N89-19838

SPACEBORNE VHSIC MULTIPROCESSOR SYSTEM FOR AI APPLICATIONS

Henry Lum, Jr.
NASA Ames Research Center
Moffitt Field, CA 94040

Howard E. Shrobe* **and John G. Aspinall**
Symbolics Cambridge Research Center
11 Cambridge Center
Cambridge, MA 02142

Abstract

A multiprocessor system, under design for space-station applications, makes use of the latest generation symbolic processor and packaging technology. The result will be a compact, space-qualified system two to three orders of magnitude more powerful than present-day symbolic processing systems.

1 Symbolic Computing

The tasks for which symbolic computing is uniquely qualified are different from those served well by conventional numerical computing. Conventional programs tend to be uniform, simple, homogeneous, and numerically intensive. Symbolic programs, on the other hand, are diverse and heterogeneous, involving a variety of mechanisms and conceptual tasks within a single program. A single symbolic computing application, for example the management of an autonomous space vehicle, will have to perform a variety of tasks such as hierarchical classification, signal interpretation, hypothesis formation, matching, and logical inference; not to mention conventional numerical tasks. It will have to employ a variety of different mechanisms such as rule-based programming, frame-instantiation, constraint propagation, numerical simulation, object-oriented programming, symbolic mathematics, and truth maintenance; all within a single large system.

The popular notion of an AI program as a single, simple rule interpreter is a gross oversimplification. In fact, symbolic computing places much more serious demands on the system architecture than would be presented by the need simply to to support a simple rule interpreter.

1.1 The Object-Oriented Viewpoint

A viewpoint of a computer that is characteristic of symbolic computation is called the *Object-Oriented Viewpoint*. In this viewpoint memory does not consist of a stream of raw bits organized into bytes or words. Rather, it consists of much larger conceptual entities which are thought of as objects. An object might be something simple like a list, an array, an integer or it might be something with higher semantic content, for example, a node in a semantic network or a data structure representing an entity in the real world.

These objects should have an identity. This means that you should be able to tell the type of an object, just by looking at it. In addition, one should be able to tell its location in memory. The techniques that are used to do this are called *storage conventions*. Ideally, the hardware should guarantee that the storage conventions are never violated.

The object-oriented viewpoint depends upon the ability to make memory seemingly infinite, in the sense that there will always be room for allocating new objects. Indeed, the goal is to free the programmer from worrying about where objects are allocated and when they are deallocated. In practice, this means that the system needs to support garbage collection, the process of reclaiming unused storage. It is necessary that unused storage be reclaimed at a rapid enough rate so that free storage is always available. Garbage collection means that the symmetry of storage is maintained; to the programmer, all storage is the same and its always available.

The second major feature of the object-oriented viewpoint is that the programmer codes using *Generic Operations*. A generic operation is defined as an abstract, conceptual operation which does not reflect the limitations of the hardware. For example, addition is a conceptual operation which is meaningful to apply to integers, floating-point numbers, vectors, polynomials, etc. Ideally, there should be a single operation, called PLUS, which does all of these, dispatching on the type of the objects being added to determine how to perform the operation.

Modern symbolic computing hardware allows this viewpoint to be supported efficiently. It is the hardware's job to check every operation and decide how to perform it based upon the types of the operands. So in effect that hardware will tell itself: "That's a

*Howard Shrobe is also a Principal Research Scientist at the MIT Artificial Intelligence Laboratory.

fixed-point number and therefore I should do integer add," or, "That's a floating point number, I should be performing floating point add." Or, "It's an extended number that I can't directly support at all, but I can support it by this sequence of other instructions."

In addition to higher level code, this approach leads to better debugging capability and supports the concept of incrementality. Since dispatching on the operand type is a runtime function, a new data-type may be added by simply defining the generic operation upon the new type. Existing software can now use the new data-type without recompilation. Any attempt to do an invalid operation on any particular piece of data is detected by the hardware, allowing the programmer to enter a debugging session in the context of the error.

2 Ivory

Symbolics is now implementing a new generation of symbolic processing architectures built upon the Ivory processor. Ivory-based architectures represent the state of the art in satisfying the requirements of symbolic processing, as described in the previous section. In particular, Ivory supplies the following.

