A VLSI Implementation of an Architecture for Applicative Programming

By

John T. O'Donnell, Timothy Bridges and Sidney W. Kitchel

TECHNICAL REPORT NO. 232

A VLSI Implementation of an Architecture for Applicative Programming

by

John T. O'Donnell, Timothy Bridges and Sidney W. Kitchel

October, 1987

This report is based on work supported in part by the National Science Foundation grant number DCR 85–21497.

This report will also appear in the *Proc. of the International Conference on Frontiers in Computing*, December 1987, Amsterdam.

A VLSI Implementation of an Architecture for Applicative Programming

John T. O'Donnell, Timothy Bridges and Sidney W. Kitchel
Computer Science Department
Indiana University
Bloomington, Indiana 47405 USA

The Applicative Programming System Architecture contains a novel Data Structure Memory (DSM) which supports fast access operations on compact linear data structures. Several problems that arise in implementations of applicative and functional programming languages can be solved efficiently using special data representations on the DSM. Each memory word in the DSM contains a very small local processor, and there is also a tree-structured communications network within the DSM. Therefore the DSM is a massively parallel SIMD machine. This paper describes a VLSI implementation of the DSM architecture and compares its performance with implementations on a conventional sequential computer and the NASA Massively Parallel Processor.

Keywords: applicative language, functional language, SIMD, VLSI, fine grain parallelism, massively parallel processors.

1. Introduction

Applicative (or functional) programming languages have gained widespread interest because of their expressiveness, elegance and good mathematical properties [4]. Unfortunately, applicative programs are still much slower than equivalent imperative ones. There are several reasons for this. One of the problems is simply that conventional computer architectures are well matched to the implementation of imperative languages, but they give poor support for the data structure operations needed for implementing applicative languages. This paper shows how a VLSI architecture can help to solve that problem.

The applicative programming system architecture (APSA) [8, 9, 10, 12] contains a massively parallel intelligent memory unit which directly implements some of the key data structure algorithms needed by applicative languages. Each word in this data structure memory (DSM) contains a small amount of processing power, and many of the data structure algorithms cause most or all of the memory words to perform an action in parallel. In addition, the DSM contains an interconnection network, and the network's nodes also contain small processors.

Because of its massive parallelism, the DSM cannot be implemented efficiently on a single processor. Because of its fine grain organization, the DSM is also inappropriate for hardware implementation using standard chips. Fortunately, there are two promising ways to implement the DSM. The first method is to program the DSM's operations on general purpose SIMD machines such as the Massively Parallel Processor [13] or the Connection Machine [3]. The second method is to implement the DSM directly with a custom VLSI design. Both of these methods have now been used successfully. An implementation of APSA on the MPP is already operational [11], and this paper describes the first VLSI implementation.

Section 2 gives an overview of APSA and briefly describes the kinds of data structures that it supports. Section 3 describes the architecture of the DSM and Section 4 shows how to program it. A program that illustrates the DSM's microinstruction set is given in Section 5. Section 6 then describes the VLSI implementation, and its performance is discussed in Section 7. APSA is compared with two related architectures in Section 8, and Section 9 summarizes the results.

2. Overview of APSA

It is conventional to view a memory system as an inert device which just holds data, while all computation takes place in the processor. Conventional RAM chips implement this kind of memory.

Since data structures reside in the memory while all activity takes place in the processor, operations on a data structure (such as measuring the size of a list or traversing a graph) require a large amount of communication between the processor and the memory [1]. Unfortunately, the data path between the processor and memory is only a few words wide, so any operation that affects all the components of a data structure must take place sequentially, and the operation requires time proportional to the size of the structure.

A better alternative is to design a memory that can directly perform global operations on a data structure, without sending individual words to the processor. In addition to reducing the communications bottleneck on the bus connecting the processor and memory, this approach also makes it possible for the memory to operate simultaneously on all the words in a data structure. We can do this by attaching a very small processor to each word in the memory, and including an interconnection network within the memory. This idea has been around for a long time, and it was used in several early associative machines and content-addressable parallel processors [2]. APSA also uses this approach.

Another way to think of conventional computers is that they consist of a control processor (the CPU) which issues instructions to be executed on an auxiliary processor (the memory). The memory can execute only two instructions: $x = fetch(a)$ and $store(x, a)$ where a is an address and x is a data value. APSA generalizes this view by supporting a much more powerful set of primitive operations.

The DSM supports fast access operations on compact linear data structures. For example, if all the elements of a list are stored in adjacent memory cells, then the DSM can index into the list, measure the size of the list, search for an element associatively, perform an arithmetic operation on some or all of the elements, insert a new element into the list, delete an element, etc. Furthermore, *all of these operations take unit time—their speed is a small constant independent of the size of the list.* In contrast, each of these operations requires linear time on conventional computers.

Figure 1 shows the global organization of APSA, which is just like ordinary computers except that there are two memory units instead of one. The RAM contains the control

