s)$, and the number of memory modules will be $\phi(d, d_m, s)$. Each processor will be connected to $\phi(d_p, d_m, s)$ memory modules, and each memory module will be connected to $\phi(d - d_m - 1, d_p - d_m - 1, s)$ processors. If we are interested in a *symmetric* architecture, with an equal number of processors and memory modules, then we must choose d_p and d_m such that $d = d_p + d_m + 1$. A more detailed discussion of projective geometries, along with some examples of interest, appears in [KAR 91].

2.4 Load assignment

With the above correspondence between subspaces of the geometry and processors (and memories), the assignment of computational load to processors can automatically be done at a fine-grain level. To illustrate, consider a binary operation

$$a \leftarrow a \circ b$$

Suppose operand a is in memory module M_i, and b is in memory module M_j. Then we associate an index pair (i, j) with this operation. (Similarly, we associate an index triplet with a ternary operation). The processor P_l responsible for doing this operation is determined by a function f that depends on the geometry :

$$l = f(i, j)$$

Thus operations having the same associated index pairs (or triplets) always get assigned to the same procesor. Furthermore, the function f is compatible with the structure of the geometry i.e., processor P_l has connections to memory modules M_i and M_j.

2.5 Perfect patterns and perfect sequences

We now introduce the concepts of *perfect patterns* and *perfect sequences* which restrict the combinations of words that can be accessed in one cycle. These combinations are designed so that no conflicts can arise in either accessing the memories or in sending the accessed data through the interconnection network. We define a perfect access pattern for a symmetric architecture based on a 4-dimensional geometry. In this architecture, memory modules are in a one-to-one correspondence with lines, and processors are in a one-to-one correspondence with planes. A memory module is connected to a processor iff the line corresponding to the memory module lies on the plane corresponding to the processor.

Suppose the number of lines (and hence the number of planes) is n. A perfect access pattern P is a collection of n non-collinear triplets,

$$P = \{ (p_i, q_i, r_i) \mid p_i, q_i, r_i \in \Omega_0, dim\langle p_i, q_i, r_i \rangle = 2, i = 1 \ldots n \}$$

satisfying the following properties:

1. Let $u_i, i = 1 \ldots n$ denote the lines generated by the first two points of each triplet

$$u_i = \langle p_i, q_i \rangle$$

Then the collection of lines $\{u_1 \ldots u_n\}$ forms a permutation of all the lines of the geometry.

2. Let $v_i, i = 1 \ldots n$ denote the lines

$$v_i = \langle q_i, r_i \rangle$$

Then the collection of lines $\{v_1 \ldots v_n\}$ forms a permutation of all the lines of the geometry.

3. Let $w_i, i = 1 \ldots n$ denote the lines

$$w_i = \langle r_i, p_i \rangle$$

Then the collection of lines $\{w_1 \ldots w_n\}$ forms a permutation of all the lines of the geometry.

4. Let $h_i, i = 1 \ldots n$ denote the planes

$$h_i = \langle p_i, q_i, r_i \rangle$$

Then the collection of planes $\{h_1 \ldots h_n\}$ forms a permutation of all the planes of the geometry.

The table below depicts a perfect access pattern for the 4-dimensional geometry.

	Triplet of Points	Triplet of lines			Planes
1	(p_1, q_1, r_1)	$u_1 = \langle p_1, q_1 \rangle$	$v_1 = \langle q_1, r_1 \rangle$	$w_1 = \langle r_1, p_1 \rangle$	$h_1 = \langle p_1, q_1, r_1 \rangle$
2	(p_2, q_2, r_2)	$u_2 = \langle p_2, q_2 \rangle$	$v_2 = \langle q_2, r_2 \rangle$	$w_2 = \langle r_2, p_2 \rangle$	$h_2 = \langle p_2, q_2, r_2 \rangle$
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
n	(p_n, q_n, r_n)	$u_n = \langle p_n, q_n \rangle$	$v_n = \langle q_n, r_n \rangle$	$w_n = \langle r_n, p_n \rangle$	$h_n = \langle p_n, q_n, r_n \rangle$

When an operation having (p_i, q_i, r_i) as the associated index triplet is performed, the three memory modules accessed correspond to the three lines u_i, v_i, w_i, and the processor performing the operation corresponds to the plane h_i. Hence it is clear that if we schedule n operations whose associated triplets form a perfect pattern to execute in parallel, then

1. there are no read or write conflicts in memory accesses
2. there is no conflict in the use of processors
3. all the processors are fully utilized
4. the memory bandwidth is fully utilized

Hence the name *perfect pattern.*

Since there is a connection between a memory module M and a processor P iff the line α corresponding to M is contained in the plane β corresponding to P, we can denote a connection by the ordered pair (α, β). Let C be the collection of all processor-memory connections:

$$C = \{(\alpha, \beta) | \alpha \in \Omega_1, \beta \in \Omega_2, \alpha \subseteq \beta\}$$

Then, if (p_i, q_i, r_i) is a triplet in a perfect pattern P, u_i, v_i, w_i are the corresponding lines, and h_i is the corresponding plane, we say the perfect pattern P *exercises* the connections (u_i, h_i), (v_i, h_i) and (w_i, h_i).

