A PYRAMID IMPLEMENTATION USING A RECONFIGURABLE ARRAY OF PROCESSORS

by

Peter A. Sandon

Computer Sciences Technical Report #601

June 1985

A Pyramid Implementation Using a Reconfigurable Array of Processors

Peter A. Sandon

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

Abstract

A pyramid processor is a natural extension of an array processor, useful for various image understanding algorithms. This paper describes an implementation for a pyramid processor which uses a single array of reconfigurable processing elements. By allowing this array to be configured to represent different layers of the pyramid structure, more processing power can be applied to the upper layers than in conventional designs. Such a design can increase the performance of the pyramid processor for algorithms that cannot take advantage of the parallelism of layers. Modifications to the basic design provide tradeoffs in performance and complexity. Layer parallelism can be recovered, for example, by using multiple reconfigurable arrays.

1. Introduction

The two-dimensional topology of the retina and visual cortex suggests a two-dimensional array as a useful data structure for acquisition and subsequent processing of visual data. To provide fast computation on such data structures, array architectures have been proposed and built [Batcher 1980, Duff 1976, Flanders, et. al. 1977]. Another data structure, suggested by the hierarchical structure of biological visual systems [Hubel & Wiesel 1962, Van Essen & Maunsell 1983], is the pyramid. This structure embeds different sized arrays into a hierarchy that has been found useful for various computations involved in the vision process [Hong, et. al. 1981, Levine 1980, Tanimoto 1978, Uhr 1978]. These data structures can be efficiently supported by pyramid architectures, which are an extension of array architectures. Although a pyramid processor has yet to be built, there are several proposals for architectures and implementations [Ahuja & Swamy 1982, Dyer 1982, Schaefer 1985].

This paper discusses a particular weakness of the standard approach to designing pyramid machines. This weakness results from extending array processors in the natural way, that is, by adding successively smaller arrays to the base array, using the same size processing element at every level. The result of such an approach is that processing power in the upper layers is insufficient to support algorithms whose processing requirements do not decrease exponentially with increases in the hierarchy level. We present a processor design for a single layer of the pyramid, which has the flexibility to represent various layers of the data structure. The nodes at successively higher levels of the pyramid are represented by successively more powerful processors. This reduces the bottleneck that otherwise occurs, and efficiently utilizes all resources at every level.

2. Pyramid Architectures

A pyramid data structure consists of a stack of two-dimensional square arrays. The arrays are arranged from bottom to top in order of decreasing size. In a typical pyramid, the linear dimension of a given array, or layer, is half that of the layer below it. Thus, a nine layer pyramid might have 64K nodes in layer 8, arranged in a 256x256 array. Above that, in layer 7 will be a 128x128 node

array, in layer 6 will be 64x64 nodes, and so forth. At the top of the pyramid, layer 0 will have a single node.

Unlike array data structures, which provide random access to their elements, array processors limit access to individual array nodes. Instead, data paths for local transmission of data between neighboring nodes is provided. Similarly, pyramid processors provide connections among neighboring nodes. Horizontal adjacencies, among nodes within a given layer, are often referred to as sibling, or neighbor, relationships. A given node may be connected to its four closest, or eight closest neighbors. Vertical adjacencies, among nodes in different layers, are referred to as parent and child relationships. A given node may be connected to one or more parents in the layer directly above, and to a number of children in the layer directly below.

A typical design for an array processor, or for a single layer of a pyramid processor, is to put a processing element (PE) at each node of the array, with a single control processor broadcasting instructions to the PEs. Each PE is generally a simple, often a single-bit, processor which has a handful of single-bit registers and an ALU. Appendix A contains a brief description of such a PE. Figure A1 is a block diagram of a PE which could support an instruction set like that given in Table I. Since there is a single controller for the entire array of PEs, every PE executes the same instruction stream as every other PE, resulting in single-instruction, multiple data (SIMD) execution [Flynn 1972]. On the other hand, for a pyramid processor, each layer has its own controller, so the layers can execute independently of one another. This allows for the possibility of having all layers executing simultaneously, providing a pipeline from the image data at the bottom of the pyramid, to the high level recognition processing in the upper layers. Although this concurrency among layers is desirable, it is often not realized, since many vision programs require both bottom-up and top-down data flow. In the absence of a pipeline, it may be difficult to utilize more than a single layer of the pyramid at a time.

The direct extension of array processors, in which each layer of the pyramid is an array of single-bit PEs, may be adequate for certain pyramid algorithms. For instance, the use of the

pyramid data structure for segmenting an image [Levine 1980] involves simple convergent operations (average and logical OR) at each level in a bottom to top pass, followed by divergent (broadcast) operations and splitting and merging in a top to bottom pass. At any given level of the pyramid, the amount of data to be processed per PE is constant. On the other hand, Levine's algorithm does not lend itself to pipelining of layers, so at any given time during execution of the algorithm, all layers but one would be idle.

