DISTRIBUTED BIT–PARALLEL ARCHITECTURE AND ALGORITHMS FOR EARLY VISION

Michael Bolotski

B. A. Sc. (Hons.) University of British Columbia

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

DEPARTMENT OF ELECTRICAL ENGINEERING

We accept this thesis as conforming

to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

August 1990

© Michael Bolotski, 1990

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of ELECTRICAL ENGINEERING

The University of British Columbia
Vancouver, Canada

Date Aug 31 / 1990

DE-6 (2/88)

Abstract

A new form of parallelism, *distributed bit-parallelism*, is introduced. A distributed bit-parallel organization distributes each bit of a data item to a different processor. Bit-parallelism allows computation that is sub-linear with word size for such operations as integer addition, arithmetic shifts, and data moves. The implications of bit-parallelism for system architecture are analyzed. An implementation of a bit-parallel architecture based on a mesh with bypass network is presented. The performance of bit-parallel algorithms on this architecture is analyzed and found to be several times faster than bit-serial algorithms. The application of the architecture to low level vision algorithms is discussed.

Acknowledgement

The first person to be acknowledged here is Rod Barman. The original architecture and algorithms were developed jointly as part of a graduate course. His other major contribution is the unrelenting insistence that the work be expanded significantly from the course project results. He wouldn't let me take the easy way to a thesis and deserves great thanks for his foresight.

I would like to thank Dr. Dan Camporese for being as approachable a supervisor as anyone can ask for, and for the frequent discussions on various aspects of VLSI design. Dr. Bob Woodham's guidance on the basic principles of how to approach problems proved to be invaluable. His tolerance of the thesis topic as it diverged from pure vision is commendable. Dr. Jim Little's willingness to discuss his practical experience with the Connection Machine is also greatly appreciated.

In the VLSI Lab, Dave Gagne deserves acknowledgment for all his help with the VLSI tools and tolerance of frequent questions. The original design of the RAM cell in Chapter 3 was done by Chris Adams and Albert Luey as part of a course project.

Finally, I'd like to thank my parents for their continual moral support and encouragement throughout my academic work.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In order for autonomous machines to interact intelligently with the world they must be be able to perceive the world and decide what the world contains. Vision is one of the richest means of perception. This thesis discusses a computer architecture and associated algorithms that allow the first stage of the vision process to be performed in real time.

Computational vision can be decomposed into three levels: low (or early) vision, which deals with image oriented features; middle vision, which manipulates geometric quantities and performs simple recognition; and high level vision, which uses reasoning to disambiguate objects. Early vision attempts to extract the physical properties of viewed surfaces, such as distance, orientation, and velocity, from image intensity data.

The computing requirements of early vision are enormous; a moderately sized image can contain over 65,000 pixels, each of which is used multiple times in a computation. Two characteristics of early vision tasks allow this level of computing power to be feasible. First, vision computations are extremely local. In order to solve the problem at a point, only a few neighbouring values are required. Second, the individual calculations at each pixel are fairly simple. These two factors lead to the design of a fine-grained massively parallel computer, with tens of thousands of very small processors. Although each processor is capable of only simple computations, one can be allocated to every pixel. In this *data-parallel* organization, each pixel can be be transformed simultaneously. Several machines based on this model have

been built, and their performance on image-oriented tasks is comparable to supercomputers.

This thesis describes an architecture that takes advantage of another level of parallelism beyond data-parallelism. Instead of storing one data item at each processor, each bit of the item could be stored at a different processor. This data organization is called *distributed bit-parallel* (DBP), since operations occur in parallel on all bits of a data word, and since the bits are distributed across multiple processors. Distributed bit-parallel techniques exploit the implicit parallelism in arithmetic operations to perform these operations in less than the linear time required on a bit-serial machine.

Chapter 2 describes previous approaches to providing massively parallel solutions to vision problems, both hardware and software. A short history of massively parallel computers is given. The background in algorithms is provided by a discussion of early vision tasks. Finally, previous work on distributed bit parallel ideas is summarized.

Chapter 3 analyzes the design of a DBP machine within the constraints of a very large scale integrated (VLSI) circuit implementation. Several communication networks are considered. The enhanced mesh is found to scale better with increasing system size and shrinking VLSI technology than variable degree networks such as the hypercube. Minor enhancements to the mesh are shown to provide fast communication over long distances.

Next, the impact of DBP organization on internal processor architecture is discussed. The most significant constraint is placed on memory, which is forced to be very small. Analysis of several algorithmic primitives shows that small memories do not result in severe performance degradation. A VLSI implementation of a DBP PE is briefly described and shown to be very compact. The chapter concludes with a design of a hypothetical DBP machine.

Chapter 4 describes various classes of algorithms suitable for a distributed bit-parallel machine, starting from the low level arithmetic primitives, moving up to vector primitives, and

then to some vision algorithms. Conventional arithmetic operations such as addition and subtraction are found to execute faster on the DBP model. Several arithmetic-level algorithms unique to this model, such as multi-operand addition and shifts are described. Vector primitives based on parallel prefix operations are discussed and shown to have efficient implementations on a mesh with bypass. Finally, simulation results of DBP implementations of early vision algorithms such as optical flow and edge detection are presented.

General observations about the DBP model are summarized in Chapter 5, while Chapter 6 draws some conclusions from this research and points to several directions for future work.

Chapter 2

Background

The performance of a computer is maximized when its architecture is well matched to the task at hand. In order to achieve this synergy, the system designer must be aware of both the architectural alternatives and the required algorithms. Some of this background is provided through the discussion of earlier approaches to constructing massively parallel machines, and of various early vision algorithms. An introduction to the distributed bit-parallel model concludes the chapter.

2.1 Massively Parallel Machines

All parallel machines consist of a collection of processing elements (PEs) interconnected by a communication network. In a massively parallel (MP) computer, this collection is very large, containing more than 10,000 processors. The challenge in designing such a system is twofold. First, each processor must be sufficiently small so that a large array will still be of reasonable size (and cost). Second, the communication network must be able to support a large number of simultaneous messages.

Several techniques are used to reduce processor size. The common first step is to share a single instruction stream among all processors. In such a Single Instruction/Multiple Data (SIMD) architecture, only one controller and instruction memory is required for the entire system rather than being replicated at each processor. Some local autonomy can be provided

by an *idle* bit, which disables a processor when set. Other methods of reducing PE size are to simplify the ALU to operate on one bit of a word at a time, in bit-serial fashion, and to reduce local memory at each PE. The architectural tradeoffs of internal PE design are discussed further in Chapter 3.

Various communication networks have been proposed for parallel machines, including the hypercube [Hil85], cube-connected cycles [PV81], mesh of trees [Lei85], and 2-D mesh. Several forms of enhanced mesh have been suggested, including mesh with bypass [KH88], mesh with broadcast [Sto83], and reconfigurable mesh [MKRS88]. The polymorphic torus [LM89b] is an implementation of a reconfigurable mesh. Despite the variety of possible networks, all commercially implemented MP architectures to date, with the exception of the Connection Machine, have used a basic mesh network. Also, most research VLSI implementations are based on enhanced meshes. The constraints leading to this choice are discussed in Chapter 3.

The first massively parallel computer to be designed was the DAP (Distributed Array Processor) developed by ICL (UK) Ltd. in 1973 [Fou87]. The DAP set the mould for future MP machines with its mesh network, bit-serial ALU, and 4K bits of memory per processor. It consisted of 4096 PEs in a 64 by 64 mesh, and was implemented with small and medium scale (SSI/MSI) components. Large scale integrated (LSI) circuit technology first influenced MP architecture in the design of the CLIP4 system, developed at the University College at London in 1974. A simple PE architecture and small 32-bit memory allowed eight processors to be placed on a chip.

The MPP machine built by Goodyear Corporation for NASA between 1979 and 1983 was the first to exceed 10,000 PEs [Pot85]. It was also implemented in LSI technology, but unlike the CLIP4, it relied on external commercial memory chips. The other innovation in the MPP was the addition of multiple shift registers with 4-, 8-, and 16-bit lengths. These shift registers allowed efficient implementation of bit-serial multiplication, at considerable area

expense.

The first MP system to be used for real-time image processing as well as the first to exceed 64K PEs was the GAPP (Geometric Array Parallel Processor), developed in 1984 by NCR. It was used for real-time processing for infrared sensor data on helicopters [Clo88]. The initial system contained 50K processors and was later expanded to 81K. This processor density was made possible by the extreme simplicity of the PE and the small local memory size of 128 bits.

Unlike the previous mesh-based designs, the Connection Machine uses a hypercube network to connect its PEs. Developed by Hillis [Hil85] at MIT and put into commercial production by Thinking Machines Corporation, the CM contains 64K PEs with a router for every 16 PEs. The 4096 routers are connected in a 12-dimensional hypercube. This general connectivity allows the CM to perform algorithms that do not rely on purely local communication patterns. There are other interesting features in the internal PE architecture: the ALU has three inputs and two outputs; one of several registers can serve as the idle bit; the sense of the idle bit can be inverted. This flexibility does not come for free: the CM has the comparatively slow cycle time of 4 MHz.

The addition of local autonomy in SIMD processors was investigated in several recent designs. In a conventional SIMD PE, the only form of autonomy is provided by the activity or idle bit; processors with the idle bit set do not execute the current instruction. A degree of adressing autonomy was incorporated in the Blitzen chip developed at the Microelectronics Center of North Carolina in 1987 [BDHR88]. Different PEs can access non-identical memory locations in the same cycle, as memory addresses can be modified by local state.

Connection autonomy is provided in the IBM polymorphic torus design [LM89b, LM89a, ML89]. Each processor can selectively short any combination of its ports together, thereby

creating non-uniform communication patterns. The other consequence of this capability is that processors can be bypassed, allowing fast non-nearest neighbour communication. Another design that provides similar functionality is the CAAAP (Content Addressible Associative Array Processor) developed at the University of Massachussetts [SNW89].

2.2 Algorithms

Early vision algorithms can be classified into two broad categories: local and global. Local operations require information only from a small neighbourhood of pixels while global operations use information from larger regions or even the entire image. The two classes require different type of primitive operations to be efficient: local algorithms require efficient local data movement and arithmetic operations; non-local algorithms require efficient long-distance communication.

In the subsequent discussion N is defined to be the number of processors in the machine, and k is defined to be the number of bits in the data word. Asymptotic time complexity of an algorithm is denoted by $\Theta()$ instead of the more familiar $O()$ since exact values rather than upper bounds are known.

2.2.1 Arithmetic Operations

Fine-grained MP machines are almost always bit-serial for architectural reasons described previously. A bit-serial ALU can change only one bit of each data word per cycle. As a result, arithmetic operations are at best $\Theta(k)$ with the size of the word. For example, integer addition is performed with the ripple carry algorithm [Hwa79], which requires $\Theta(k)$ time. Similarly, data is transferred between processors over a one-bit wire, and also requires $\Theta(k)$ time. More complex operations, such as multiplication, require $\Theta(k^2)$ time. Many

bit-serial PEs contain shift registers to allow an efficient implementation of multiply using the shift-and-add algorithm.

2.2.2 Local Algorithms

Many vision tasks have been formulated as regularization [TA77] problems. These include edge detection [TP80], surface reconstruction [Har86], computation of optical flow [HS81], computation of lightness [Hor86], shape from shading [IH81], and binocular stereo matching [Nis81]. Regularization problems require the solution of associated Euler-Lagrange equations, which are linear (for quadratic regularization functions) and involve only first derivatives. The discrete approximations to these equations use only nearest-neighbour information. Any mesh-connected machine can solve the discrete equations efficiently, if the basic arithmetic operations are in turn efficient.

2.2.3 Non-Local Algorithms

A second class of tasks, closer to middle vision, requires non-local information. Many algorithms fall in this category, and representative list is provided here:

- Labelling connected components. All pixels within a given intensity range that are connected are to be labelled with a unique label. This is a simple technique for segmenting the image into a small number of regions.
- Finding the convex hull. The convex hull of a set of points is the smallest convex region enclosing all of the points.
- Histogram-based computations. A histogram of pixel intensities can be used for image enhancement if the intensity range is too small. Histograms are also used to estimate image noise in the Canny edge detection algorithm [Can86].

2.3 Distributed Bit-Parallel Architecture and Algorithms

The idea of using multiple processors to operate on a single data item is not new. Even the first MP machine, the DAP, allowed columns of processors to be configured into a ripple-carry chain. This capability was not central to the architecture and operated six times slower than bit-serial operations.

The reconfigurable processor array (RPA) [Rus89] uses a mixed approach to DBP. Although each PE processes two bits at a time, and a data word is shared among a cluster of PEs, considerable hardware resources are devoted to supporting bit-slice operations. There is support for 8-bit wide compare and barrel-shift operations, an 8-bit wide stack, an activity bit stack, and a general bit stack. As a result, each PE is quite complex for a fine-grained machine.

The RPA still performs binary addition with a linear time ripple carry algorithm. The idea of using multiple processors and embedded tree structures to achieve logarithmic performance was introduced in [BWG85]. The processor array developed in that work was designed to serve as the ALU for a conventional mainframe computer, and included local program memory and circuitry to interface the array to a program counter. The array consists of simple (1-bit ALU, 32 bits of memory) PEs, arranged in a mesh with partitionable buses. This partitioning allows embedding of binary trees in the mesh. The main drawback of the approach is the use of k^2 processors to process a k-bit word, a technique clearly unusable with word sizes larger than 16 bits.

The first fully distributed bit-parallel organization in the context of a parallel computer was introduced in the design of the *Silt* machine [BB89, BBCL90]. The central idea of DBP is the distribution of each data item among a group of processors, called a *cluster*, with each

processor storing one bit. This organization, coupled with the ability of the mesh-with-bypass network to embed binary trees, was used to construct arithmetic algorithms that operate in time logarithmic with the number of bits in the cluster.

Each *Silt* PE is very simple, consisting of a one-bit, two-input ALU, and 29 bits of memory. The PEs are connected by a mesh-with-bypass network that allows horizontal and vertical bypassing. If a PE's bypass bit is set, the input link is connected directly to the output link; otherwise, the PE's output port is connected. Broadcast is supported since a PE can still read from the input link while bypassed. Architectural factors leading to this design are explored in the next chapter.

Chapter 3

Distributed Bit-Parallel Architecture

The DBP organization and algorithms require a very special machine architecture for practical implementation. There are two primary constraints imposed by this organization. First, the number of processors must be very large, and second, the inter-processor communication speed must be high.

The requirement for a large number of processors arises from the distribution of one bit per processor. If an array of N data items of k bits is to be processed, then $N \times k$ processors are required, an increase in processors by a factor of k compared to a bit-serial organization. This constraint demands that the ALU and communication circuitry be kept very simple. Even so, the very large number of PEs can only be provide by a VLSI implementation.

A simple one-bit ALU can be constructed with very little silicon area, but will only perform a limited set of operations. Complex calculations, like multiplication, require many of these simple operations, most of which involve communication between processors. Communication time must therefore be very fast. Also, if operations on a data item distributed within a cluster are to be sub-linear with the size of the cluster, then communication time within the cluster must also be sub-linear. Constraints on the network are clearly more severe than those on the internal PE architecture and the network is therefore discussed first.

3.1 The Communication Network

In summary, a network suitable for a DBP architecture must:

- Provide fast local communication.
- Scale to a very large number of processors.
- Be suitable for VLSI implementation.

These constraints are closely interrelated and cannot be analyzed independently. The approach taken in this section is to discuss network design choices and their suitability under the various constraints.

3.1.1 Message Passing vs. Circuit Switching

A basic design decision is whether the communication network will use message passing or circuit switching. In a message passing system, a message and its destination address are combined into a packet and handed off to the routing circuitry, which takes care of the delivery. In a circuit-switched system, an electrical connection is established between the sender and the receiver.

Message-based systems have several convenient features:

- The network is abstracted away from the algorithms.
- The algorithm complexity is substantially reduced.
- The physical network can be changed without modifying the program.

The main disadvantage of message passing compared to circuit switching is speed (although area consumption is also higher). Routing circuitry must examine each message's address

and forward it to the appropriate destination. Although this procedure can be quite efficient with large packets if wormhole routing [Dal87] is used, it performs poorly on the single-bit messages of DBP algorithms. Communication on the direct electrical connection of a circuit-switched system is much faster. Since speed is the overriding consideration, the circuit-switching method is more appropriate for a DBP architecture.

3.1.2 Fixed vs. Variable Degree Networks

A DBP machine typically has between 8 and 64 times as many processors as a bit-serial one, since one processor is allocated per bit. The communication network must therefore scale well with a large number of nodes. In fixed degree networks such as the mesh or torus, the number of connections at each PE is independent of system size. In a variable degree network such as a hypercube, both the number of wires per processor and the longest wire length grow with the size of the system. This growth is limited by packaging and propagation delay constraints respectively.

The Packaging Constraint

A convenient quantity to use when discussing packaging constraints is the *flux*, defined by Maresca and Li [ML89] as the number of wires connected to a package, which can be a chip, a module, or a board. Flux constraints arise due to limitations on the number of IO pads or on wire density.