A sequence of perfect patterns is called a *perfect sequence* if each connection in C is exercised the same number of times collectively by the patterns in the sequence. It follows that if such perfect sequences form the basis for instructions executed on the architecture, it leads to a uniform utilization of even the wires connecting processors and memories. It is then possible to connnect the processors and memories so that the number of wires in the system only grows *linearly* with the number of processors. A definition of perfect patterns for 2-dimensional geometries, and a discussion on how to generate perfect patterns based on automorphisms of the underlying groups, appears in [KAR 91].

2.6 Instructions for the system and its elements

There is a 3-level hierarchy of instructions in the system. At the lowest level, an *elementary instruction* is an instruction that each individual hardware element follows – for a processor this is an add, multiply, no-op etc.; for a memory module it consists of an address and a read/write flag; for a switch element in the interconnection network it is a pattern that encodes its configuration in a given cycle.

At the next level, a *system-level instruction* is a conflict-free collection of elementary instructions – one for each element of the system, grouped together on the basis of a perfect pattern. Thus a system-level instruction specifies the operation of the system as a whole. A *Perfect Sequence of system instructions (PS-Instruction)* is a list of system instructions based on a perfect sequence of the geometry.

At the top level, a collection of third-level instructions, each consisting of a list of system-instructions, implement a high-level operator acting on a large data structure e.g, a vector, a matrix, linked list etc. Unlike the other two levels, this level of instructions is not hard-wired and can be changed by loading different instructions into special instruction sequence memories (see below). The machine can be specialized for a new application by designing, compiling and loading a new third-level instruction set suitable for that application. These third-level instructions are designed to exploit massive parallelism at a fine-grain level, using the structure of the geometry. The main program runs on the host, and passes control to the accelerator whenever it encounters a third-level instruction.

A *compiler* running on the host or another machine (see [DKR 91]) takes a data-flow graph of a third-level instruction as input, and uses its knowledge of perfect patterns and sequences to produce as output a collection of programs, one for each element of the system. Each program consists of a list of elementary instructions for the associated element. Each hardware element has an associated local *instruction sequence memory* for storing the instruction sequence to be used by that element. The initial loading of these instruction sequences is done at the start of each new subroutine by the host. Once loaded, the instruction sequence may typically get executed many times before being overwritten by the instruction sequence for some other subroutine. Depending on the size of these instruction memories, instruction sequences for several subroutines can reside simultaneously in these memories.

2.7 Memory system

The memory system of the accelerator is partitioned into n_m modules $M_1 \ldots M_{n_m}$. The set of words that can be accessed in a single machine cycle must be based on a perfect pattern. Thus the type of memory access possible is between the two extremes of random and sequential access, and could be called "structured" access. The total set of allowed combinations is still

powerful enough so that a sequence of such accesses is as effective as random access. On the other hand, such a structured access memory has a bandwidth much higher than a random access memory implemented using comparable technology.

2.8 Processing elements

The processing elements in the accelerator are pipelined arithmetic and logic units. The instruction sequence to be followed by a processor can also contain *move(i,j)* instructions which move operands from memory module M_i to M_j, as well as "no-ops". In the case of operations that take several machine cycles to produce a result, "execution" of the corresponding instruction means initiation of the operation. The compiler must ensure that the pipeline delay in availability of the result is taken into account when the data-flow graph is processed. The instruction sequence to be followed by a processor is stored in its local instruction memory. This instruction sequence does not contain addresses of operands, but only the type of operation to be performed (including moves and no-ops). There is no concept of "fetching" an operand – instead the processor simply operates on whatever operands are available at its input ports at the start of each machine cycle. There is also a local data memory associated with each processor used to store data operands used only by that processor, thus reducing traffic on the interconnection network. For instance, in an application such as multiplication of a sparse matrix by a vector, the matrix elements can be stored in the local memories of processors, and the input and output vectors can be stored in the shared, partitioned global memory.

2.9 Application example

The paper by Karmarkar [KAR 91] illustrated some of the key features of the architecture by describing how to map some commonly used operations in scientific computation (viz., multiplication of a vector by a sparse matrix, and the pivoting step in Gaussian elimination) onto the architecture. In this paper, we look at another application viz., the join of two binary relations in a relational database. As illustrated below, this application has a property similar to the Gaussian elimination example, which can be exploited to obtain fine-grain parallelism on this architecture.