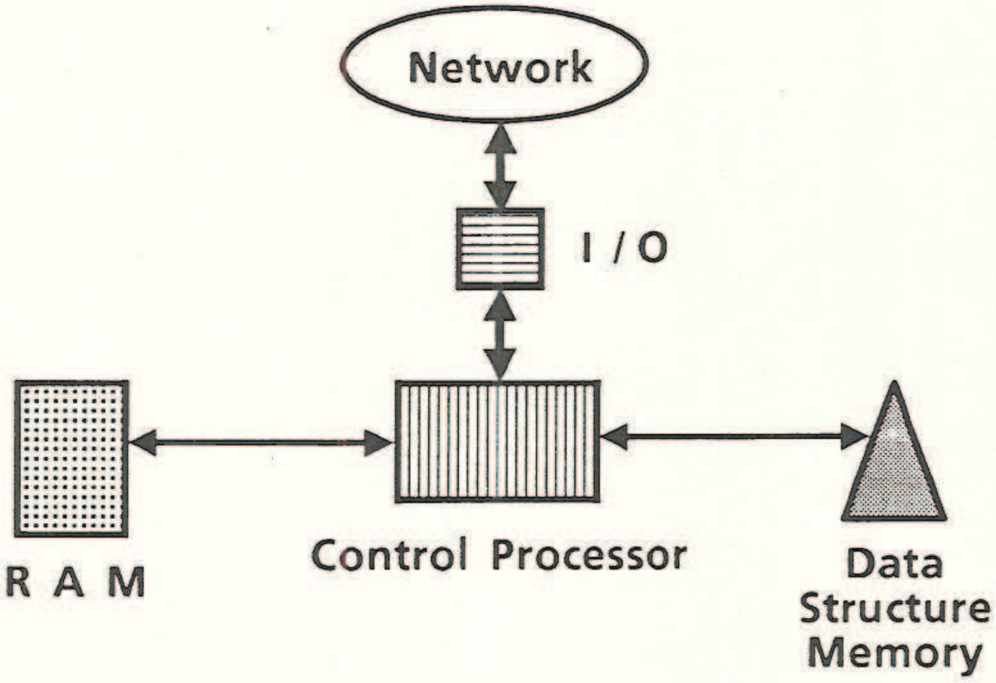


Figure 1. Global organization of APSA

processor's program and all data that can be accessed efficiently by *fetch* and *store*. The DSM stores and manipulates compact linear data structure representations (such as variable size vectors, lists, environments, aggregates, etc.).

The purpose of the DSM is to reduce the time complexity of key data structure algorithms. It does this at the cost of extra hardware complexity compared to a RAM. The next section briefly describes the organization of the DSM.

3. The data structure memory

There are two kinds of operation that must be supported by the DSM's hardware. First, there are global operations that involve communication among a set of adjacent memory words. Examples include broadcasting an instruction to all the memory cells, indexing into a list, and resolving multiple responders after an associative match. Second, there needs to be a fast way to shift a set of adjacent words in parallel in order to allow insertions or deletions in a structure.

The DSM contains two internal interconnection networks that are used to implement these two kinds of operation (Figure 2). A binary tree of combinational logic nodes implements the global operations, and data paths connecting each memory cell to its predecessor and successor permit the machine to shift all the words in a data structure in parallel. Therefore the DSM is organized as a binary tree with additional connections between adjacent leaves.

In principle, the tree nodes do not need any local processors or memory. All the necessary DSM operations can be implemented with a purely combinational tree (i.e., a

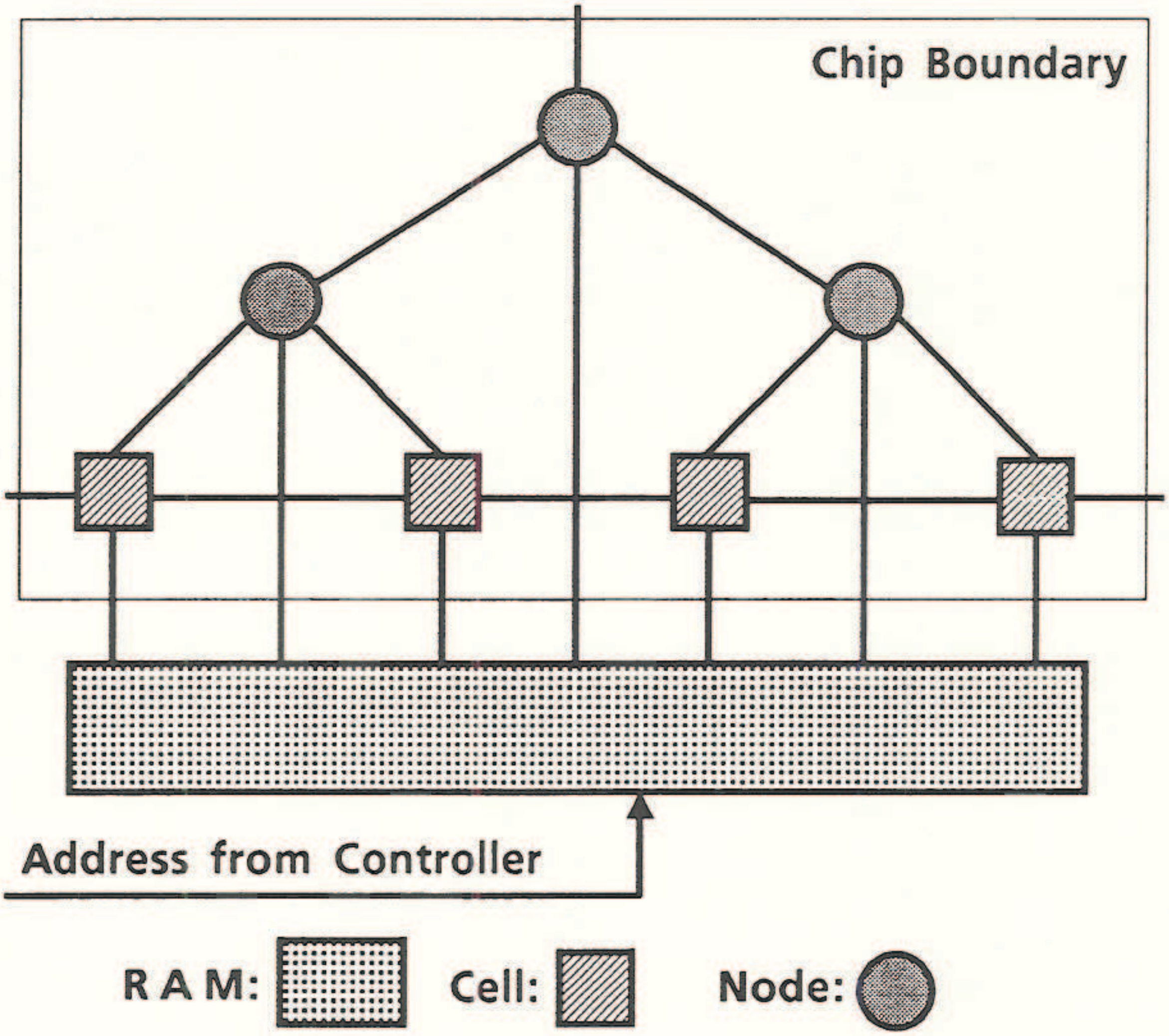
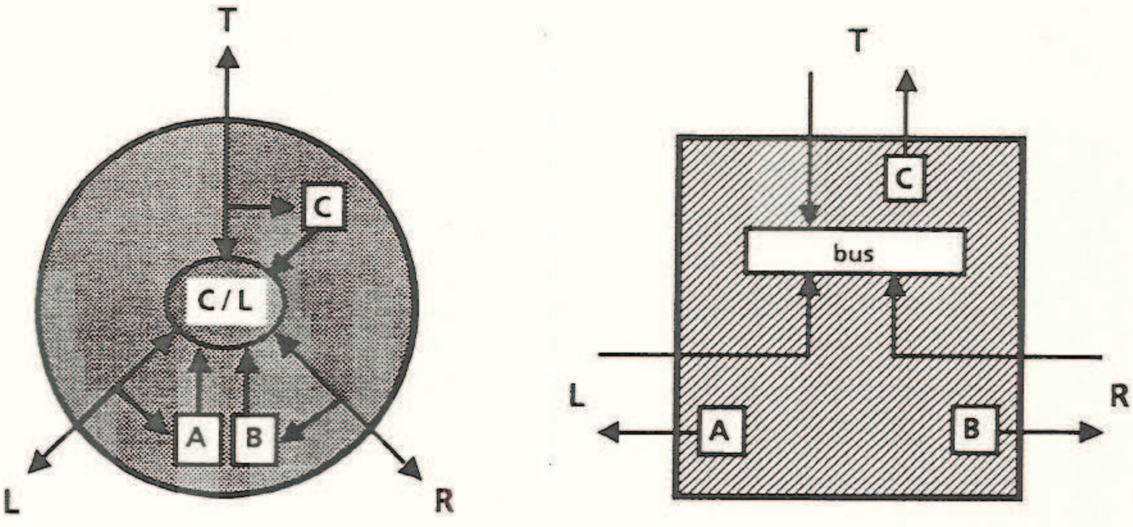