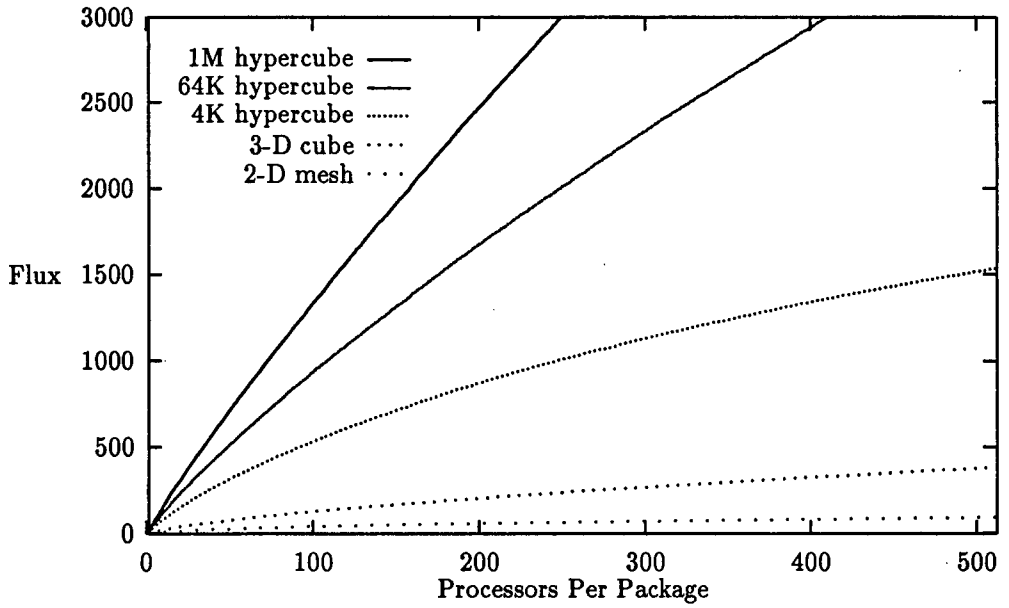


Figure 3.1. Flux requirement as a function of processors per package. The hypercube network is shown three times under various assumptions. A 64K processor system is the base case. The Connection Machine uses a truncated hypercube, in which a router chip is responsible for 16 processors and only the routers are connected in a 12-dimensional hypercube. The flux of this truncated arrangement is equal to that of a 4K PE pure hypercube.

A comparison between the flux requirements of the variable-degree hypercube and the fixed-degree mesh and 3-d cube is shown in Figure 3.1. The three equations in the plot are:

$$F_{hypercube}(n,p) = p(\log_2 n - \log_2 p) \tag{3.1}$$

$$F_{mesh}(p) = 4\sqrt{p} \tag{3.2}$$

$$F_{cube}(p) = 6\,(\sqrt[3]{p})^2 \tag{3.3}$$

where F is the flux (or number of I/O pads), p is the number of processors per package, and n is the number of processors in the machine. Note that this flux is only due to the communication links; it does not include wires for external memory, microinstruction, or power.

The Longest Wire Constraint

The *diameter* of a network is the maximum number of PEs in a path between any two PEs. This value corresponds to the number of steps required to send a message between any two processors. Maximum wire length between any two processors in any network with superlinear diameter grows with the number of nodes as shown in Equation 3.4. Leighton [Lei83, Theorem 5-2], showed that any layout of a graph G with diameter d and minimum layout area A has some edge with the length:

$$L = \Theta\left(\frac{\sqrt{A}}{3d}\right). \tag{3.4}$$

For all networks, the layout area is at least proportional to the number of processors (in a heavily connected network such as the hypercube, additional area is consumed by wires), so that $A \propto N$. The diameter of a mesh network, is $\Theta(\sqrt{N})$, so that L is $\Theta(1)$. This result is obvious, since wires in a mesh only connect adjacent processors and their length is independent of system size. The diameter of a hypercube is $\Theta(\log_2 N)$, and the longest wire length is given by:

$$L_{hypercube} = \Theta\left(\frac{\sqrt{N}}{\log_2 N}\right). \tag{3.5}$$

A problem with long wires is that they don't scale well with shrinking VLSI technology. Speed of light limitations occur on-chip at distances of approximately 30 mm with current technology [Dal87]. If hypercube wires are all off-chip, as in the case of the CM, the limitation is not as significant, but propagation time is still limited to $\frac{\sqrt{\epsilon_r}}{c}$, or 150 mm per nanosecond. This off-chip wiring restriction also excludes the use of multi-chip modules, and therefore leads to additional pin and board delays.

There are problems even below the fundamental speed of light limits. Although the capacitance of short wires decreases with feature size, resistance increases at an identical rate,

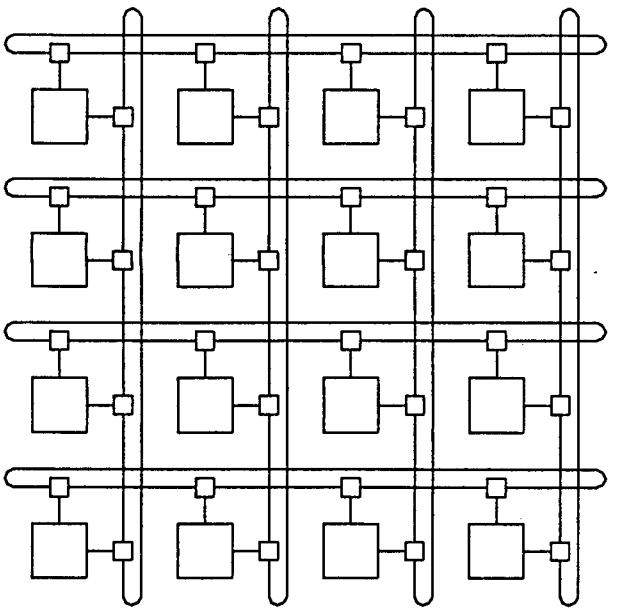


Figure 3.2. A torus network. The larger boxes represent the PEs and the smaller boxes are the links. Bypass capability can be provided if a link can short circuit its input and output.

and the RC product stays constant. Logarithmic delays across long wires can be achieved with a chain of increasingly larger inverters [MC80], but since the current driving ability of transistors decreases with feature size, this chain must grow as features shrink.

3.1.3 The Enhanced Mesh

The two-dimensional mesh network maps well onto the two-dimensional structure of VLSI circuits and scales well with a large number of PEs. The disadvantage of an ordinary mesh is that the number of cycles required to propagate a message between any two processors is at worst $\sqrt{2N}$. This diameter is usually reduced to $\sqrt{\frac{N}{2}}$, by connecting opposite edges of the mesh together to form a torus, as shown in Figure 3.2.

Communication between PEs within the same cluster still requires several cycles in a conventional torus. This restriction is removed by allowing a processor to be bypassed, effectively

removing itself from the network. A bypass capability allows the powerful features of binary tree embedding and broadcast, and only minor circuitry is required for its implementation. A single multiplexer that connects either the locally generated signal or the incoming signal to the output port is sufficient. A slight delay is still incurred at each bypassed PE due to this circuitry. As a result, communication time is still proportional to distance, but the constant of proportionality is very small.

The ratio of cycle time to link bypass delay has a significant effect on the efficiency of long distance messages. It is instructive to compare this ratio for two implementations of a mesh with bypass. In a VLSI prototype of the polymorphic torus [ML89], the cycle time is 120 ns, while the bypass delay is 2 ns, for a propagation delay of 1 cycle per 60 bypassed PEs. In the simulation of the *Silt* PE described in Section 3.3, the respective values are 40 ns and 3 ns, leading to a ratio of 1 cycle per 13 PEs. The disparity in bypass rates is caused by the artificially slow cycle time of the torus chip, which should be capable of a 25–30 ns cycle time if implemented in a more advanced 2μ VLSI technology.

A fast multi-PE bypass circuit that increases the *Silt* bypass rate to 30 PEs per cycle is described in Section 3.3 . A message crossing the entire length of a mega-processor system therefore requires 33 cycles, excluding chip and board delays. In addition, transmission can be pipelined, so that an entire cluster can be sent in approximately 40 cycles. With this technique, the message path is divided into four (in the case of 16-bit clusters) segments, with each segment acting as a pipeline stage.

Although a DBP machine is $\sqrt{k}$ times wider in each dimension than a bit-serial machine, and therefore has a longer propagation delay, $\sqrt{k}$ bits are transmitted per cycle, so that delays for DBP and bit-serial systems are identical when an entire cluster is transmitted. If the distance is sufficiently long, the transmission of each bit in the cluster can be pipelined and total message delay approaches bypass delay. In this situation the smaller size of the

bit-serial organization is superior due to fewer bypassed PEs and therefore shorter bypass delay.

3.1.4 Dealing With Non-Uniform Propagation Delay

Messages sent over short distances arrive faster than those sent over long ones. The delay is dependent on the algorithm, since it is impossible to identify the transmission distance by static code analysis. The polymorphic torus adopts a clock-stretching scheme to deal with this problem, and expects the programmer to specify expected delays. A more efficient solution is to allow messages to propagate and perform potentially useful operations while waiting for the message to arrive. Programmer intervention is still required with this method.

The hardware can be constructed to allow the message bit to propagate continuously from a link until a write is issued along the same dimension. For example, after a bit has been written west, it will propagate until either the west or east port is written to. The advantage of this scheme over a variable-clock method is that useful computation can be done while waiting for messages. For example, a bit can be written east, several computation cycles not involving communication links can be done, and finally the west port can be read. Also, since writes north or south do not interfere with east/west writes, communication in these directions can also be overlapped. This approach is similar to the idea of filling the branch delay slot with useful instructions in deeply pipelined RISC architectures, and compiler techniques from that field can be adapted to reorganize *Silt* code for improved efficiency.

The problem of hardware affecting the basic algorithm still remains; programs must be re-compiled for architectures of different sizes, and the programmer is burdened with specifying the expected message distance if optimally efficient code is to be generated. Of course, operations that do not involve the links do not have to be considered. Also, operations that

exchange information only within a cluster can assume that local messages arrive within one cycle.

3.1.5 Conclusion

The enhanced mesh network meets the criteria described at the beginning of this section. It provides fast local communication via the bypass mechanism; has complexity independent of overall system size; and maps well onto a VLSI implementation. More sophisticated mesh enhancement than the simple bypass are possible, but require more area and are slower. The network chosen in the *Silt* design is the mesh with bypass. The next task is to design the internal architecture of each PE.

3.2 The Processor

The overriding constraint on the design of a DBP processor is the need to keep area consumption as low as possible. As mentioned earlier, a DBP organization requires k times as many PEs as a data-parallel one. If total silicon area used by the two machines is to be similar, a DBP PE must be approximately k times smaller than a bit-serial PE. This requirement may seem unreasonable since a bit-serial PE is already very small. However, since memory is the dominant consumer of area, reducing the local memory of a DBP PE by a factor of k achieves the desired effect. In order for PE size reduction to track memory size reduction, the area of non-memory components must be relatively small. Clearly, memory is the most constrained component of the PE, followed by the communication links, and the ALU.

3.2.1 Memory

The most important considerations in the design of the memory system is the quantity and location. The two aspects are tightly interrelated, but the location is the most constraining of the two. The task of choosing an optimal memory size is difficult, since there are both VLSI and algorithmic constraints.

On-Chip vs. Off-Chip Memory

There are two basic choices for placing the memory: on-chip and off-chip. The advantage of using off-chip memory is twofold. First, memory chips are manufactured using a specialized fabrication technology and are considerably denser and faster than memory implemented on-chip with a general–purpose technology. Second, more precious chip area can be dedicated to processors.

A minor disadvantage of off-chip memory is that flux constraints are likely to allow only one memory pin per processor. The PE will therefore be able to access only one bit per cycle, relying on internal registers for other operands. In contrast, on-chip memory can be made dual ported (at the expense of some area) in order to provide two operands per cycle and therefore keep the ALU busy.

The overriding disadvantage of off-chip memory is the poor scaling with shrinking technology. If the number of processors per chip is N, the number of memory pins required is $\Theta(N)$, while the available perimeter is $\Theta(\sqrt{N})$. Earlier discussion demonstrated the tight constraints on chip pins due to the network wires alone. Off-chip memories square the flux demands. The only option available for a very-fine grained machine is on-chip memory.

The development of fine-grained machines is likely to diverge as feature size shrinks. Systems using off-chip memory will stay completely flux-limited and must therefore increase PE complexity in order to use all available chip area. The most probable direction of complexity increase is to move from a bit-serial to a bit-slice parallel ALU that processes several bits simultaneously. However, as the ALU width increases, so does the required memory bandwidth, which again encounters the flux barrier.

VLSI Considerations for Memory Capacity

The basic tradeoff in VLSI memory design is that of area and time. For a fixed memory capacity, access time can be decreased at the expense of circuit area, and vice versa. Smaller memories are both faster and denser, but incur a higher fractional overhead of non-memory circuitry per bit. This overhead consists of the processing and communication elements, as well as the memory cells lost to algorithmic overhead. Analysis in Appendix A shows that the area–time product is minimized by small memory sizes. In effect, breaking a large memory into smaller pieces and adding processing ability to each piece does not significantly increase cost.

Algorithmic Considerations for Memory Capacity

When local memory is too small for a particular algorithm, data must be frequently loaded from external memory. Since these operations are very expensive, typically requiring hundreds of cycles, overall performance drops significantly as the system does no useful work while it waits for the data to arrive. This effect is known as the von-Neumann bottleneck.

The average number of cycles required to access a data item is given by:

$$T_{avg}(p) = (1 - p) + p \times T_{load} \tag{3.6}$$

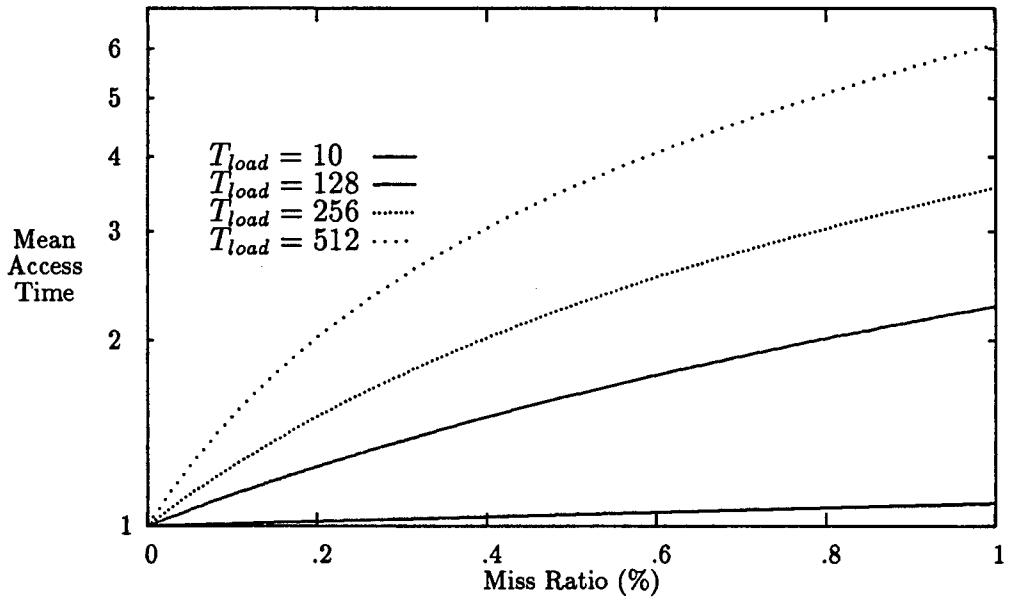


Figure 3.3. Average access time as a function of miss ratio. T_{load} is the time required to load a register from off chip. The 10-cycle load time occurs when PEs are used as memory servers.

where p is the fraction of instructions accessing external data, and T_{load} is the number of cycles required to load a bit from external memory. This function is plotted in Figure 3.3. The range of T_{load} values was chosen to be between 128 and 512, based on a design in Section 3.4 that requires 256 cycles.

Since a 16-bit addition requires approximately 40 instructions (as shown in Chapter 4), all of which require only local memory access, the miss rate for this instruction is at worst 2.5%, assuming a local sum is being accumulated. This miss rate results in a catastrophic performance drop by a factor of 6 (when T_{load} is 256 cycles). However, this occurs only if every add caused a cache miss, which is highly unlikely. If multiplication, which requires approximately 500 instructions, is a common operation, then the miss rate is below 0.2%, resulting in a performance penalty of 1.5.

For more complex algorithms, overall performance is impacted less by external loads. For example, if a 3 by 3 neighbourhood convolution is common, the miss rate is below 0.02%, since the operation requires approximately 5000 cycles. At this miss rate, the slowdown is only 5%. Similarly, there are many computational primitives that require thousands of cycles but only a few memory bits. If the miss rate is maintained below 0.04%, the system will operate at least at 90% of full memory speed.

If a particular algorithm requires considerably more memory than is available locally, thereby causing frequent external memory accesses, a possible solution is to use additional processors as memory servers. Every other processor can remain idle and bypassed for most operations. The time required to access a server is approximately 10 cycles, which is much better than the 256 cycles for off-chip memory. The disadvantage is the reduced number of computational elements.

Conclusion

The preceding discussion has shown that a small memory is advantageous from a VLSI viewpoint and does not degrade performance significantly. The size chosen for the initial *Silt* implementation is 57 bits. This somewhat strange value is based on an 8-bit address space which also contains the link registers, as discussed in a later section.

3.2.2 ALU

The everpresent size constraint is again encountered in the ALU design. Fortunately, only the compact two-input, 16-function ALU is required. Other circuits such as the full adders and shift registers common to bit-serial processors are unnecessary: both of these functions are performed under program control rather than by hardware.

3.2.3 Communication Links

The minimal mesh enhancement required to support DBP algorithms is horizontal and vertical bypass. The other extreme of local connection autonomy is a full crossbar interconnect, in which any port can be connected to any combination of ports. This latter technique is adopted by the CAAPP processor and by the polymorphic torus.

The full interconnectivity provides a considerable enhancement to the communication capability. For example, messages can turn corners without explicit routing steps. Message propagation speed becomes dependent only on the length of the path and not the number of jogs. This effect can be used to advantage in a connected-components algorithm as described in [SNW89].

The price of the improved performance is both area and speed. Compared to the bypass technique, six state bits are required to store the connectivity information instead of two. Similarly, six transmission gates are required, along with 12 additional inverters. In fact, assuming the simplest possible implementation, full connectivity requires three times the area resources of a simple bypass.

The additional circuit complexity results in slower operation. A full crossbar switch doubles the number of transmission gates in the bypass critical path. Since every additional delay in the bypass path is multiplied by the number of bypassed processors, the speed penalty is quite severe. Also, the setup time for a particular configuration is longer, since three bits must be set to select the configuration.