Consider two binary relations R and S. Let r and s be tuples in R and S respectively. We denote the components of r by $(r[1], r[2])$. A computation of the natural join of relations R and S, denoted by $R\,|X|\,S$, brings together tuples r and s only if they match on the join attribute. Thus $(r[1], r[2])$ and $(s[1], s[2])$ must have at least one element in common. This property can be exploited as follows:

1. For each tuple r in R, map each of its components to a point in the projective space by means of some hash functions f_1, f_2

$$\alpha = f_1(r[1]), \beta = f_2(r[2]) \quad \alpha, \beta \in \Omega_0$$

   We associate each tuple with the line determined by $\langle \alpha, \beta \rangle$. If $\alpha \neq \beta$, this line is unique. If $\alpha = \beta$, we can associate the tuple with any line passing through point α. Do the same for the tuples in S.

2. Define an architecture in which memory modules are in one-to-one correspondence with one-dimensional subspaces i.e., lines, and processors are in one-to-one correspondence with two-dimensional subspaces i.e., planes. Note that the memory system of the architecture would typically be a collection of disks, and each memory module would be a disk.

3. Then, for each tuple in R (and S), assign it to the memory module (i.e., the disk) corresponding to the line associated with the tuple. With this assignment, each relation is distributed across the disks in the system, and not stored on a single disk, as would be case in a straightforward implementation. The portion that is stored on each disk is determined according to the rules described here.

4. Consider tuples $r \in R, s \in S$. Let the pair of points associated with the attributes of tuple r be (α_r, β_r), and the pair of points associated with tuple s be (α_s, β_s). From the discussion above, tuples r and s are brought together only if $dim\langle \alpha_r, \beta_r, \alpha_s, \beta_s \rangle \leq 3$ (since the tuples must match of the join attribute and so either $\alpha_r = \alpha_s$, or $\beta_r = \beta_s$). In general, $dim\langle \alpha_r, \beta_r, \alpha_s, \beta_s \rangle = 3$, and so the tuples r and s determine a plane, say δ, in the projective space. Then the task of bringing tuples r and s during computation of the join is assigned to the processor corresponding to the plane δ. Since the lines $\langle \alpha_r, \beta_r \rangle$ and $\langle \alpha_s, \beta_s \rangle$ lie on the plane δ, the processor has connections to the memory modules where r and s are stored.

5. Recall that, for a symmetric architecture, d, the dimension of the projective space, is given by $d_m + d_p + 1$, where d_m, d_p are the dimensions of the subspaces corresponding to the memory modules and processors respectively. Thus, if we are interested in having an equal number of processors and memory modules, we would choose a four-dimensional geometry i.e., $d = 1 + 2 + 1 = 4$.

Note that the fundamental property exploited here is that two tuples are brought together during join computation only if they match on the value of at least one component. The scheme we have outlined above is applicable to other computations which have this property e.g., the pivoting step in Gaussian elimination, and computing the transitive closure of a directed graph.

3 Some system design issues

After examining various tradeoffs, we partitioned the system into building blocks. These are the processing elements (PEs) and the memory modules (MMs) shown in Fig 2. The switch is not a separate element, but is distributed across the system, and appears as part of these building blocks. These blocks could be combined on a single chip, or implemented as separate chips. The architecture allows simple designs for the memory modules and the processing elements. For instance, a processor can be a simple pipelined ALU, while the memory can be fully pipelined. The ALU instruction memory stores the instruction sequence for the ALU and its associated local data memory. The sequence of addresses for the data memory is stored in its associated sequencing unit. Since this sequence is known in advance, full pipelining is possible in decoding addresses, accessing bits in memory and, in case DRAMs are used, in dynamic error detection and correction. We present a design for a pipelined SRAM with a cycle time of 5ns in §4.1. The sequence memory for each distributed switch element contains a list of control words that determine its configuration during each cycle.

As discussed in §2.5, instructions based on perfect sequences ensure that each processor memory connection is exercised the same number of times. We exploit this property by grouping several machine cycles together in "frames" corresponding to the length of a perfect sequence. Data is sent over all connections simultaneously, with the data movement required for the computation in the current frame being completed in the preceding frame. This approach requires connections of reduced width. Consequently, overall system design can be quite compact. To illustrate this point, consider the problem of interconnecting a symmetric architecture based on a 4-dimensional geometry and having 155 processors and 155 memory