A different use of the pyramid structure is seen in the recognition cone algorithm [Uhr 1978]. Here, simple features in the image layer are detected and combined together into more complex features, which are then represented in the next higher layer. Successively higher layers of the pyramid represent successively more complex features of the original image. Not only are there more features being represented, but complex features may require more computation to process. In this case, the computational load per PE is higher in the upper layers than in the lower layers. This forms a bottleneck in the upper pyramid, where simple processors are inadequate to efficiently process complex data. An obvious solution to this problem is to use more powerful processors in higher layers of the pyramid. This solution is the major motivation for the pyramid processor design described below.

3. A 'Flat' Pyramid

Rather than restrict the representation of each node in the pyramid to a single-bit PE, we have suggested more powerful processors at higher layers of the pyramid. One approach to such a design, would be to use the same size array at all levels, but to connect the single-bit PEs into multiple (4,16,64...) bit PEs to represent nodes at higher levels. Now, if we are willing to restrict execution to one layer at a time, we can imagine compressing all the layers into a single plane of PEs, which can be configured to represent any given layer of the pyramid. This is the design we now describe.

Given an NxN array of single-bit PEs, the array can be used to represent the base array, that is, level N, of the pyramid by configuring each PE as a 1-bit processor to represent one node of the array, NxN nodes in all. To represent the layer above the bottom layer, layer N-1, each set of four

PEs is configured as a 4-bit processor to represent one node of the array, there being N/2xN/2 nodes in all. Thus, this layer has one quarter the number of processors, as required for the pyramid structure, but each processor is four times as powerful as the ones in the base layer. Similarly, the next higher layer can be represented by configuring the array as 16-bit processors in a N/4xN/4 array. Extending this approach to the apex of the pyramid, level 0 would be represented by a single NxN-bit processor.

To implement such a design requires specification of its three major components and the interconnections of these components. The three components are the processing elements, the PE memory and the single controller which broadcasts instructions to all PEs. We will discuss the implementation in terms of an array that can be reconfigured to represent one of the bottom four layers of a pyramid. Rationale for this choice is given in the next section.

The Processing Element

The major implication of the planar pyramid approach on the design of the PE that composes it, is that the design must allow for the PE to be configured as a 1-bit processor, or as a component of a 4-bit processor, a 16-bit processor or a 64-bit processor. A description of a simple, single-bit PE is given in Appendix A. The discussion of the planar pyramid PE is based on the PE described.

Implementation of the PE described in Appendix A requires a register file and an arithmetic/logic unit (ALU). Within the register file are two registers for buffering data to and from neighboring PEs. Since neighborhoods change as the configuration is changed, each neighbor register must be connected to a different neighboring PE for each different configuration. Thus, for r possible configurations, the neighbor out register must have an r-way selector to connect it to the appropriate output bus. The selector is controlled by a mode register which contains the current configuration value.

Notice that we could have used the same neighbor connections for all layers, and simply performed multiple shifts to simulate the effect of having different neighbors at different levels. Unfortunately, this eliminates one of the most important characteristics of this type of pyramid, namely the

logarithmic distance between units. By performing multiple shifts to simulate the connectivity of various layers, the distance between units becomes proportional to the number of units on a side of the array, rather than the logarithm of that number of units.

The effect of reconfiguration on the ALU is reflected only in the arithmetic operations. In particular, the carry values used in the ADD function depend on the current configuration of the PE. Support for reconfigurability requires only the mode register and a carry generation circuit that is sensitive to the value of that register. Since we are considering four configuration modes, a two bit mode register is sufficient. The carry generate and mode register are shown schematically in Figure 1. A ripple type carry is shown in Figure 1, but a Manchester chain or carry look-ahead could be used to improve performance.