3.2.4 Autonomy Considerations

Another design aspect of processor is the amount of local autonomy available to each processor. All SIMD machines must have at least an activity bit that controls whether a processor is idle during the current turn. This is a minimal requirement for conditional execution. Additional autonomy at the expense of area may be provided by a variety of methods:

- *Operational autonomy by modification of the ALU operation.* This technique is taken to the extreme in the NAP processor [FK88] where a lookup table is used for all ALU operations and the instruction only indicates the address of the operation. A simpler version of this feature can be provided by allowing the ALU operation to be chosen from either the instruction word or from special registers based on a bit in the instruction. An even simpler design specifies two ALU operations in the instruction and selects between them based on a local bit.
- *Address autonomy by modification of the memory address.* This form of autonomy is implemented in the Blitzen chip [BDHR88] by ORring address lines with local registers. Althrough the VLSI implementation of the *Silt* PE discussed in Section 3.3, uses global address decoding, address autonomy can be provided by adding a multiplexer to select between two decoded row lines.
- *Activity autonomy.* This is a generalization of the simple idle bit technique. For example, the Connection Machine PE allows any one of several registers to be used as the idle bit. The CAAAP processor's idle bit can be overridden with a global signal. Another possibility is to place a secondary idle register in series with the primary one and to select whether the secondary is used with a global signal.

These approaches incur varying area and delay costs, and it is not clear which features result in sufficient program reduction to offset the increased cost. The *Silt* PE uses only the last

form of address autonomy mentioned above: a global control signal and a secondary idle bit. A 10–15% speed increase was obtained at the small expense of a multiplexer and five additional transistors. The other options can only be evaluated when a larger collection of representative programs is available.

3.2.5 Conclusion

The architecture resulting from the preceding analysis is essentially the same as the one described in the original *Silt* project. The network is a straightforward mesh with bypass; the bypass enhancement is implemented locally within each PE. The ALU has two inputs and supports all 16 logical two-input operations. The 57-bit memory is dual-ported, so that two bits can be read and one bit can be written in the same cycle. Four memory bits are dedicated to the communication links, two memory bits serve as the horizontal and vertical bypass flags, and another two as the primary and secondary idle flags.

3.3 A VLSI Implementation

A DBP processor was implemented in VLSI in order to determine comparative area and timing requirements of the memory, processing, and communication elements. The technology is Northern Telecom's double-level metal, single-level poly, 3μ CMOS process. The completed chip contains 16 processors and one set of address decoders.

Each PE consists of 64 bits of dual-port static RAM, two pairs of communication links, the ALU, and the write driver circuitry. The dedicated registers (idle, bypass, links) are integrated with the rest of the RAM array to minimize area and simplify decoding. Each communication wire is unidirectional. A block diagram of the processor is shown in Figure 3.4.

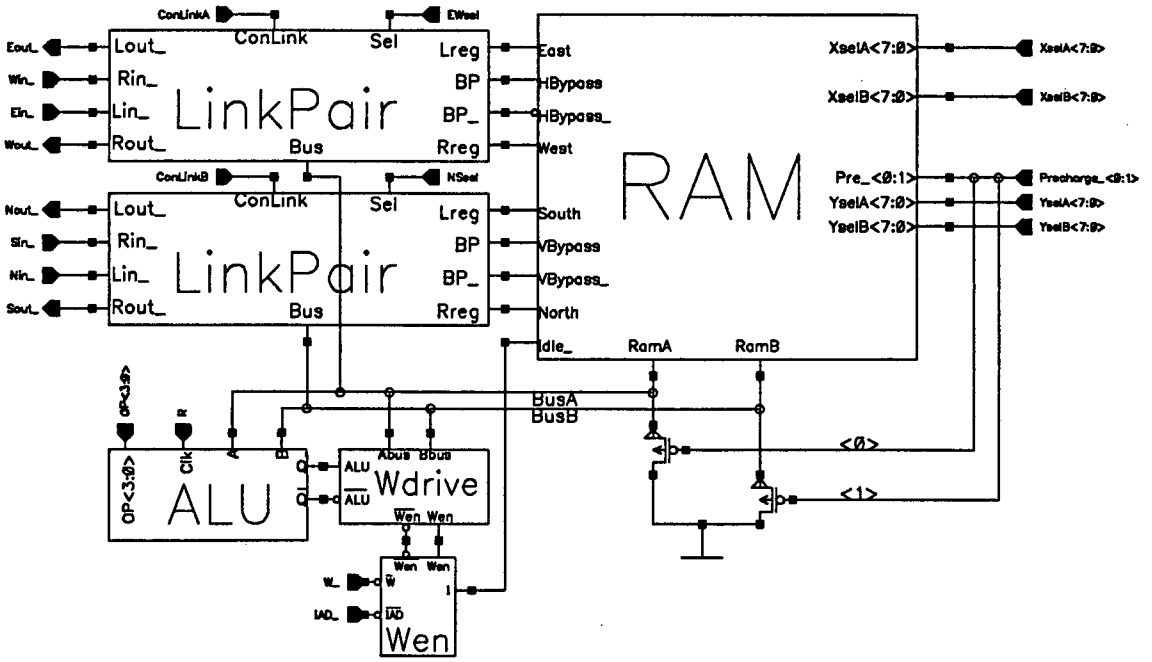


Figure 3.4. Processor Block Diagram.

3.3.1 Theory of Operation

A full cycle is divided into read and write cycles. During the read cycle, two different read addresses are decoded and applied to the RAM. The memory values are read onto the A and B buses and propagate through the ALU. The ALU output is latched at the end of the read cycle. The B bus reads inverted values from the RAM as it is connected to the opposite inverter from the A bus.

At the beginning of the write cycle, the write address is duplicated on both A and B address lines. The latched ALU output enables appropriate drivers onto the A and B buses which write the data to RAM. If the idle bit is set, the write signal is inhibited (unless the destination or source address is the idle bit itself).

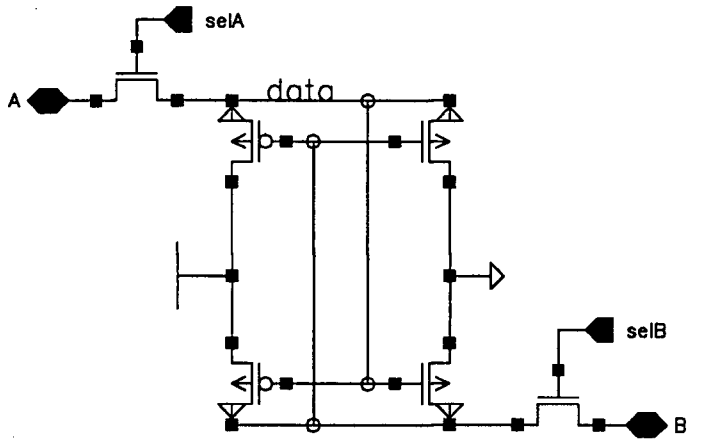


Figure 3.5. Standard RAM schematic

Transmission of a bit requires only that the sending PE write to a link register within its own memory. This register is connected via a bypass circuit to the adjacent PE. The bit starts propagating as soon as the write is complete, and is available at the adjacent PE at the beginning of the next read cycle. Since the receiving PE can enable the incoming value directly onto its ALU buses, communication only requires one cycle.

3.3.2 ALU

The ALU consists of the classic 2-input, 16-operation, pass-transistor based implementation [MC80]. A standard 14-transistor latch is connected to the ALU output. The ALU is very fast due to its simplicity; the output is available before the end of the read cycle.

3.3.3 Memory

The RAM cell is a standard six-transistor cell as shown in Figure 3.5, with the exception

that BIT and $\overline{\text{BIT}}$ are used as separate read ports instead of as a differential output signal. As a result, sense amplifiers cannot be used to speed up the read time. A dual-ported RAM with differential signals would require eight transistors. Since the RAM is very small, sense amplifiers would not speed up operation significantly, and their area would be a sizeable fraction of the overall RAM size.

3.3.4 Communication

Two unidirectional wires are used for each mesh direction. This approach eliminates a multiplexer at each port, reducing both area and propagation delay. The multiplexers must still be used at the chip boundary to reduce the pin count. These multiplexers decode the destination address to determine whether the pin is to be driven or read.

The operation of the link circuitry shown in Figure 3.6 is straighforward. The bypass registers select between the incoming and locally generated signals. The incoming signals must be inverted, since they are sent out through an inverting buffer. Another multiplexer selects which of the ports to place on the ALU bus. Only one bus line is provided for each pair of links. As a result, the north and south ports can only be accessed on the B bus and the east/west ports can only be accessed on the A bus. The pass transistor can be used instead of a transmission gate because the buses are precharged.

Long Distance Bypass

The problem with the preceding circuit, and with any purely local bypass scheme, is that the signal incurs the delay of at least one transmission gate at each processor. This delay is considerably longer than direct wire transmission. Simulated propagation delay through the preceding link circuit was 3.3 ns. Based on a 40 ns cycle time, a message will be able

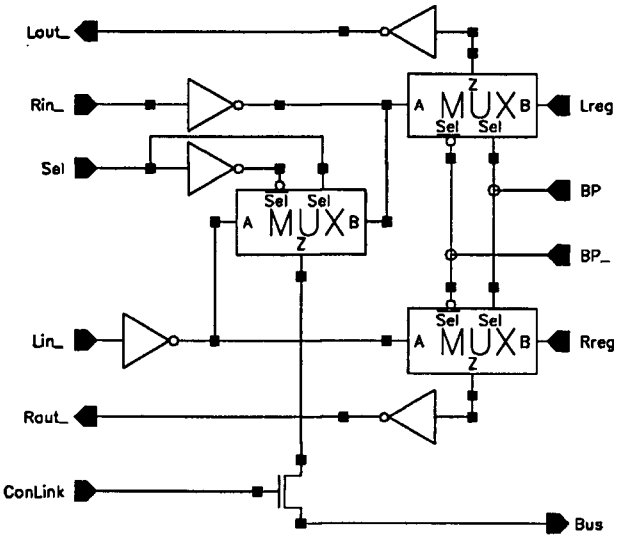


Figure 3.6. The link schematic. Incoming signals L/Rin are buffered by an inverter and connected to two multiplexers. The first multiplexer selects whether this signal or the local link register L/Rreg is sent to the output, based on the bypass register BP. The second multiplexer determines whether the left or right signal is connected to the ALU bus.

to propagate through only 13 processors per cycle. This performance can be improved by detecting a continuous sequence of bypassed processors and bypassing all of them with a single wire. This method avoids propagation delay through the links. The bypassed sequence is detected by connecting N bypass registers to an AND gate, and using the output of the gate to select between the output of the final processor or the bypass wire, as shown in Figure 3.7. Simulation of this circuit shows that propagation delay is reduced to approximately 6 ns across 8 PEs. Up to 53 processors can be bypassed in a single cycle.

The area increase for an N-processor bypass is $\frac{N}{2}+1$ wires horizontally and vertically. The AND gate can be placed under the wires. If another layer of wiring is available, the area increase becomes negligible. The worst case occurs when bypassing $nk+(k-1)$ PEs, where

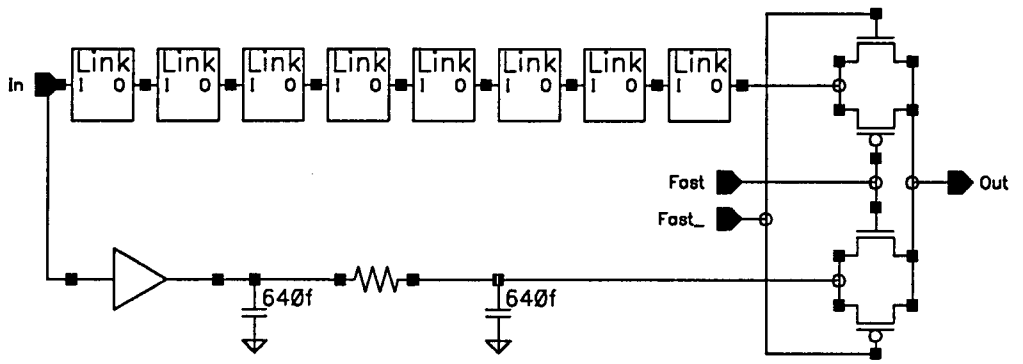


Figure 3.7. The fast bypass circuit incurs the delay of a single buffer and a wire. The bypass registers of each link and the eight-input AND gate that drives the Fast input are not shown.

k is the number of PEs with a quick path.

3.3.5 Processing Element Layout

A physical layout of a *Silt* PE is shown in Figure 3.8. The word line drivers to the right of the memory block are not shown since they are shared with the adjacent processor. Alternating PEs are rotated about the vertical to allow this sharing.

Control lines are routed in second level metal, or metal-2, vertically through the processor. Interestingly, the PE barely fits underneath these wires. The wide instruction word of the CM is impractical for this implementation, since the active area under the additional control wires would be wasted.

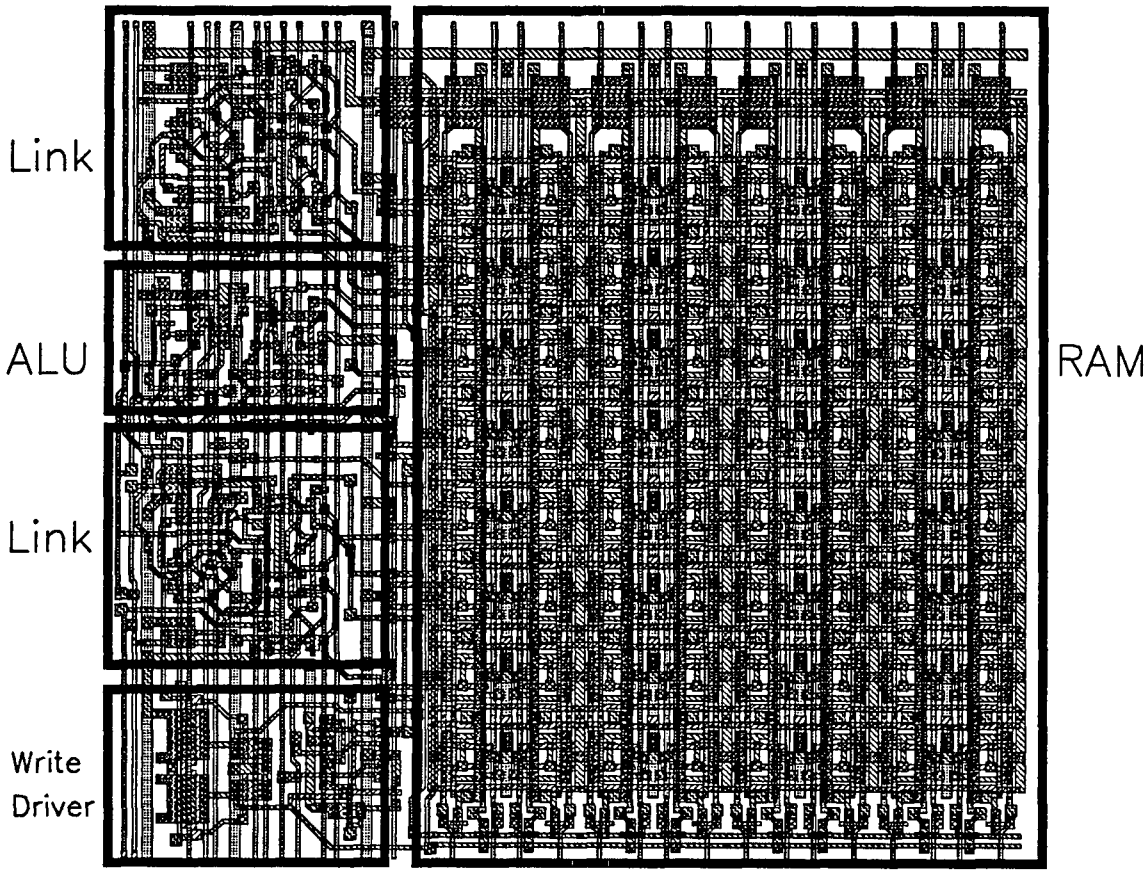


Figure 3.8. Physical layout of the PE. Dual port RAM occupies the middle and right part of the chip. The leftmost RAM column is tapped and connected to the ALU and links. The non-memory elements are arranged in a column, with the ALU placed between the two link pairs and the RAM write drivers at the bottom. Row drivers are not shown.

3.4 A Potential Silt Machine

Two variants of a *Silt* system are described below. The first is a prototype, and the second a potential commercial system. Both versions adopt the same approach to providing external memory.

3.4.1 Off-chip Memory

All early vision algorithms discussed in Chapter 4 fit into local memory. No additional level of memory hierarchy is therefore required between the on-chip memory and the disk (or sensor). However, if multiple vision modules are to be integrated, more local memory may be required to hold temporary results from each module. This memory will be accessed infrequently, and may therefore be located on a separate board. This board can be placed in parallel with each *Silt* board. To access the off-chip memory associated with each processor requires loading the data at one edge of the board. The time required to load a register in each processor, assuming a 64K (256×256) PE board, is 256 cycles.

Dense low power 4 Mbit memories with 23 nanosecond access times are currently in commercial production. Assuming a packaged chip size of 2 cm by 2 cm, each memory chip covers approximately the same area as 4096 *Silt* PEs (in the production system), allowing 1024 bits of off-chip memory per PE.

3.4.2 A Prototype System

This section describes a prototype system that can be constructed with established technology. The VLSI implementation described in Section 3.3 allows 256 PEs on a 1 cm^2 chip. The pinout requirements consist of the 64 link pins and approximately 30 other pins for address lines, op code, power, and clocking, for a total of 94 pins.

A small prototype system containing 64K processors can be assembled with 256 chips. Four boards, each containing 64 chips, can be connected in a torus configuration. Each board edge requires approximately 140 connections, if the non-link pins are distributed along all four edges.

The cost of the prototype (and the production system) can be reduced with the use of multi-chip module (MCM) technology. In this technology, unpackaged dice are mounted directly on a silicon or ceramic substrate. The benefits of eliminating a level of packaging include reduced time delay between chips, increased density, and lower cost. Commercially available MCMs can be 2.5 cm square and can have up to 500 IO leads [Joh90]. Proprietary designs up to 10 cm square and research MCMs with thousands of IOs have been reported.

Mesh-based architectures are highly suitable for this technology, since the chips can be almost butted together, reducing the area overhead of wiring, and eliminating the need for expensive multi-level interconnects. A 25 cm^2 module has sufficient area for 16 $1cm^2$ chips, and the entire prototype would then consist of 16 MCMs.

Data can be loaded into the array by inserting a line of multiplexers at the boundary of one of the boards. The multiplexers allow the loading of external data at the rate of 256 bits per cycle. At a clock rate of 20 MHz, the IO rate is 640 MB/sec.