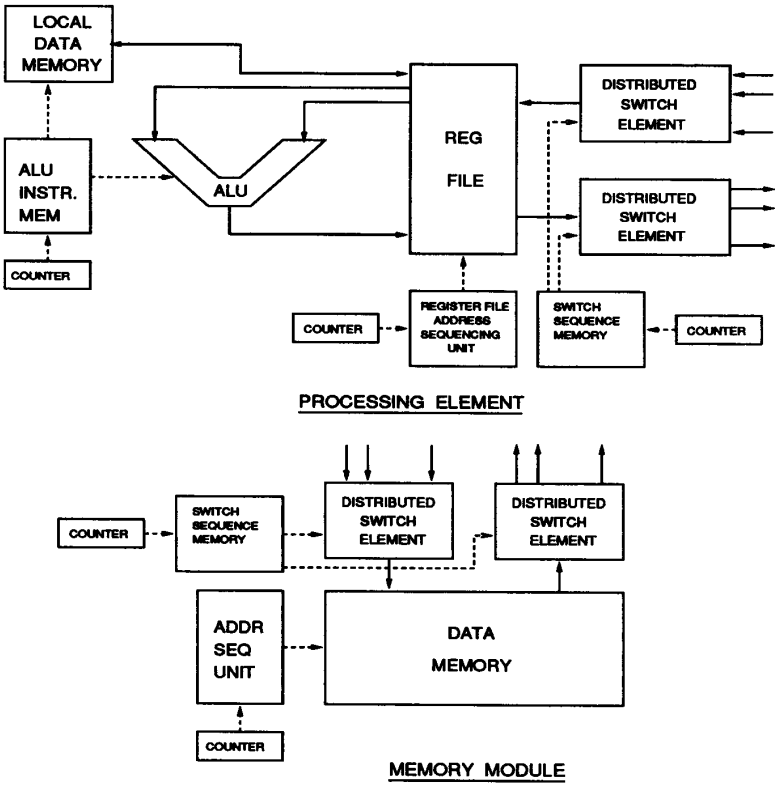

Figure 2: Building Blocks of the System

modules. Given the connections between processors and memories, we can, for a given arrangement of these elements, estimate the maximum wiring density on the board. We do

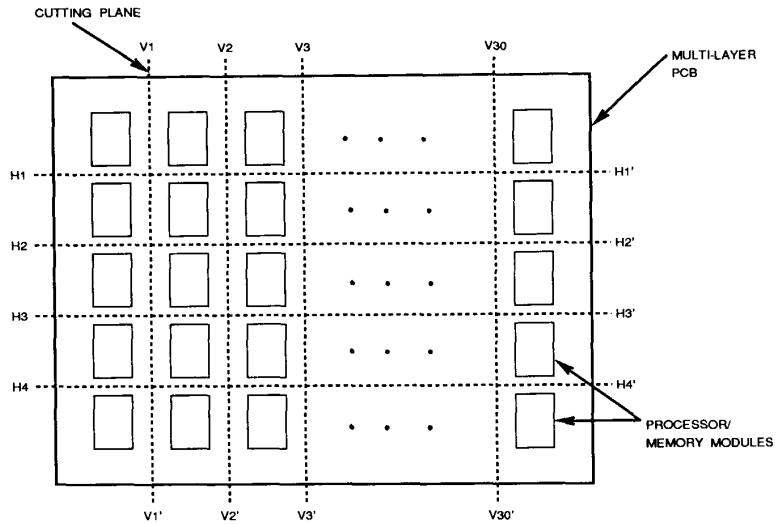

Figure 3: Arrangement of elements for wiring density estimates

this by counting the total number of wires crossing each of the cutting planes shown in Fig 3. The wire count for the horizontal cut $H_i - H_i'$ is the total number of wires crossing the plane $H_i - H_i'$ in the figure. Similarly for wire counts of vertical cuts $V_i - V_i'$. By using an interior point method, we showed that the lower bound on the number of wires crossing the central cuts $(V15, V16\&H2, H3)$ was 940. By using another interior point method for finding optimal cuts in a graph, we found a solution with 1036 wires crossing the central cuts. Even for a simple scheme shown in which the elements were laid out in a natural 5×31 arrangement based on the group theoretic structure of the geometry, results showed that the maximum wire count, across the central cut, is 1488 wires, which is within a factor of two of the lower bound (see Fig 4). (Here we have assumed that each processor-memory connection has 4 wires.) The results imply a required cumulative wiring density of 100 wires/inch for all the layers on a $15'' \times 15''$ printed circuit board(PCB), which can be easily achieved on available multilayer PCBs. For a synchronous parallel machine, such a single board design is preferable, because clocking and routing are considerably simpler, and higher speeds are possible. The fact that the number of wires in the system grows only linearly with the number of elements also means that we can afford to have dedicated point-to-point unidirectional connections between processors and memories, rather than connect them using a network based on shared buses. The advantage of having point-to-point connections is that each wire in a link can be constructed as a transmission line with a controlled termination impedance, and high bit rates are possible. In §4.3 we discuss a design for a high speed I/O buffer done entirely in silicon which could be used for such a high speed interconnection.