Figure 2. Organization of the Data Structure Memory (DSM)

---

tree where nodes contain logic gates but not flip flops). However, this organization requires wide data paths up the tree (on the order of 100 bits wide), leading to several practical problems for a VLSI implementation. It is easier to build a bit serial tree machine with 1-bit wide data paths. In order to implement some of the DSM's instructions, a bit serial machine needs to latch certain values in the tree nodes. Consequently, the tree nodes need to have local memories and local processors, just like the leaf cells.

These hardware constraints suggest a two level architecture. The high level architecture (an abstract machine) performs operations directly on the data structures. Its tree nodes are purely combinational and its data paths and registers are wide. The high level architecture would be expensive to implement in hardware (although it would be possible to do so). In contrast, the low level architecture is bit serial and its tree nodes contain memory. The low level architecture is designed to be easy to implement in hardware while still containing enough power to emulate the high level architecture. This approach, called "microprogramming", is common practice in processor design.



a) Node

b) Cell

Figure 3. Processing element ports

4. The microinstruction set

This section describes the bit serial architecture of the DSM from the programmer's point of view, and the following section shows how to implement several high level data structure operations as microcode.

The purpose of the DSM is to manipulate linear data structure representations which reside in "memory cells". Each cell contains some local addressable memory (typically 128 bits or more) along with a local bit serial processor. The tree structure permits communication among the cells, but it is not used to store permanent data. The memory cells are the leaves of the tree, and we will call them "cells". The non-leaf nodes in the tree will be called "nodes".

There are two tree communication operations, called upsweep and downsweep. An upsweep causes each node to apply a logic function f to its left and right subtree inputs (li and ri respectively). The result $f(li, ri)$ is sent immediately to the top output port (to) and the input values are latched in the A and B registers. The downsweep operation is similar. Figure 3 shows the relation between the ports and registers during sweeps.

The nodes and cells have similar architectures, although they differ in the way they handle communications and arithmetic. For convenience, we use the term "processing element" or PE to refer to a node or a cell. Figures 4 and 5 show the internal architecture of a cell and node.

Each PE contains a local bit-addressable memory and four 1-bit registers called A, B, C and D. There is also a local function box which can compute an arbitrary boolean function of two or three inputs. The registers, memory port, communications ports and function box belonging to a PE are all connected on a local bus within the PE. Each PE has its own bus.

When a microinstruction requires access to the PE local memories, the control processor