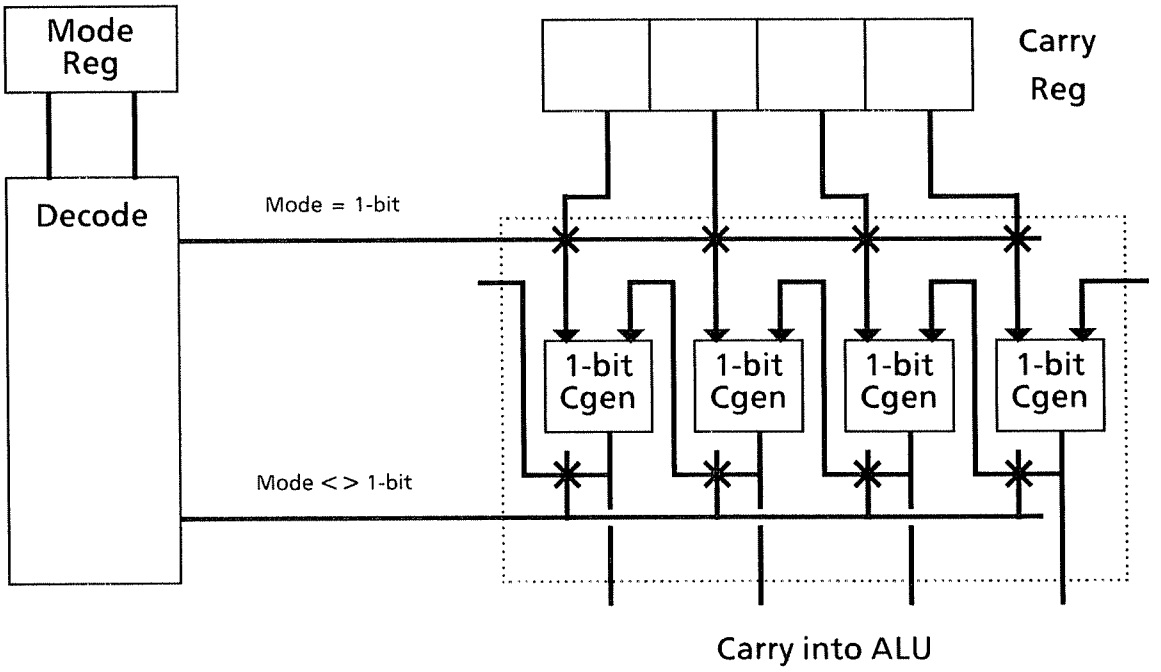
Since a shift operation is generally not useful in a 1-bit processor, it was not included in the PE design. A multiple-bit design could take advantage of such an operation, however, in which case its execution would be gated in a manner similar to that for the ADD instruction.

The Memory Unit

Like the PEs themselves, the associated memory must be reconfigurable. This is necessary in order that a one-bit PE be able to access single bit operands from the same memory that a 4-bit processor is accessing 4-bit operands. Consider, for instance, a block of 32 single bit memory cells. A set of 4 one-bit processors might utilize this memory as eight single bit memory locations per processor. The processor, memory address and bit significance associated with each cell would then be as shown in Figure 2a. If the processors were working with single byte operands, for instance, it would take eight addresses to store one operand. Now, a single 4-bit processor would utilize these 32 cells as eight four-bit memory locations. Without reconfigurable memory, the four bit processor will access one bit from each of four different single-bit processors. A more useful access method is for the 4-bit processor to have access to four bits at a time from one single-bit processor, in order that the access have significance as a four bit value. In that case, the memory organization should be as shown in Figure 2b. To extend the discussion one step further, a sixteen bit processor would util-