- Runtime type checking – Parallel tag processor, late-branch ROM and comprehensive trap logic support generic arithmetic and pointer manipulation.
- Virtual Memory Support – On chip translation buffer, microcoded cache-miss backup and the support of CDR-coded lists (more compact physical memory representation).
- Specialized Lisp operations – Pipelined memory interface and high level microcoded primitives support efficient implementation of operations such as CAR and CDR.
- Garbage Collection – On chip hardware to facilitate efficient GC algorithms such as the Ephemeral GC [Moon, 1984].
- Fast call and return – Specialized datapaths, parallel operations, and fast cycle time support the complex calling strategies required by Lisp.
- Fast "vector" instructions for garbage collection, data-base searching and graphics applications.
- A fast coprocessor interface, primarily used to provide high floating point performance.
- A programmable interleaved memory interface to allow a wide range of memory system speeds and architectures to be used - ranging from small high speed caches to four-way interleaved standard MOS memories.

2.1 Data Architecture

In line with the requirements of the object-oriented viewpoint, every word in memory contains either an object reference or is part of the representation of an object. A machine word contains 40 bits, which are assigned as in Figure 1.

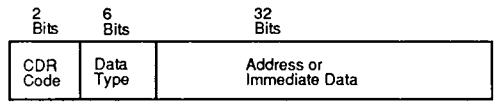

Figure 1: Ivory Memory Word

The *data type field* indicates what kind of information is stored in a word. The *cdr-code field* is used for various purposes. For header data types, the cdr-code field is used as an extension of the data-type field. For stored representations of lists, the contents of this field indicate how the data that constitute the list are stored. This results in a compact representation of lists. The *address or immediate data field* is interpreted according to the data type of the word. This field contains either the address of the stored representation of an object, or the actual representation of an object.

Ivory supports the rich variety of objects found in symbolic processing environments as described in the previous section. General Lisp data structures such as symbols, lists, arrays, strings, and characters are all directly manipulated by the instruction set. For numeric data types, Ivory includes very efficient support (immediate object representation) for 32-bit integers and 32-bit IEEE single-precision floating point numbers. It also supports infinite precision integers, 64-bit IEEE double-precision numbers, rational numbers, and complex numbers.

2.2 Virtual Memory

Ivory implements a 4 gigaword virtual address space. The 32-bit virtual word address is divided into a 24-bit virtual page number and an 8-bit page offset. The virtual page number is mapped via the *Page Hash Table* (PHT) to get a 24-bit physical page number.

While the 256-word page size may seem small by traditional processor standards, it is appropriate for symbolic processors. Symbolic processors tend to have many small functions, small data objects, and little locality of reference. These factors tend to limit the advantages of a larger page size, and the smaller page size allows better allocation of physical memory.

2.3 Garbage Collection

The Ivory memory architecture supports two methods for garbage collection (GC). Both strategies are incremental in nature and identical to the Symbolics 3600 implementation [Moon, 1984; Moon, 1985]. The two methods differ in how they decide to actually reclaim storage. In both cases the garbage collection

process *condemns* or identifies storage it would like to reclaim. This storage is considered to occupy *old space* while other storage is termed *new space*. If the processor attempts to read an object reference to old space, a trap will be taken and the *Transporter* will be invoked. This is a software routine which copies the storage containing the object representation into new space. It also updates the pointer to the old object in memory to point to the copy of the object in new space. In order to signal the trap which invokes the transporter the memory interface looks at the data type of a word to determine if it is a pointer, and if the address field points to old space.

The *Dynamic GC* is used to reclaim objects that have lifetimes on the order of tens of minutes to hours and days. It performs a single linear scan of all of new space, reading every memory word. During this scan the memory interface will cause the transporter to be invoked if the word is a pointer to old space. At the end of the scan, all of new space must point only to new space and the storage used by old space can be reclaimed. This scan is done incrementally so as not to hurt interactive performance.

The *Ephemeral GC* (EGC) is used for reclaiming objects that have lifetimes of the order of seconds to minutes. This scheme is based on the observation that most of the objects created in the system have a relatively short lifetime. The EGC attempts to reclaim the most recently allocated objects by breaking up storage into *levels*, corresponding to how recently an object was created. Ephemeral GC requires the memory system to maintain a database of pages which contain pointers to a more recent level. When the EGC condemns the most recent level it uses the database to scan only those pages which potentially contain pointers to the condemned level. To support this, the memory interface must notice when it is writing a pointer to a more recent level into a page. In Ivory, this information is maintained in the PHT for pages which are in physical memory, and in a companion structure for pages which reside in secondary storage.