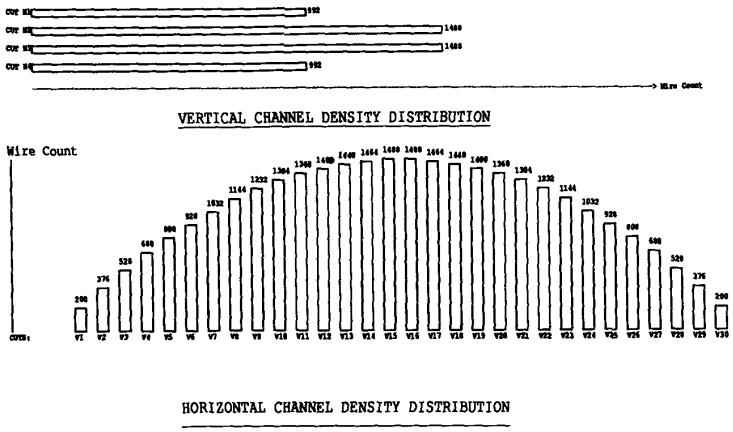

Figure 4: Wiring density estimate results

4 Design of some key elements

4.1 Pipelined SRAM design

Pipelining memories has traditionally been difficult because the sequence of addresses that will be generated by a processor is not known a priori. Since address sequences for memories in this architecture are known before computation is initiated, pipelining is possible, and cycle times can be considerably reduced. We have designed a pipelined SRAM that decodes addresses in a hierarchical fashion, employing latches in decoding and read/write stages to achieve high throughput rates. Most of these ideas will be applicable in the case of DRAM design also. ADVICE [LWN 80] simulations for a 4 Kbit test memory in 0.9μ CMOS with extracted parasitics are shown in Fig 5. In the figure, *CLK* is the clock, *EQU* is a pulse that equalizes the bitlines, *WL* is the word line pulse, and *D* and *DN* are the complementary outputs of the sense amplifier. The simulations predict a 5 ns cycle time under worst-case conditions, and thus indicate such a pipelined SRAM could meet the bandwidth requirements of a processor.

4.2 Processor design

Since the processor does not fetch operands, but simply operates on the two operands that are available at its input ports at the beginning of each cycle, it can be a simple floating point arithmetic and logic unit. A critical section of a floating point unit is the multiplier, so we have concentrated on the design of a fast pipelined multiplier, which we discuss next.

Binary multiplication in most floating point processors is implemented in two stages viz. generation of the partial products, and addition of these partial products. For multiplication of

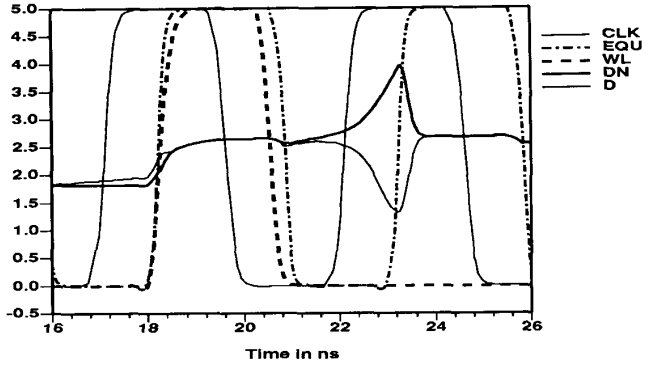

Figure 5: Simulation results for pipelined SRAM

large numbers, most of the time is spent in this addition of partial products. Several methods have been used in the past, of which Wallace tree structures [WAL 64] using Carry Save Adders (CSAs) as the basic building blocks have been the most common. Each CSA takes 3 inputs (two operands & a carry-in) and produces 2 outputs (a sum & a carry-out), so the number of stages in the Wallace tree for addition of n partial products is $\lceil log_{1.5}(n) \rceil$. Other reduction schemes have been proposed, using building blocks called *counters* [SOM 91]. A *(p,q)* counter has p inputs and q outputs, and the number of stages in the Wallace tree for the addition of n partial products is $\lceil log_{p/q}(n) \rceil$. While counters reduce the routing complexity of the Wallace tree by reducing the number of stages, building high speed monolithic high order (many input) counters has been a challenging task, because speeds of CMOS circuits reduce as the number of inputs increases.