Figure 1 - Mode Register and Carry Generation

a) A 4-bit Carry generator



b) A 16-bit Carry generator

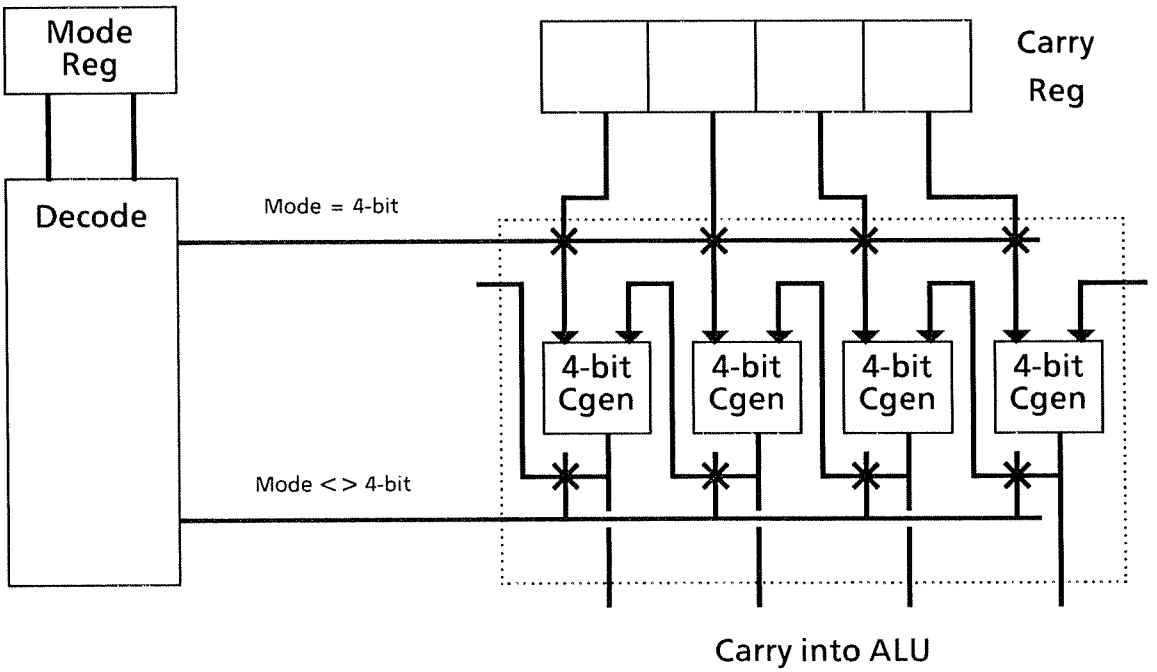


Figure 2 - Memory Organization

a) Four one-bit processors

P3 A0 B0	P3 A1 B0	P3 A2 B0	P3 A3 B0	P3 A4 B0	P3 A5 B0	P3 A6 B0	P3 A7 B0
P2 A0 B0	P2 A1 B0	P2 A2 B0	P2 A3 B0	P2 A4 B0	P2 A5 B0	P2 A6 B0	P2 A7 B0
P1 A0 B0	P1 A1 B0	P1 A2 B0	P1 A3 B0	P1 A4 B0	P1 A5 B0	P1 A6 B0	P1 A7 B0
P0 A0 B0	P0 A1 B0	P0 A2 B0	P0 A3 B0	P0 A4 B0	P0 A5 B0	P0 A6 B0	P0 A7 B0

b) One four-bit processor

P0 A0 B3	P0 A0 B2	P0 A0 B1	P0 A0 B0	P0 A4 B3	P0 A4 B2	P0 A4 B1	P0 A4 B0
P0 A1 B3	P0 A1 B2	P0 A1 B1	P0 A1 B0	P0 A5 B3	P0 A5 B2	P0 A5 B1	P0 A5 B0
P0 A2 B3	P0 A2 B2	P0 A2 B1	P0 A2 B0	P0 A6 B3	P0 A6 B2	P0 A6 B1	P0 A6 B0
P0 A3 B3	P0 A3 B2	P0 A3 B1	P0 A3 B0	P0 A7 B3	P0 A7 B2	P0 A7 B1	P0 A7 B0

c) One sixteen-bit processor

P0 A0 BF	P0 A0 BB	P0 A0 B7	P0 A0 B3	P0 A0 BF	P0 A0 BB	P0 A0 B7	P0 A0 B3
P0 A0 BE	P0 A0 BA	P0 A0 B6	P0 A0 B2	P0 A0 BE	P0 A0 BA	P0 A0 B6	P0 A0 B2
P0 A0 BD	P0 A0 B9	P0 A0 B5	P0 A0 B1	P0 A0 BD	P0 A0 B9	P0 A0 B5	P0 A0 B1
P0 A0 BC	P0 A0 B8	P0 A0 B4	P0 A0 B0	P0 A0 BC	P0 A0 B8	P0 A0 B4	P0 A0 B0

P - Processor number
A - Address
B - Bit significance
(hexadecimal)

ize the 32 memory cells as two 16-bit locations. Since the 16 bits associated with a given location should correspond to four consecutive addresses of the four bit processor, the memory organization in this case would be as shown in Figure 2c. Similar arguments apply to accessing the memory as a 64-bit processor.

It is clear then, that for a given memory address, the memory cell whose data is gated onto a given data line depends on the configuration of the processor that is accessing memory. The memory unit therefore requires a mode register identical to that used in the PE, as well as the substantial additional circuitry required to associate four different addresses and four different data lines to each individual memory cell.

The Controller

A single controller issues instructions to all PEs in the reconfigurable array. The controller is responsible for timing the issue of instructions such that each instruction completes before the succeeding instruction begins. In addition, the controller provides addresses and timing signals to memory, allowing each PE to read and write memory at the appropriate time. For this design, the controller must also set the mode registers in the PEs and memory units, to specify the particular layer currently being represented.

Each of the PE instructions can be carried out in a single cycle. However, in this particular design, data to and from memory is buffered to reduce cycle time. As a result, an instruction that writes to memory collides with an instruction that reads from memory two cycles later. To avoid such collisions, instruction sequences must be rearranged, or No-op instructions, such as 'LDA Areg', inserted into the sequence. In addition, potential write after read hazards must be detected and prevented.

A timing diagram for issue and execution of instructions is shown in Figure 3. Figure 3a shows an instruction stream in which the Memory-In register is used as an operand of each ALU operation, but results are placed in the Areg. In this case, one instruction per cycle may be issued.

Figure 3 - Instruction Issue Timing

a) Sequence without Memory Writes

Inst	cycle 1	cycle 2	cycle 3	cycle 4	cycle 5
1	Issue Addr Issue Inst	Read Mem Ireg valid MI valid	ALU op Areg valid		
2		Issue Addr Issue Inst	Read Mem Ireg valid MI valid	ALU op Areg valid	
3			Issue Addr Issue Inst	Read Mem Ireg valid MI valid	ALU op Areg valid

b) Sequence with Memory Reads and Writes

Inst	cycle 1	cycle 2	cycle 3	cycle 4	cycle 5
1	Issue Addr Issue Inst	Read Mem Ireg valid MI valid	Issue Addr ALU op MO valid	Write Mem	
2		Issue Addr Issue Inst	Read Mem Ireg valid MI valid	Issue Addr ALU op MO valid	Write Mem
3					Issue Addr Issue Inst

Figure 3b shows a sequence of instructions, each of which involves a memory read and memory write. In this case, only two instructions may be issued every four cycles.