2.4 Stack Execution

Ivory uses a stack-based model of execution. The stack is divided into frames, one for each active function. The stack is used for passing arguments, allocation of local variables, and intermediate results of computations. A stack frame is indexed by three pointer registers; the frame pointer (FP), the local pointer (LP), and the stack pointer (SP).

The frame information consists of two words; the offsets of the LP and SP registers from the FP, and the continuation of this frame. The continuation is either the return address of this function, or the next function to call. The FP is used to access the frame information and the arguments to the function. The SP is used to access intermediate results of computations. The LP must be distinct from the FP because arguments are pushed by the caller, and may be pushed in several different ways.

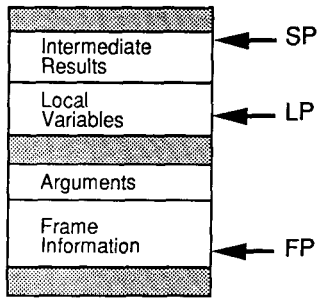

Figure 2: Ivory Stack Frame

2.5 Instruction Set

Ivory performs different operations depending on the data-type of the word that is fetched as an instruction. Most object references push themselves onto the top of the stack. This capability is used to supply full word constant operands. A special data-type is used to push the contents of the word whose address is specified in the address field of the instruction word. Other data-types are allocated to perform specialized types of function calls. There are data-types for calling compiled functions and generic functions using the address field to point to the function. A further type is used to call the contents of a memory word as a function. This is used to implement dynamically linked functions. Finally there is a set of 16 data-types which are further decoded into two "packed" instructions. The 32-bit immediate data, along with 4 bits from the data-type field are combined to form two 18-bit instructions.

The Ivory instruction set incorporates full run-time error and exception checking. Exceptions are cases which are not an error, but cannot be handled by the processor hardware without software intervention. The checking performed by the instruction set includes full checking of data-types, subscript range, uninitialized variables, undefined functions, and detection of integer and floating-point overflow. Exceptions are handled by causing a trap through a vector in memory.

Even though strict error-checking performed is by the instruction set, it is possible to extend the instructions supplied to handle new object types. By utilizing exception handlers and the New Flavors [Moon, 1986] object-oriented programming system, it is possible to define a new object type that masquerades as an existing type in all programs. Conversely it is possible to treat the architecturally-defined data-types as part of the object-oriented system. This permits the def-

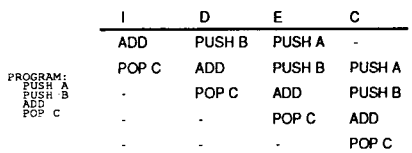

Figure 3: Ivory Pipeline Stages

inition of generic functions (as described in the first section of this paper) which can operate uniformly on instances and objects with different data-types.

2.6 Microarchitecture

The Ivory microprocessor implements a 4-stage pipeline as shown in Figure 3. The first stage fetches the instruction, decodes it, and adjusts the program counter. The second stage fetches the initial microinstruction, computes the operand address, and adjusts the stack pointer. The third stage fetches the operands and computes the result. The fourth stage stores the result, unless a fault has occurred, in which case it restores the state of the third stage.

Instructions spend only a single cycle in the first two pipeline stages, but can spend an arbitrary number of cycles in the execute stage. Simple instructions, such as PUSH, ADD, and EQ execute in a single cycle. Conditional branches are resolved in the second (D) stage. A taken conditional branch executes in two cycles; a not-taken branch in one.

Figure 4 shows a block diagram of the Ivory CPU. Instructions are fetched directly from a 32 word (up to 64 instructions) direct-mapped instruction cache. The cache, which is filled by an autonomous prefetcher, serves to buffer instructions arriving from memory and hold small program loops. A bypass path provides the instruction directly from memory when the first stage is stalled on a cache miss.

The operand address calculation data path contains the stack frame pointer registers and a 32-bit adder/subtractor. It computes the address of the operand and stack pointer adjustment according to the macroinstruction. This data path is also used in parallel with the main data path to accelerate function call/return.