We have avoided the use of high fan-in CMOS circuit by employing a different scheme for representing the partial products. The design we describe employs a novel adder unit which takes in four 2-bit operands, and produces two 2-bit results. In a certain sense, one could think of this multiplier as operating in base 4. As shown in Fig 6, each adder unit takes four inputs (each

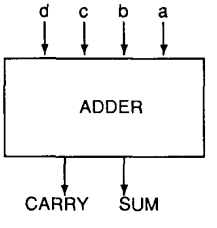

Figure 6: 4-input adder

a base 4 digit), and produces a sum and a carry (each a base 4 digit). Thus the adder reduces the number of partial products by a factor of 2 in every stage, and the depth of the tree for addition of n partial products is $\lceil log_2(n) \rceil$. For single precision floating point arithmetic, 5 adder stages would be needed for multiplication of mantissas. Each digit is represented in a fully decoded

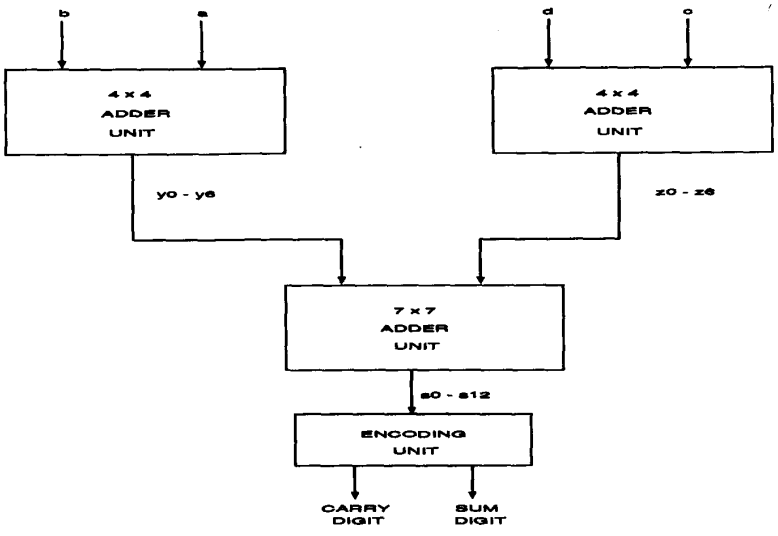

Figure 7: Structure of a 4-input adder

form using 4 wires, exactly one of which may be active. The structure of the adder is shown in Fig 7. Two 4×4 adder units add a pair of input digits, each producing a sum in base 7. A 7×7 adder unit then adds these two intermediate sums, and the resulting sum is reencoded as a pair of base 4 digits by the encoder unit shown. Each 4×4 adder unit is implemented with an array of pass transistors as shown in Fig 8. A similar scheme is used for the 7×7 adder unit. We have used transistors with low threshold voltages to ensure that the outputs of the 7×7 adder unit (which are $2V_T$ below V_{DD}, where V_T is the threshold voltage) are still large enough to drive the gates which reencode the output. There are many ways of combining the adders to form a single pipeline stage. In our design, we have combined four adder units to form a single pipeline stage, as shown in Fig 9. Results of ADVICE simulations of the circuit with extracted parasitics indicate that the delay for a single adder is less than 2ns, and the delay for the pipeline stage formed from 4 adders is less than 10ns (under worst case conditions). These results are plotted in Fig 9, where D is one of the inputs to the first adder in the chain, Y & Z are intermediate outputs of its 4×4 units, and S is an output of its 7×7 unit. SUM is an output of the encoder after the fourth adder in the chain. These results indicate that the multiplier could work at 100 MHz, and could be used with the pipelined SRAM described in §4.1. Designs for the partial product generation unit, the final carry lookahead adder, normalization circuit and exponent adder will be presented in a subsequent paper.