The program issued by the controller to the PEs is stored in the controller memory. Sequence control is accomplished in three ways: masking, data independent branching and data dependent branching. Masking involves the Greg in the PE. Whenever the Greg of an individual PE has a value '1', the PE executes the instructions as they are issued. When the Greg value is '0', all instructions are ignored. Data independent sequence control involves unconditional branches or branches conditional on a count value within the controller. Data dependent sequence control involves the Sreg in each PE. The Sreg values of all PEs are ORed together into a result which the controller can test. Such a test can be used to determine if all PEs are in some known state or if none of the PEs are in a given state.

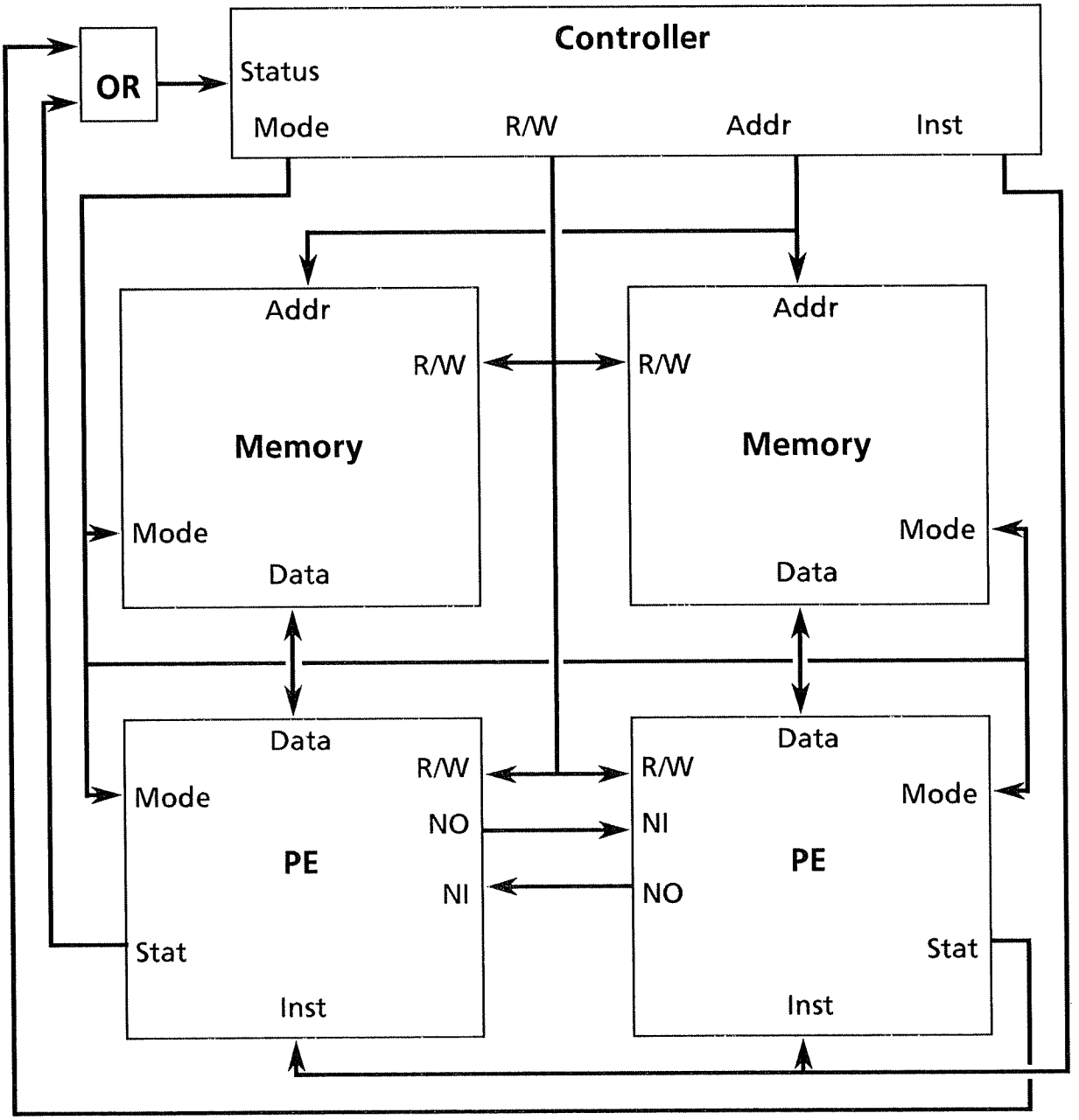
Component Interconnections

The connections among the three sub-units are illustrated in Figure 4. The connections are similar to those required for a standard array processor, except for the neighbor connections, which look deceptively simple in the figure. Due to the reconfigurable nature of the array, each PE must be connectable to r x n other PEs, where r is the number of possible configurations, and n is the number of neighbors for a single configuration. This is a limiting factor of the design, due to the large pin count it represents. For instance, if we use a 4-connected array, with 16 PEs per chip to support four configurations would require 256 pins just for the neighbor connections. Until the technology exists to support such a chip, we must reduce pin requirements either by reducing the number of PEs per chip or the number of configurations supported, or we must multiplex pins. The first alternative requires a high chip count, the second results in performance degradation.