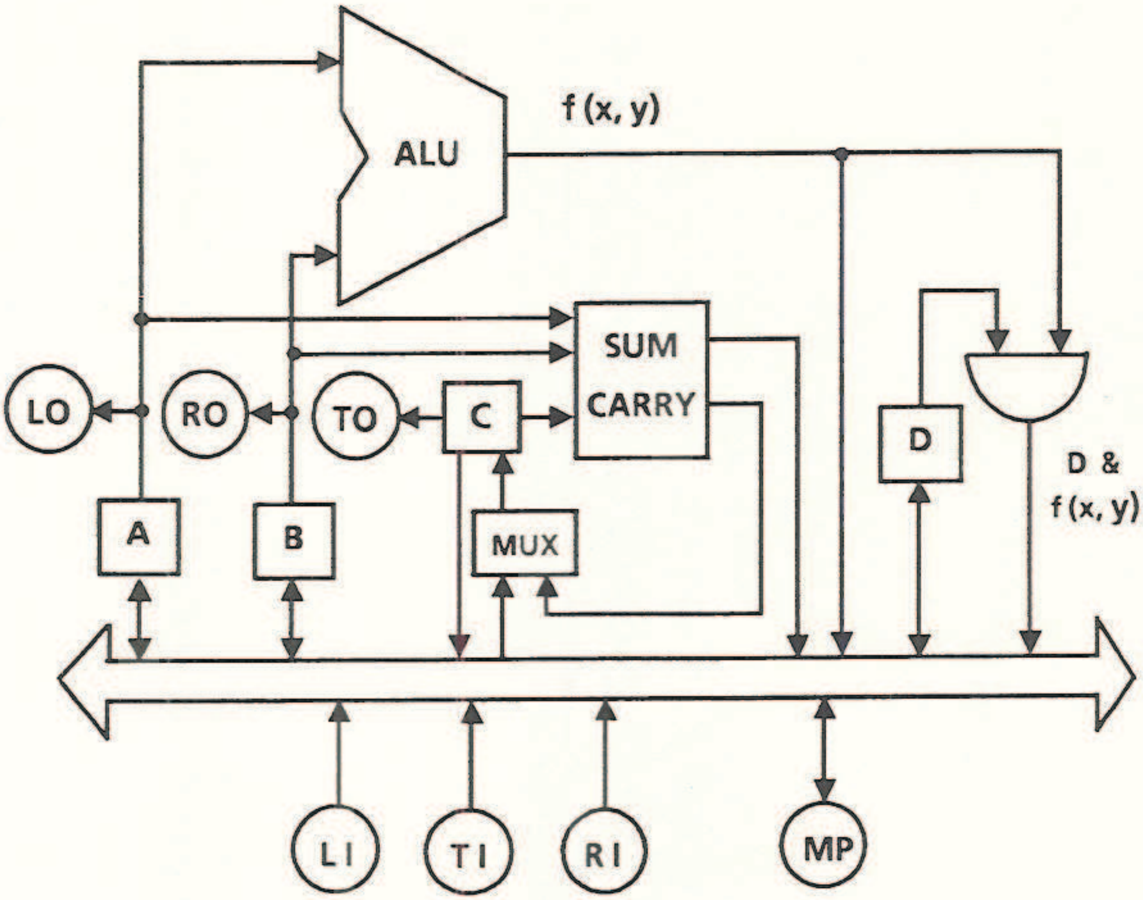


Figure 4. Cell architecture

places the address on a bus that goes to all the local memory RAMs. Therefore all PEs must access the same memory location; the hardware does not allow a PE to compute its own memory address independently of the other PEs. Relaxing this restriction would require a large increase in hardware complexity.

Each microinstruction contains a specification of which PEs are to execute it, an opcode and zero or more operands. The specification of which PEs should execute a microinstruction contains two fields. The *cnd* field (for "conditional") is a flag which may be uncond, meaning that the selected PEs should unconditionally execute the microinstruction; or cond, which specifies conditional execution depending on the value of the PE's local D register. The *petype* field specifies the class of PEs that are to execute the microinstruction. This may be `cell`, `node` or `all`.

A description of each microinstruction is given below. It is convenient to divide them into three classes: (1) the microinstructions that may be executed on any PE, (2) the microinstructions which apply only to the cells and (3) the sweep microinstructions, which involve the cells and nodes in different ways.

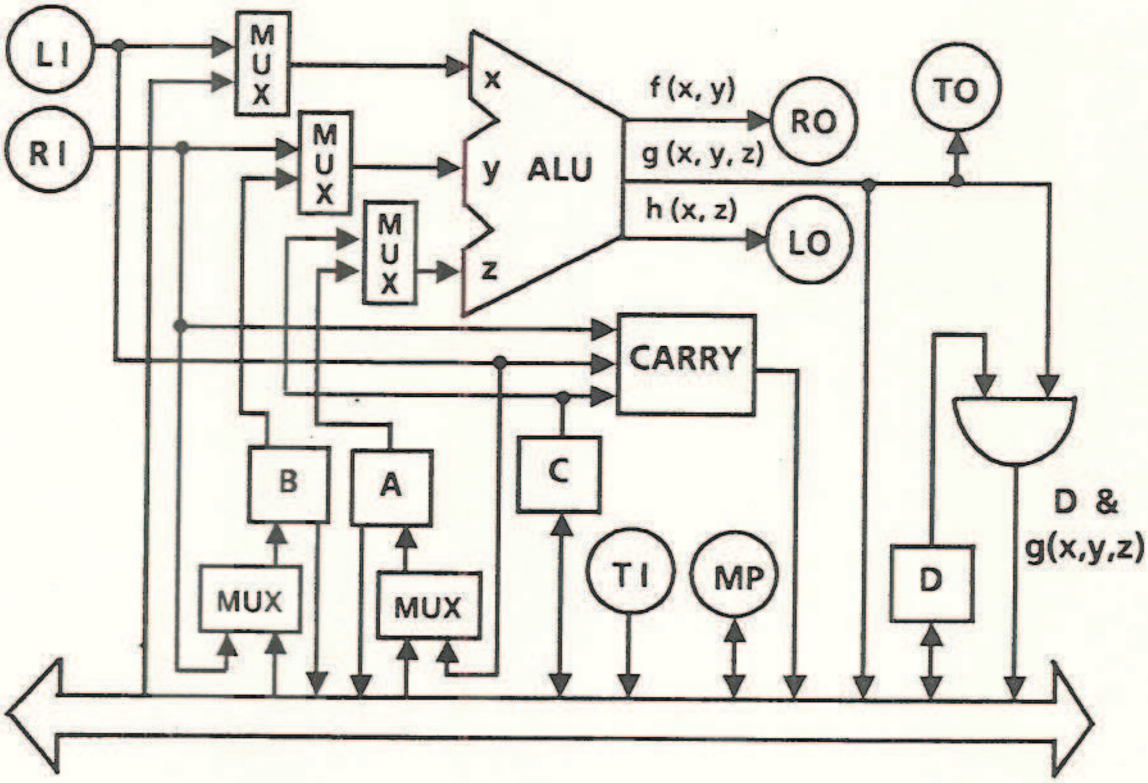


Figure 5. Node architecture

The following instructions may be executed by all PEs.

- *cnd petype* `load` *reg* `:=` *addr*

  The bit in the PE's local memory with address *addr* is loaded into *reg*.

- *cnd petype* `store` *addr* `:=` *reg*

  The value in *reg* is stored into the local memory at address *addr*.

- *cnd petype* `move` *reg* `:=` *reg*

  The value in the source register (right hand side) is copied into the destination register (left hand side).

- *cnd petype* `logic` *reg* `:=` *fcn2*

  The two-bit logic function is applied to the values in the `A` and `B` registers, and the result is copied into the destination register.

- *cnd petype* `andmask` *fcn2*

  The value `D` · *fcn2*(`A`,`B`) is placed in the `D` register.

The following instructions are only executable on the cells, so they do not need to specify *petype* .

- *cnd* `add`

  The sum of `A`, `B` and `C` is stored into `A`, while the carry is stored into `C`.