The main data path contains a 128 word top-of-stack cache, a 32-word scratchpad (which contains a duplicate of the top word on the stack in a fixed location), the ALU, and tag checking logic. The ALU includes an adder, boolean unit, shift/mask logic, and support for one-bit-per-cycle integer multiply/divide. Tag checking is done in parallel with with the ALU operation, so that in the common cases no time penalty is paid for type checking. Similarly, ECC checking of data from memory is done in parallel with the ALU using the on-chip ECC logic. Bypass paths for both the top-of-stack cache and scratchpad forward the result of the previous instruction to the ALU as necessary.

The Ivory processor supports a pipelined memory bus which can have up to four outstanding requests at once. An associative queue of outstanding request addresses is maintained for detecting when instructions arrive from memory and installing them into the instruction cache. The memory interface protocol is implemented by an independent state machine which arbitrates between on-chip users of the memory system and other bus masters.

3 Multiprocessing with Ivory

In addition to the features of Ivory described in the previous section, there are several design features of the Ivory processor specifically intended to support multiprocessor architectures. They are:

- Support for Futures.
- Support for Special Variable Binding.
- Synchronization primitives.

3.1 Futures

Futures are a Lisp language construct which appear in parallel extensions to Lisp such as MultiLisp [Halstead, 1985] and QLISP [Gabriel & McCarthy, 1984]. A future is a compound structure which represents a promised value coupled with a process that computes the value. The future is a first class data structure which can be stored in other data structures, loaded and stored even though the process computing its value has not terminated. However, if the value of the future is ever required for a computation (e.g. it is one input to an arithmetic operation) then the processes attempting to *touch* the future blocks until the future's value is delivered. This facilitates a very flexible, demand driven style of parallel processing.

Ivory provides a special hardware datatype for futures which is known about by the microcode and the tags processor. This datatype acts as an invisible pointer; if a future has been delivered, the instruction attempting to use its value is not interrupted, but simply follow the future pointer to its actual value. If the value has not been delivered, the hardware causes a trap; the operating system can then suspend the requesting process until the value is delivered.

Without the hardware support provided for the future datatype, the compiler would have to emit code to check the datatype of *every* value manipulated by the program. This is because any value might be a future. Experiments with the QLISP system at Stanford University have shown that this leads to unac-

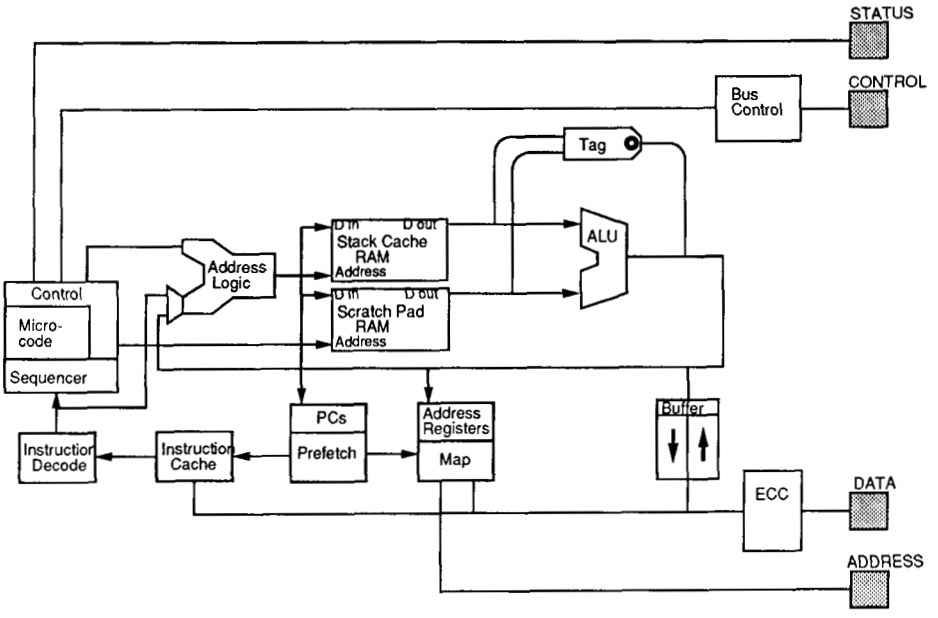

Figure 4: Ivory Processor Block Diagram

ceptable overhead in their implementation on stock parallel processing hardware.