3.4.3 A Production System

This section describes a production system implemented with currently available technology. A 1.25 μm technology would allow 1024 processors to be manufactured on a 1 cm^2 chip. Each such chip has 32 link IOs per edge. Four chips could be placed on a single MCM, and still require fewer than 300 IOs. Each MCM requires only 3 cm per side. A 64 module board would measure under 30 cm per side, and only four boards are required for a mega-processor system. Such a system would be able to process a 256×256 image at 16 bits per pixel, and is roughly equivalent to a 64K PE Connection Machine.

A 32 processor cluster requires 56 cycles to perform a 32-bit addition. Assuming a conservative 40 MHz clock rate, each cluster can perform 7.1×10^5 adds per second. Since the system

described above contains 32K such clusters, the overall peak computation rate is 23.4×10^9 32-bit adds per second.

If IO occurred at one edge of the mesh, 1024 bits could be loaded per cycle. The potential IO rate is therefore 4.8 gigabytes per second. This rate can only be sustained with the use of multiple disk drives. It is likely that the IO rate will be limited by system components other than the *Silt* boards.

Chapter 4

Distributed Bit-Parallel Algorithms

The ultimate goal of a vision machine is to perform vision tasks quickly and efficiently. Since the algorithms are generally known, the problem becomes finding efficient implementations of the primitives used to construct the algorithms. This approach is obviously top-down, from theory to algorithm to implementation. Frequently, the process is driven from the other direction; operations which are efficient on a particular architectures are used to create algorithms whose results are acceptable if not theoretically correct.

This discussion adopts the bottom-up approach, proceeding from intra-cluster arithmetic operations to inter-cluster algorithmic primitives to the assembly of those primitives into vision algorithms. As a preliminary, a parallel prefix algorithm used in several primitives is introduced.

Most of the algorithms described in this chapter were implemented and tested with a simulator. The simulator is a *C* language program that simulates the detailed behaviour of each processor. The programs were written using a smart assembler with several high-level language features. A description of the *Silt* language and the actual source code for the primitives described below is provided in Appendix C.

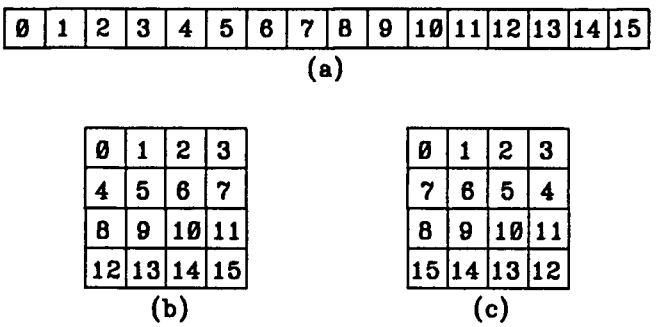


Figure 4.1. Three possible cluster configurations: (a) line; (b) row major; (c) snake.

4.1 Cluster Data Organization

The first question that must be answered before any algorithms are written is how a data item is to be distributed across a cluster of processors. Several possibilities are shown in Figure 4.1.

Each configuration requires a different number of steps to perform common operations:

- *Inter-cluster move.* The two square clusters move $\sqrt{k}$ bits per cycle, requiring only $\sqrt{k}$ cycles for the entire cluster. The line configuration moves the entire cluster vertically in one step, but requires k cycles to move it horizontally, for an average of $\frac{k+1}{2}$ cycles.
- *Average tree distance.* The line and snake configurations double the number of bypassed PEs at each level of the tree up to $\frac{k}{2}$. The row major organizations doubles until $\frac{\sqrt{k}}{2}$ are bypassed in each direction.
- *Broadcast distance.* The square configurations can broadcast along each axis, for a total distance of $2\sqrt{k}$; in the line configuration, the message must propagate across the full length of the line.

- *Shift.* Each processor in a line configuration must only write in one direction and read from the opposition direction. The square clusters must bring one edge column across the cluster to the other edge. The snake organization also has the overhead of writing in two directions.
- *Shift by $n\sqrt{k}$.* When the shift distance is a multiple of row width, the row major cluster must only shift vertically by one position. The snake cluster can use this trick only for even multiples of row width since the bit order is reversed in alternate rows. The snake configuration must repeat a one position shift $n\sqrt{k}$ times.
- *Shift multiple values.* The row major organization can recoup some of the overhead in setting and clearing bypass links by transmitting several items for each link setup.

Operation	Line	Row Major	Snake	Bit Serial
Inter-cluster move	$\frac{k+1}{2}$	$\sqrt{k}+1$	$\sqrt{k}+1$	$2k$
Average tree distance	$\frac{k-1}{\log k}$	$\frac{2(\sqrt{k}-1)}{\log k}$	$\frac{k-1}{\log k}$	-
Broadcast Distance	k	$2\sqrt{k}$	k	-
Shift	2	8	11	k
Shift by $n\sqrt{k}$	$n\sqrt{k}$	$n+2$	$\frac{1}{2}(n\sqrt{k}+n+2)$	$n\sqrt{k}$
Shift n values	$2n$	$5n+3$	$2n$	nk

Table 4.1. Merits of Cluster Organizations

This information is summarized in Table 4.1 assuming that only horizontal and vertical bypass is available. If the local mesh enhancement is improved to a full crossbar interconnect, these cycle counts will change, but not significantly. The advantages of the row-major form are readily apparent, and this organization is the one chosen for subsequent algorithms.

Each processor in a cluster stores its intra-cluster ID at a cost of $\log_2 k$ memory locations. Groups of processors can be selected with a few logical operations on those ID bits. For example, all processors in the top row have $I_3 = I_2 = 0$, where the ID number is stored in $I3 - 0$. Similar calculations can determine the bottom, left, and right edges.

A	=	[	4	1	7	8	3	2	1	5]
+-scan(A)	=	[	4	5	12	20	23	25	26	31]
max-scan(A)	=	[	4	4	7	8	8	8	8	8]
+-reduce(A)	=	31								
min-reduce(A)	=	1								
S	=	[	1	0	0	1	0	0	0	0]
seg +-scan(A)	=	[	4	5	12	8	11	13	15	20]
seg min-scan(A)	=	[	4	1	1	8	31	2	1	1]

Figure 4.2. Example of various scan and reduce primitives. The S vector indicates the segment breaks used for the segmented primitives.

4.2 Parallel Prefix Operations

Several algorithms discussed in this chapter are based on *scan*, or *parallel prefix* operations. Scan operations take a binary operator $\oplus$, and an ordered set $[a_0, a_1, \ldots, a_{n-1}]$ and return the ordered set $[a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \ldots a_{n-1})]$. Common scan operations include *or-*, *and-*, *max-*, *min-*, and *+-scan*. There are also corresponding *reduce* primitives which reduce the values in a vector to a single value using a binary associative operator. For example, *+-reduce* calculates the sum of a vector. Both scans and reduce primitives have *segmented* versions, in which vectors are implicitly divided into segments by another vector that marks the beginning of each segment. Examples of these primitives are shown in Figure 4.2.

The parallel prefix of a binary associative operator can be implemented efficiently on any network that allows embedding of binary trees. An example of such an implementation is shown in Figure 4.3. Tree-based scans require two sweeps: an upsweep to propagate information to the internal nodes, and a downsweep to send global information down to the leaves. Software implementations of this algorithm, in which tree levels exist in time rather than space, require temporary memory locations to store the values generated on the up

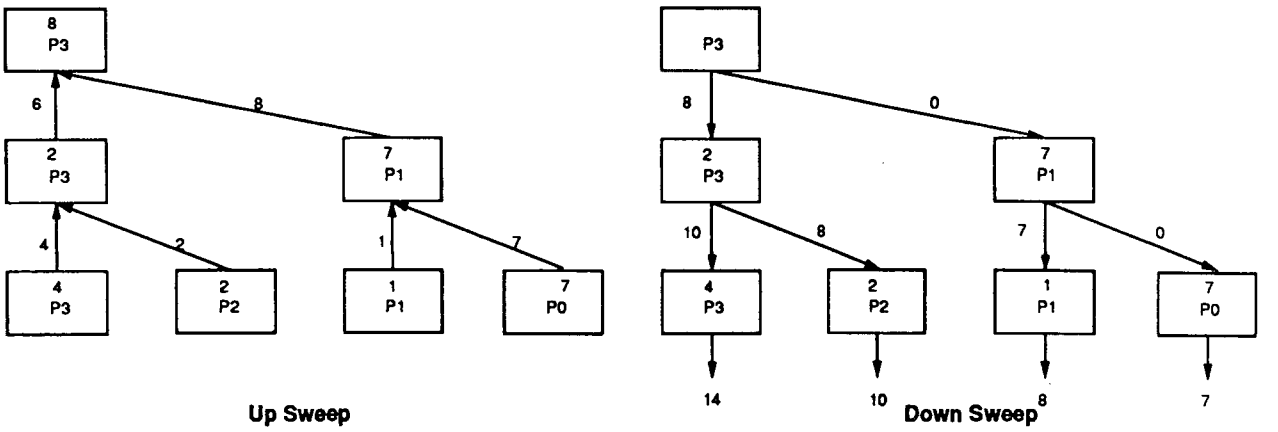


Figure 4.3. Tree-based scan implementation of *+-scan*([6 5 15 5]). In the up sweep each PE retains the value sent by the right subtree and propagates the sum upwards. In the down sweep, each processor sends the value received from the upper tree level to the right subtree and its sum with the retained value to the left subtree.

sweep, since these values will be used again on the downsweep. A useful property of this design from the VLSI viewpoint is its bounded fanin and fanout: each processing node has two inputs and one fanout, independent of total circuit size.

An alternate parallel prefix circuit developed by Ladner and Fischer [LF80] is shown in Figure 4.4. This circuit is both faster and requires less temporary storage than the tree-based circuit. The speed advantage occurs because the LF circuit only performs one sweep instead of two. Auxiliary storage is unnecessary for the same reason: computed values are used only at subsequent levels, and can therefore be discarded. In the tree-based method, $\log_2 N$ memory locations are required to store intermediate results. The memory savings of the LF method are substantial when the parallel prefix is performed at the inter-cluster level. On a 64K cluster mesh, 16 fewer memory locations – fully 25% of the 64-bit local memory – are required.

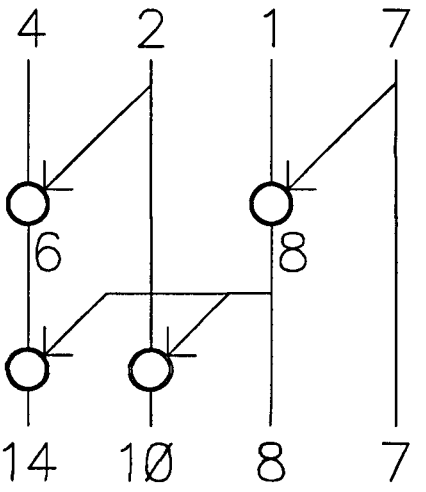


Figure 4.4. The Ladner-Fischer parallel prefix circuit. In this example, the binary associative operation in the bubble is addition, and the circuit computes *+-scan*(7 1 2 4).

The LF circuit is not used in hardware designs for two reasons. First, the number of processing nodes is much higher than in the tree-based circuit, $\frac{N \log N}{2}$ instead of $N-1$. Second, the worst-case fanout is $\frac{N}{2}$ as compared to 2. Neither of these limitations applies in a software implementation of the circuit.

In a DBP implementation, each level of the circuit exists in time rather than in space, so that N physical processors are available at each level. The high fanout requirement is satisfied by the broadcast capability of the mesh with bypass network. Note that architectures which embed trees efficiently but cannot perform broadcasts are not suitable for this method.

The procedure for expanding the circuit size is straightforward. To create a circuit of size $P(n)$, two circuits of size $P(\frac{n}{2})$ are combined with a row of $\frac{n}{2}$ processing nodes as shown in Figure 4.5. This process illustrates the principle of operation: at each tree level, the cumulative value of the lower half is applied simultaneously to all elements in the upper half, while the lower half is unchanged. An intuitive explanation for the speed advantage

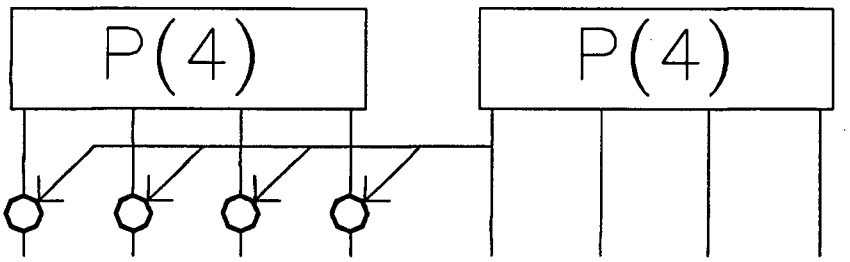


Figure 4.5. Expanding the LF circuit.

of this method is that at every time step $N/2$ of the processors are doing work, while in the tree method at level i, only $N/2^i$ processors are active. The average PE utilization rate is therefore much higher.

When executing a *reduce* operation, where only the final value is required, the tree method performs better than the LF methods because no downsweep is needed, and no temporary memory is used. The time requirements then seem identical, but the LF method incurs the overhead in setting up the broadcast step. When implemented on a square mesh, there are two broadcasts (one horizontal, one vertical) in order to reach the N/2 processors.

4.3 Intra-Cluster Arithmetic Primitives

The simplest DBP primitives are similar in function if not in implementation to arithmetic primitives in the instruction set of a conventional uniprocessor computer. These include binary addition, logical and arithmetic shift, testing of a particular bit, and bitwise logical operations. There are also several instructions that do not have counterparts on typical computers, such as multi-operand additions and shifts.

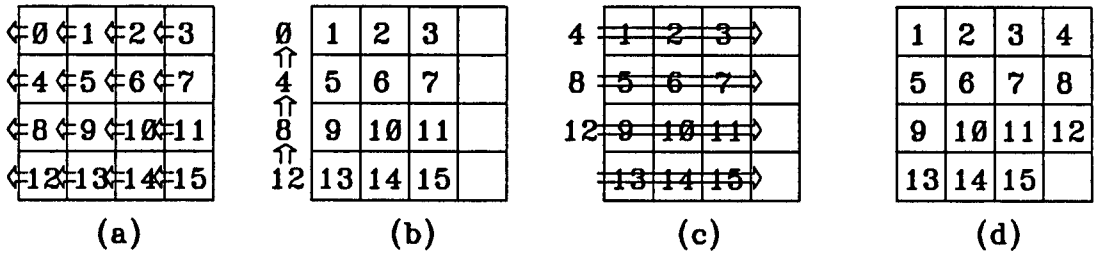


Figure 4.6. DBP implementation of logical shift right. (a) all PEs shift west. (b) the leftmost column now stored in an adjacent cluster is shifted north. Idling overhead is implicit whenever only a subset of PEs performs an operation. (c) internal PEs are bypassed and the column is shifted to the other side. The overhead in this step is due to setting and clearing the bypass registers.

4.3.1 Shift Operations

The implementation of the basic logical shift operation is based on the bypass capability of the network. Most of the bits only shift one position left or right, but one column of the cluster must move to the opposite edge and then shift either up or down. The bypass allows the column to move in constant time. Finally, the most significant (MSB) or least significant bit (LSB) is cleared, depending on whether the shift is to the left or the right. This sequence of operations is shown in Figure 4.6. The arithmetic shift requires slightly more work, as the MSB must be retained.

There are several variations of the shift operation:

- *Multi-bit shift.* The word can be shifted several positions instead of one.
- *Distance $n\sqrt{k}$ shifts.* If the shift distance is a multiple of the row size, the bits move up or down only. The overhead of sending the column to the other edge is removed. This feature is used to advantage in the normalization stage of floating point algorithms.

- Multi-operand shift. Several numbers can be shifted one position simultaneously, reducing the overhead of changing the bypass configuration.

4.3.2 Collect and Distribute

One disadvantage of a DBP architecture is a consequence of the need for an entire cluster to act as a unit. Idling a data item requires distributing the idle command to all processors in a cluster. This operation is more expensive compared to a bit-serial organization, since the status bit that determines whether the idle will occur must first be collected at the distribution point and then distributed to the cluster.

For example, the sequence `if (A > B) { ... }` is implemented by subtracting B from A and idling all clusters for which the result is negative. The sign of the result is available only at the MSB processor and must be distributed to the rest of the cluster. Similarly, tests for the even/oddness of a number must distribute the results from the LSB processor to the cluster. These two locations, MSB and LSB, are very convenient, since the values can be immediately broadcast along the top row and right column, without a collect phase. In other situations, a bit must be extracted from an intermediate position in the cluster and brought to a convenient broadcast location.

Although the collect operation could be performed with a global OR, a sequence of nearest-neighbour moves is faster for short distances due to the overhead of setting and clearing up bypasses. For a 16-bit cluster, 2 cycles on average are required to move any bit to a corner; the 64-bit cluster requires 4 cycles. Once at a corner, the bit can be distributed with two broadcasts, one each along an edge row and column. The cycle requirements are independent of word length for practical cluster sizes.

4.3.3 Addition

Binary addition may be reformulated as a binary associative operation as shown in Appendix B, and can therefore be implemented efficiently by any network that computes parallel prefix. Due to the considerably reduced area, fanout and wiring requirements, scan circuits are usually implemented with the tree method [BK82] rather than the LF method. The DBP implementation described in [BB89] is also based on the tree.

As discussed earlier, the LF circuit can be mapped efficiently onto the mesh with bypass. A circuit implementing 4-bit addition is shown in Figure 4.7.

The main difficulty in converting the circuit to an algorithm is emulating the wires in the circuit; in other words, choosing which PEs are to compute new results, and which are to retain the old ones. The obvious solution is to idle the passive processors, but this incurs unnecessary overhead. A more efficient alternative is to transmit identity values to the inactive processors. The two update rules at each procesor are:

$$G \leftarrow G + G' \cdot P \tag{4.1}$$

$$P \leftarrow P \cdot P'. \tag{4.2}$$

Identities for the G and P updates are 0 and 1, respectively. These identities can be generated by AND and OR operations at the time of transmission.