- *cnd* `neighbor` *dir*

  The direction *dir* must be either `lookleft` or `lookright`. If it is `lookleft` then each cell stores the `B` register value of its left neighbor into its own `A` register; this shifts data to the right. If *dir* is `lookright` then each cell stores the `A` register value of its right neighbor into its own `B` register, which shifts data to the left. If *cnd* is `cond` then only cells with `D`=1 will store a value into a local register, but every cell makes its `A` and `B` values available to its neighbors, regardless of the value stored in `D`. The left neighbor of the leftmost cell is the left port *LP*, and the right neighbor of the rightmost cell is the right port *RP*.

The following instructions involve all the nodes and all the cells.

- *b* = `uplogic` *fcn3*

  Cells: $to \equiv \texttt{C}$

  Nodes: $\texttt{A} := li \quad \texttt{B} := ri \quad \texttt{C} := \text{carry}(li, ri, \texttt{C}) \quad to \equiv fcn3(li, ri, \texttt{C})$

  Each node receives its left and right inputs (*li* and *ri*) from the processing elements below it. Nodes at the bottom level receive their inputs from the `C` registers of the cells below them, while higher nodes receive their inputs from the combinational outputs of the nodes below them. Each node stores its left input *li* into its `A` register and stores its right input *ri* into its `B` register. Its combinational output (sent to its top port *to*) is $fcn3(li, ri, \texttt{C})$. The combinational output of the top port can be read by the controller and stored in a variable *b*.

- `dnlogic` *b* $fcn2_1$ $fcn2_2$

  Each node receives a combinational input *ti* from its top port and stores it in its local `C` register. The control processor supplies the top input value *b* for the top node in the tree. The two function arguments are used to compute the combinational outputs for the left and right ports respectively. The left port output is $fcn2_1(ti, \texttt{A})$ and the right port output is $fcn2_2(ti, \texttt{B})$. Each cell stores its top port input into its local `C` register.

The following instructions are similar to `dnlogic`, except they specify the logic function differently. While `dnlogic` allows the programmer to specify separate functions for the left and right outputs, the `dnleft` and `dnright` instructions allow a single more complex function to be used for one of the outputs, while the other output is simply connected to the top input. These instructions are useful for implementing complex instructions that treat linear data structures asymmetrically. These instructions are not implemented in the current VLSI chip.

- `dnleft` *fcn3*

  Nodes: $\texttt{C} := ti \quad lo \equiv fcn3(\texttt{A}, \texttt{B}, ti) \quad ro \equiv ti$

  Cells: $\texttt{C} := ti$

- `dnright` *fcn3*

  Nodes: $\texttt{C} := ti \quad lo \equiv ti \quad ro \equiv fcn3(\texttt{A}, \texttt{B}, ti)$

  Cells: $\texttt{C} := ti$

5. APSA programming techniques

Since the DSM hardware implements very low level microinstructions, a high level algorithm should not use them directly. Instead, we define an abstract architecture with an intermediate level of data structure instructions. These instructions must be designed to be easy to use in the high level algorithm, but it must also be possible to implement them efficiently using the low level microinstructions. This section illustrates this approach by showing a typical data format for the abstract machine, along with the microcode used to implement several typical intermediate level instructions.

Each abstract machine word contains a type field, a value field and several flags. All of these fields and flags are stored in a cell's local RAM, although they will be brought into the registers (one bit at a time) whenever an instruction is manipulating them. The contents of a cell can be represented by a box with the following format:

● ○ *type* *value*

The type field specifies whether the value field is a symbol, integer, pointer, available, etc. The "select" flag is used to locate one or more cells that are currently being used, while the "mark" flag is used for insertions and deletions. The symbol "●" indicates that the select flag is set in a cell, while the symbol "○" indicates that the mark flag is set.

A common operation is to search the memory for a cell that contains a certain value. The search instruction *match*(x) sets a select in every cell whose type field contains *sym* (indicating that the cell contains a symbol) and whose value field contains the specified symbol x. For example, suppose that the memory contains

avail	*sym* b	*sym* a	*sym* d	*sym* e	*sym* a

Now if we execute the instruction *match*(a) then the result will be

avail	*sym* b	● *sym* a	*sym* d	*sym* e	● *sym* a

The following microprogram implements the match instruction. It begins by putting the constant 1 into the A register in every cell; this is done by specifying a logic function that returns 1 on all inputs. Then the program iterates over the bits in x, clearing the A register in cells which have values that differ from x. First the loop fetches the value bit from a cell into the B register. Then it computes the new value of the A register, which is $A \cdot (B = x[i])$, where $x[i]$ denotes the current bit in x. However, there is no way to make the cells compute an expression like $B = x[i]$ directly. Instead, the program first tests the current operand bit $x[i]$, which is in the control processor, in order to decide how to calculate the new value of A: if the operand bit is 1 then $A = A \cdot B$ but if the operand bit is 0 then $A = A \cdot \neg B$. At the end of the loop the value of A is stored into the select flag location in the local memory.

```
match (x)
   uncond cell logic A f2con1
   for i := 0 step 1 until value_size
      begin
         uncond cell load B [value_addr+i]
         if x[i] = 1
            then uncond cell logic A f2and
            else uncond cell logic A f2andnot
      end
   uncond cell store select A
```

Instructions that look at a sequence of adjacent cells must usually use the tree sweep instructions. One of the simplest of these instructions is "leftmost", which clears the select flag in all cells except the leftmost cell that already had select set. This instruction provides a convenient way to deal with multiple responders after a match. For example, executing the leftmost instruction with the memory state shown above produces:

avail	*sym* b	● *sym* a	*sym* d	*sym* e	*sym* a

On a conventional computer, this instruction requires time proportional to the size of the memory. However, the DSM can execute it in a constant number of microinstructions, regardless of the size of the memory. Six microinstructions are sufficient:

```
leftmost()
   uncond cell move C := A
   b = uplogic (f3or2)
   uncond node move B := A
   dnlogic b f2arg1 f2andnot
   uncond move cell B := A
   uncond cell logic A f2and
```

This algorithm consists of an upsweep which identifies which descendants of every tree node contain a cell with select set, followed by a downsweep that turns off the select flag in the right subtree of any node whose left subtree has a cell with select set.

