The Architecture of FAIM-1

Judy M. Anderson, William S. Coates, Alan L. Davis, Robert W. Hon,
Ian N. Robinson, Shane V. Robison, and Kenneth S. Stevens
Schlumberger Palo Alto Research

FAIM-1 can be scaled to provide two to three orders of magnitude improvement over conventional AI machines with minimal inconvenience to programmers.

This article describes a symbolic multiprocessing system called FAIM-1.* FAIM-1 is a highly concurrent, general-purpose, symbolic accelerator for parallel AI symbolic computation. The paramount goal of the FAIM project is to produce an architecture that can be scaled to a configuration capable of performance improvements of two to three orders of magnitude over conventional architectures. In the design of FAIM-1, prime consideration was given to programmability, performance, extensibility, fault tolerance, and the cost-effective use of technology.

Programmability. Although the FAIM-1 machine architecture is unconventional, the software environment provides a concurrent programming language and an application development system that are based on models familiar to members of the AI community. This environment permits immediate evaluation of the FAIM-1 architecture when that architecture is used for existing applications, and it eases the burden on programmers of future applications.

Performance. Effective use of concurrency is the primary mechanism employed by the FAIM-1 system to increase significantly performance over conventional sequential systems. Hardware concurrency is exploited in the operation of

- the individual processing elements,
- subsystems within the processing elements, and
- components within each subsystem.

Software concurrency is exploited in two distinct forms. *Spatial concurrency* involves a set of independent tasks, each working on a partitioned piece of the problem. *Temporal concurrency* involves pipelined execution in which stages can be viewed as concurrent tasks operating on elements of a stream of data at different times.

Extensibility. High priority was given to creating a design permitting arbitrary expansion of the hardware resources. Expansion in FAIM-1 requires minimal rewiring, and no modification to either the user software or system software. The communication topology is planar, and therefore will not become a liability as technology advances to permit evolution from multiple-chip to single-chip processing elements. Wiring complexity scales linearly with processing-element count. All hardware module interfaces are self-timed[1] to permit individual components to evolve independently in performance, implementation technology, and functionality. Self-timed circuit design is a type of circuit design discipline that does not use a global clock to guarantee synchronization. In this style, each component keeps time internally and provides interface handshaking signals to coordinate with its partner subsystems. The result is an architecture that is easy to modify and exhibits a favorable cost/performance ratio under scaling.

Fault tolerance. Any solution to the fault-tolerance problem inherently contains redundancy. The FAIM-1 contains significant redundancy, both in terms of

*Work on the FAIM (Fairchild AI Machine) project was begun at the Fairchild Research Laboratories and has since moved to another research lab within Schlumberger.

0018-9162/87/0100-0055$01.00 © 1987 IEEE

processing elements and in the way these elements are interconnected. FAIM-1 is designed to be fault tolerant at the processing-element level, but not at the gate or circuit level. The resource-allocation mechanism permits the reassignment of tasks to processors, the message-routing algorithm is capable of routing messages around failed paths and processors, and the system software supports self-diagnosis.

Technology. The architecture is designed to take advantage of the cost and performance of both advanced VLSI circuit technology and advanced packaging technology (immersion-cooled wafer hybridization is an example of the latter) as they become available.

The focus of this article is on the physical architecture of the FAIM-1 system. However, to understand some of the design decisions, one must examine the salient aspects of both the software system structure and the programming language that the physical architecture supports. The next section ("Software structure") presents a synopsis of both topics; subsequent sections present the hardware architecture.

Software structure

The architecture of the FAIM-1 serves as a high-speed evaluation engine for the concurrent programming language OIL (Our Intermediate Language), and supports a style of distributed multiprocessing system structure that is embodied in the runtime operating system.

The OIL language and OIL objects. OIL is a high-level, concurrent, symbolic-programming language. The design of OIL was influenced by current AI programming practices, in which a number of AI programming languages are widely used. A distillation of these languages leaves three main linguistic styles: object-oriented programming, logic programming (primarily Prolog), and procedural programming (primarily Lisp). A complex AI application may require several (or all) of these programming styles. Emulating one programming style within another is inefficient, so there is a need for a better linguistic mechanism, one that efficiently incorporates or efficiently supports the essential features of the major styles. OIL has been designed to provide concurrent versions of each of the three linguistic styles.

An OIL program is a collection of objects that communicate by sending messages. The nature of the communication structure explicitly indicates the top level of concurrency represented by the program. Individual objects may themselves be concurrent program fragments. An OIL object consists of some local state information (typically in the form of variables) and several *ports* through which messages are sent and received in FIFO order. A *behavior*, associated with each port, describes what the object does in response to a message. The behavior is a program that may modify the local state and/or send messages to other objects. Atomic OIL objects are of two distinct types: *logical* and *procedural*. Logical behaviors are written in a declarative style similar to a parallel version of Prolog.[2] Procedural behaviors are written imperatively in a lexically scoped dialect of Lisp that is similar to T.[3] Objects can be nested heterogeneously to form other objects that permit control to pass between declarative and imperative behaviors.

An OIL object consists of

- *state*, which is represented as the collection of variables and data structures that are considered local to the object;
- *ports*, that is, the set of entry points that may receive messages;
- *entries* (an entry is a subset of ports); and
- *behaviors*, that is, code that may be either procedural or logical.

Objects can be created dynamically or statically. Upon creation, the state of an object is set to its initialization value. An object takes action when a message arrives. To support distributed procedure calls and parallel process synchronization, subsets of ports may be grouped into *entries*. Behaviors are associated with entries rather than ports. When all of an entry's ports have a message, then the associated behavior is said to be *fireable*. In cases where only a single port's message invokes a behavior, that port is also labelled as an entry. In cases where an object is defined in terms of other objects, the inheritance is static.