Memory savings in this algorithm are substantial. Only three memory locations are used, compared with $2 \log k - 1$ locations required by the tree-based method. The 16-bit addition using the LF method was implemented and found to be 10% faster than the tree-based addition. This speed advantage increases with word size.

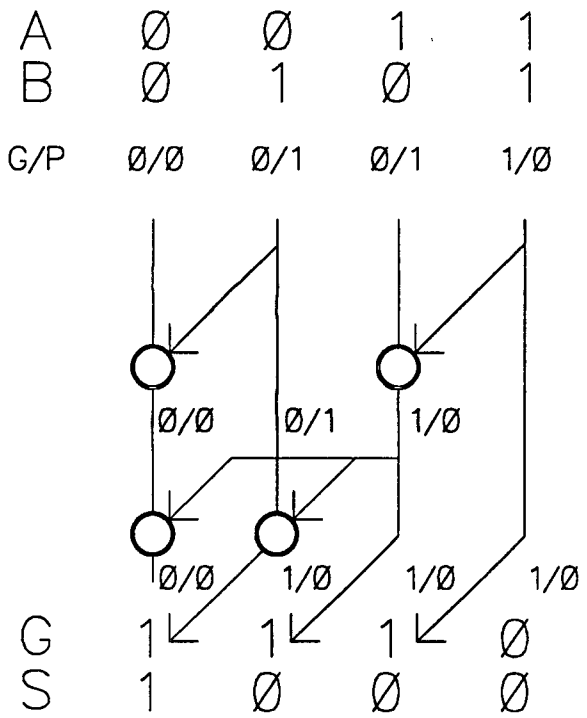


Figure 4.7. *A* and *B* are added to produce *S*. The first stage produces *G* and *P* with an AND and XOR operation respectively. The carry is generated in a parallel prefix fashion by the circuit, and *S* is obtained by the XOR of *A*, *B*, and *G*.

4.3.4 Multi-Operand Addition

The sum of a sequence of n numbers can be evaluated much faster than the $n \log_2 k$ operations required by n additions with the use of a software version of a circuit known as the carry-save adder (CSA) [Hwa79]. The delay of each intermediate CSA stage is independent of word size; only the final step uses the logarithmic time adder. This adder is used in the implementation of the multiplication algorithm discussed in Section 4.3.5.

A *carry save number* M is a pair of binary sequences S and C. The value of M is the sum of the values of S and C. The addition of a carry save number $M = [S, C]$ with a binary integer B, yielding a carry save result $[S', C']$ can be performed in a bitwise manner:

$$S' = [\text{sum}(S_{n-1}, C_{n-1}, B_{n-1}), \ldots, \text{sum}(S_1, C_1, B_1), \text{sum}(S_0, C_0, B_0)] \quad (4.3)$$

$$C' = [\text{carry}(S_{n-2}, C_{n-2}, B_{n-2}), \ldots, \text{carry}(S_0, C_0, B_0), 0] \quad (4.4)$$

The addition time is therefore independent of word width. A circuit implementation of a multi-operand adder is shown in Figure 4.9.

The time required to add a sequence of n numbers is

$$T_n = t_{setup} + (n-2)t_{csa} + t_{add}, \quad (4.5)$$

where n is the number of integers to be summed, t_{setup} is the initial setup time, t_{csa} is the size-independent addition time, and t_{add} is the time for a full logarithmic add. The speedup for various numbers of operands is shown in Figure 4.8. The relative speedup is not considerable, since addition time is already logarithmic with word size. However, algorithms which rely on the summation of an $M \times M$ neighbourhood require $2M$ additions, and will therefore operate 2 to 3 times faster.

4.3.5 Multiplication

A fast integer multiplication algorithm known as the modified Booth algorithm [Sor61] is suitable for bit-parallel implementation. It uses a recoding technique to cut in half the number of required additions.

A single iteration examines three adjacent bits of the multiplier and adds 0, ± 1, or ± 2 times the multiplicand to the sum. The multiplicand is then shifted by two places. This sequence is shown in more detail in Figure 4.10.

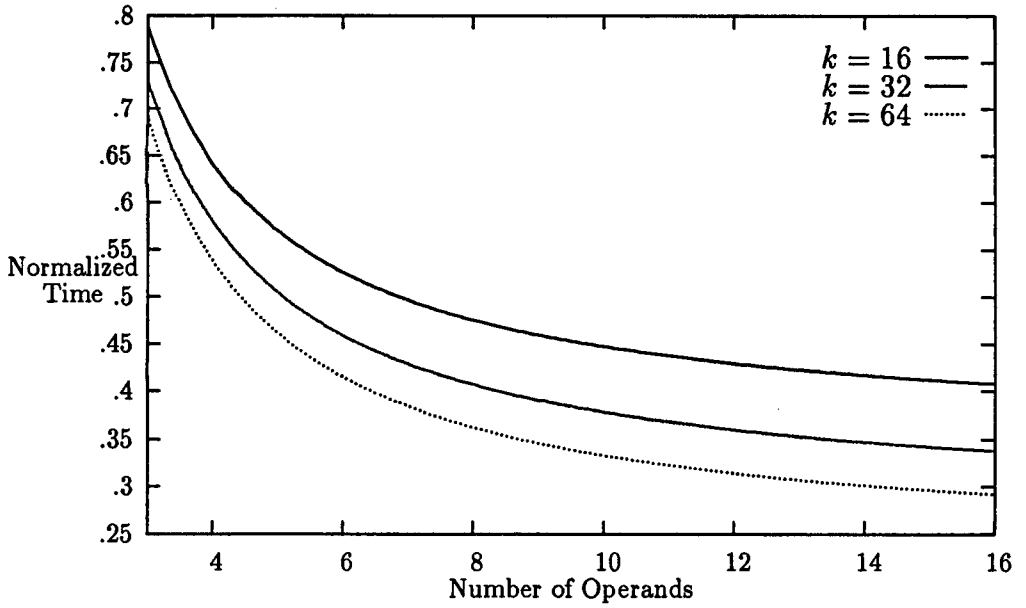


Figure 4.8. Multi-operand addition speedup for various word sizes. As the number of operands increases, the time approaches the ratio of CSA-stage delay to full addition delay.

The complete algorithm consists of $\frac{k}{2}$ iterations of the steps shown in Figure 4.10, followed by an add operation to convert the carry-save accumulator to standard form. The multiplication of two 16-bit numbers to form a 32-bit result requires 655 cycles, or 26.2 μsec (based on a 40 ns cycle time). Multiplication of two 8-bit numbers requires 352 cycles, or 14 μsec.

4.3.6 Floating Point Operations

Early vision algorithms generally do not require the precision of floating-point arithmetic, especially since the sampled image intensities are usually 8-bit quantities. However, floating-point is important for a few algorithms and may well be useful for higher level vision tasks. The DBP implementation of floating-point is presented in outline form in this section. Details of overflow checking are not considered, but are fully described in standard arithmetic texts

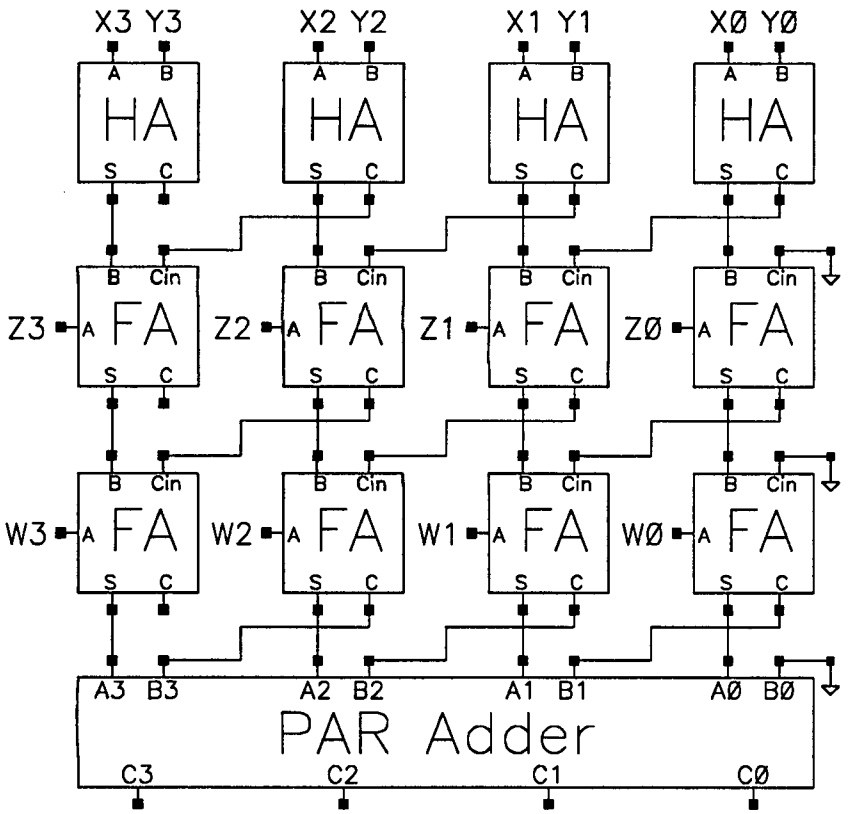


Figure 4.9. A 4-operand carry-save adder. The four numbers X, Y, Z, and W are added to form C. A logical shift left occurs after every csa stage. The abbreviations HA and FA stand for half-adder and full-adder respectively. The block labelled **Par adder** represents the standard 2-operand binary addition operation, requiring logarithmic time.

such as [Hwa79].

Floating Point Addition

The four main phases of the addition algorithm are:

1. Check for zeros.
2. Align the mantissas.

1. Decode lowest two bits of A to determine multiplier $(0, \pm 1, \pm 2)$.
2. Copy B into temporary register T.
3. Shift B left.
4. If multiplier is 2, copy B into T.
5. Shift B left.
6. If multiplier is 0, clear T.
7. If multiplier is negative, invert T, and set the carry in.
8. Add T to the carry-save accumulator.
9. Shift A right by two.

Figure 4.10. A cycle of the Booth multiplication algorithm. Several optimizations have been made to the basic algorithm. For example, the very first iteration performs only the initial half-adder step of the carry-save addition. The seemingly conditional operations in steps 6 and 7 above are implemented unconditionally by ANDing and XORing the data item with the condition bit.

---

3. Add the mantissas.
4. Normalize the sum.

In the first step, if one of the two numbers being summed is 0, the answer is the other number, and the calculation ends. This is implemented with a cluster-wide OR followed by a distribute operation. The third step is a trivial integer add operation.

The most time consuming operations are mantissa alignment and normalization. A conventional SIMD implementation of mantissa alignment iterates across the smaller exponent, shifting the mantissa and incrementing the exponent until it is equal to the larger one. Each iteration requires a shift, comparison, and an increment. In a 32-bit floating-point representation with an 8-bit exponent, and a 23-bit mantissa, 23 iterations are required. The DBP implementation shown in Figure 4.11 needs only five iterations since it executes in time logarithmic with the mantissa length. No incrementing or comparison is required. A similar logarithmic-time algorithm can be used to implement the normalization step.

1. Set K to the difference of the two exponents.
2. Set counter i to 1.
3. Set M to the mantissa of the smaller number.
4. If the LSB of K is 1, shift M right by i,
5. Shift K right by 1, double i, and repeat from step 4.

Figure 4.11. Mantissa alignment for floating point addition. The number of cycles is logarithmic with mantissa length. This operation in step 4 is very fast in most cases, since shifts by $\sqrt{k}$ are efficient.

Floating Point Multiplication and Division

Unlike fixed-point arithmetic, floating-point multiplication is simpler than addition. The main steps are:

1. Check for zeros.
2. Add exponents.
3. Subtract bias.
4. Multiply mantissas.
5. Normalize mantissas.

All of these operations can be implemented by algorithms developed earlier. The sequence for floating-point division is identical except for step 4 where a fixed-point division is performed instead of a multiplication.

4.3.7 Constant Generation

Constants with a regular bit pattern may be generated algorithmically instead of being loaded externally. For example, a constant that is a power of two may be generated in $\log_2 k$

steps by logical operations on the intra-cluster processor id. More complex patterns require a longer sequence of operations. The problem of determining a sequence of operations to generate a particular bit pattern is isomorphic to the problem of multilevel minimization of a boolean function. The latter problem has been studied extensively in the field of VLSI synthesis [BHMSV84], [BRSV87]. Several existing synthesis tools can be used to determine a good sequence of operations.

An example of this procedure is shown in Figure 4.12, produced with the minimization tools *espresso* and *MIS-II* [BHMSV84]. The constant was generated with 6 instructions instead of the 256 required to load it from off-chip. Also, several constants can be generated simulataneously by specifying a function with multiple outputs.

An example of this procedure is shown in Figure 4.12, produced with the minimization tools *espresso* and *MIS-II* [BHMSV84]. The constant was generated with 6 instructions instead of the 256 required to load it from off-chip. Also, several constants can be generated simulataneously by specifying a function with multiple outputs.

4.4 Inter-cluster Operations

The global primitives on a mesh with bypass operate in one of three modes: broadcast, tree, or mesh. The most appropriate method is chosen, depending on the algorithm. For example, reduce operations operate in tree mode, while parallel prefix in general operates in broadcast mode.

Logarithmic time inter-cluster communication can be performed on a DBP architecture by embedding a regular binary tree in the mesh of clusters. As in the intra-cluster embedding, each cluster must have a unique number, which serves as the implicit cluster address. The address therefore requires $\log_2 N$ bits, where N is the number of clusters in the machine.

```
0000 0      1000 1      10-0 1      I3' I2 I1 + I2' (I0' (I1 + I3) + I1' I0)
0001 1      1001 1      -010 1
0010 1      1010 1      011- 1
0011 0      1011 0      -001 1
0100 0      1100 0
0101 0      1101 0
0110 1      1110 0
0111 1      1111 0
```

(a) (b) (c)

```
X = I1 | I3;
X = X & ~I0;
Y = ~I1 & I0;
X = X | Y;
X = X & ~I2;
Y = ~I3 & I2;
Y = Y & I1;
X = X | Y;
```

(d)

```
X = ~I1;
when (~I0)
  X = I1 | I3;
Y = ~I3 & ~I1;
when (~I2)
  Y = X;
```

(e)

Figure 4.12. (a) the desired constant, 1990 (binary 0000011111000110), as a function of the PE ID. (b) the sum-of-products form of the minimized equation generated by *espresso*. (c) the factored form of the equation generated by *MIS* from (b). (d) the straighforward translation of the equation into *Silt* code. Note the notation change: ~ indicates negation, & is logical AND, and | is logical OR. (e) an optimized version of (d), using the software equivalent of a multiplexer circuit.

The address can be stored in a distributed fashion and processed in parallel, similarly to a data item. Address processing is slightly less efficient if the cluster size is smaller than the number of bits in the address.

4.4.1 Global OR

Although the primitives presented so far execute for a fixed number of steps, most complex algorithms require conditional execution. In order for the controller to change the flow of execution, it must be able to detect a data-dependent result in the PE array. For example, the algorithm may specify a set of operations to be iterated until no change occurs. The array must notify the controller that this condition is true. A common technique for providing this feedback is a sum-OR tree (also known as a some/none tree), in which the output of a register in each PE is connected to a tree structure of OR gates. The controller can access the state of the tree root, and is therefore able to monitor the condition of the array.

This feature can be implemented in software on a mesh with bypass, saving valuable area and power resources. A simple algorithm is described in [LM89b]: all processors which contain a 0 bypass themselves and then all processors write their values. Thus if any PE contained a 1, the first PE would receive a 1; if no PE contained a 1, the first PE would receive a 0. The problem with this approach is that if all PEs contain a 0 and therefore disconnect, the bus will be in an unknown state, and the algorithm is forced to assume that an undriven bus defaults to a 0. This assumption is unwarranted since the circuitry required to drive the bus to a known state occupies area, consumes power, and reduces overall speed.

A simple sequence of operations can compute the global OR without making the known-state assumption. The basic idea is to ensure that at least the first processor drives the bus. In the first step, all processors but the first disconnect if their value is 0. All processors then write their value out. The first PE thus receives the OR of all values except its own. Finally, the PE ORs its own value with the received value.

4.4.2 Bitonic Sort

Since the *Silt* machine can function as a mesh, it can run the well known bitonic sort algorithm [NS79]. A normalized comparison to a bit-serial machine is instructive, as sorting is a common benchmark of performance. The equations below are obtained from the work of Nassimi and Sahni [NS79] and are similar to an alternate algorithm by Kumar and Hirschberg [KH83].

$$N_{route} = 14(n-1) - 8\log_2 n \tag{4.6}$$

$$N_{compare} = 2\log_2^2 n + \log_2 n \tag{4.7}$$

$$N_{exchange} = 6.5\log_2^2 n + 2.5\log_2 n \tag{4.8}$$

The number of exchange instructions differs from that given by [NS79] because each compare step contains an interchange operation, which is described under the exchange category.

On a DBP machine the type of the operation influences performance significantly, since the route operation takes $\Theta(\sqrt{k})$, comparison takes $\Theta(\log_2 k)$, and interchange takes $\Theta(1)$. Also, the cost of idling is non-trivial since the information that the cluster should idle must be propagated to each PE in the cluster.

The time required for a full bitonic sort of 16-bit numbers on a 64K processor machine is shown in Table 4.2. The bit-serial machine is assumed to have a single ported memory, so that addition requires three cycles per bit.

The number of operations is much easier to compute on a bit-serial machine since each of the above operations takes time linear to the number of bits in the word. Consider a 64K cluster machine, which requires 16-bit cluster IDs. Assume that addition on a bit-serial machine requires three cycles per bit, as does the exchange operation.

Clearly, the design of algorithms must take into account the considerable advantage of a DBP

Operation	Freq	Bit Serial		DBP	
		Cycles	Total	Cycles	Total
Route	3506	16	56096	4	14024
Compare	136	48	6528	55	7480
Exchange	436	48	20926	3	1308
Total			83550		22812

Table 4.2. Cycle Counts for Sorting, 64K Cluster System

organization for data movement and exchange operations. Curiously, the CM-2 requires approximately 19,000 cycles to implement a bitonic sort [Ble89]. The advantage of the fewer routing steps required by hypercube network is offset by the 1-bit bandwidth of a bit-serial organization.