6. The VLSI implementation

We have successfully implemented a prototype of APSA in VLSI hardware and measured its performance. The initial results are very encouraging: the DSM hardware is faster than expected, and its simplicity should enable us to construct a moderately large machine.

Since the control processor is a conventional architecture, we used a standard IBM PC AT compatible machine. The control algorithms are written in C. There were only two systems that had to be implemented in hardware: the DSM is fabricated as a VLSI chip, and the interface between the DSM and the control processor is a circuit board based on standard Programmable Logic Device (PLD) chips.

The DSM chip contains a complete tree architecture with four cells and three nodes. In addition, it contains an isolated node which makes it possible to build a larger DSM out of two smaller ones. We also placed several smaller test circuits on the chip, in order to be able to study the performance of partial circuits in case the full tree didn't work. Fortunately the entire DSM architecture worked correctly.

The nodes and cells are very simple architectures, so there is room for a number of them on a typical chip layout. We combined them with a standard "H tree" geometry [7], which leads to a relatively efficient usage of the chip area.

Before designing the cell and node circuits, one major organizational problem must be solved: how should the local PE memories be implemented? There are two alternatives: either (1) incorporate the local memories into the cell and node circuit designs, or (2) use off-chip RAM to represent the local memories. A number of issues are involved in this decision.

- Standard RAM chips have benefited from an enormous amount of refinement, and we could not hope to achieve the same memory density in a custom VLSI chip.
- Placing the local memories on the DSM chip would make the nodes and cells larger, reducing the number of processing elements that would fit.
- Standard RAM chips contain far more memory than needed for the DSM processing elements. That means we would never need to worry about running out of memory. However, building the local memories into the DSM chips would require us to decide in advance how many bits are needed. Since the purpose of this project is to support research into parallel data structures, this decision would be very difficult.
- Using separate RAM chips would increase the total chip count.
- The prototype DSM chip contains 8 processing elements (a tree with 4 cells and 3 nodes, plus an isolated node for connecting chips into bigger trees). Therefore a single RAM chip organized into 8-bit bytes could provide one bit for each processing element, and only one RAM chip would be needed for each DSM chip.

For the prototype DSM chip, it is clearly better to use separate RAMs. However, it will be necessary to reconsider this question for larger DSM designs in the future. For example, if a DSM contains 128 processing elements, then either 16 RAMs will be needed for each DSM (greatly increasing the chip count) or else 16 sequential RAM cycles will be needed to implement memory access instructions (severely slowing the machine).

Most of the DSM design effort went into laying out the cell and node architectures. Fortunately these architectures are very similar: they both consist of four ports (top, left, right and memory), four registers (A, B, C and D) and a function box, all connected on a local bus. The most important difference between a node and cell is in the ALU design.

Each PE contains a local bus used to connect the four PE registers, function box, adder, and D masking of the ALU output. The four 1-bit registers are implemented as 2-phase latches driven by ϕ_1 and ϕ_2 clocks. Access to and from the bus is provided through complementary pass transistors. The boolean function box is implemented by a multiplexor where the operands serve as the multiplexor control. The adder and D register masking of the ALU are implemented as specific combinational logic circuits.

Each PE has four I/O ports, three to provide access to one of the DSM's two interconnection networks and the fourth serving as a memory port to off-chip RAM. The data paths to and from the interconnection network ports are different in the tree nodes and the leaf cells. In the tree nodes it is sometimes necessary to look at two inputs concurrently, therefore some input ports are routed directly or through secondary buses instead of to the main bus. On the other hand, the leaf cells can only store the value of an input port into one of the registers, so all inputs can be routed directly onto the main bus.

Outputs to the interconnection networks are also different for tree nodes and leaf cells. The tree nodes' outputs are combinational outputs of a special ALU. This ALU takes three inputs, x, y and z, and produces three outputs: $f(x,y)$ goes to the right output, $g(x,y,z)$ goes to the top output, and $h(x,z)$ goes to the left output. This ALU is implemented as a tree of multiplexors. The first level of muliplexors implements the functions f and h, while the final level F combines them into the function g. Therefore the functions f, g and h are not independent, but rather $g(x,y,z) = F\big(f(x,y),\, h(x,z)\big)$. Since some of the inputs to the ALU are combinational and the outputs to the tree network are combinational, this design implements combinational paths all the way between the root and the leaf levels of the tree. The PEs at the leaf level of the tree have no combinational outputs. Therefore the leaf cell outputs are wired directly from the A, B and C registers.

The DSM chip contains only registers, function boxes, and interconnections. All instruction decoding and control are centralized on the interface board. The DSM chips receive individual control lines that specify register loads, bus control, function selection and RAM read/write operations. This organization eliminates much redundant circuitry and reduces the area of individual PEs. As a result more PEs can be placed on a chip. The cost is that more pins are needed, but this is not a problem because the data ports are only one bit wide.

We designed the PE modules using the Berkeley 1986 VLSI Tools [14], and the circuit layouts were done using the Magic interactive VLSI editor. These designs were simulated using the ESIM event simulator and the Crystal timing analyzer. The resulting modules were tiled into an H-tree layout of the DSM using the MQUILT tiling tool. The layout specification input to this tool was generated manually since there were only 10 PE's per chip, but could easily be automated for larger layouts. The final layout contained a connected tree with seven PE's, an isolated tree node, and two isolated leaf cells. A small amount of additional circuitry was included to test the properties of the register design used in the PE's and to test the signal delay caused by pad circuitry.

The chips were then fabricated by the MOSIS fabrication facility [16]. Since this was a prototype design, we selected the cheapest technology, which was scalable CMOS P-well with 3.0 micron feature size [17]. The prototype fit onto a medium size chip of 4.6×6.4 mm and 64 pins. The scalable layout rules will allow us to fabricate this same design in denser technologies; 1.2 micron is now available through MOSIS.

Careful simulation of the design was the key to getting a working chip. The first fabrication run failed because there was an error in the standard library pad layout that we used, and we had not simulated the pads themselves. Therefore we modified the pad layout, and also learned how to make the simulator handle the pads. This meant that there were no shortcuts in the simulations. As a result, the second fabrication run was fully functional.

