Design Tradeoffs in the Development of the Low-Level Processor for the Image Understanding Architecture

Charles C. Weems, Deepak Rana
David B. Shu, J. Gregory Nash

COINS TR 90-09

February 1990

Design Tradeoffs in the Development of the Low-Level Processor for the Image Understanding Architecture*

Charles C. Weems, Deepak Rana
Computer and Information Science Department
Lederle Graduate Research Center
University of Massachusetts
Amherst, MA 01003

David B. Shu, J. Gregory Nash
Hughes Research Laboratories
M/S RL 69
3011 Malibu Canyon Road
Malibu, CA 90265

Point of Contact: Chip Weems, (413) 545-3163, Weems@CS.UMass.EDU

* This work was supported in part by the Defense Advanced Research Projects Agency Under contract number DACA76-86-C-0015, monitored by the U.S. Army Engineer Topographic Laboratory, and contract number F49620-86-C-0041, monitored by the Air Force Office of Scientific Research.

Design Tradeoffs in the Development of the Low-Level Processor for the Image Understanding Architecture

1. Abstract

Now that the design of the Image Understanding Architecture (IUA) is cast in silicon and the first prototype is nearing completion, it is a good opportunity to consider the development process in retrospect and relate what we have learned.

The IUA is the first parallel processing system that has been designed from conception to address the computational requirements of knowledge-based, real-time computer vision. The vast diversity of processing required for state-of-the-art vision systems presents a unique challenge to the computer architect. The requirements of vision have led not only to the development of a processor that simultaneously supports multiple granularities and models of parallelism, but also to the development of at least one new model of parallelism.

This paper describes the architectural decision-making process that went into the design of the low-level (sensory) processing portion of the IUA. Prior to this discussion, the processing requirements of a typical vision system are presented, and the overall design of the IUA is briefly reviewed to establish the proper context.

2. Introduction

Computer vision using color imagery requires a processor capable of accepting 23 megabytes of input per second, and interpreting it to construct a three dimensional model of the sensor's environment. An interpretation may require hundreds of objects of many different types to be identified [Draper, 1989]. Complicating factors include the presence of noise, occlusion, uneven lighting, shadows, specular reflections, and motion effects. A typical real-time scenario allows a few seconds for initial orientation, and then requires updating of the environmental model up to thirty times per second.

Vision researchers [Hanson, 1986] have shown that pattern recognition techniques, by themselves, are inadequate for this task. Consider an image of a window: Parts of the window can be reflective, transparent, or both; it also introduces distortion and specularity. Despite the fact that the window has no

characteristic pattern, we perceive it as a separate object because of our knowledge of the properties of glass, and the window's surroundings. Even if the window were perfectly transparent, the presence of the window-frame would allow us to infer the existence of a pane of glass. In fact, most of what we "see" in natural scenes is really inferred from partial information.

Thus, it is clear that vision involves both sensory and knowledge-based processing. However, between these two levels of abstraction it is useful to introduce one or more additional levels of symbolic processing. Symbols range in complexity from descriptions of extracted image events (such as edges or regions) through perceptually useful groupings of events (such as geometric figures or surfaces), to descriptions of object parts or entire objects. Although the level of abstraction of these intermediate representations may vary, they share a similarity of structure and processing. Thus, vision researchers tend to classify algorithms and representations into three levels: low (sensory), intermediate (symbolic), and high (knowledge-based). For the architect, a vision system therefore presents three distinct sets of computational requirements (Table 1).

	Low Level	Intermediate Level	High Level
Computation	• Fine grained • 256K 8-bit pixels • 8-bit integer arithmetic • Limited real arithmetic • Comparisons	• Medium grained • Thousands of "tokens" • 16-bit integer arithmetic • 32-bit real arithmetic • Building "token" records • Maintaining lists of token relationships	• Coarse grained • Hundreds of "schemas" • 32 bit real and integer arithmetic • List traversals • Symbolic processing
Communication	• Local neighborhood • Across connected components • Structured patterns • Broadcast • Up • Summary feedback • High-speed I/O • Fine grained messages	• Local neighborhood • Long distance • Broadcast • Down and up • Summary feedback • Medium length messages	• Blackboard access • Control info to lower levels • Queries to lower levels • Data up from lower levels • Coarse grained messages
Control	• SIMD-Associative • Multi-SIMD • Locally associative SIMD • Central control • Local activity control	• SIMD-Associative • Synchronous-MIMD • MIMD directed by higher level • Central and local control	• MIMD • Distributed control • Attention focusing mechanisms • Coordination with central control of lower levels

Table 1. Computation, Communication, and Control Requirements of Each Level of Abstraction in a Vision System

While it may be argued that a general-purpose processor can fulfill all of the requirements of vision, the goal of real-time performance necessitates the use of special-purpose processors that take advantage of the different requirements at each level. Another key to achieving real-time performance is to allow processing to take place at all levels simultaneously, which leads to the idea of linking together three different parallel processors. But because of the massive amount of communication between levels, and the requirement for flexible, top-down control, the different parallel processors must be designed from the start to be tightly coupled with each other. It was basically this analysis that lead to the concept of the IUA.

3. Overview of the IUA

The Image Understanding Architecture [Weems, 1989] represents a hardware implementation of the three levels of abstraction inherent in our view of computer vision. It consists of three different, tightly coupled parallel processors (Figure 1). The low- and intermediate-levels are controlled by a dedicated Array Control Unit (ACU) that takes its directions from the high level. As Figure 1 indicates, each of these processors provides a different granularity and different modes of parallelism. The low level is a fine-grained, processor-per-pixel array of bit-serial processing elements. The intermediate is a medium grained, multicomputer array of 16-bit processors. The high level is a coarse-grained multiprocessor oriented toward manipulating models, inference, running LISP, etc.

We are currently building a 1/64th slice of the IUA as a proof-of-concept demonstration. The discussion that follows describes the full IUA, except where it is specifically noted that a feature pertains only to the prototype.