4.4.3 Non-Power of Two Trees

Situations arise where vector lengths are not exactly powers of two. A binary tree structure can still be implemented, as long as the binary associative operator has an identity. For instance, the identity for addition is 0; the identity for logical AND is 1. At each level of the tree, senders whose value to be ignored send out the identity value. This operation is shown in Figure 4.13.

4.4.4 Direct Segmented Scan Operations

Blelloch [Ble89] describes an implementation of segmented operations using several unsegmented operations. For instance, a segmented *+-scan* is implemented by executing an unsegmented *+-scan*, copying the first element in each segment across the segment with a segmented *max-scan*, and subtracting each element. This method is termed an *indirect scan*, as scan operations are not performed directly within each segment. A disadvantage of this

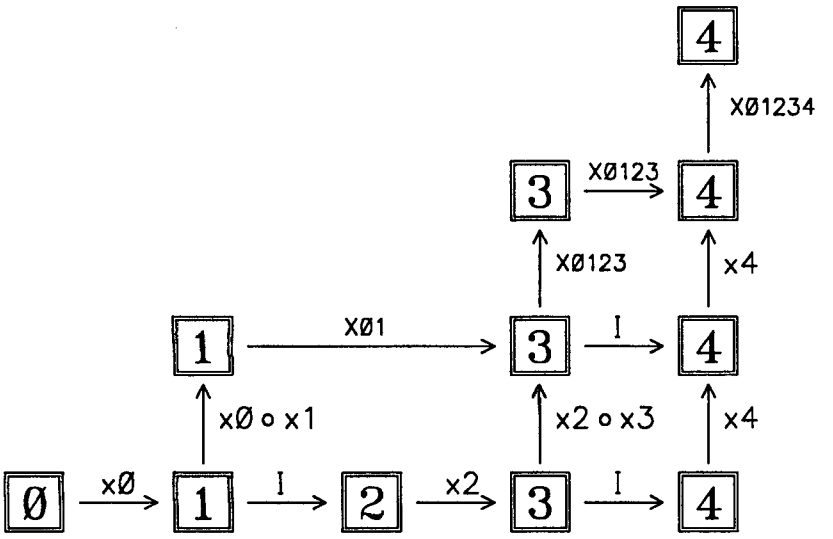


Figure 4.13. Parallel prefix on non-power of two sequences. I is the identity value for the operation. $X0123$ is used to abbreviate the more cumbersome $x_1 \circ x_2 \circ x_3$.

method is that the global result may cause an overflow condition in a large network. This next section describes an implementation of a *direct scan* on a mesh with bypass network.

The two problems involved in implementation on the mesh is that segment size could be non-uniform, and that the sender and receiver PEs could be on different rows of the mesh. The first problem was solved in Section 4.4.3. The solution to the second problem is to use a column of processors as routers for messages that traverse several rows. The algorithm is presented in Figure 4.14.

1. All processors except senders, receivers, segment end, and eastern edge processors are bypassed.
2. Senders transmit a 1 bit east. All others transmit a 0 bit. The bit does not propagate across segment boundaries or the eastern edge. After this step, edge PEs that received a 1 know that they take part in the second routing step. Receiver PEs that received a 0 also participate in the second step.
3. Senders transmit a data item. Every processor stores the item in a temporary register T.
4. All non-routing edge PEs bypass themselves vertically.
5. Routing PEs broadcast the data item south.
6. All processors send their *northern* input to their eastern output.
7. All second-phase receivers replace their T register with their western input.
8. All receiver PEs move their T register into the destination register.

Figure 4.14. Routing on a Mesh

4.5 Vision Algorithms

The vision algorithms discussed below are classified as local or non-local. As noted earlier, local algorithms require fast arithmetic, while non-local algorithms require fast communication.

4.5.1 Local Algorithms

Edge Detection

The Marr-Hildreth edge detection algorithm [Mar82] consists of filtering the image with a Gaussian filter, computing the Laplacian of the filtered image, and locating the zero crossings of the final image. Serial implementations of the algorithm take advantage of the linear nature of both filters and convolve the image directly with the Laplacian of a Gaussian ($\nabla^2 G$). A

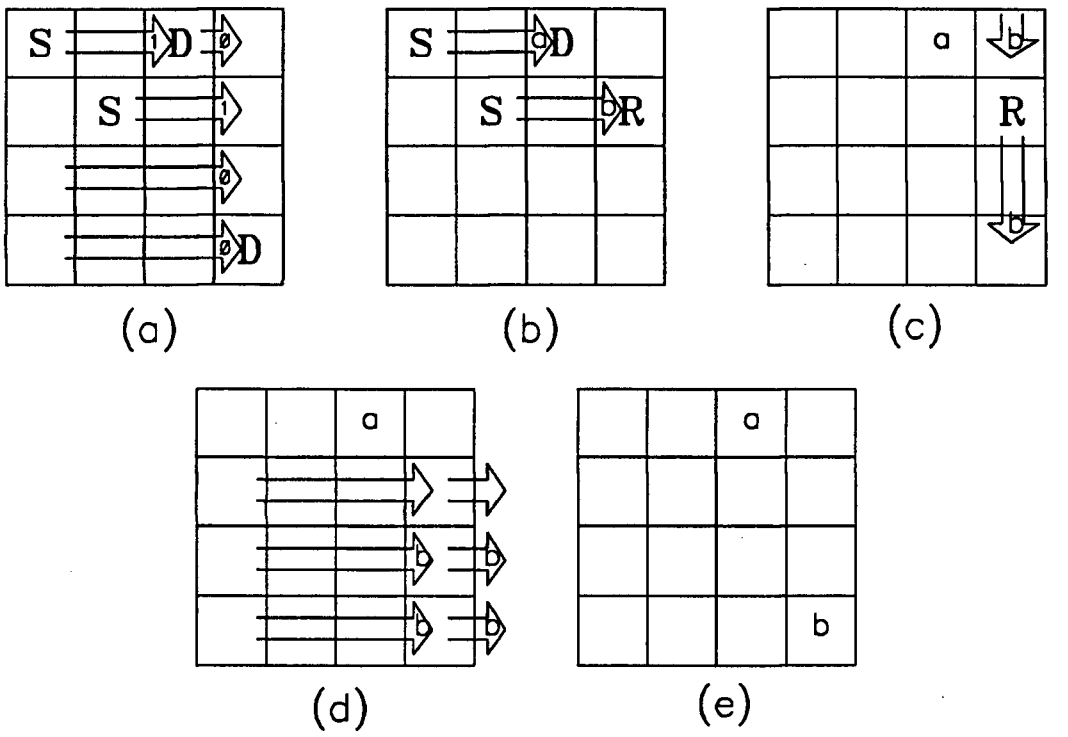


Figure 4.15. Direct segmented scan example. Processors marked S and D are the source and destination, respectively. (a) Senders broadcast a 1; all others broadcast a 0. (b) Edge PEs that received a 1 become routers (marked as R). Source PEs send out the actual data item. (c) Routers send the received data south which is (d) rebroadcast east. (e) Destination PEs that received a 0 in phase (a) accept the data item; other PEs keep the value sent in (b).

fine-grained parallel implementation achieves better performance by convolving with two simple filters instead of with one complex filter.

Convolution with a Gaussian can be approximated by repeated convolution with a triangular filter as described in [LBC89]. A triangular filter with weights $\frac{1}{4}$, $\frac{1}{2}$, and $\frac{1}{4}$ requires only arithmetic shifts. Further, since the Gaussian filter is separable, it can be implemented by two one-dimensional convolutions. The effective σ of a Gaussian filter implemented with this

technique is given by

$$\sigma = \sqrt{\frac{m+1}{2}} \tag{4.9}$$

where m is the number of iterations. This filter was implemented on *Silt* in [BB89], and has been optimized with the multioperand addition technique. The algorithm uses 16-bit clusters to maintain precision when adding and dividing 8-bit intensity values. One iteration requires 194 cycles.

The second stage is convolution with a discrete Laplacian kernel. The following kernel was shown by Horn [Hor86] to have desirable properties.

$\frac{1}{6}$

1	4	1
4	-20	4
1	4	1

Another advantage of the kernel is that most weights are simple sums of powers of two. For example, 20 is the sum of 16 and 4, so that multiplication by 20 requires a shift by 2, a shift by 4, and a summation. These multi-operand and multi-distance shifts can be implemented efficiently as shown earlier. This algorithm was used to obtain the edges shown in Figure 4.16.

The Laplacian and zero crossing calculation required 644 cycles, including a thinning step on the zero crossings. Total edge detection time for a Gaussian with a σ of 1.58, requiring four iterations of the triangle filter, is 1420 cycles, or 56.8 μseconds.

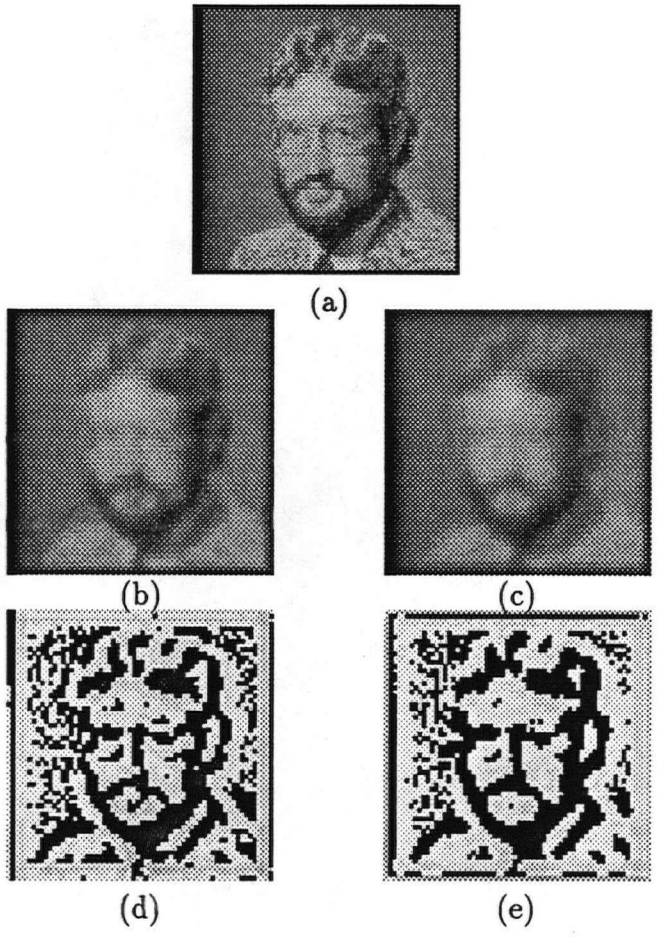


Figure 4.16. Edge detection with the $\nabla^2 G$ filter. (a) original image; (b) smoothed with a Gaussian, 2 iterations; (c) smoothed with a Gaussian, 4 iterations; (d) and (e) zero crossings of $\nabla^2 G$.

Surface Reconstruction

Surface reconstruction consists of determining the surface shape (height and slope) from a set of potentially sparse and noisy measurements. Harris [Har86] introduced the coupled depth/slope model for surface reconstruction and developed a set of equations suitable for a mesh-based massively parallel computer. In the equations below, u is the surface height, and p and q are the slopes in the x and y directions respectively.

$$u_{i,j}^{k+1} = \frac{1}{4}\left[(u_{i-1,j}^{k} + p_{i-1,j}^{k}) + (u_{i,j-1}^{k} + q_{i,j-1}^{k}) + (u_{i,j+1}^{k} - q_{i,j}^{k}) + (u_{i+1,j}^{k} - p_{i,j}^{k})\right] \quad (4.10)$$

$$p_{i,j}^{k+1} = \frac{1}{5}\left[(u_{i+1,j}^{k+1} - u_{i,j}^{k+1}) + p_{i,j-1}^{k} + p_{i,j+1}^{k} + p_{i-1,j}^{k} + p_{i+1,j}^{k}\right] \quad (4.11)$$

$$q_{i,j}^{k+1} = \frac{1}{5}\left[(u_{i,j+1}^{k+1} - u_{i,j}^{k+1}) + q_{i,j-1}^{k} + q_{i,j+1}^{k} + q_{i-1,j}^{k} + q_{i+1,j}^{k}\right] \quad (4.12)$$

A noticeable feature of the equations is the summation of multiple values at a pixel, which is computed efficiently by the multioperand addition algorithm. The other expensive computation is the division by 5 (or multiplication by binary 0.001100110011). This operation can be implemented by a special purpose sequence of shifts and adds. The shifts can be made especially efficient by decomposing the number into two fractions each with a repetition interval is four. Also, the shifts can be optimized by shifting the p and q sums together using the multiple-value shift discussed earlier. A single iteration of the algorithm requires approximately 1500 cycles.

4.5.2 Non-Local Algorithms

Histogram Calculation

Several methods for calculating image histograms using scan operations were described in [LBC89]. The sorting method is most appropriate for the *Silt* architecture. Image intensities are first sorted and the beginning of each sequence of identical intensities is identified. A segmented +-reduce operation determines the length of each segment, and the sums are permuted to the processor with the same ID as the intensity value. The most expensive operations here are the two sorting steps, which require approximately 25,000 steps each as shown earlier.

After a histogram is obtained, histogram equalization can be performed efficiently. The cumulative distribution is obtained with a *+-scan* on the histogram, and division by the total number of image pixels is trivial. Alternatives also exist for the task of actually transforming each pixel. First, if the sorted pixels retain their original coordinates, the equalization can be done while the pixels are still organized into segments, and the resulting pixels can be permuted to their original locations. Another possibility is to broadcast each original/equalized intensity pair to the entire array one at a time. For reasonably-sized histograms, the second approach is considerably more efficient.

Optical Flow

The optical flow of a changing scene is the apparent motion of the brightness field [Hor86]. Many algorithms have been proposed to determine optical flow. Several of these fall under the framework of regularization and may therefore be solved similarly to the surface reconstruction described earlier. An interesting algorithm that operates semi-locally is described in [LB88]. The basic idea is simple yet powerful.

The original image is displaced in a variety of directions. Each resulting displacement "layer" is correlated with the moved image in a large (11 by 11) window. In the case of binary image features, the correlation at each level is the number of feature pixels in the 11 by 11 neighbourhood that are present both in the moved image and the displaced image. Finally, each pixel chooses the layer with the highest correlation value in a WTA (winner take all) step. Images manipulated by the algorithm may consist either of simple brightness values or of more complex features such as edges.

The most time consuming step of the algorithm is the correlation. If binary feature detection is used, the multiplication step of the correlation is a simple logical AND. The remainder

of the work consists of summing the values in each window. Region summation can be done with scan operations, as described in [LBC89], but for small regions a shift-and-add technique is faster, especially since the multi-operand addition algorithm can be used.

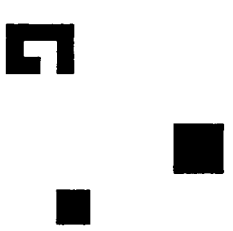
Since the number of layers is $(2\delta + 1)^2$ where δ is the maximum displacement, the winner-take-all (WTA) step is quadratic with the displacement. The time required for this step can be reduced if additional processors are available, as the WTA operation is binary and associative.

This algorithm has been implemented on the *Silt* simulator for the binary (present/absent) feature case. The results are shown in Figure 4.17. The edge detection algorithm discussed earlier could be used to obtain the features (in this case edges) from a raw intensity image.

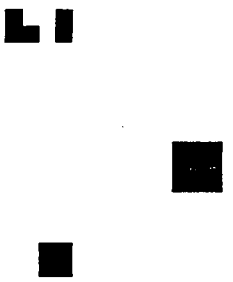
The time requirement for a displacement of 2 pixels and a 5×5 summation region is 8930 cycles, or 357 μseconds. The implementation was sequential in terms of layer processing; each displacement is processed serially at each pixel. After the correlation step, if the current value exceeds the best value, a counter corresponding to the current layer is remembered. The multi-operand addition technique is used to increment the counter efficiently. A full addition is required only at the end of the algorithm, after all displacements have been considered.

Connected Components

The task of determining connected components consists of labelling all pixels that have the same property (typically brightness), and are connected (typically 4- or 8-connected) with the same identifier. A scan-based algorithm described in [LBC89] fits well onto the mesh with bypass architecture.



(a)



(b)

```
2 2 2 2 1 1 0 0 0 0 0 0 0 0 0 0
2 2 2 2 1 1 0 0 0 0 0 0 0 0 0 0
2 2 2 2 1 1 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 3 3 3 3 3 3 3 0
0 0 0 0 0 0 0 0 3 3 3 3 3 3 3 0
0 0 0 0 0 0 0 0 3 3 3 3 3 3 3 0
0 0 0 0 0 0 0 0 3 3 3 3 3 3 3 0
9 9 9 9 9 4 0 0 3 3 3 3 3 3 3 0
9 9 9 9 9 4 0 0 0 0 0 0 0 0 0 0
9 9 9 9 9 4 0 0 0 0 0 0 0 0 0 0
9 9 9 9 9 4 0 0 0 0 0 0 0 0 0 0
```

(c)

```
3 5 5 3 2 2 0 0 0 0 0 0 0 0 0 0
3 5 5 3 2 2 0 0 0 0 0 0 0 0 0 0
3 5 5 3 2 2 0 0 0 0 0 0 0 0 0 0
2 3 3 2 1 1 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 2 3 3 3 2 1 0
0 0 0 0 0 0 0 0 2 4 6 6 6 4 2 0
0 0 0 0 0 0 0 0 3 6 9 9 9 6 3 0
0 0 0 0 0 0 0 0 3 6 9 9 9 6 3 0
0 0 0 0 0 0 0 0 3 6 9 9 9 6 3 0
0 0 0 0 0 0 0 0 2 4 6 6 6 4 2 0
1 2 2 2 2 1 0 0 1 2 3 3 3 2 1 0
1 2 2 2 2 1 0 0 0 0 0 0 0 0 0 0
1 2 2 2 2 1 0 0 0 0 0 0 0 0 0 0
1 2 2 2 2 1 0 0 0 0 0 0 0 0 0 0
```

(d)

Figure 4.17. Optical flow calculation with maximum displacement of 2 pixels and a 5×5 summation region. (a) original image. (b) displaced image. The top shape moved up by one pixel and partially off the image; the right shape moved down by one pixel and the lower shape moved three units down and one to the left. (c) displacement layer chosen by the algorithm. (d) region sum for winning layer.