4. Configuration Alternatives

We now discuss advantages and disadvantages of the reconfigurable array, as well as alternatives for using the array to implement a pyramid.

Figure 4 - Component Interconnections



The standard configuration is that previously described: a single N x N array of PEs that can be configured as any one of the log N + 1 layers of a pyramid. The major advantage of such an implementation is the additional processing power at all layers above the base layer. The major disadvantage is that parallelism among layers is lost. This is acceptable if little parallelism can be realized in the given application. There is a hardware tradeoff in this approach as well. In using a single N x N array of PEs, we have eliminated all PEs associated with the non-base layers, as well as the vertical connections among those PEs. On the other hand, the reconfigurable array is more complex than the base layer of a conventional pyramid, requiring the horizontal connections of all layers to be merged into a single layer. In addition, PE and memory are complicated by the reconfiguration hardware.

The possible savings in hardware, and corresponding loss of parallelism of pyramid layers may be acceptable in many cases, particularly in experimental and low-cost systems. However, for real-time, computation-intensive applications, solutions to a very different problem are desired. Given that we can afford a lot of hardware, how might we modify this design to increase performance.

One simple way is to replicate the reconfigurable array, so that each array implements a single layer of the pyramid. We connect these arrays using the vertical connections normally found in pyramid processors, except that here each processor is connected only to a single processor above and a single processor below. Now we have restored the parallelism of layers, but have also increased the processing power beyond the conventional pyramid, due to the multibit processors. In fact, the flexibility of the system is increased as well, since each reconfigurable array represents an entire pyramid. Now the structure of the pyramid can be dynamically configured to fit the particular algorithm being executed. For instance, the first array might process an image through the first three layers, using convergent operations, then pass it to the next array. The second array could apply divergent operations, which correspond to passing the results of computations back down the pyramid. The third array could then be used to perform more convergent operations, and so forth. Thus a sequence of operations involving data flow both up and down the pyramid can be pipelined

through the stack of arrays, where pipelining would have been impossible in the conventional pyramid.

The complexity of implementing an array that can represent all layers of the pyramid has been mentioned. At the same time, the value of carrying the reconfigurability all the way to the top layer of the pyramid should be questioned. Increasing the width of the processor data path is likely to increase performance up to the point where the data path width corresponds to the precision of operands being processed. Thus 4-bit and 16-bit processors are likely to bring an improvement over single-bit processors, and 64-bit processors may also be reasonable. Beyond 64-bits, however, no improvement will likely result from increasing the data path width. Therefore, our approach seems applicable to the implementation of only three or four layers of the pyramid. In fact, the problems of connecting each processor to its corresponding neighbors at every level would tax current VLSI technology when implementing any more than three layers, as mentioned previously.

Given that we can only apply the reconfigurable array to the implementation of the lower three or four layers of the pyramid, how do we implement the upper layers? The standard approach would be to use one processor per node. If we use 64-bit processors to match those at the top of the lower layers, we will still have the bottleneck problem, but the bottleneck will be wider. To extend the approach of putting more processing power per node in higher layers, we might consider grouping processors into networks for each node. Possibly a network of SIMD processors would be useful, but probably an MIMD network would be necessary for many applications.

Returning to our 9 level pyramid, we could use 1-bit, 4-bit, 16-bit and 64-bit processors at levels 8, 7, 6 and 5, respectively. Then, at layer 4, each node could be represented by a network of 4 64-bit processors. At layer 3, 16 64-bit processors per node could be used. This would require a design similar to that used for the bottom layers, although each PE is now a 64-bit processor with its own controller. The complexity of such a design is far beyond any current proposals, but seems to be a natural extension of the attempt to reduce the bottleneck in the upper pyramid.

Finally, we can suggest a compromise in the tradeoff between decreasing complexity and increasing processing power. Instead of increasing processing power by a factor of four at successively higher layers of the pyramid, we could use a factor of two. This decreases the complexity of sharing every processor among all layers, since each processor will now be used for the base layer plus one other layer. In addition, this approach extends to almost the entire pyramid, since only the top two layers would have data paths wider than 64 bits. Arguments for various configurations similar to those made above could be made for this design. Assuming that the computational load per node does not increase by a factor of four in each successive layer, this last approach may provide the best balance among the layers, without introducing the complication of MIMD networks for the top of the pyramid.