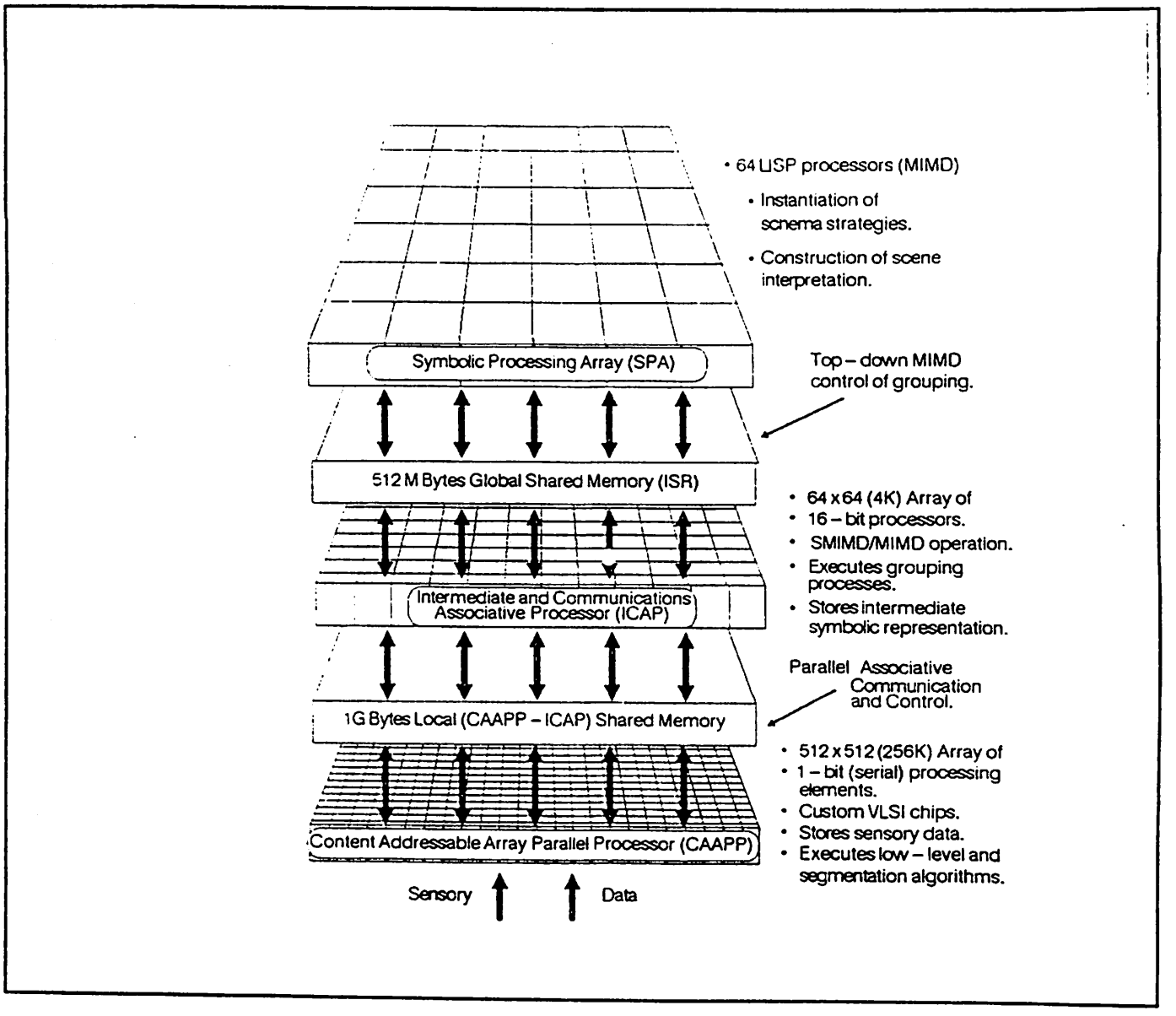


Figure 1. IUA Overview

At the high level, the IUA is purely a MIMD parallel processor. The low level operates in pure SIMD or local-SIMD mode, and the intermediate level operates in synchronous-MIMD or pure MIMD mode. Briefly, local-SIMD and synchronous-MIMD differ from the familiar SIMD and MIMD modes as follows. In local-SIMD, the PE's execute a single instruction stream, but are arranged into disjoint groups, with each group able to locally broadcast values, and compute its own summary values in parallel with other groups. Synchronous-MIMD mode is more like SIMD than MIMD: the processors execute the same program with autonomous

instruction pointers so that they can branch independently, but they resynchronize for each stage of processing. Synchronous-MIMD has the advantage of being as simple to program as a SIMD system but without the SIMD time penalty of having to sequentiallize the paths in a branch.

4. Tradeoffs in the Development of the Low-Level Processor

Our approach to the design of the low-level processor was based on our experience with content addressable, or associative, parallel processors [Foster, 1976] for real-time applications. The guiding principles that came out of this work were:

One processor per data element wherever possible. Folding multiple "virtual" processors onto a smaller number provides greater flexibility, but creates a significant overhead and reduces the predictability of a system's real-time behavior.

Common vision operations should be roughly as fast as scalar operations on uniprocessors. The low level should be thought of as an "image operand machine."

Fast I/O with minimal disruption of processing. The processor must not starve for I/O. Long idle periods during I/O waste cycles and wreak havoc with meeting real-time deadlines.

Fast global summary of processing results. A real-time processor cannot be slow to make decisions. For a parallel processor, a centrally-controlled test-and-branch requires the local states of the processors to be condensed to a single value. Thus, the rapid summary of results is essential.

One cycle per instruction. Multi-cycle instructions greatly complicate control in a SIMD array, and make real-time performance analysis more difficult.

Complete modularity, with no features that limit the size of the array. Changing the size of the array should not require changing the architecture.

> Don't push the limits of technology and architecture at the same time. Keep the design simple and regular, and use a stable technology. Avoid "tricks" (like fancy clocking schemes), and creeping elegance.

Our analysis of low-level vision algorithms showed that the majority would best be served by a mesh-connected array, augmented with the features of an associative processor (i.e. global broadcast with local partial matches, activity control with global override, and dedicated response hardware) [Weems, 1984]. Our approach was to design a "straw man" architecture that provided those features, and was buildable with the technology available in the early 1980's.

One of the tradeoffs to consider was whether to build a bit-serial or bit-parallel associative processor. In a bit-parallel system, every memory bit contains a comparator, and all of the bits of broadcast values are compared at once. In a bit-serial scheme, there is only one comparator per memory word, and each bit of a value must be broadcast and compared sequentially. Thus, a bit-parallel associative memory can perform a comparison N times faster, where N is the number of bits in a word. However, the factor of N only applies to equality comparisons. Inequalities are tested by transmitting one bit at a time, through a "comparand mask," because the bits are not independent and so cannot be compared simultaneously with simple bit comparators.

In our earlier work, we found that equality tests account for 49% of all comparisons, and 25% of all operations in a variety of associative processing applications. Since 8-bit comparisons are the most frequent in image processing, a bit parallel system would provide a factor of 8 x 0.25 = 2 speed increase at a cost of roughly double the hardware. This balance is shifted toward the bit-serial approach by other factors: The availability of real-estate that would otherwise go to comparators allows the inclusion of a full ALU, which speeds-up arithmetic, and either double the memory per processor or double the number of processors. We therefore chose a bit-serial approach at the low level.

A software simulator was constructed that allowed us to program numerous low-level vision algorithms on the architecture, and to analyze it for utilization and bottlenecks. Part of the analysis involved the examination of both static and dynamic instruction counts, and the number of algorithms that used each

instruction [Weems, 1984]. Comparing static and dynamic counts helps to identify those instructions and architectural features that are most frequently used in loops (loads, stores, arithmetic, mesh communication) and those that typically appear outside of loops or in loop control (global summary, carry initialization, activity control). The number of algorithms that use each instruction helps identify anomalies in the usage.

We also performed a feasibility study by laying out a 16 processor custom VLSI chip (using 3-micron NMOS, the predominant technology at the start of 1982). Each of the processors contained a bit-serial ALU, five registers, 32 bits of static memory, and communication circuitry. In addition to the processors, the chips contained instruction and address decoders and global summary circuitry. The chips were very conservatively designed, using only 28 pins, a 173 mil square die, 6700 transistors, and dissipating 0.35 watts.

This preliminary design study pointed out several key areas requiring improvement. Specifically, neighbor communication was too slow, the memory was too small, I/O required too much overhead, the separation of activity control from response added significant overhead in some frequently used loops, using the registers for communication and arithmetic created a bottleneck, allowing arithmetic operations only on the registers was another bottleneck, and response count (one of the global summary operations) was too slow. The remaining discussion addresses the tradeoffs examined in dealing with each of these points.

4.1 Neighbor Communication

At the time of the preliminary design, 64 pin DIP devices were available but uncommon, and 40 pin devices were the standard "large" package. Since our goal was to put 64 processors on a chip, 32 pins would be required for full mesh connection between chips, leaving too few pins for other functions. Mesh I/O was thus serialized between chips (i.e. streamed over a single pin in each direction -- N, S, E, and W). However, most of the low-level vision algorithms used the mesh extensively and were thus significantly slower than optimal.

To improve communication speed, one option was to multiplex some of the pins for communication during additional clock phases. But this violated our guideline of minimizing the cycle time and distributing no more than a two-phase clock.

Another option was to build the 64-processor chips with four blocks of 16 processors, each with a serial connection to its neighboring block. Such a scheme doubles the speed of communication, but remains a factor of four slower than optimal.

In the end, technology came to our rescue. When pin grid arrays became available, the pin limitation disappeared, and full mesh communication across chip boundaries became possible. We also arranged to read from two neighbors at once, and thus a processor can collect data from four neighbors in only two cycles.

Another tradeoff that we analyzed was a four-way versus an eight-way mesh. We found that only a few algorithms take advantage of an eight-way mesh, and the increase in performance is quite small unless extra hardware is added to allow operations on eight inputs at once. Even then, the improvement does not justify the cost of tripling the number of I/O pins on the processor chip (from 32 to 96) and at circuit board boundaries where, assuming an 8 x 8 array of processor chips on a board, 768 I/O lines are required.

In addition to local communication, several low-level vision algorithms require communication between processors that are spatially distant in the mesh. One option is to augment the mesh with an additional network, such as a pyramid or a hypercube, but we found that this added too much complexity, required too many I/O connections, and in some cases required hardware that placed limits on the array size. Some examples of other architectures that have taken this approach can be found in [Uhr, 1972], [Hillis, 1986], [Ibrahim, 1984].