Size	Vax 780	MPP	VLSI
2×2	610	12	.16
4×4	2,500	20	.40
8×8	10,500	29	.64
16×16	53,000	39	.86
32×32	175,000	51	1.16
64×64	706,000	68	1.40
128×128	2,872,000	95	1.70

Table 1. 1-bit wide upsweep times (microseconds)

7. Performance results

Table 1 gives a rough comparison of the speeds of the three implementations of the DSM. The numbers give the time (in microseconds) required to perform a 1 bit wide upsweep, using the logic function $to = li$ OR ri.

The first column in Table 1 gives the "layout size" of the DSM, assuming a square H-tree layout. Therefore the total number of leaf cells is i^2 for an $i \times i$ layout.

The "Vax 780" column gives the time required by a simulator written in C running on a Vax 780 computer. This simulator requires time proportional to the number of leaf cells.

The "MPP" column gives the upsweep time for the implementation running on the Massively Parallel Processor [11]. Since the MPP assigns a separate processing element to each leaf cell and to each node at a given level in the tree, the time it takes to do the logic functions is proportional to $\log N$ where N is the memory size. However, the MPP provides only nearest-neighbor communication, so the communication time is slower than $\log N$.

The "VLSI" column gives the upsweep time for the hardware VLSI implementation. Here the total upsweep time is almost logarithmic, because the wire delay from a node to its parent is very low. We do not have enough chips to build a 16K cell machine, but we were able to measure the upsweep time by constructing an unbalanced tree. The left ports of the upper tree nodes were connected to real DSM chips, while the right ports were connected to constant logic values. This does not produce a usable DSM, but it does allow fairly realistic timing measurements. However, very large fabrications of the DSM will probably have additional delays due to longer wires and connections between boards.

Table 1 makes it dramatically clear that implementing APSA on a von Neumann architecture is pointless, while true parallel implementations—using either general SIMD machines or VLSI—are fast enough to be practical. The VLSI implementation of the DSM is slower than a RAM by a constant factor, but it provides a linear speedup for algorithms that traverse linear data structures.

8. Comparison with related work

APSA is loosely related to two other fine-grain parallel implementations of symbolic languages: the FFP Machine (or FFPM) and Connection Machine Lisp. This section discusses the similarities and differences among these three systems.

The FFP Machine [5, 6] is a massively parallel implementation of the FFP language [1]. The FFPM and APSA were developed independently at about the same time. Both of these architectures contain a linear array of processors (called cells in APSA and L-cells in the FFPM) with an additional binary tree of processors (called nodes in APSA and T-cells in FFPM). Numerous variations on tree architectures have been studied for a long time, but APSA and FFPM use the general organization in novel ways.

APSA and FFPM both contain a tree structure and both are intended for implementing functional programming languages, but these similarities are rather superficial. They differ in several fundamental aspects:

- The FFPM is restricted to string reduction because it cannot represent pointers. In contrast, APSA was designed from the beginning to handle pointers well. APSA is unique in its ability to perform both string reduction and graph reduction efficiently. The primary problem with the FFPM is that string reduction may cause enormous communication overhead. This happens when two expressions being evaluated in parallel need access to the same data structure. With string reduction separate copies of the data structure must be made for each evaluation, while graph reduction allows the data structure to be shared via pointers.
- The FFPM is an MIMD architecture which runs a parallel algorithm distributed through its cells, while APSA is an SIMD architecture which runs a sequential algorithm in its control processor. In this respect, APSA is like the Massively Parallel Processor and the Connection Machine, and it is quite different from FFPM.
- The FFPM is an asynchronous machine, while APSA is synchronous.
- The abstract version of APSA can be implemented on standard SIMD machines (and this has already been done), but the FFPM cannot.
- The hardware complexity of the tree nodes and cells is much smaller in APSA than in the FFPM (by approximately 2 orders of magnitude).

Connection Machine Lisp [15], like most programming languages for SIMD architectures, is a sequential language that supports parallel operations on data structures. The primary data structure in CM Lisp is the *xapping*, which is an unordered set of index-value pairs. Indices and values may be arbitrary Lisp objects. CM Lisp contains several interesting operators that map functions over the elements of a xapping in parallel, and there are several mechanisms for controlling this parallelism.

APSA and the Connection Machine illustrate different tradeoffs between hardware complexity and programming style. The tree structure in APSA is able to perform parallel operations on the objects in a linear data structure representation, but it does not have enough communication paths to handle non-linear representations. The Connection Machine does not have this restriction because of its powerful routing network. However, the Connection Machine requires much more complex hardware than APSA.

9. Conclusion

APSA contains a massively parallel intelligent data structure memory that supports a number of powerful operations on linear data structure representations. These operations are especially useful in implementing applicative programming languages and for symbolic processing.

The data structure memory has a very regular architecture which is well suited for VLSI implementation. We have constructed a prototype chip and measured its performance. The VLSI machine implements the critical tree sweep operations approximately 50 times faster than an implementation on the general purpose Massively Parallel Processor. A sequential implementation on a Vax 780 runs more than a million times slower.

The usefulness of APSA will depend on whether the algorithmic speedup provided by the DSM can overcome the cost of its hardware and its slower performance. The initial results look very encouraging, since several interesting algorithms run faster on APSA by $O(n)$ where n is the problem size. Furthermore, the organization of the DSM is very well suited for denser circuit fabrication techniques such as wafer-scale integration.