5. Concluding Remarks

A two-dimensional reconfigurable array of processing elements has been suggested as a way to represent a three-dimensional pyramid processor. The advantages of such an approach are: 1) the reconfiguration of the processors allows more powerful processors to be used in the upper layers of the pyramid, 2) the vertical connections, between pyramid layers, are simplified, and 3) the number of processors required to represent the entire pyramid is less than that required by a conventional representation. Drawbacks to this approach include the inability to execute more than one pyramid layer at a time, the impracticality of extending the approach to include all layers of the pyramid, and the complexity of the horizontal connections of PEs due to reconfigurability.

We have suggested compromises in the design to allow various advantages and disadvantages to be traded off with one another. In particular, increasing the data path width of processors by a factor of two rather than four in successively higher layers, may provide adequate processing power in the upper pyramid, without increasing the complexity of the design unduly.

Acknowledgements

I am grateful to Len Uhr and Chuck Dyer for the suggestions they have made to improve this paper.

Appendix A - A PE Architecture

The PE required by the reconfigurable array has a fairly conventional architecture. The major differences between this PE and one that might be used in a conventional pyramid involve the different modes in which addition occurs, and the lack of parent to child connections. We describe here a conventional PE that could be used in the reconfigurable array by modifying the ADD operation to account for the configuration mode.

We begin our description of the PE by defining its instruction set, presented in Table 1. Figure A1 is a data path diagram of the PE.

The instruction set provides two types of function: transferring data among registers and applying arithmetic or logical operations to data in registers. There are four data transfer instructions, LDA, STA, LDC and LOADN. The LDA instruction allows data from one register to be moved to the accumulator. This includes the load from memory function which uses the MIreg as the source reg. The STA instruction allows data from the accumulator to be moved to another register. This includes the store to memory function which uses the MOreg as the sink reg. The MIreg is loaded every cycle from the memory-in bus, leaving to the controller the responsibility of having the correct data on the bus at any given time . Similarly, the MOreg value is at all times reflected in the memory-out bus and the controller performs a memory write cycle at the appropriate time. The LDC instruction is simply a move using the ZERO pseudo register value (or its complement) as the source reg, and the carry register as the sink reg, resulting in a Set/Clear Carry operation.

The LOADN instruction is used to ingate the NIreg. The NIreg and NOreg provide the facilities for PEs to communicate with adjacent PEs on the same level of the pyramid. This is accomplished in three steps. The NOreg is first loaded with data to be passed to a neighbor. The controller then manipulates a set of control lines which determine the direction in which data will be passed, either North, South, East or West. Finally, the NIreg is loaded from the NIbus which contains the correct data from some other PE's NOreg.

Table I - Instruction Set

Instructions :

LDA	<src>	move data from src to Areg
STA	<snk>	move data from Areg to snk
LDC	<imm>	set/clear Creg
ADD	<aside> <bside> <res>	add aside, bside and carry to produce res
AND	<aside> <bside> <res>	logical AND of aside and bside
OR	<aside> <bside> <res>	logical OR of aside and bside
XOR	<aside> <bside> <res>	logical exclusive or of aside adn bside
LOADN		load NIreg from neighbor-in bus

Operation Fields :

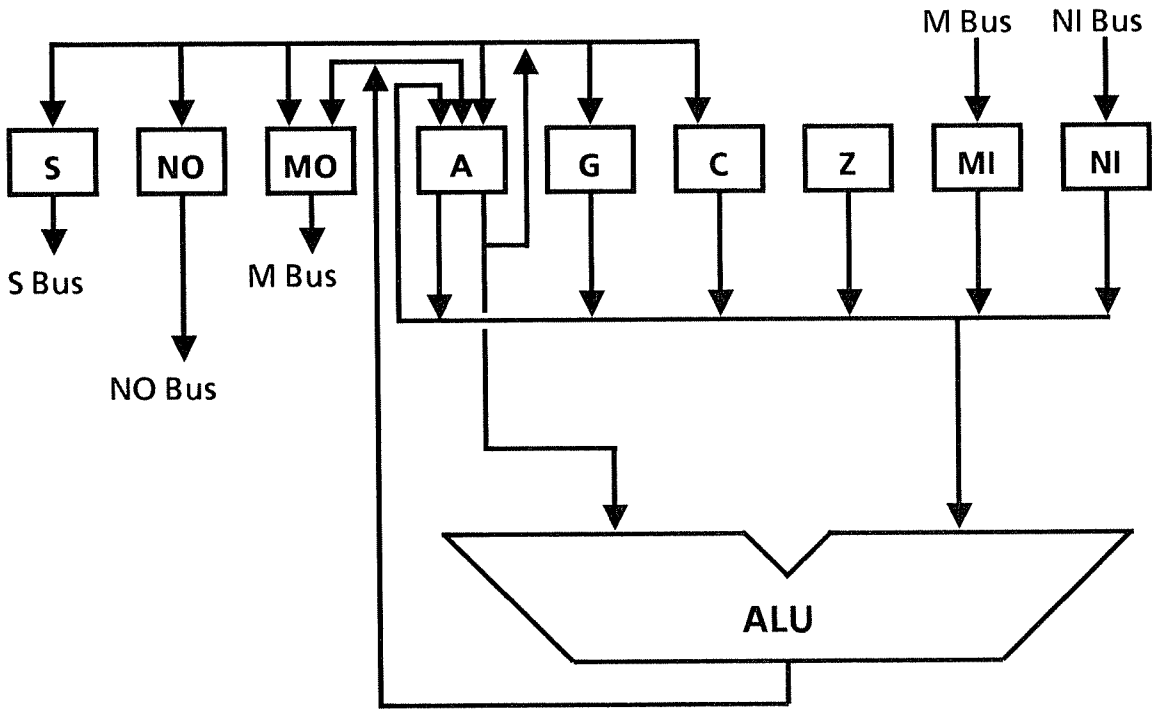
<src>	:	Areg NIreg Creg MIreg Greg ZERO
		-Areg -NIreg -Creg -MIreg -Greg -ZERO
<snk>	:	Areg NOreg Creg MOreg Greg Sreg
<imm>	:	ZERO -ZERO
<aside>	:	Areg -Areg
<bside>	:	MIreg -MIreg ZERO -ZERO
<res>	:	Areg MOreg