An alternate approach is to enhance the mesh itself, so that no additional connectivity is required between processors. An electrically switched mesh can allow signals to be sent between distant processors. If one considers a row or column of processors as a bus, a processor could open a switch in one direction, breaking the bus, and then transmit a message in the opposite direction that would propagate at electrical speed as far as the next break. Processors that have not broken the bus are listeners. This scheme is similar to those that were later proposed by [Kumar, 1985], [Miller, 1987], and [Li, 1987], and is a generalization of the propagate operation in the CLIP-4 [Duff. 1978], and the "flash-through" mode of the ILLIAC III [McCormick, 1963]. Because all of the processors in a bus

segment can listen to the message being transmitted, there is also some similarity to a broadcast protocol multiprocessor [Levitan, 1984].

In an implementation with chained pass transistors and buffers, the propagation speed can be quite slow. A practical implementation requires the use of precharging and inverted logic to eliminate the buffers. However, once the buffers have been removed from the circuitry, there is no longer any need to restrict a bus to a single talker. If multiple processors output a bit onto a bus segment, the wired-OR of those signals can be read back. To those familiar with associative processors [Foster, 1976], it is immediately obvious that such an operation is equivalent to a some/none test for responders, except that it is performed independently within each segment.

There is also no need to restrict the busses to rows and columns. They can be any contiguous group of processors in the mesh. In a vision algorithm, for example, each region in an image could be electrically isolated from its neighbors, allowing local broadcast and some/none operations to occur simultaneously in all regions (see Figure 2). An important result of these capabilities is that maximum or minimum values can be determined within regions, which can then be used to label connected components (a frequently used operation in vision). Our simulations show that connected component labelling can be performed in this network in roughly 50 microseconds on a 512 by 512 array, assuming a 100 nanosecond cycle. The Coterie Network has many other uses, including matrix arithmetic, FFT, convex hull computation, simulating a pyramid processor, etc.

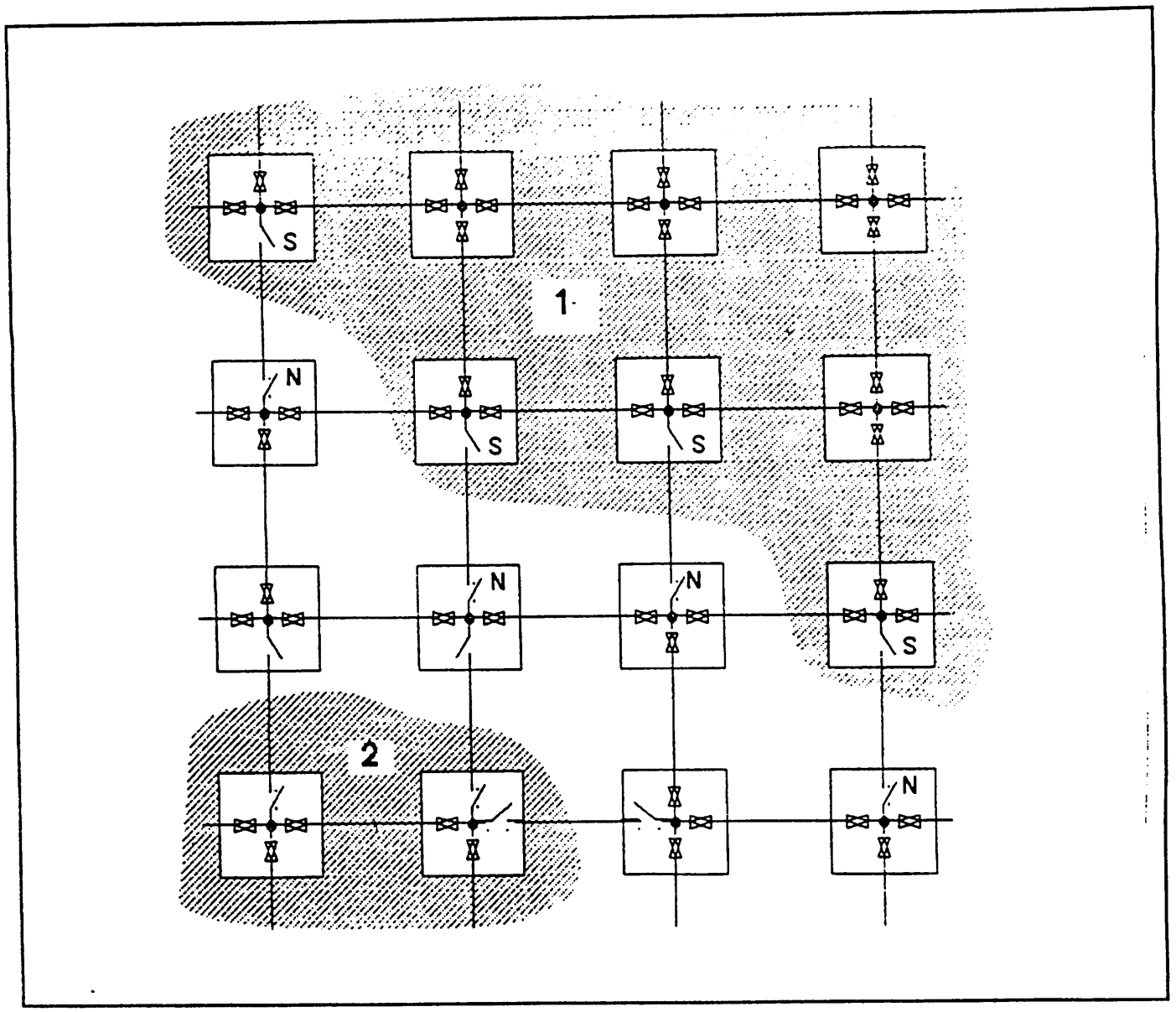


Figure 2. Isolated Processor Groups Corresponding to Regions in an Image

This new network has properties that are significantly different from the usual mesh. Because the processors in a group share common properties and purposes, we call the group a Coterie, and the network is known as the Coterie Network. In our current design, the actual implementation permits signals to cross orthogonally or bypass a node diagonally (see Figure 3). Thus, Coteries need not be physically adjacent to be linked.