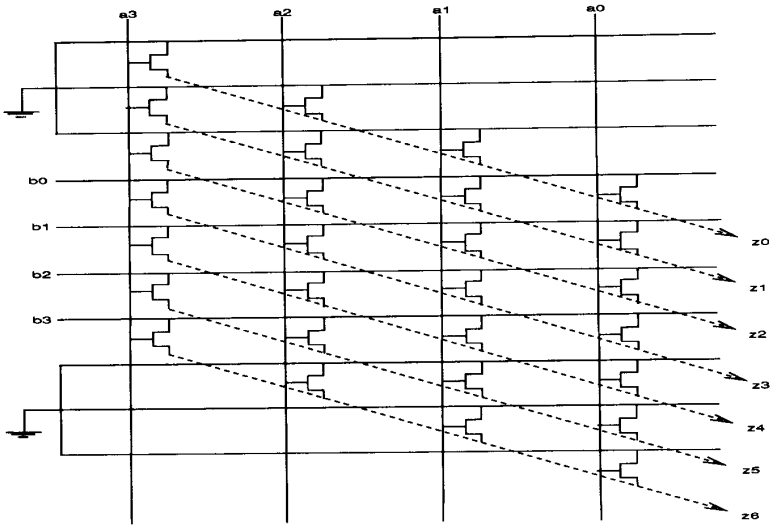

Figure 8: 4 X 4 Adder Unit

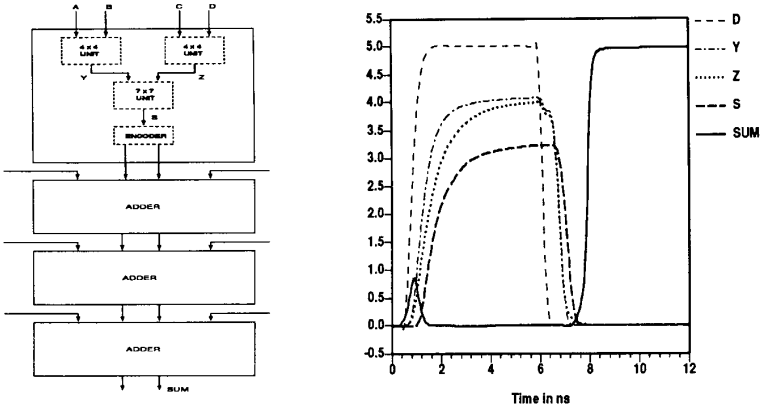

Figure 9: Simulation results for a chain of four adders

4.3 Switch Design

As mentioned in §3, the switch is not a separate element, but is distributed across the system. With each processing and memory element, there is an associated distributed switch element. This element consists of a multiplexer, a demultiplexer, and circuitry for bit serialization and synchronization of incoming data with system clock. We discuss here the switch design for

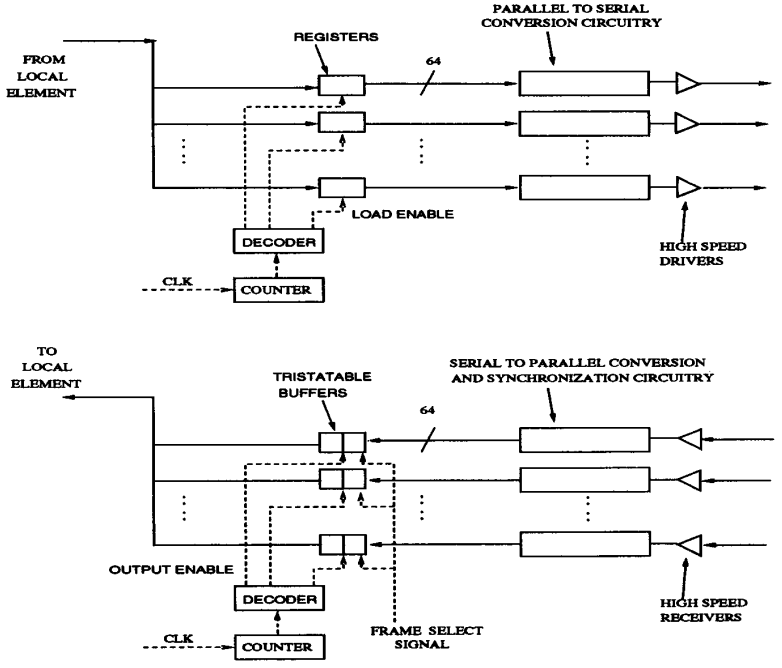

Figure 10: Distributed Switch Elements

a symmetric architecture having 57 processors (and 57 memories). In this architecture, each processor (memory) is connected to eight memories (processors). Recall that instructions based on perfect sequences ensure that each processor-memory connection is exercised the same number of times. If the perfect sequences are designed so that each connection is exercised exactly once, then each perfect sequence will consist of 8 perfect patterns. Thus each frame (§3) will comprise of 8 machine cycles. Since data being sent in the current frame will be used only in the next frame, each port can be accessed in a cyclic manner. Thus each port is accessed every eighth cycle and data can be time multiplexed over eight cycles. So, for instance, if data words were 64 bits wide, each port could be only 8 bits wide. Fig 10 shows the design for a distributed switch element. Since the buffer are accessed cyclically, the control needed for the switch is simple, and counters can be used instead of a switch sequence memory. The frame select signal at the receiving end selects the set of buffers which have been filled in the preceding

frame. The synchronization circuitry at each receiving port recovers the transmission clock and synchronizes the received data with the local system clock. The number of wires per connection can be further reduced if high speed bit serial links are used. In the above example for instance, if a 400 Mbit/s point-to-point link were used, then two links per port would meet the bandwidth requirements of processors and memories operating at 100 MHz. If each link used a differential pair of wires, the number of wires per element would be 16, resulting in considerable reduction in the wiring.