where -reg is the complement of the register value

Register Definitions :

Areg	-	Accumulator	:	provides operands for and gets results from most ALU operations
Creg	-	Carry	:	provides carry-in and receives carry-out from ADD operations
Greg	-	Mask	:	enables/disables the processor
NIreg	-	Neighbor In	:	loaded from NOreg of adjacent PE for neighbor communications'
NOreg	-	Neighbor Out	:	loaded with data to be passed to an adjacent PE
MIreg	-	Memory In	:	loaded every cycle from the memory-in bus
MOreg	-	Memory Out	:	loaded with data to be stored in memory
Sreg	-	Status	:	provides input to the grand-OR circuit
ZERO	-	Pseudo reg	:	always contains the value zero

Figure A1 : Data Path Model for PE



In the typical case, the ALU instructions take the value of the accumulator as one operand, the value of a memory location (held in the MIreg) as the second operand, apply a given operation, and return the result to the accumulator. Alternatively, the ZERO pseudo register may be used as an operand instead of memory, and the result of the operation may go to memory, rather than to the Areg. In all cases, the complemented values of the source registers may be used as operands.

References

Ahuja, N. and S. Swamy, "Interleaved Pyramid Architectures for Bottom Up Image Analysis," *IJCPR*, vol. 6, pp. 388-390, 1982.

Batcher, K. E., "Design of a Massively Parallel Processor," *IEEE Transactions on Computing*, vol. 29, pp. 836-840, 1980.

Duff, M. J. B., "CLIP4: A Large Scale Integrated Circuit Array Parallel Processor," *IJCPR*, vol. 4, pp. 728-733, 1976.

Dyer, C. R., "Pyramid Algorithms and Machines," in *Multicomputers and Image Processing*, ed. K. Preston, Jr. and L. Uhr , pp. 409-420, Academic Press, New York, 1982.

Flanders, P. M., D. J. Hunt, S. F. Reddaway, and D. Parkinson, "Efficient High Speed Computing with the Distributed Array Processor," in *High Speed Computer and Algorithm Organization*, ed. D. J. Kuck, D. H. Lawrie, and A. H. Sameh, pp. 113-128, Academic Press, New York, 1977.

Flynn, M., "Some computer organizations and their effectiveness," *IEEE Trans. Computers*, vol. 21, pp. 948-960, 1972.

Hong, T. H., M. Shneier, and A. Rosenfeld, "Border Extraction Using Linked Edge Pyramids," U. of Maryland Tech Report TR-1080, July 1981.

Hubel, D. H. and T. N. Wiesel, "Receptive fields, binocular interaction and functional architecture in the cat's visual cortex," *J Physiol (London)*, vol. 160, pp. 106-154, 1962.

Levine, M. D., "Region Analysis Using a Pyramid Data Structure," in *Structured Computer Vision*, ed. S. Tanimoto and A. Klinger, pp. 57-100, Academic Press, New York, 1980.

Schaefer, D. H., "A pyramid of MPP processing elements - experiences and plans," *Proc. 18th Int. Conf. on System Sciences*, 1985.

Tanimoto, S. L., "Regular Hierarchical Image and Processing Structures in Machine Vision," in *Computer Vision Systems*, ed. A. R. Hanson and E. M. Riseman, pp. 165-174, Academic Press, New York, 1978.

Uhr, L., "Recognition Cones and Some Test Results," in *Computer Vision Systems*, ed. A. R. Hanson and E. M. Riseman, pp. 363-372, Academic Press, New York, 1978.

Van Essen, D. C. and J. H. R. Maunsell, "Hierarchical organization and functional streams in the visual cortex," *Trends in Neurosciences*, vol. 6, pp. 370-375, September 1983.