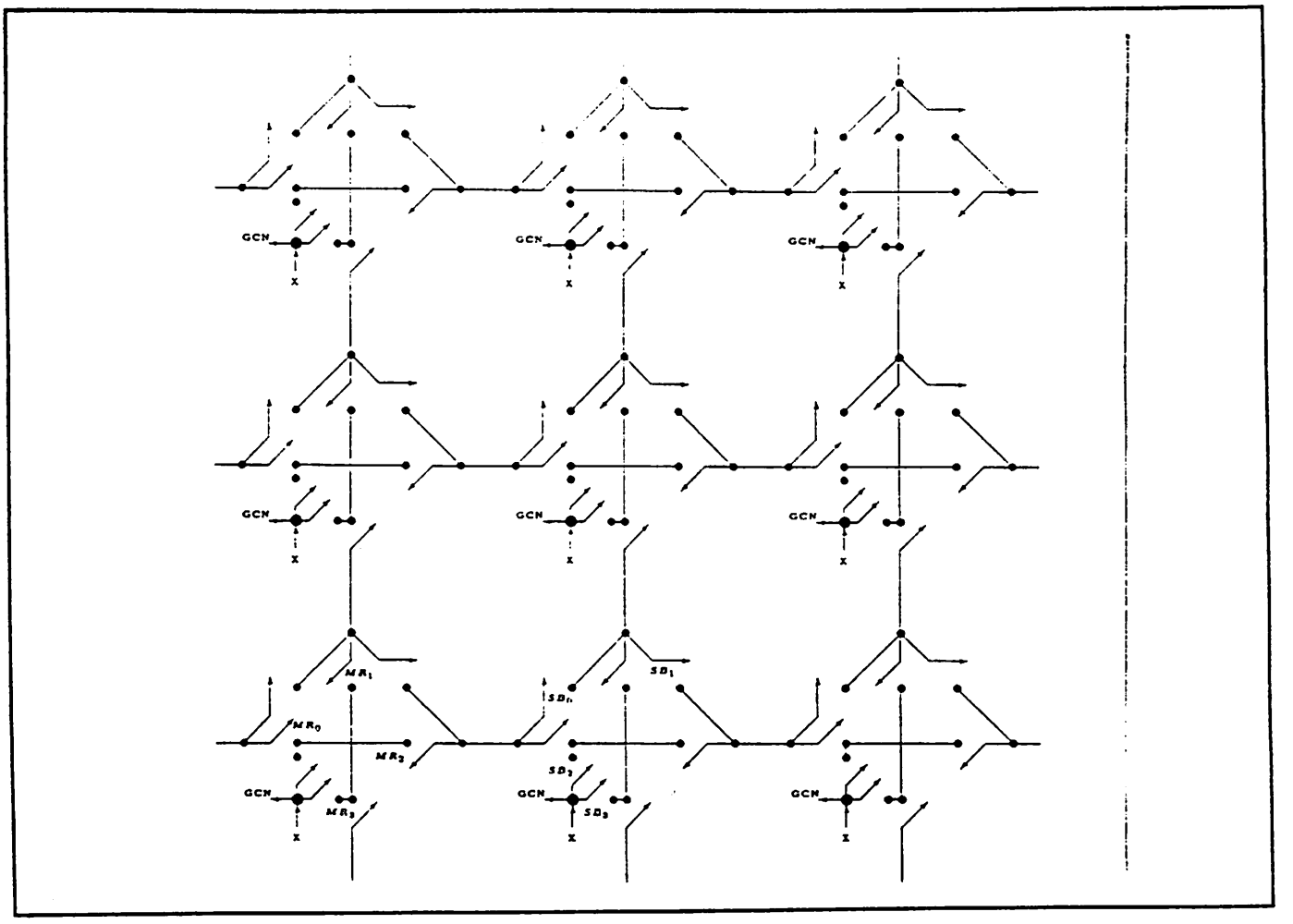


Figure 3. A Three by Three Coterie Network, Showing Crossing and Bypass Links

The Coterie Network results in a new mode of parallelism that falls somewhere between SIMD and MIMD. While there is still a single instruction stream, broadcasting of data values and collection of summary information are no longer restricted to a central entity; a capability that has previously been restricted to MIMD architectures. It may be noted that the Connection Machine also has such capabilities, but the Connection Machine is not a purely SIMD system. Each Connection Machine router node is a finite state machine that operates semi-autonomously. Its router can thus be considered a mono-algorithmic MIMD processor that is attached to the SIMD processor array. The Coterie Network, on the other hand, contains no active logic and operates only in response to broadcast instructions.

4.2 Memory

Although we squeezed our suite of analysis algorithms into 32 bits per processor (sometimes by grouping processors together and pooling their memory), it was clearly far too little storage. The two obvious options for expanding processor memory are to add memory to the processor chip, or to use external memory. Each of these approaches has its problems:

1. There isn't space for a large memory on the processor chip.

2. Too many pins are required for all of the processors to access external memory in parallel.

3. Accessing external memory limits the processor clock rate.

Some array processors (MPP [Batcher, 1980], DAP [Hunt, 1981], Connection Machine) have opted for external memory because they can use standard parts, and because they place only a few processors on a chip. Others (CLIP-4, GAPP [Davis, 1984], ASP [Lea, 1988]) have placed memory on the chip. We anticipate chips with more than 64 (1024 is feasible) processors, and increased clock rate, which favors on-chip memory. However, we also saw the need for at least tens of thousands of bits per processor, which only off-chip memory could support. Our solution to this dilemma is to do both.

Each of our processors contains an explicitly managed data cache on the chip. In our current implementation, this cache contains 320 bits, but the architecture provides for expansion to 1024 bits. The cache is divided into pages of 128 bits, with the lowest two pages being directly transferrable to an external backing store in blocks of 8 bits. The higher pages are used like a register file, to store the most frequently used data, and their values must be moved to the lower pages to be transferred to external memory.

The split functionality of the cache is due to the silicon area required to implement the swappable pages, which take up two thirds of the current real estate. Their large size results from the need to perform a corner-turning operation when accessing external memory. The processors access a column of 64 bits in the cache, one bit per processor. But the memory must be transferred to and from external memory in 64 one-byte chunks, where each byte is an eight-bit field from a processor cache.

Why arrange the data this way? Because the external memory is dual-ported with the intermediate-level processor, and constitutes the primary data path between the processing levels. (See Figure 4.) The low-level data must be corner-turned on its way to the backing store, so that the intermediate-level processor can work with it directly. Otherwise, the intermediate-level processor would spend too much time rearranging bits.

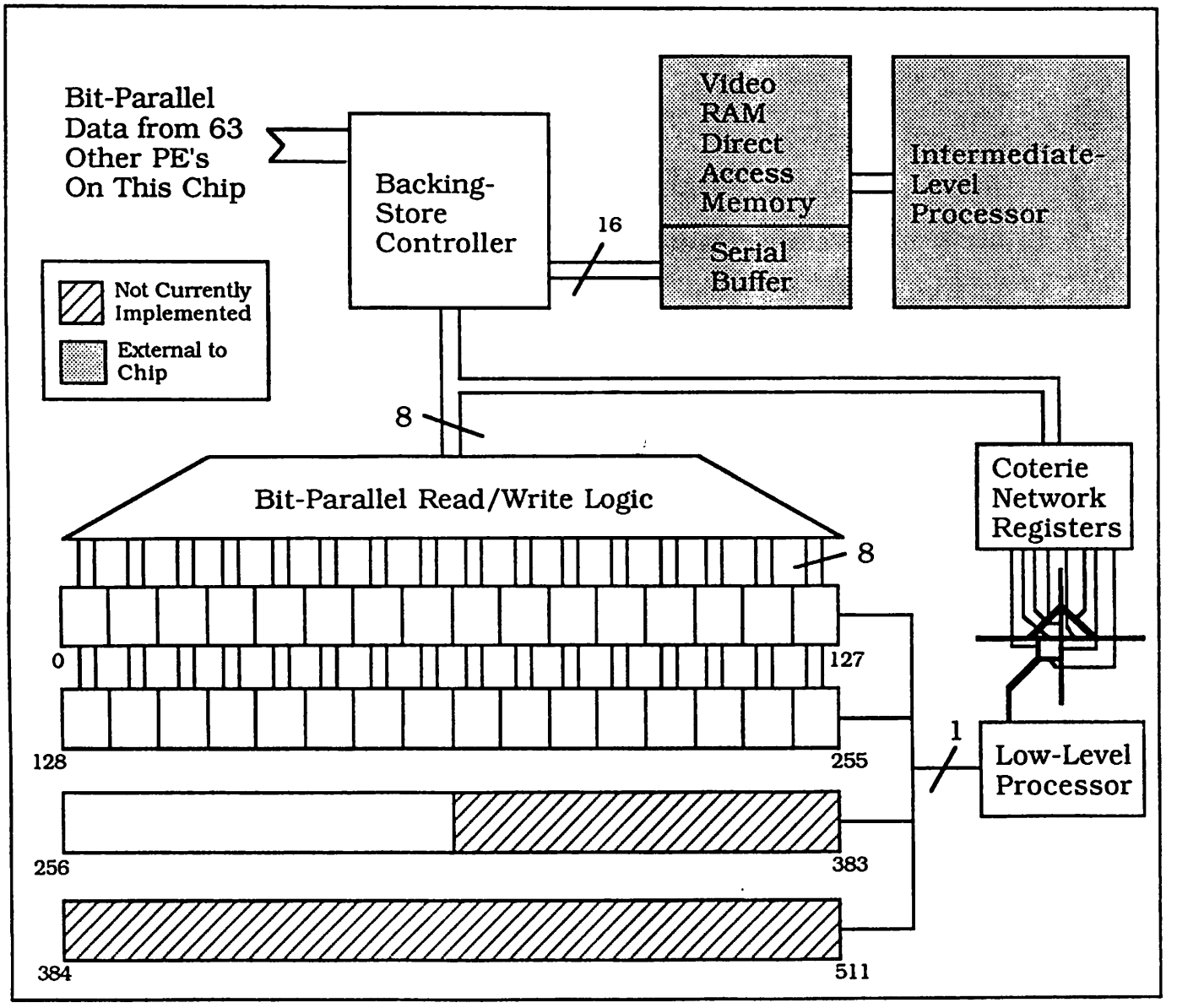


Figure 4. Memory Architecture

Once corner-turning is implemented, two other useful features drop into place. The swappable pages have eight-bit data paths, allowing us to move eight bits at

once (a factor of eight speed-up over the bit-serial data path, and useful for aligning mantissas in floating-point operations). The registers that control the Coterie Network switches can be attached to the 8-bit data path, allowing the entire network to be reconfigured with a single instruction.