3.2 Special Variable Bindings

Lisp allows two types of variable bindings. Lexical binding of variables is the type familiar in all block structured languages. Dynamic binding (also known as special binding), however, changes the globally accessible value of a location through the dynamic extent of the binding. This change, however, is visible only within the process which bound the variable; all other processes see either the global value or their own dynamically bound value.

Classically, special variable binding is performed using shallow binding. This involves overwriting the location with its newly bound value while saving the old value in a special stack. Shallow binding optimizes the speed of access to the variable. Shallow binding can be (and is) used in sequential machines that support multiple processes. When a process relinquishes control of the processor, its special variable bindings are undone; the process which gains control of the processor must first establish its special variable bindings before beginning its execution. This makes processes switching costly, even for sequential machines.

In the parallel processing world the shallow binding technique doesn't work at all. This is because two distinct processors can be concurrently executing separate processes each of which wants to bind the location to a unique value. Since shallow binding works by overwriting the single location there is no way for two processes both to bind the same location.

This forces the use of the much slower deep binding technique for special variables. In a deep binding scheme, each process maintains an ordered mapping between locations and bound values. The mapping must be ordered since a process can repeatedly rebind the value of a single location and the latest binding should hold. This data structure must be sequentially searched for a variable binding; this is typically a very slow process.

Ivory provides hardware support to optimize deep binding. A special datatype, called *bound location*, is used to indicate that a location has been dynamically bound. Whenever Ivory encounters such a datatype, it traps to a microcode routine that searches a hashtable for the value of the binding. The hashtable uses a key derived from both the identity of the binding process and the address of the location bound. The binding and unbinding instructions keep this hash table up to date. A probe into this table is very fast;

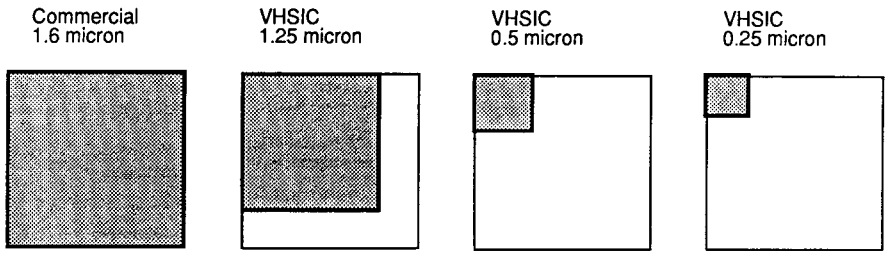

Figure 5: Relative sizes of a single Ivory processor in four different technologies.

if there is no entry in the hash table, then a classic deep binding search is initiated.

This technique has two advantages over the techniques possible without datatype checking hardware. First, for locations which have not been dynamically bound, there is no cost above that of shallow binding, since the special techniques are only invoked for locations whose datatype is *bound location*; these markers are only placed in a location that is dynamically bound. The second advantage comes from the hardware assisted hashing used to fetch the binding. In conventional processors neither of these techniques are available. The lack of the bound location datatype is particularly critical since any location may be dynamically bound and hence any load or store must check for this.

3.3 Locking and Synchronization

Parallel processing in the presence of side-effects requires techniques for establishing critical regions, mutual exclusion from data structures and joint synchronization of processors to rendezvous points. These are very difficult to achieve efficiently without hardware support. Ivory provides a *store conditional* instruction that can serve as the basis for all of these facilities. Store conditional takes three arguments; the first is a location, the second two are the new value and the test value. If the location currently contains a value EQ to the test value, then the new value is stored in the location. The value returned by the instruction can be tested to see if the store succeeded.

Semaphores, atomic updates and locks can all be implemented using this single atomic update primitive.

4 Technology for a Spaceborne Processor Architecture

The ultimate Spaceborne VHSIC multiprocessor will result from a combination of the powerful computer architecture ideas of Ivory with evolving VHSIC hardware technologies. At each stage of this evolution, the overall performance, integration and reliability of the system will be increased. Three hardware technologies are particularly important:

- Fine-line VHSIC chip technology, moving from 1.25 micron, to .5 micron and finally to .25 micron Rad-Hard CMOS.
- Super-chip technology which integrates a significant portion of the total system onto a single large die (2 inch square), using redundancy techniques to achieve adequate yields.
- Button-Board System Packaging which allows very dense packaging of boards into modules without the use of backplanes and connectors.