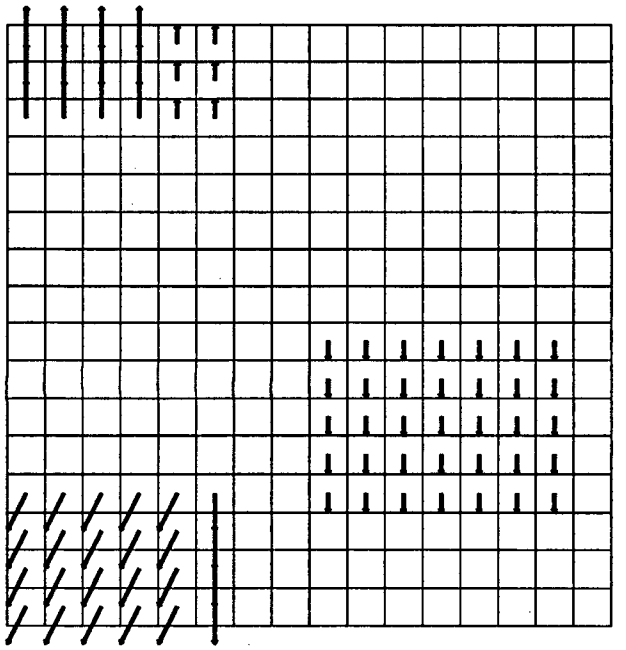


Figure 4.18. Optical flow field resulting from the raw data in Figure 4.17. Some confusion is caused by the movement of the top shape off the image, and by the movement of the lower shape by a distance exceeding the maximum displacement layer.

First, each pixel determines whether it is on a horizontal boundary. Then, a segmented *min-reduce* is performed on each row. Finally, the minimum value is broadcast to all pixels in the segment. This operation is then repeated in the vertical direction. The horizontal-vertical sequence must be repeated several times for non-convex objects at each iteration, another corner in the object is turned. For example, a U-shaped object is labelled in two iterations.

Disadvantages of the algorithm are obvious: diagonal objects require one iteration per diagonal pixel, and complex objects such as spirals are even worse.

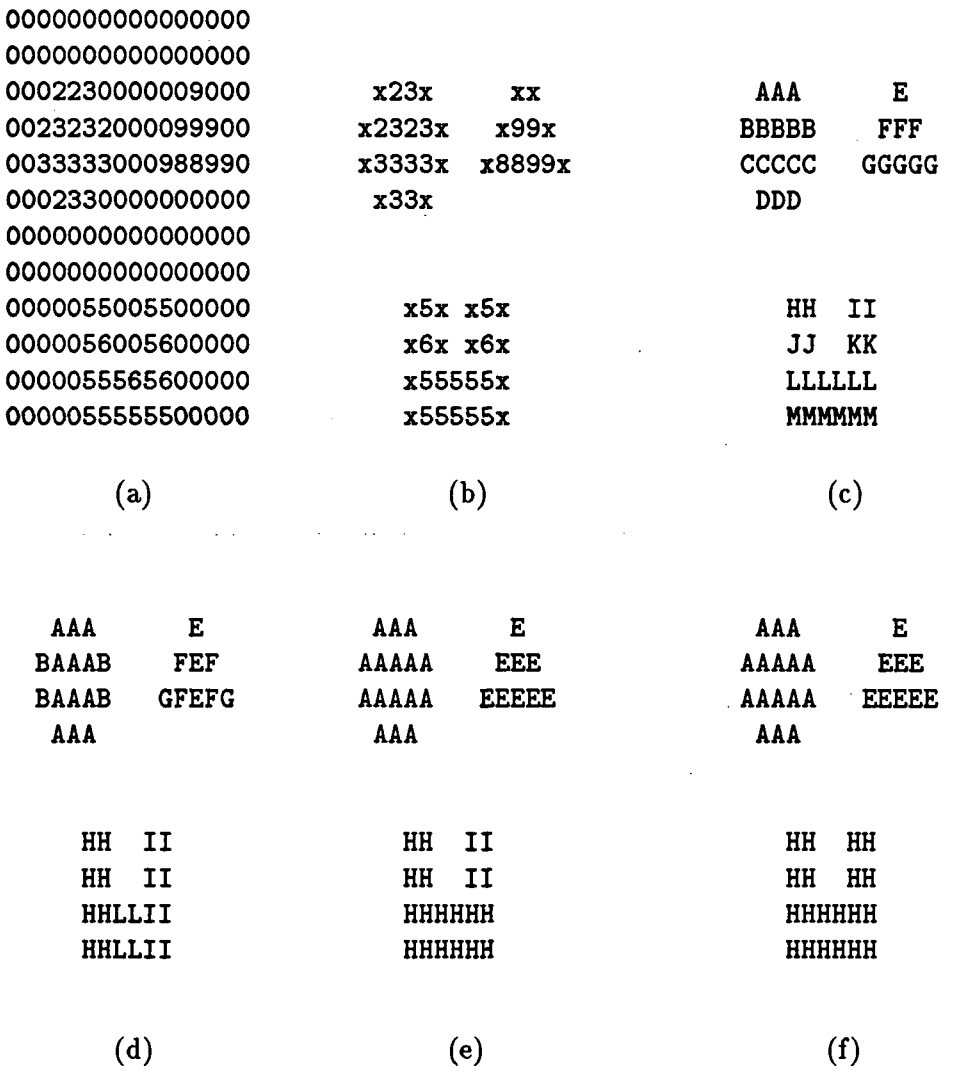


Figure 4.19. Connected component algorithm. (a) the original intensity image; (b) edge pixels have identified themselves, with a threshold of 2; (c) one horizontal iteration has been completed; (d) a vertical iteration completed; (d) another horizontal and (e) vertical iteration are required to label the U-shape.

Chapter 5

Discussion

Several observations have been made through the course of writing this thesis that are not particular to any specific section. These comments are briefly discussed here.

5.1 Asymptotics And Constants

In massively parallel systems of practical size, the constants in front of the asymptotic performance equations are very significant. For example, the number of cycles required for a bitonic sort was shown earlier to be approximately 23,000 for a DBP mesh, as compared to 19,000 on the Connection Machine's hypercube network. In fact, since the cycle time is expected to be almost an order of magnitude less on the *Silt* machine, sorting will be several times faster on the mesh. In this case, the performance improvement is due primarily to the faster data movement operation: a DBP network moves four bits per cycle compared to a bit-serial network's one bit. In the case of data moves internal to a PE, the ratio is even worse.

As another example, messages are generally believed to require $O(\log_2 n)$ steps to traverse a hypercube network. This is only true if communication time is independent of distance; since the longest link in a hypercube is $O(\frac{\sqrt{N}}{\log_2 N})$ and transmission time is at best logarithmic with distance, the total delay is much longer. For long wires and fast cycle times, the speed of light limitation results in propagation time linear with length, so that a hypercube

performs identically to a mesh, at a much higher wiring penalty. These results indicate that theoretical analysis of systems and algorithms must be very careful to use realistic models of the implementation environment.

5.2 Message Passing and Circuit Switching Revisited

Earlier discussion pointed out the relative advantages of message passing and circuit switching, and decided on a circuit-based network. However, the mesh with bypass has an element of message passing: since circuit switching is available only in horizontal and vertical directions, any network embedding that turns corners must be implemented using a mixture of circuit and message switching. Each corner node must store the direction that it must forward the message which consumes either two or four bits, depending on whether decoding time or memory is at a premium, and each corner turn requires four cycles, as the routing direction must be checked in SIMD fashion.

This restriction is made obvious in the connected components algorithm of Section 4.5.2 where the information can only propagate diagonally at the rate of one pixel per four cycles. The Coterie network discussed in [SNW89] implements complete circuit switching: an entire region can be interconnected electrically. This approach may be better for certain applications, despite the increased area requirements.

5.3 Flux-Constrained Processor Design

If VLSI technology keeps outstripping interconnect technology, the main limitation on chip design will be the number of pins. At that point, more area will be available for each processor. Many options exist for the allocation of that area. They include:

- *Increase local memory.* This is the obvious choice, but may not be necessary if most algorithms either fit in local memory, or have a tolerable performance penalty due to external memory accesses.
- *Increase complexity of ALU.* There isn't much increase possible if the memory bandwidth stays constant. One possibility is to use different busses for memory and links, so that up to two memory bits and two input bits may be operated on simultaneously.
- *Increase complexity of communication circuitry.* This is one of the most promising possibilities.
- *Increase degree of autonomy.* Several options were discussed in Section 3.2.2. More code must be analyzed to see which scheme results in the best program simplification.
- *Widen memory bandwidth.* Increase the number of read ports to three or the number of write ports to two. The ALU complexity must increase as well to take advantage of the available bits.
- *Speed up local memory.* For example, differential outputs can be provided for the dual port memory and sense amplifiers used to reduce read time.

Before any of these enhancements is selected, the cost of the additional area must be considered very carefully. An architectural feature should not be added until it can be demonstrated to reduce computation time sufficiently to pay back for the increase in area. This analysis requires that a set of representative programs be written assuming the existence of each feature.

5.4 Circuits To Algorithms

All circuits that have been converted to DBP algorithms share a set of common characteristics.

- Circuits are composed of multiple homogenous layers. Within each layer all processing units (PUs) are nearly identical. This criterion ensures that processor utilization is high at every stage.
- Each PU performs a simple computation. Since each PU is emulated by a single processor, the computation is done internal to the PE and is at best linear with the number of operands.
- Each PU has a small fanin. If many wires lead into a PU then emulation of the circuit involves a good deal of communication overhead. Also, a high fanin means that the computation has several inputs and is unlikely to be simple.
- PU interconnections have few crossovers. Since a mesh does not have any crossing wire, every crossover in the emulated circuit involves idling some processors.

Any circuit displaying these features is a good candidate for conversion into an algorithm. Unlike circuit design, a small fanout is not an important criterion, since the mesh with bypass supports efficient broadcast.

5.5 Memory, Time, and Area Tradeoffs

Many of the DBP arithmetic operations shown are derived from physical circuits. Although the spatial structure of the circuit is converted to the temporal structure of the algorithm, the information content has not changed. However, the algorithmic representation of the circuit connectivity is cheaper in terms of area, which in this case is a more expensive resource than time. Similarly, when memory is the resource to be conserved, constants can be generated through computation. The extreme example of representing information implicitly in the algorithm rather than explicitly as constants is the Gaussian convolution technique described in Chapter 4. Here, a simple sequence of local operations produces the same results as a

convolution without the expense of storing the convolution coefficients.

5.6 Programming A DBP Machine

Optimizing SIMD code is straighforward and opportunities for optimization are frequent. For example, many arithmetic primitives move the final result into a destination register. This register is then frequently moved into a communication port. A compiler can optimize the last step of the primitive by moving the result directly into the port, avoiding the register. Another possibility for optimizations occurs because the arithmetic primitives are careful about preserving the state of the bypass registers. Since these registers may be set unconditionally by the next primitive, a compiler can eliminate the unnecessary state restoration. Data flow analysis is made especially easy by the SIMD nature of the architecture because branches do not exist. These optimizations are similar in spirit to those possible in a RISC architecture.

Chapter 6

Conclusions

In summary, the thesis developed a distributed bit-parallel architecture and algorithms closely matched to that architecture. The resulting combination provides sufficient computational power to perform early vision tasks in real time. The main reason for this power is very high computation and communication bandwidth.

The computation bandwidth is due to the large number of processing elements. Since each bit of a word is allocated to a processor, the entire word can change each cycle. In contrast, a bit-serial machine can change only one bit in each word per cycle. This requirement for a large number of PEs demands that each PE be as small as possible. The main effect of this restriction is to reduce local memory size. Analysis of VLSI and algorithmic constraints showed that a small memory results in only a minor performance penalty. A VLSI implementation of a 16-PE chip demonstrated that a DBP PE can be made as small as 600 μm on a side, even in a non-aggressive VLSI technology.

The unique ability of a DBP organization to operate on each bit of each data word simultaneously was used to develop efficient arithmetic primitives. A technique for multi-operand addition based on the carry-save adder was implemented and shown to be several times faster than repeated two-operand addition. Since summations over image areas are common in vision algorithms, the multi-operand addition technique is very useful. Another application of this technique is the accumulation of partial sums during a multiplication algorithm.

The high communication bandwidth of the architecture is due to several factors. First, the network is circuit switched, eliminating routing delays. Second, the bypass capability allows distant processors to communicate quickly. This capability was enhanced by the development of a fast bypass circuit. Third, the network supports a one to many broadcast feature. Finally, communication occurs over a wide bus. Since a cluster is arranged in a square, $\sqrt{k}$ bits are transmitted at once.

The broadcast capability of the network led to the development of a fast parallel prefix algorithm based on the Ladner-Fischer circuit, superior in both time and memory requirements to previous tree-based algorithms. In fact, while tree-based algorithms require storage proportional to the tree height, the new algorithm requires only constant storage.

Inter-cluster parallel prefix computation benefits even more from the algorithm. While a tree-based scan of 64K clusters requires 16 temporary locations, the new algorithm requires only one. As a result, much more memory is available for storing image data, and the need for large local memories at each PE is reduced.

A summary of various arithmetic, scan, and vector operations is shown in Table 6.1. The time requirements shown are based on a 40 ns cycle time.

Operation	Cycles	Time (μsec)
Add, 16-bit	43	1.7
Multiply, 16-bit	670	26.8
Add, 32-bit	52	2.1
Multiply, 32-bit	1400	56
Edge Detection	1420	56.8
Optical Flow	9000	360

Table 6.1. Summary of DBP algorithms

6.1 Future Work

Future research directions are numerous, since this work has barely scratched the surface of the potential of the DBP model. A partial list is shown here.

- *Cluster reconfiguration.* If a particular cluster organization is more suitable for a given algorithm, the row-major layout can be suspended for the duration of that algorithm.
- *Multiple PEs per data bit.* If at a particular stage in an algorithm more processors are available than there are data bits, highly parallel structures can be created for even faster execution. For example, a parallel multiplier can perform multiplication with time logarithmic with word size rather than linear, if sufficient PEs are available.
- *Multigrid methods.* The convergence rate of many nearest-neighbour vision algorithms can be increased with the use of multigrid methods, as the low frequency components will propagate much faster. The mesh with bypass is well suited for this technique since clusters can communicate over large distances quickly.
- *Random number generation. Silt* operating as cellula automata can be used as a high–quality random number generator [HMC89] in the implementation of several stochastic vision algorithms [MMP87].

References

[BB89] Roderick A. Barman and Michael Bolotski. Silt: Why Fine Grained Isn't Fine Enough. Graduate Course Term Report, May 1989.

[BBCL90] Roderick A. Barman, Michael Bolotski, Daniel Camporese, and James J. Little. Silt: A Bit-Parallel Approach. In *International Conference on Pattern Recognition.* IEEE, 1990.

[BDHR88] D. W. Blevins, E. W. Davis, R. A. Heaton, and J. H. Reif. Blitzen: A Highly Integrated Massively Parallel Machine. In *Frontiers of Parallel Computation '88*, 1988.

[BHMSV84] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. L. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis.* Kluwer Academic Publishers, 1984.

[BK82] R. P. Brent and H. T. Kung. A Regular Layout for Parallel Adders. *IEEE Trans. on Computers*, May 1982.

[Ble89] Guy E. Blelloch. Scans as Primitive Parallel Operations. *IEEE Trans. on Computers*, 38(11):1526–1538, Nov 1989.

[BRSV87] R. K. Brayton, R. Rudell, and A. L. Sangiovanni-Vincentelli. MIS: A Multiple-Level Logic Optimization System. *IEEE Trans. Computer-Aided Design*, CAD-6:1062–1081, Nov 1987.

[BWG85] Peter Beadle, Janet Wiles, and Leslie M. Goldschlager. *Implementation of an ALU by a Parallel Machine*, chapter VLSI: Algorithms and Architectures, pages 153–165. North-Holland, 1985.

[Can86] J. F. Canny. A Computational Approach To Edge Detection. *IEEE Trans. Pattern Analysis and Machine Intellegence*, PAMI-8:679–698, 1986.

[Clo88] Eugene L. Cloud. The Geometric Arithmetic Parallel Processor. In *Frontiers of Parallel Computation '88*, 1988.

[Dal87] William J. Dally. *A VLSI Architecture for Concurrent Data Structures.* Series in VLSI, Computer Architecture, and Digital Signal Processing. Kluwer Academic Publishers, Boston/Dordrecht/Lancaster, 1987.

[FK88] Jeff Fried and Bradley C. Kuszmaul. NAP (No ALU Processr), The Great Communicator. In *Frontiers of Parallel Computation '88*, pages 383–389, 1988.

[Fou87] Terry Fountain. *Processor Arrays: Architectures and Applications.* Harcourt Brace Jovanovich, 1987.

[Har86] John G. Harris. The Coupled Depth/Slope Approach To Surface Reconstruction. Master's thesis, MIT, May 1986.

[Hil85] D. Hillis. *The Connection Machine.* Distinguished Dissertations. The MIT Press, Cambridge, Massachusetts, 1985.

[HMC89] P. D. Hortensius, R. D. McLeod, and H. C. Card. Parallel Random Number Generation for VLSI Systems Using Cellular Automata. *IEEE Trans. on Computers*, 38(10):1466–1472, Oct 1989.

[Hor86] Berthold Klaus Paul Horn. *Robot Vision.* MIT Electrical Engineering and Computer Science Series. The MIT Press, Cambridge, Massachusetts, 1986.

[HS81] B. K. P. Horn and B. G. Schunck. Determining Optical Flow. *Artificial Intelligence*, 17:185–203, 1981.

[Hwa79] Kai Hwang. *Computer Arithmetic.* John Wiley and Sons, 1979.

[IH81] H. Ikeuchi and B. K. P. Horn. Numerical Shape from Shading and Occluding Boundaries. *Artificial Intelligence*, 17:141–184, Aug 1981.

[Joh90] Robert R. Johnson. Multichip modules: next-generation packages. *IEEE Spectrum*, March 1990.