Each processor has access to 32K bits of external memory. Backing store transfers take 16 instruction cycles per byte. Since it takes a minimum of 8 cycles for the processors to operate on a byte, it is possible to starve them 50% of the time in the worst case. However, typical operations take two or more cycles per bit and will not be able to outrun transfers. The transfers are done in the background through cycle-stealing, so processing can be overlapped.

The path to the backing store requires only 16 pins for data, and a few extra for control. This was accomplished by using video RAM chips (VRAM) for the backing store. The processor chip is connected to the serial port of the VRAM (16 bits wide), and streams the data to or from this port at double the instruction rate. Thus there is no need to generate addresses, except when transferring the serial buffer to the direct access memory. The memory map of the intermediate-level processor includes this direct access memory, thus providing the dual-porting required for interlevel communication.

With an 8-bit path to memory, it would seem that an 8-bit ALU should be used. There are three reasons why this wasn't done. First, there isn't enough space on the chip. Second, an 8-bit processor would outrun cache transfers, and require many more pins to provide sufficient bandwidth for accessing external memory. Third, a fixed operand length wastes a substantial amount of memory since operands of other lengths do not fit perfectly (1-, 6-, 9-, 10-, 12-, and 18-bit values are nearly as common 8-bit), which is a significant problem with such a small cache. However, we have left room in the instruction set to permit 8-bit operations when the technology is available to support them fully.

4.4 Input/Output

Our original I/O scheme was similar to that used in the MPP, with data shifted in from an array edge. Since the original design used serialized mesh communication across chip boundaries, 3.3 milliseconds would be required to fill

a 512 by 512 array with an eight-bit image, assuming a 100 nanosecond clock. Ten percent of a frame time is an unacceptable I/O penalty in a real-time application.

With parallel mesh communication, the time is reduced to 410 microseconds. Initially we felt that this would be fast enough, but later realized that even a pause of this length could interfere with real-time deadlines. Also, in a multisensory task, I/O is much more significant.

Another consideration is that vision systems occasionally fall behind and need to subsample the incoming stream of frames in order to catch up. It is also desirable to be able to retrieve previous frames. We were thus faced with redesigning the I/O after beginning to design the new chip, which greatly limited our options.

Our interim solution is to associate an additional VRAM with each processor chip, connected to the South edge of the chip's mesh network (Figure 5). A special instruction tri-states the North edge of the chip during I/O. Then the data in the serial buffer of the VRAM is shifted in from the South, and stored by the processors, using the corner-turning circuitry. To output to the VRAM, the Coterie Network is used to send data from the North edge of the chip to the South, where it is streamed into the VRAM's serial port. Transfers to and from this staging memory take only 8 microseconds for an array-sized 8-bit image. The memory is large enough to hold sixteen seconds of imagery, providing a reasonable time window.

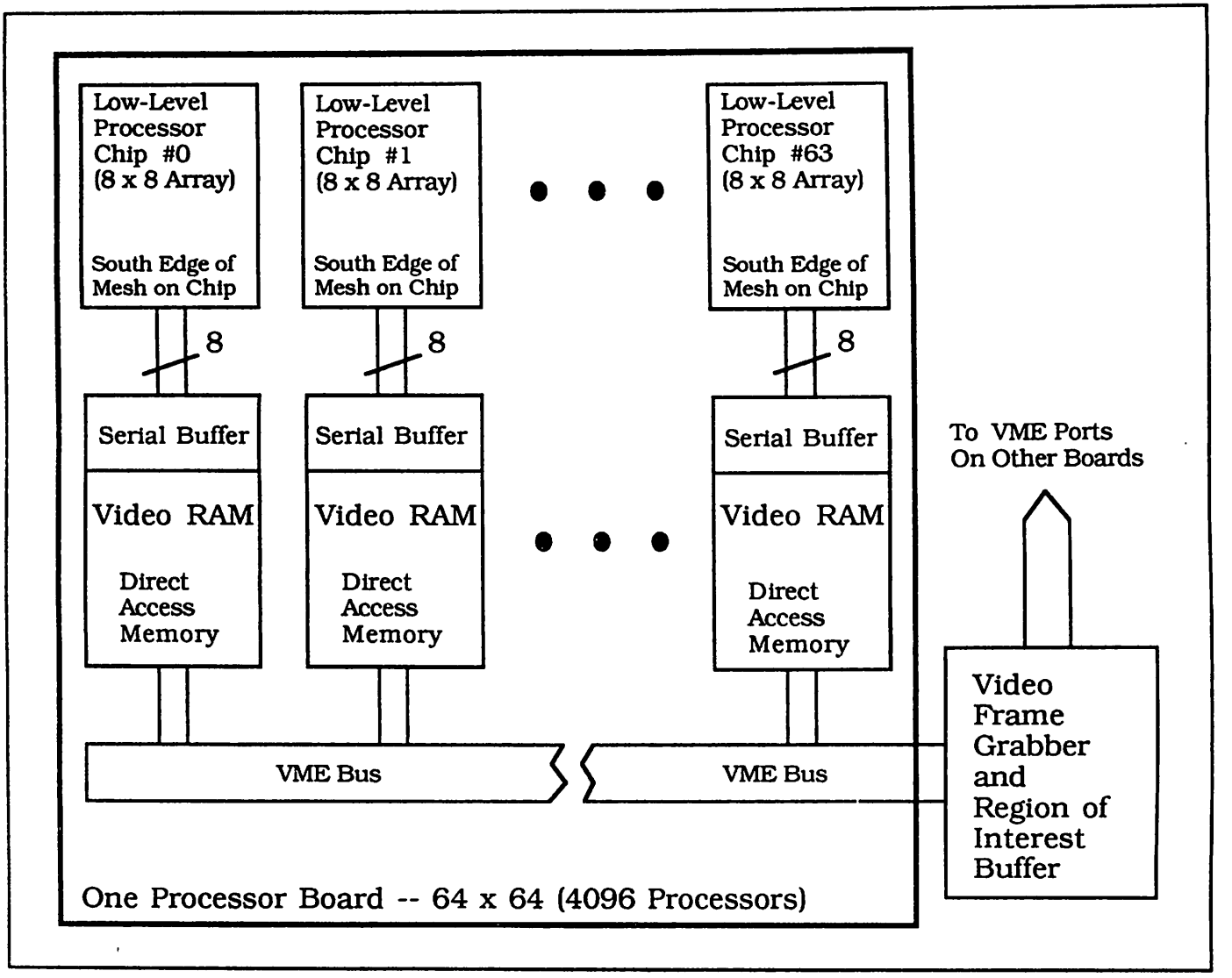


Figure 5. I/O Architecture.

The direct access portion of the VRAM is connected to a VME port, and appears as a block of memory in the VME address space. In the full-scale system, it is intended that each processor card have its own VME port so that parallel I/O to the staging memory can take place. In our prototype, the single VME port is connected to a smart frame grabber that moves data in and out through a region-of-interest buffer.

This last-minute modification is admittedly clumsy. The preferred option is to dual-port a segment of the backing store with the frame grabber instead of the

intermediate-level processor. This approach reduces interface complexity and simplifies the programmer model, and will be incorporated in the next chip design. It will then be possible to transfer an image plane between the staging memory and processors in 1.6 microseconds.

4.5 Response and Activity Control

Traditional associative processor designs use a response flag to both control local activity and provide summary information to the central control. In our earlier work we found this to be inefficient, since it was often necessary to report a result unrelated to activity. In our preliminary design, we took the opposite approach, separating response from activity by providing a register for each. However, our instruction usage analysis showed frequent copying of the response register into the activity register or vice versa. One third of the algorithms used combined response and activity.