4.1 Chips and Superchips

The first commercial Ivory chip is implemented in 1.6 micron CMOS technology and runs with a cycle time in the vicinity of 150 ns. At this clock rate Ivory's performance is roughly 3 times that of the Symbolics 3600. A second version of Ivory is now available with cycle times in the vicinity of 100 ns.

Radiation doses as high as 10^5 rads are expected in the space station environment. Since mechanical shielding takes up space and weight, the use of rad-hardened technology is preferable. VHSIC provides a CMOS technology with adequate radiation resistance for space-station applications. In addition, error-correcting logic on the memory bus of Ivory (already provided in the commercial version of Ivory) may be enhanced for increased reliability to single event upset (SEU).

Figure 5 shows how the die size for an Ivory chip will shrink as it is reimplemented in newer VHSIC technologies. Since Ivory is implemented in a technology independent design system, it can be retargeted to these new technologies in a matter of days through a totally mechanical process. These dense processes also allow the cycle time to be reduced; cycle times below 50 ns should be achievable with the .5 micron process.

Figure 6 shows the outline of a super-chip containing Ivory chips implemented in .5 and .25 micron technology. Even with several redundant copies of the

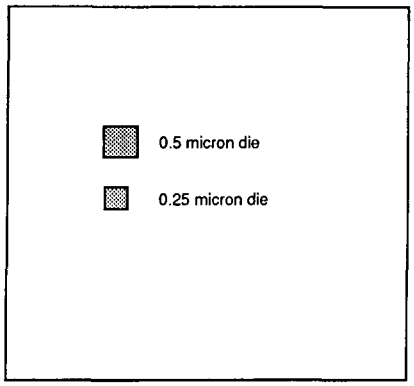

Figure 6: Ivory processors in a superchip.

Ivory die, the superchip contains room for an extensive cache memory. The advantage of this approach is that the cache can be more closely coupled to the processor, avoiding the delays associated with crossing chip boundaries.

Using this technology also allows the chip architecture to evolve further, leading to additional performance improvements. Such processors should be capable of performance in the range of 15 to 20 times that of the Symbolics 3600.

For small to modest scale parallelism, two interconnection technologies are appropriate. For system of 4 to 8 Ivory processors, a shared bus with snooping caches is capable of providing adequate bandwidth and a coherent memory image shared by all processors. However, a single bus system is not attractive when fault-tolerance considerations are added in.

A cross-bar interconnection scheme can support a larger number of processor (up to 32) and provide greater fault tolerance. Figure 7 shows an example of a crossbar interconnect.

4.2 Button Board Interconnect

Buttonboard Packaging is a new technology at TRW which provides performance and density improvements over those possible with conventional packages. Buttonboard packaging replaces bulky edge connectors with "button"-shaped contacts and small PC-board strips with low interconnect density and few layers. Buttons may be placed anywhere on a circuit board to provide an interconnect between boards. This allows a reduction of the path length of inter-board signals, thereby reducing propagation delays.

In the envisaged design, component-carrying boards alternate with signal-routing boards in a stack which is fastened together to make a monolithic whole. Results of prototype testing seem to indicate that a buttonboard-based design will easily meet electrical and mechanical requirements for the space station.

Figure 8 shows how a crossbar system can be implemented using button board packaging. One interesting feature of this packaging technique is that additional processors (or memory) can be added simply by dropping in an additional processor card. A full scale SVMS system should therefore capable of a peak performance of greater than 500 times that of the Symbolics 3600.

References

[Baker *et al.*, 1987] C. Baker, D Chan, J. Cherry, A. Corry, G. Efland, B. Edwards, M. Matson, H. Minsky, E. Nestler, K. Reti, D. Sarrazin, C. Sommer, D. Tan and N. Weste The Symbolics Ivory Processor: A 40 Bit Tagged Architecture Lisp Microprocessor. In *Proceedings ICCD-87.*

[Edwards *et al.*, 1987] B. Edwards, G. Efland and N. Weste The Symbolics I-Machine Architecture: A Symbolic Processor Architecture for VLSI Implementation. In *Proceedings ICCD-87.*

[Gabriel & McCarthy, 1984] R.P. Gabriel and J. McCarthy. Queue-based multiprocessing Lisp. Symposium on Lisp and Functional Programming, August 1984.