[KH83] M. Kumar and D. S. Hirschberg. An Efficient Implementation of Batcher's Odd-Even Merge Algorithm and Its Applications in Parallel Sorting Schemes. *IEEE Trans. on Computers*, C-32(3):254–264, March 1983.

[KH88] D. Kim and K. Hwang. Mesh-connected array processors with bypass capability for signal/image processing. In *Hawaii Conf. on Syst. Sci*, 1988.

[LB88] James J. Little and Heinrich H. Bülthoff. Parallel Optical Flow Using Local Voting. AI Memo 929, MIT, July 1988.

[LBC89] James J. Little, Guy E. Blelloch, and Todd A. Cass. Algorithmic Techniques for Computer Vision on a Fine-Grained Parallel Machine. *IEEE Trans. Pattern Analysis and Machine Intellegence*, 11(3):244–257, 1989.

[Lei83] Frank Thomson Leighton. *Complexity Issues in VLSI.* Foundations of Computing Series. The MIT Press, Cambridge, Massachusetts, 1983.

[Lei85] Tom Leighton. Tight Bounds on the Complexity of Parallel Sorting. *IEEE Trans. on Computers*, C-34(4):344–354, Apr 1985.

[LF80] R. E. Ladner and M. J. Fischer. Parallel Prefix Computation. *Journal of the ACM*, 27(4):831–838, Oct 1980.

[LM89a] H. Li and M. Maresca. Flux and Fluid. *IEEE Trans. on Computers*, 38(9):1346–1351, Sep 1989.

[LM89b] Hungweng Li and Massimo Maresca. Polymorphic-Torus Architecture for Computer Vision. *IEEE Trans. Pattern Analysis and Machine Intellegence*, 11(3):233–243, 1989.

[Mar82] David Marr. *Vision.* Freeman, 1982.

[MC80] Carver Mead and Lynn Conway. *Introduction to VLSI Systems.* Addison-Wesley Series in Computer Science. Addison-Wesley, 1980.

[MKRS88] R. Miller, V. K. P. Kumar, D. Reisis, and Q. Stout. Meshes with reconfigurable buses. In *5th MIT Conference on Advanced Research in VLSI*, 1988.

[ML89] M. Maresca and H. Li. *Toward connection autonomy of fine-grain SIMD parallel architecture*, volume Parallel Processing for Computer Vision and Display, chapter 5, pages 77–86. Addison-Wesley, 1989.

[MMP87] J. Marroquin, S. Mitter, and T. Poggio. Probabilistic Solution of Ill-Posed Problems in Computational Vision. *Journal of the American Statistical Association*, 1987.

[Nis81] H. K. Nishihara. Intensity, Visible-Surface, and Volumetric Representations. *Artificial Intelligence*, 17:265–284, Aug 1981.

[NS79] David Nassimi and Sartaj Sahni. Bitonic Sort on a Mesh-Connected Parallel Computer. *IEEE Trans. on Computers*, C-27(1):2–7, Jan 1979.

[Pot85] Jerry L. Potter, editor. *The Massively Parallel Processor.* MIT Press Series in Scientific Computation. The MIT Press, Cambridge, Massachusetts, 1985.

[PV81] F. Preparata and J. Vuillemin. The Cube-Connected-Cycles: A Versatile Network for Parallel Computation. *Communication of the ACM*, 24(7):300–310, July 1981.

[Rus89] Andrew Rushton. *Reconfigurable Processor-Array: a bit-sliced parallel computer.* Research Monograph in Parallel and Distributed Computing. Pitman, 1989.

[SNW89] David B. Shu, Greg Nash, and Charles Weems. *Image Understanding Architecture and Applications*, chapter 9, pages 297–355. Springer-Verlag, 1989.

[Sor61] O. L. Mac Sorley. High speed Arithmetic in Binary Computers. *Proceedings of the IRE*, 49:67–91, 1961.

[Sto83] Q. F. Stout. Mesh-Connected Computers with Broadcasting. *IEEE Trans. on Computers*, C-32(9):826–830, Sep 1983.

[TA77] A. N. Tikhonov and V. Y. Arsenin. *Solutions of Ill-Posed Problems.* Winston and Sons, 1977.

[TP80] V. Torre and T. Poggio. On Edge Detection. AI Memo 768, AI Laboratory, MIT, 1980.

Appendix A

Memory Cost

A memory array is organized as a matrix of bit cells with each column connected to a bus as shown in Figure A.1. During a memory access, one cell per column is enabled onto the bit line, and the appropriate column is selected by multiplexing logic at the bottom of the matrix.

The time required to drive the bus wire using a tree structure of cascaded drivers is given by [MC80, chapter 8]

$$t = \alpha \tau b_0 \log_\alpha S, \tag{A.1}$$

where τ is a constant, b_0 is the initial transistor spacing, S is the number of transistors, and α is the tree branching ratio. This area is replicated S times, once for each column. The area of the tree structure is

$$A_{memory} = \log_\alpha S b_0^2 S^2. \tag{A.2}$$

The row enable wires also have non-negligible capacitance and must be driven with a series of progressively larger inverters. The area occupied by the inverter chain is

$$A_{row} = K_n b_0^2 \frac{eS^2}{e-1}, \tag{A.3}$$

where K_n is the ratio of n-channel transistor area to RAM cell area, b_0^2.

Therefore, the area of each PE is given by

$$A_{PE} = b_0^2 \left[K_{ac} + S^2 \left(\frac{\log S}{\log \alpha} + \frac{K_n e}{e-1} \right) \right] \tag{A.4}$$

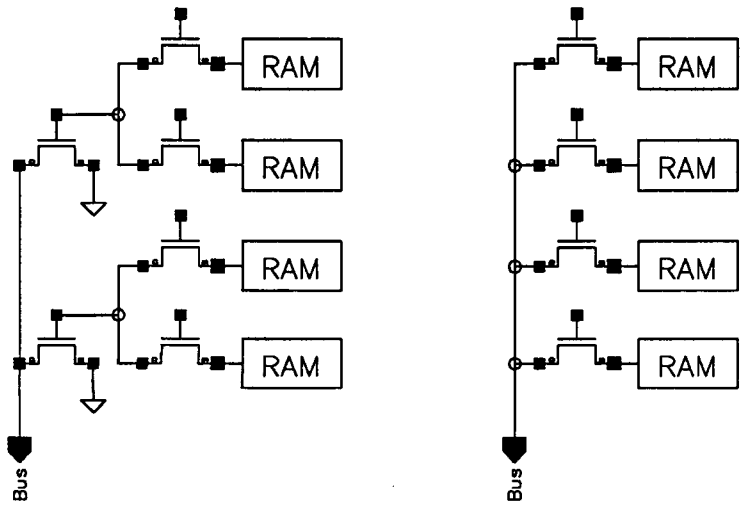


Figure A.1. RAM bit line organization. The tree array is shown on the left, and the linear array on the right. The transistor sizes on successive levels of the tree are scaled by α.

where K_{ac} is the area of the non-memory elements of the processor, measured in units of RAM cells. For the remainder of the discussion, consider b_0 to be normalized to 1.

There are several potential cost functions. We could attempt to minimize the overall area, the area-time product AT, or the area-time squared product, AT^2. The cost as a function of memory capacity $m = S^2$, assuming $\log \alpha = 2$. The other source of overhead is m_c, the constant number of bits required in each processor during the execution of DBP algorithms, and therefore unavailable for the storage of external data. Examples are intra-cluster processor IDs, segment flags, and the temporaries required during the addition and multiplication algorithms. This overhead is usually in the range of 10-20 bits.

The three cost functions, with the parameter memory capacity $m = S^2$ are shown below:

$$C_A(m) = \frac{1}{m - m_c}\left[K_{ac} + m\left(\frac{\log m}{4} + \frac{K_n e}{e-1}\right)\right] \tag{A.5}$$

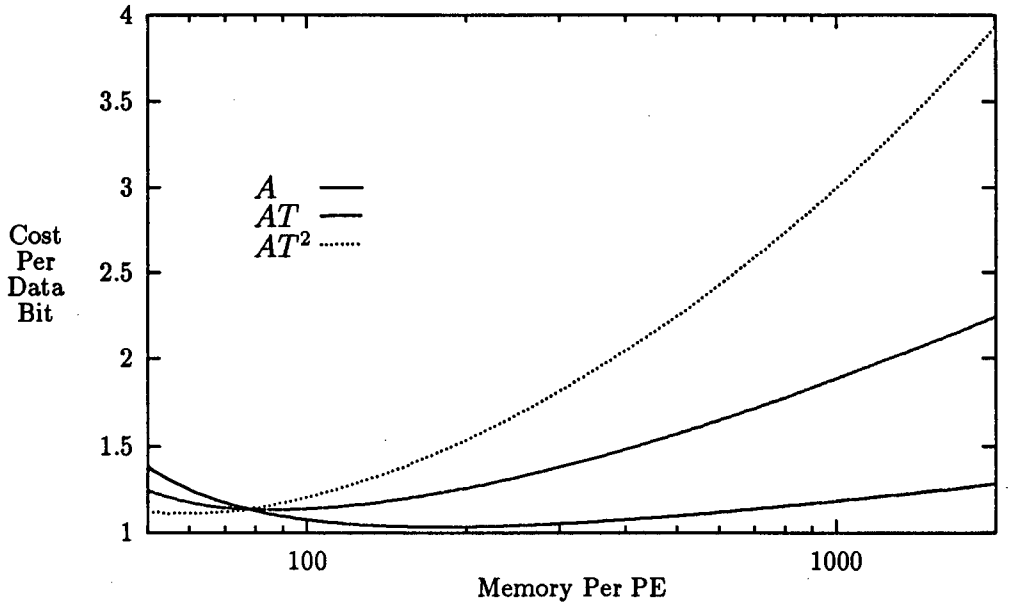


Figure A.2. Cost per Effective Data Bit

$$C_{AT}(m) = \frac{1}{m - m_c} \log m \left[K_{ac} + m \left(\frac{\log m}{4} + \frac{K_n e}{e - 1} \right) \right] \quad \text{(A.6)}$$

$$C_{AT^2}(m) = \frac{1}{m - m_c} \log^2 m \left[K_{ac} + m \left(\frac{\log m}{4} + \frac{K_n e}{e - 1} \right) \right] \quad \text{(A.7)}$$

The optimal value of $\log \alpha$ independent of m is 2, as shown in [MC80]. The VLSI implementation described in Section 3.3 had $K_{ac} = 30$ and $K_n = 0.1$. A graph of A, AT, and AT^2 cost functions is shown in Figure A.2.

The graph shows that a small memory size is in some sense "optimal" under several cost models. This is obviously not an absolute statement since small memories will suffer from cache misses more frequently. What it does demonstrate is that small memory sizes are not inherently bad because of non-memory overhead.

A.1 Area of Row Drivers

The capacitive load to be driven, C_L is SC_g where S is the number of columns and C_g is the drain capacitance of the n-channel pass transistor. The capacitance of the final output stage of the cascaded inverters should be equal to C_L. The gate capacitance of an inverter, $C_i = XC_g$. For a CMOS inverter with the p-channel transistor twice the size of the n-channel transistor, $X = 3$. The area of an inverter A_{inv} is XA_n, where A_n is the area of the pass transistor.

$$C_L = C_i e^n \tag{A.8}$$

$$n = \log\left(\frac{C_L}{XC_g}\right) = \log\frac{S}{X} \tag{A.9}$$

where n is the number of stages. The total area of the stages is

$$A = A_{inv}\sum_{i=0}^{n} e^i \tag{A.10}$$

$$\approx A_{inv}\frac{e^{n+1}}{e-1} \tag{A.11}$$

$$\approx A_{inv}\frac{Se}{X(e-1)} \tag{A.12}$$

$$\approx A_n\frac{Se}{e-1} \tag{A.13}$$

Since there are S rows, the final row driver area is

$$Arow = A_n\frac{S^2}{e-1} \tag{A.14}$$

Appendix B

Binary Addition

The equations for the sum S of two binary numbers A and B are

$$S_i = A_i \oplus B_i \oplus C_{i-1}, \tag{B.1}$$

$$C_i = A_i \cdot B_i + A_i \cdot C_{i-1} + B_i \cdot C_{i-1}. \tag{B.2}$$

for $k \geq i \geq 0$. This formulation corresponds to the standard bit-serial method of adding two binary numbers. These sequential operations may be transformed into a sequence of associative operations by redefining S_i and C_i as

$$S_i = p_i \oplus C_{i-1}, \tag{B.3}$$

$$C_i = g_i + p_i \cdot C_{i-1}, \tag{B.4}$$

where

$$g_i = A_i \cdot B_i, \tag{B.5}$$

$$p_i = A_i \oplus B_i. \tag{B.6}$$

The g_i and p_i are the *generate* and *propagate* signals. The generate signal indicates that the current bit position will generate a carry; the propagate signal indicates that the current bit position will propagate an incoming carry. These signals are computed locally at each bit position.

C_i can now be expanded:

$$C_i = g_i + p_i g_{i-1} + p_i p_{i-1} g_{i-2} + \ldots + p_i \ldots p_1 C_0. \tag{B.7}$$

Define the operator o as

$$(g,p) \circ (g',p') = (g + (p \cdot g'), p \cdot p'). \tag{B.8}$$

Rewrite Equation B.7 strictly in terms of the o operator to yield

$$(G_i, P_i) = \begin{cases} (g_1, p_1) & \text{if } i = 1, \\ (g_i, p_i) \circ (G_{i-1}, P_{i-1}) & \text{if } 2 \leq i \leq \mathrm{n}\ , \end{cases}$$

In other words, (G_i, P_i) consists of the o operator applied to all lower bits; this type of operation is called parallel prefix. The first stage of addition proceeds by calculating the o of each pair of (g_i, p_i). After $\Theta(\log_2 k)$ steps, parallel prefix of the lower half of the word is available at the top of the tree.

The prefix is then propagated unchanged to the right subtree and concatenated with the current prefix before being propagated to the left subtree . At the bottom of the tree, the prefix values are used directly to compute the sum.

In the implementation, the tree structure exists only in the time dimension; no processors are dedicated as interior tree nodes.

Appendix C

Sample Source Code

In the subsequent discussion, segments of *Silt* assembly code will be presented. The four communication links are denoted by `N`, `E`, `W`, and `S`. The idle bit is `I`, and the alternate idle bit is `J`. The use of the alternate bit is indicated by prefixing the expression with a `!`. Each processor's ID within a cluster is stored in `I0-3`. Other language elements are `when`, `repeat`, and `bit`.

```
proc begin_csa(a,b, ps, pc) {
  ps = a ^ b;
  pc = a & b;
  lsl(pc);              // can optimize 1 op by inlining
}

proc add_csa(ps, pc, c) {
  bit t1;
  bit t2;

  t1 = pc ^ ps;         // S' = S ^ C ^ Q
  t1 = c ^ t1;          // new sum

  t2 = pc | ps;         // new carry
  t2 = t2 & c;
  pc = pc & ps;
  pc = pc | t2;         // C' = SB + Q(S+C)
  ps = t1;

  lslT(pc, t1);         // shift the carry
}

proc end_csa(ps, pc, dst) {
  add(ps, pc, dst);
}
```

Figure C.1. Carry Save Addition routines. The initialization corresponds to the half-adders in the circuit; the intermediate step is repeated for operand after the second and corresponds to the full adders; the final stage is a full parallel adder.

```
proc add(a, b, c) {
  bit P;
  bit G;
  bit T;
                /* initialization */
  P = a ^ b;
  G = a & b;
                /* level 0 */
  E = G & ~I0;  // send bit or identity
  T = P & W;
  G = G | T;
  E = P | I0;   // send bit or identity
  P = P & W;
                /* level 1 */
  H = ~I0;
  E = G & ~I1;  // send bit or identity
  T = P & W;
  G = G | T;
  E = P | I1;
  P = P & W;
                /* level 2 */
  H = ~I0 | ~I1;
  W = G & ~I2;  // send bit or identity
  S = E;
  when (~H)
    S = G & ~I2;
  T = P & N;
  G = G | T;
  W = P | I2;
  S = E;
  !  S = P | I2;// J is still H here
  P = P & N;
                /* level 3 */

  V = ~I2;      // bypass alternate rows
  W = G & ~I3;
  S = E;
  when (~V & ~H)
    S = G & ~I3;

  T = P & N;
  G = G | T;

  V = 0;
  lslT(G,T);    /* logical shift left */

  c = a ^ G;
  c = b ^ c;
}
```

Figure C.2. LF-based 16-bit addition. Note that the code required at each level increases slightly due to the overhead of broadcasting to multiple rows.

```
proc lsr(b) {
  W = b;                // shift all west
  b = E;
  H = ~I0 | ~I1;        // bypass cluster
  when (~H) {           // PEs in adjacent column
    E = b;              // send column back
    N = W;              // shift up
    b = S & ~msb;       // accept and clear MSB
  }
  H = 0;                // restore network
}


proc asr(b) {
  bit t;

  t = msb & b;          // save MSB
  W = b;                // Divide A by 2
  b = E;
  H = ~I0 | ~I1;        // bypass cluster
  when (~H) {           // PEs in adjacent column
    E = b;              // send column back
    N = W;              // shift up
    b = S & ~msb;       // accept and clear MSB
    b = b | t;          // replace with kept msb
  }
  H = 0;                // restore network
}


proc lsl4(a) {
  S = a;                // send cluster south
  a = ~I3 & ~I2;        // determine if LS row
  a = ~a & N;           // clear if in LS row
}
```

```
proc lsr2(a) {    // shift by 2 positions: shift by 4 then back by 2
  bit T;

  T = I2 & I3;    // T == 1 in one of the top 2 bits
  T = T & I1;
  W = a;
  N = a;

  W = E;
  a = E & ~T;

  E = S;
  E = W;
  when (I1)
    a = W & ~T;
}

proc lsr_2val(a, b) {   // two operand lsr (shift 2 at once)
  W = a;                // send A
  a = E;                // accept A
  W = b;                // send B
  b = E;                // accept b
  H = ~I0 | ~I1;
  when (~H) {
    E = b;
    N = W;
    b = S & ~msb;
    E = a;
    N = W;
    a = S & ~msb;
  }
  H = 0;
}
```

Figure C.3. The first procedure shifts a word by two positions. The second shifts two words by one bit. In the second case, the overhead of setting and clearing the bypass bit is reduced.