Where we originally saw only two options, there turned out to be a third. Both split and combined activity and response can be provided by allowing writes into either of the two registers or both simultaneously. This small change provided a 20 percent speed improvement in equality comparison with a broadcast value. A similar change allows inequality tests to be performed in about 50% less time, by permitting the response register to be loaded with an operand at the same time that a result is stored in the activity register.

4.6 Eliminating Bottlenecks Through the Registers

In our preliminary design, the response register served two additional purposes that simplified the hardware. It was an arithmetic operand register, and the interface to the communication network. Thus we found that data in the response register was continuously being shuffled to and from memory.

One option was to add separate registers specifically for arithmetic and communication. However, adding registers significantly enlarges the processor layout because not only are the registers large themselves, but each register has its own output line that must run through the processor. (To fit 64 processors on a chip, their layout had to be as narrow as a word of memory. Thus, there is no room to add lengthwise lines.)

A second approach is to bypass the registers. Originally, we used a purely register-to-register processor design, because our memory designs were too slow for a data-fetch and execute to occur without lengthening the instruction cycle. The advent of a second metal layer in VLSI processes enabled us to increase the speed of the memory to the point that a read-modify-write operation can be performed in a 100 nanoseconds. Thus we redesigned the processor to use memory as an operand, avoiding the need to use the response register as either an input to the ALU or a result buffer.

Similarly, communication with neighboring processors can take place through memory-to-memory transfers. Moving a bit to another processor in one of four directions is just another of the "modify" options in the read-modify-write cycle.

Having removed the overloaded functions from the response register, we found that it was actually underutilized. Therefore, we added the function of supplying data to the Coterie Network, and reading the network's output. This is a natural extension to its global summary function, since the Coterie Network is often used to form a local summary of members in a Coterie. Local summary, long distance communication, and global summary are generally exclusive of each other, so they rarely conflict in their use of the response register. Figure 6 shows the final design of the low-level processing element, while Table 2 presents its instruction set.

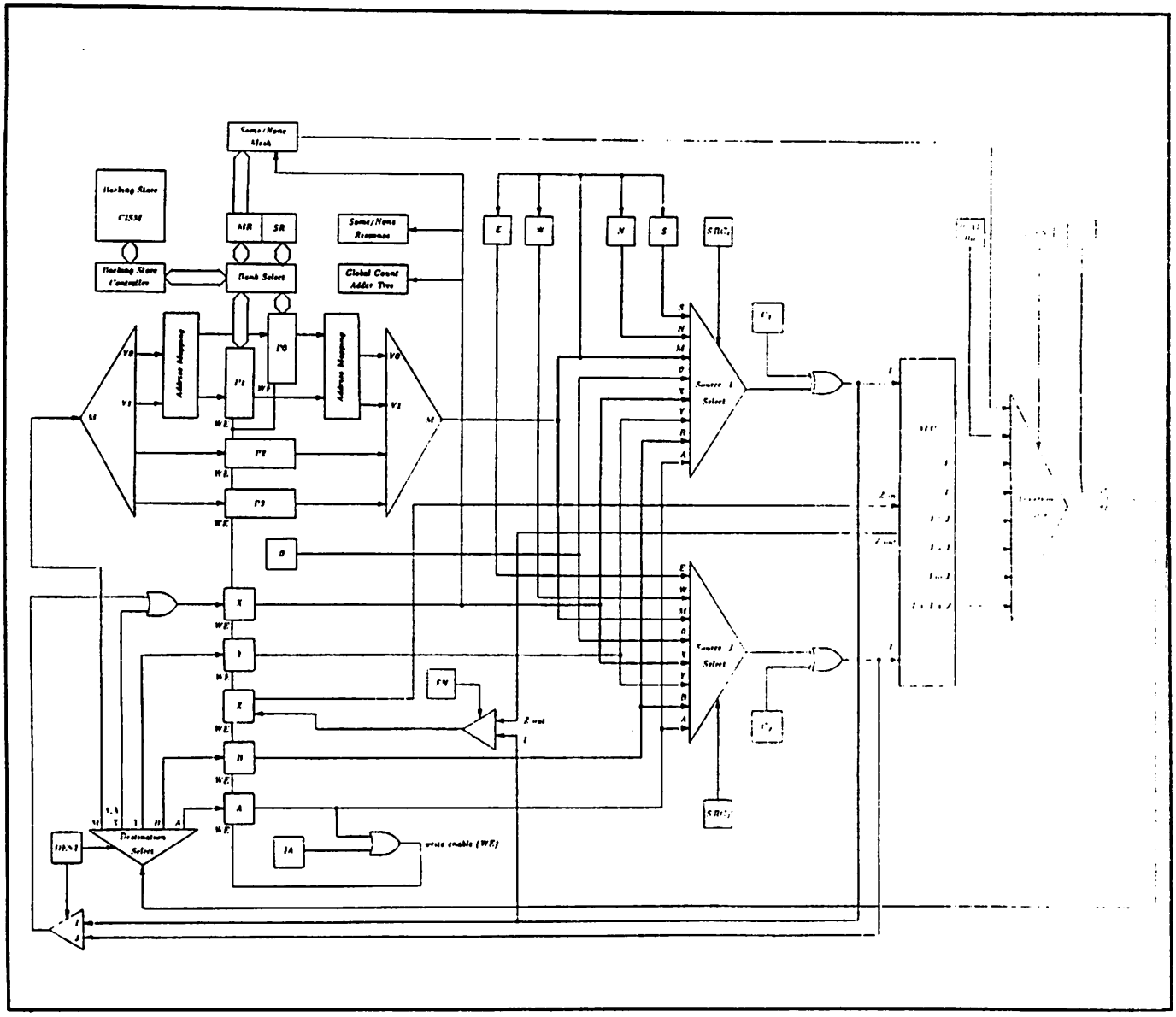
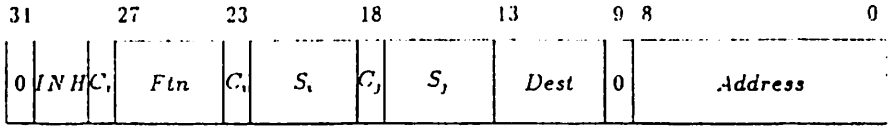


Figure 6. Architecture of the Low-Level Processing Element



INH	(Inhibit)
0	Non-inhibit — always active
1	Inhibit if A = 0
2	Inhibit if A = 0 or S/N = Some
3	Inhibit if A = 0 or S/N = None

$$I \Leftarrow C_i \oplus S_i$$
$$J \Leftarrow C_j \oplus S_j$$

	S_i	S_j	Ftn	Dest	
0	ZERO	ZERO	Coterie $\Rightarrow R$	X_{pe}	0
1	C	C	$I \Rightarrow R$	A, X	1
2	S	E	$J \Rightarrow R$	$A, X \Leftarrow I$	2
3	N	W	$\overline{I \wedge J} \Rightarrow R$	$A, X \Leftarrow J$	3
4	Y	Y	$\overline{I \vee J} \Rightarrow R$	Y	4
5	X	X	$\overline{I \oplus J} \Rightarrow R$	X	5
6	B	B	$\overline{I + J + Z} \Rightarrow R$	B	6
7	A	A	$ICAP\ C \Rightarrow R$	A	7
8	memory	memory	$I \Rightarrow Z$	memory	8
9	—	—	$memory \Rightarrow MR$	—	9
10	—	—	$memory \Rightarrow MR, SB$	—	10
11	—	—	$MR \Rightarrow memory$	—	11
12	—	—	$MR, SB \Rightarrow memory$	—	12
13	—	—	—	—	13
14	—	—	—	—	14
15	—	—	—	—	15

$$Dest \Leftarrow C_r \oplus R$$

Table 2. Low-Level Processor Instruction Set

4.7 Faster Response Count

The preliminary design formed a count of responding processors by circulating the response registers on a chip past a counter. The chip-level counts were then summed by columns, bit serially, with a pipeline of adders. A similar structure at the array edge was used to sum the column totals. This was very cheap to build, requiring only a counter and a full adder on each chip. However, it required 26.8 microseconds to develop a count for a 512 by 512 array.