Acknowledgements

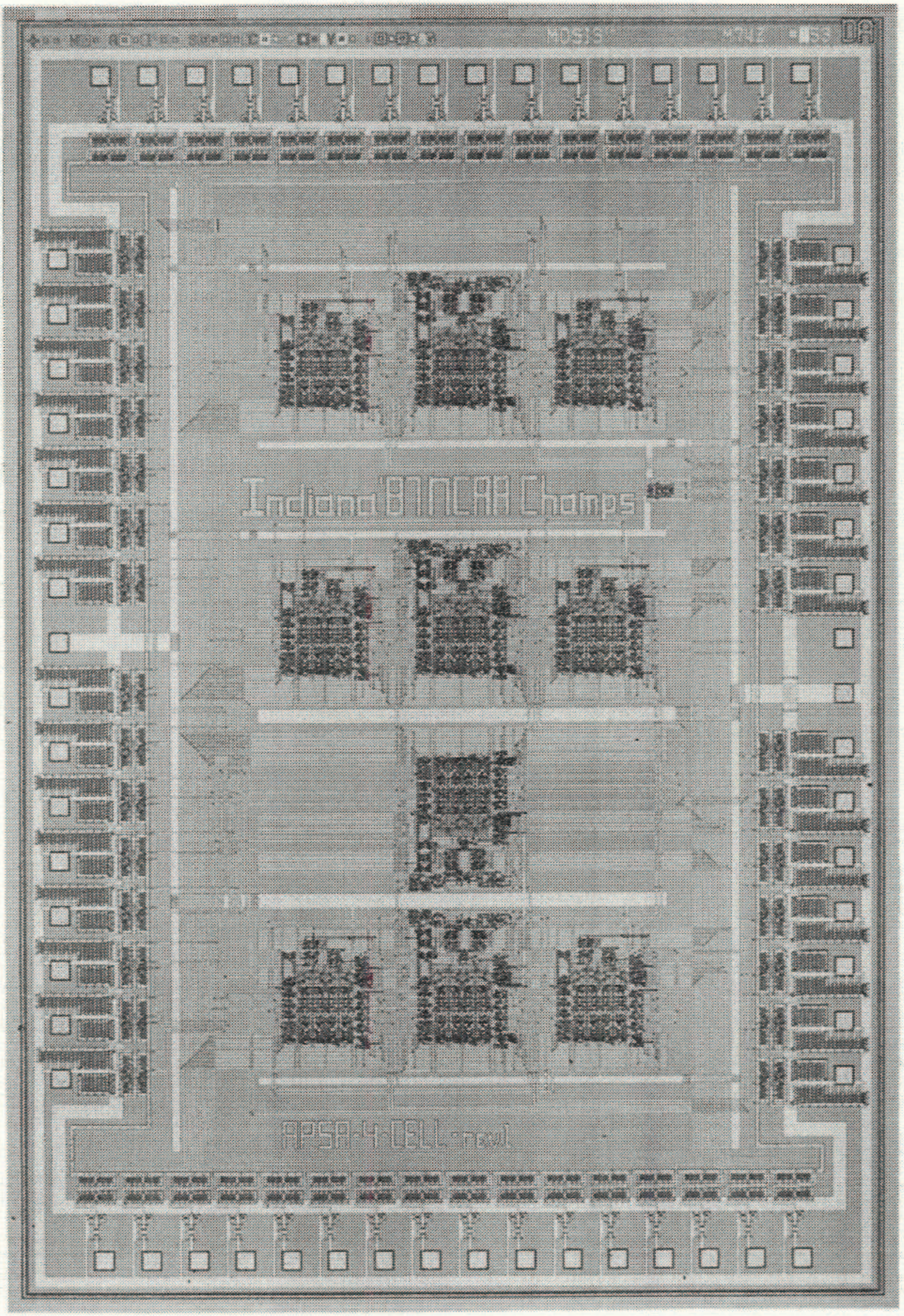
NSF and DARPA supported the MOSIS VLSI fabrication facility. The staffs at ISI and MOSIS helped us get started, David Winkel gave useful advice on design problems, and Robert Wehrmeister supported our design tools.

References

1. John Backus, "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs", *Communications of the ACM*, Vol. 21, August 1978, pp. 613–641.
2. Caxton C. Foster, *Content Addressable Parallel Processors*, New York: Van Nostrand Reinhold Co., 1976.
3. W. Daniel Hillis, *The Connection Machine*, Cambridge: The MIT Press, 1985.
4. John Hughes, "Why functional programming matters", Report 16, Programming Methodology Group, Chalmers University of Technology, Gõteborg Sweden, Nov. 1984.
5. Gyula Magò, "A network of microprocessors to execute reduction languages", *Int. Journal of Computer and Information Sciences*, Vol. 8 No. 5 and Vol. 8 No. 6, 1979.
6. Gyula Magò and David Middleton, "The FFP Machine—a progress report", *Proc. International Workshop on High-Level Language Computer Architecture*, May 1984, pp. 5.13–5.25.
7. Carver Mead and Lynn Conway, *Introduction to VLSI Systems*, Reading, Mass.: Addison-Wesley, 1980.
8. John T. O'Donnell, *A Systolic Associative LISP Computer Architecture with Incremental Parallel Storage Management*, Technical Report 81-5, Computer Science Department, University of Iowa, Iowa City, 1981.
9. John T. O'Donnell, "An architecture that efficiently updates associative aggregates in applicative programming languages", *1985 IFIP Symposium on Functional Programming Languages and Computer Architecture*, Springer-Verlag (Sept. 1985) 164–189.
10. John T. O'Donnell, "An efficient architecture for implementing sparse array variables," *Proceedings of the Twenty-third Allerton Conference on Communication, Control and*

Computing, pp. 986–995, Coordinated Science Laboratory, University of Illinois, October, 1985.

11. John T. O'Donnell, "Parallel VLSI architecture emulation and the organization of APSA/MPP," *Proceedings of the First Symposium on the Frontiers of Massively Parallel Scientific Computation*, NASA Goddard Space Flight Center, Sept. 1986.
12. John T. O'Donnell, "Finely grained parallelism in an applicative architecture," *Proceedings of the Workshop on Future Directions in Computer Architecture and Software*, pp. 372–374, Army Research Office, May, 1986.
13. Jerry L. Potter, ed., *The Massively Parallel Processor*, Cambridge: The MIT Press, 1985.
14. Walter S. Scott, Robert N. Mayo, Gordon Hamachi, and John K. Ousterhout, editors, *1986 VLSI Tools: Still More Works by the Original Artists*, Computer Science Division, University of California, Berkeley, December, 1986.
15. Guy L. Steele, Jr. and W. Daniel Hillis, "Connection Machine LISP: fine-grained parallel symbolic processing," pp. 279–297, *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming.*
16. USC Information Sciences Institute, *MOSIS: MOS Implementation System, User's Manual*, 1986.
17. Neil Weste and Kamran Eshraghian, *Principles of CMOS VLSI Design: A Systems Perspective*, Reading, Mass.: Addison-Wesley, 1985.


MOSIS
DA
Indiana '87 NCAA Champs
APSA-4-CELL-new