[Halstead, 1985] Halstead, R. MultiLisp: A Language for Concurrent Symbolic Computation ACM TOPLAS, October 1985.

[Moon, 1984] D.A. Moon Garbage Collection in a Large Lisp System. 1984 ACM Symposium on Lisp and Functional Programming, August 1984, pp. 235-246.

[Moon, 1985] D.A. Moon Architecture of the Symbolics 3600. Proceedings of the 12th IEEE International Symposium on Computer Architecture, 1985, pp. 76-83.

[Moon, 1986] D.A. Moon Object-Oriented Programming with Flavors. Proceedings of OOPSLA '86, pp. 1-8.

[Shrobe *et al.*, 1987] H.E. Shrobe, J.G. Aspinall and N.L. Mayle. Towards a Virtual Parallel Inference Engine. To appear in *Proceedings AAAI-88.*

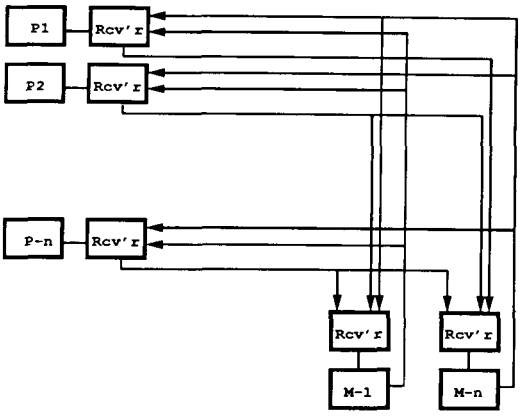

Figure 7: A crossbar interconnect. Processors are labeled **P**, memories **M**.

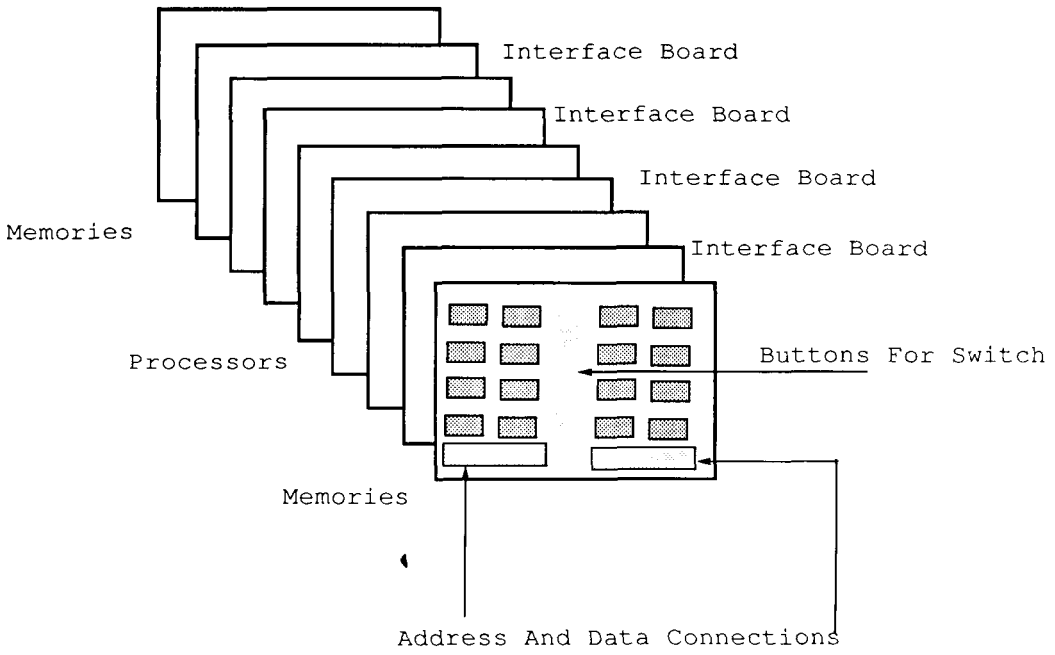

Figure 8: Button-board implementation of a crossbar interconnect.