Our analysis showed that counting is frequently done in bursts. For example, summing a set of values in the array involves counting the ones in each bit position. Thus, to sum an array of 16-bit values, 16 count operations must be performed. Another example, is creating a histogram of a set of data; which could require 256 counts for an 8-bit field. Thus, one histogram could take nearly 7 milliseconds, or 21 percent of a frame time.

For real-time applications, a count must be developed at least an order of magnitude faster. One technique proposed by Favor [Favor, 1964] and improved by Foster [Foster, 1971], uses a pyramid of adders. However, the fan-in from 262,144 processors to a single sum is too great to be practically realized this way.

Pin constraints on the processor chip, and space and pin limitations for the external circuitry required that the chip-level counts be output serially. Within the chip, Foster's scheme is used to produce a response count at the end of each instruction cycle. A special instruction latches the count into a shift register. Although there are 64 processors on the chip, the register is eight bits wide (one more than the seven bits needed), so that the count circuitry can be shared with the intermediate level processor (provision is made for summing an 8-bit value output by the intermediate level -- see Figure 7).

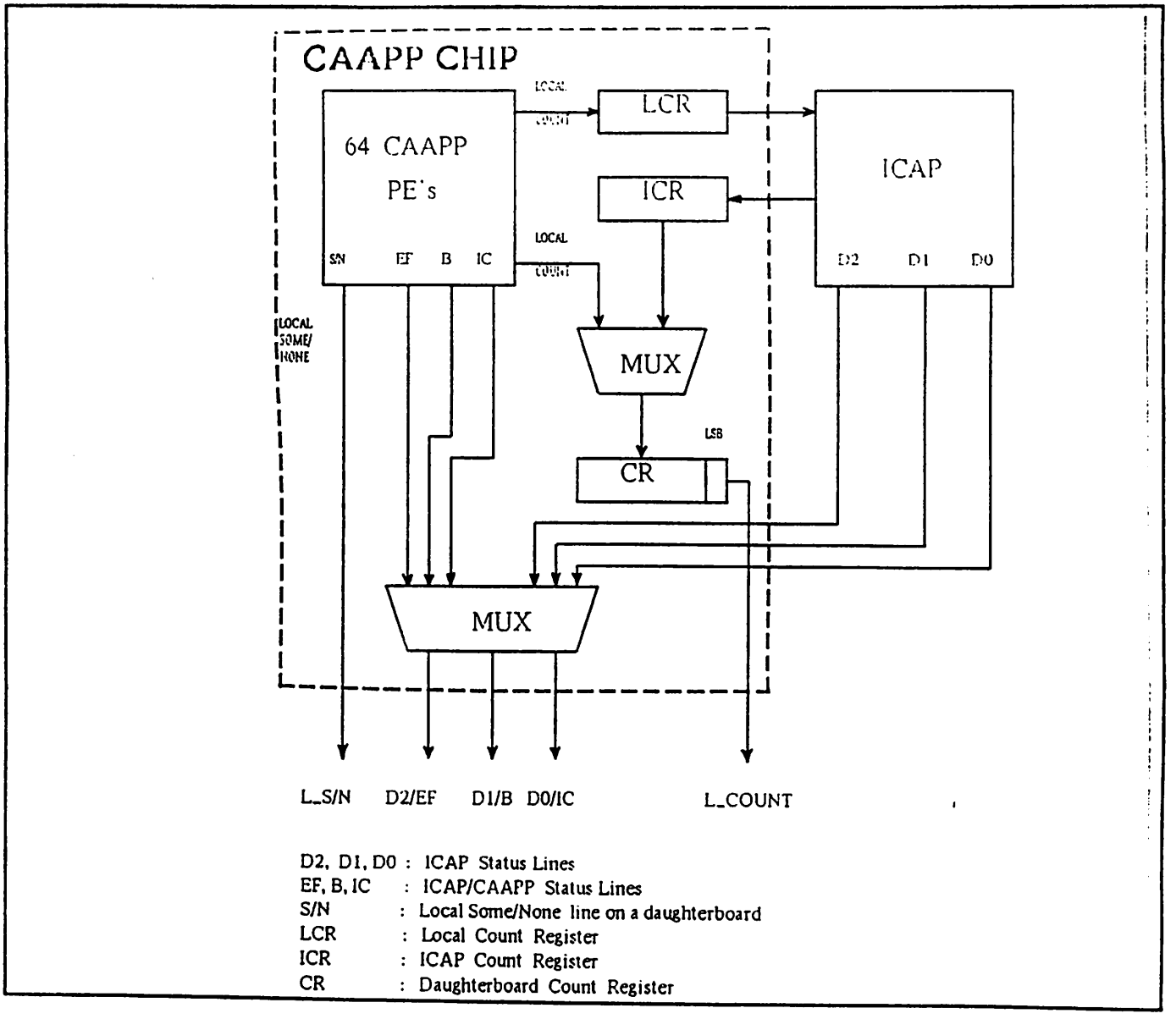


Figure 7. On-chip Response-Count Registers.

One option is to use an instruction to shift out each bit of the count. However, this idles the processors during output of the count. Our analysis showed that typically some short operation is performed between the counts in a burst. This may be as simple as loading another bit into the response register, but is likely to be a multibit comparison. Thus, we decided to use a finite state machine that automatically shifts the count out of the chip, least significant bit first, one bit per cycle. Output begins as soon as a count is latched. The processors are then able to overlap computation with the development of the current count.

Once the chip-level counts are output, there are many ways to sum them up. We initially considered a scheme with an adder pyramid on each circuit board, and a polling mechanism to form the final count: a finite state machine would read each of the 64 board-level counts and sum them in an accumulator. This reduces the counting time to 7.4 microseconds. To further increase the speed, the pyramid could be extended to sum columns of boards, and then just the eight columns would be polled. The time then drops to 1.8 microseconds, or about the same as a 16-bit comparison operation. Unfortunately, the hardware for this scheme is complex and costly.

Our final solution was to serialize the entire adding process. A custom VLSI chip was designed with 64 serial inputs, one serial output, and six parallel outputs (Figure 8). One cycle after the low order bits of a set of partial counts are input, the low order bit of the result appears at the serial output. The high order bits of the result appear in the parallel outputs.