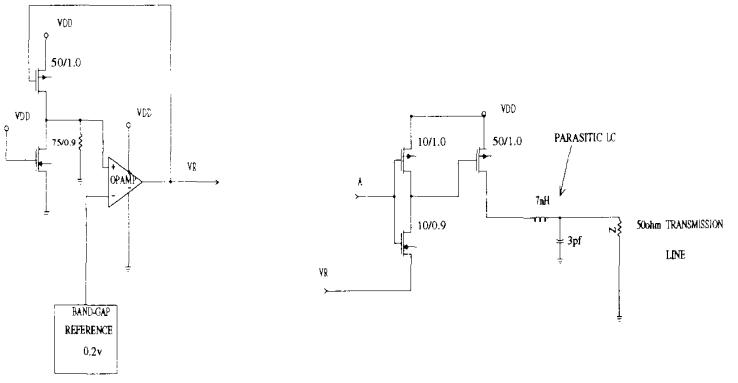

Figure 11: Transmitter Design

We have been experimenting with designs of various high-speed transmission and receiving circuits. A primary consideration in transmitter-receiver design is the reduction of power dissipation. This has prompted us to use very low voltage swings for data transmission. Figs 11 and 12 present circuit designs for a transmitter and receiver which use 200 mV signal swings. Low voltage swings also generate less noise. At the receiving end the signal is directly coupled into the source terminal of the NFETS. The PFET-NFET pair is biased at its trip point by another identical pair resulting in a high amplification. This receiver is susceptible to transistor mismatches, and so special care was taken in the layout of the circuit. The pull down NFET to ground also serves as a matching termination resistance for the transmission line. The gate voltage of this device is controlled by an opamp to track temperature and process variations and provide an accurate termination impedance under all conditions. At the transmitting side, the driving PFET is sized so that the voltage presented on the line is 200 mV peak to peak. The gate voltage of this PFET is also controlled by an opamp and is adaptively varied to maintain the same voltage swing under various operating conditions. Simulation results of the circuit with associated parasitics extracted from the layout shown in Fig 13 indicate that the circuit can operate at 500 Mbit/s under worst case conditions.

5 Future Work

We are looking into other receiver and transmitter topologies with the aim of reducing the power dissipation, and mismatch sensitivity. The results of these investigations, and designs for the remaining sections of the multiplier and the floating point adder, will be presented in a

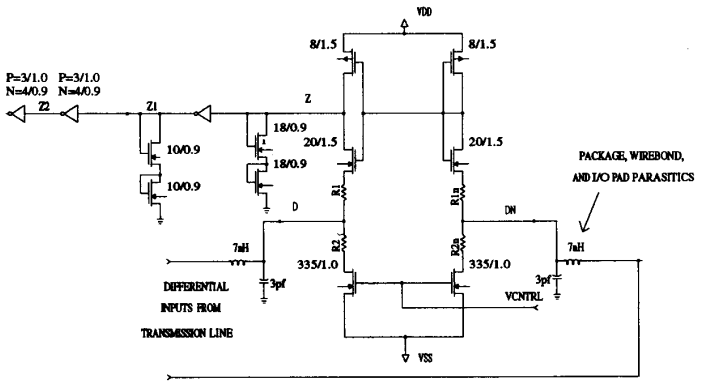

Figure 12: Receiver Design

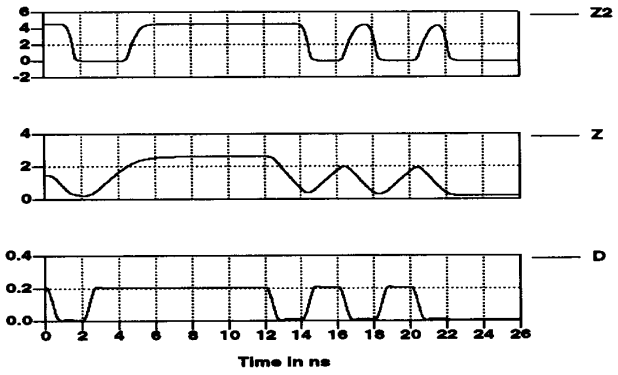

Figure 13: Simulation results for the receiver circuit

subsequent paper. Other issues such as clock synchronization, heat dissipation, design of the interface to the host, will also be addressed in subsequent papers.

Acknowledgements. We thank Joe Condon, Thad Gabara, T.R.Viswanathan, J. Fischer,King Tai, and A. J. Rainal for helpful discussions.

References

[DKR 91] I. S. Dhillon, N. K. Karmarkar, K. G. Ramakrishnan, *An Overview Of the Compilation Process for a New Parallel Architecture*, Proceedings of the Fifth Canadian Supercomputing Conference, N.B., Canada, Jun 1991

[KAR 91] N. K. Karmarkar, *A New Parallel Architecture for Sparse matrix Computation based on Finite Projective Geometries*, Proceedings of Supercomputing '91

[LWN 80] L. W. Nagel, *ADVICE for Circuit Simulations*, Proc. Intl. Symp. on Circuits and Systems, Houston, Texas, Apr 1980

[SOM 91] Paul J. Song, Giovanni De Micheli, *Circuit and Architecture Trade-offs for High-speed Multiplication*, IEEE JSSC, Vol. 26, No. 9, Sep 1991

[WAL 64] C. S. Wallace, *A suggestion for a fast multiplier*, IEEE Trans. On Electronic Computers, Vol. EC-13, Feb 1964