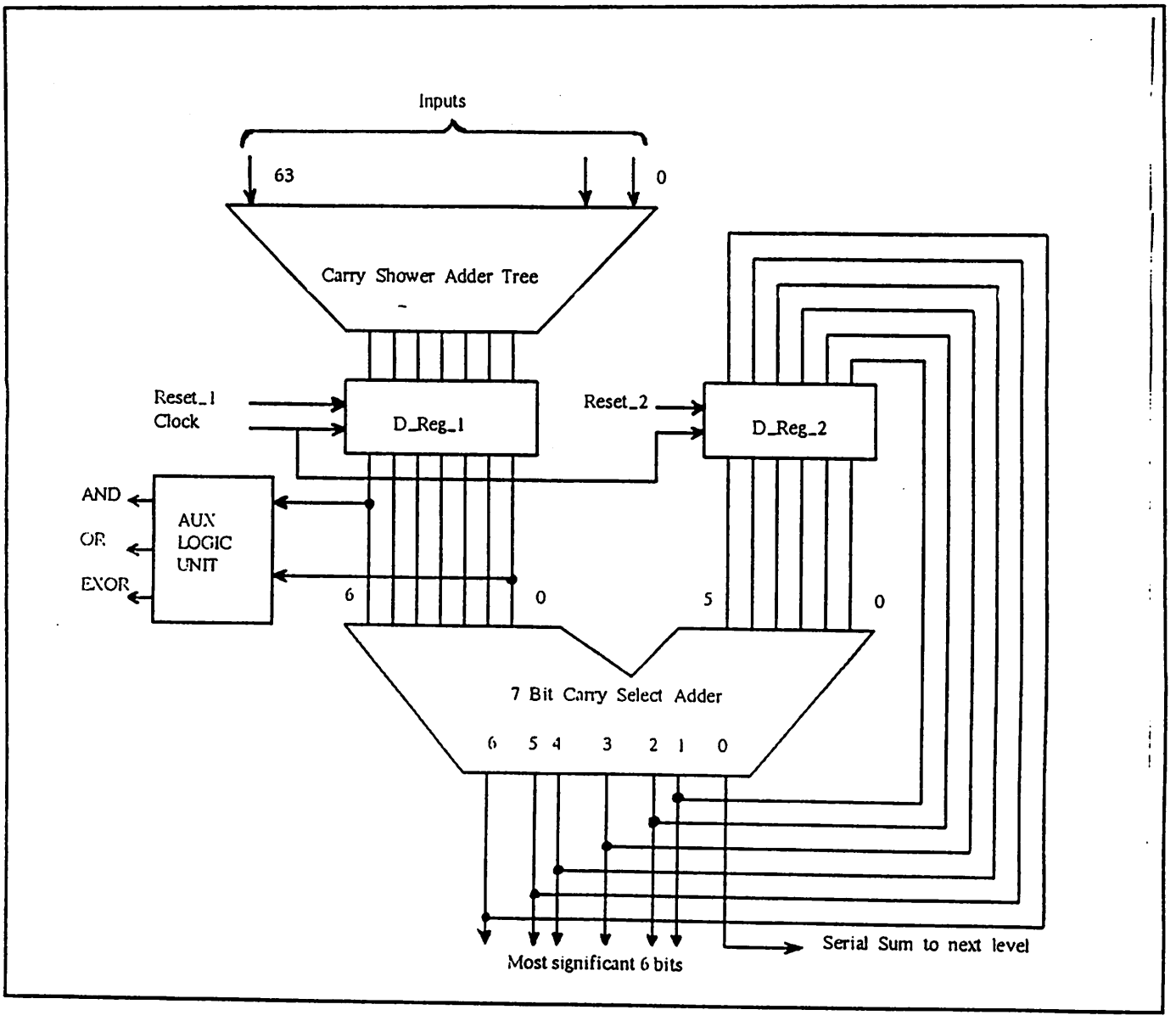


Figure 8. Functional Diagram of Feedback Concentrator Chip.

If another set of bits are input, the high-order portion is recirculated and added to the next result through a carry-select adder [Hwang, 1979]. Thus, on the third cycle, another result bit appears at the serial output and the high-order portion is again present at the parallel output. The process can be repeated to sum 64 inputs of any bit length with the low order portion of the result being shifted out serially and the high order six bits being available in parallel one cycle after the last set of bits has been input. To flush the entire result out serially, zeroes are input for six cycles after the last set of bits.

Serial flushing of the high order portion allows the chips to be cascaded. Thus, only two levels of the chips are required to form a count for the entire array. The first bit of a count reaches the central controller at the end of the third cycle, and 13 cycles later the last of the low order bits is output (we always assume an 8-bit value is being output by the processor chips) and the high order portion is available in parallel. Thus, only 1.6 microseconds are required to count the response registers in an array of 262,144 processors, using 65 copies of a single custom chip.

The same chip, which is called the Feedback Concentrator, also provides the boolean summaries some/none, some/all, and exactly one responder. It can be thought of as a general purpose 64-input reduction unit.

5. Current Status

The low-level chip, with 64 processors, has been fabricated via the DARPA MOSIS facility and is fully functional. It contains roughly 130,000 transistors and is built in 2-micron CMOS. The IUA circuit boards are now being fabricated and assembled to form a prototype slice with 4096 low-level processors, 64 intermediate-level processors, and a single high-level processor. A new version of the processor chip is nearing completion. It will be fabricated in 1.25 micron technology, and contain 256 processors.

6. Conclusions

Real-time knowledge-based sensory processing presents a unique challenge to the computer architect. Providing the necessary types and amount of processing power necessitates the design of special-purpose architectures that combine massive parallelism in different forms with an emphasis on real-time performance and maximal efficiency. Success depends on an in-depth understanding of vision systems, real-time processing, architecture, and technology.

The low-level processor of the IUA is the result of extensive analysis and tuning to support sensory preprocessing. The primary features that distinguish it from other data-parallel arrays are a combination of flexible and extremely fast I/O, fast global summary of processing results, local broadcast and summary through the

Coterie Network, on-chip data cache with off-chip backing store, and a massively parallel interface to an intermediate-level processor.

7. Bibliography

[Batcher, 1980] Batcher, K. E., Design of a Massively Parallel Processor, IEEE Trans. Comp., Vol. C-29, No. 9, September 1980.

[Davis, 1984] Davis, R., Thomas, D., Geometric Arithmetic Parallel Processor-Systolic Array Chip Meets the Demands of Heavy-Duty Processing, Electronic Design, October 31, 1984, pp. 207-218.

[Draper, 1989] Draper, B.A., Collins, R.T., Brolio, J., Griffith, J., Hanson, A.H., Riseman, E.M., The Schema System, International Journal of Computer Vision, Vol. 2, No. 3, January, 1989.

[Duff, 1978] Duff, M.J.B., Review of the CLIP Image Proceeding System, Proceedings of the National Computer Conference, 1978, AFIPS, pp. 1055-1060.

[Favor, 1964] Favor, J., A Method for Obtaining the Exact Count of Responses Using Full and Half Adders, AP-111770, Goodyear Aerospace Corp., Akron, Ohio, 1964

[Foster, 1971] Foster, C. C., Stockton, F. D., Counting Responders in an Associative Memory, IEEE Transactions on Computers, Vol. C-31, No. 12, Dec. 1971, pp. 1580-1583.

[Foster, 1976] Foster, C. C., Content Addressable Parallel Processors, New York: Van Nostrand Reinhold, 1976.

[Hanson, 1986] Hanson, A. R., Riseman, E. M., A Methodology for the Development of General Knowledge-Based Vision Systems,. In: Vision, Brain, and Cooperative Computation, M. Arbib and A. Hanson (Eds.), MIT Press, Cambridge, 1986.

[Hillis, 1986] Hillis, D.W., The Connection Machine, MIT Press, Cambridge, 1986.

[Hunt, 1981] Hunt, D.J., The ICL DAP and its Application to Image Processing, in Languages and Architectures for Image Processors (M.J.B. Duff, S. Levialdi eds.), Academic Press, London, 1981.

[Hwang, 1979] Hwang, K., Computer Arithmetic, John Wiley and Sons, New York, 1979.

[Ibrahim, 1984] Ibrahim, H.A.H., Image Understanding Algorithms on Fine-Grained Tree-Structured SIMD Machines, Ph.D. Dissertation, Department of Computer Science, Columbia University, N.Y. 1984.

[Kumar, 1985] Kumar, V.K.P., Raghavendra, C.S., Array Processor with Multiple Broadcasting, Proc. 12th Annual Symp. Computer Architecture, Association for Computing Machinery Press, 1985.

[Lea, 1988] Lea, R.M., ASP: A Cost-effective Parallel Microcomputer, IEEE Micro, October, 1988, pp. 10-29

[Levitan, 1984] Levitan, S. P., Parallel Algorithms and Architectures: A Programmers Perspective, Ph.D. Dissertation, Computer and Information Science Department, University of Massachusetts at Amherst, May 1984.

[Li, 1987] Li, H., and Maresca, Polymorphic Torus Network, Proc. Intl. Conf. Parallel Processing, Penn State Press, State College, PA, 1987.

[McCormick, 1963] McCormick, B.T., The Illinois Pattern Recognition Computer -- ILLIAC III, IEEE Trans. on Elect. Computers, Dec., 1963, pp. 791-813.

[Miller, 1987] Miller, R., Kumar, V.K.P., Reisis, D., Stout, Q.F., USC Tech. Rept. IRIS #229, Univ. of Southern California, Los Angeles, CA, 1987.

[Uhr, 1972] Uhr, L., Layered Recognition Cone Networks that Preprocess, Classify and Describe, IEEE Trans. Comp., 21:758-768, 1972.

[Weems, 1984] Weems, C. C., Image Processing on a Content Addressable Array Parallel Processor}, Ph.D. Dissertation Computer and Information Science Department, University of Massachusetts at Amherst, September 1984.

[Weems, 1989] Weems, Charles C., Steven P. Levitan, Allen R. Hanson, Edward M. Riseman, David B. Shu, and J. Gregory Nash, The Image Understanding Architecture, International Journal of Computer Vision, Vol. 2, Kluwer Academic Publishing, Boston, MA, 1989, pp. 251-282.