An object may consist of an arbitrary number of behaviors and ports, and each behavior is viewed as a potentially independent code fragment. Since the object's state is accessible by any of that object's behaviors, a potential source of nondeterminacy exists. To prevent this situation, the behaviors are viewed as a set of mutually exclusive transactions. When a behavior is started up, all other behaviors are inhibited until that behavior terminates. Furthermore, a behavior may close ports and thereby temporarily inhibit message delivery on those ports until they are subsequently re-opened.

The roles of logic and procedural components are quite distinct. The logic component is used to express in a succinct manner nondeterministic pattern-driven search, while the procedural component is used for sequential algorithm specification, manipulating unique objects, and expressing history-sensitive algorithms. Neither component projects its semantics on the other. Therefore, procedural objects do not have multiple versions of their environments, and logical objects do not have changeable state variables. Communication between logical and procedural objects is based on message streams managed by manipulation of continuations in the sending and receiving objects. The exact semantic significance of continuations is somewhat different on each side, but both use continuations as "handles" to obtain subsequent values in a stream. A logic component accepts an input stream of goals, and produces an output stream of solutions. In general, there are several logical solutions per goal. A procedural component accepts an input stream of function calls, and produces an output stream of response messages, usually with one response per input function call.

The binding of variables observes the semantics of the object in which the variables are defined. Thus, a logical variable has multiple alternative bindings, in keeping with the nondeterministic semantics of logic objects, and a procedural variable has values that are changed by direct assignment. Moreover, components are not allowed to violate the binding policies imposed by the variable's proprietor. Bound logic variables are invisible to procedural accessors; instead, such accesses directly return the variable's value. Unbound logic variables are detected as such in procedural components, but cannot be bound by them. Procedural variables are passed by value to logic components when used in parameters. Hence, procedural variables (as assignable entities) are invisible in logic components.

The programmer may also annotate OIL code with *pragmas*, which describe some of the expected runtime dynamic behavior of the code. These programmer-supplied hints are used by the static resource allocator to partition code and data onto the physical resources of the machine. The pragma information gives the programmer control over some aspects of the allocation strategy. If no pragma information is supplied, the program will still run, although perhaps not as efficiently.

Procedural OIL. The procedural component of OIL is a parallel modification of a lexically scoped dialect of Lisp called T.[3] The primary modifications to T facilitate the use of concurrency; they include a concurrent reformulation of the basic semantics, and the addition of parallel control and data structures, operations on parallel data structures, and specifiable evaluation strategies that permit a variety of parallel evaluation methods. The primitive special forms are exactly as defined by T with two exceptions. The first exception involves *Cond*, which is the normal T conditional. Cond also exists in OIL, but *Cond=* has been added, and is the parallel OIL version. Cond= causes the guards to evaluate in parallel, and the first one that evaluates to True is pursued. Semantically, this implies that an arbitrary evaluation of "True" by the guard forces selection, since the notion of what is first is not controllable by the programmer. The second is to permit special forms for specifying pragmas and type information. These are, respectively, the *Pragma* and *Proclaim* special forms. Pragmas fall into three categories:

- estimates of the probability of taking a particular branch in a decision,
- estimates of the size of dynamic data structures, and
- hints about appropriate allocation decisions.

In all other respects procedural OIL is isomorphic to T.

Logical OIL. Logic behaviors are written in a parallel form of Prolog that is syntactically similar to DEC-10 Prolog. The same notation is used for clauses, lists, and several "evaluable predicates." In general, operations such as arithmetic functions, which do not have an inherent sequential semantics, are the same in both languages. Operations like assert and retract are not supported.

An OIL logical program consists of a set of named objects, each of which contains any number of clauses. The names of the objects serve as names of *worlds*. The logic objects can be nested, giving the effect of additional worlds. The programmer may indicate a goal as *solve*(x, W), meaning that goal x is to be solved in the context of world W. This implies that the rules for solving x are defined in world W. An inner world inherits all the rules of the outer worlds. If the programmer omits the world specification, then a default single-world model is used. The programmer may also indicate private rules that are not inherited by inner worlds. Logical objects differ from procedural objects in that they cannot have internal state variables. Programmers must use procedural code to describe operations that modify an object's local state. Variable bindings invoked under unification can affect local state only after they are passed back to a procedural module, which assigns the value to one of the object's state variables.

Clauses of logic programs are compiled into sets of primitive processes. These primitive processes are objects that use local state variables to represent a portion of the global runtime environment. The processes respond to incoming messages by changing state and generating messages for other processes. The two types of primitive processes are *AND processes* and *OR processes*. AND processes execute the bodies of nonunit clauses, and OR processes manage execution of procedures that are defined by more than one clause. In the compiled logic program, there is at least one AND process for the right hand side of each nonunit clause, and at least one OR process for each *procedure* (set of clauses with heads that have the same functor and arity). An overall view of the computation would show an AND/OR process tree, with AND processes creating OR descendants to solve each literal in a goal statement, and OR processes creating AND descendants to solve bodies of matching clauses.

In AND processes, the basic actions, the internal states, and the reaction to a particular message depend on the desired control model. The programmer may provide mode declarations on logical variables; for example, the literal $p(x?,y!)$ implies that when p is evaluated, it will consume a binding for x and produce one for y. The parallelism in logical OIL is primarily the nondeterministic pattern-directed search mechanism that is obtained through OR parallelism.[4] Limited forms of AND parallelism similar to that proposed by DeGroot[5] are also provided in cases where independent AND processes can be identified at compile time or by a simple runtime ground check. The use of mode declarations significantly enhances the programmer's ability to control the amount of AND parallelism that can be exploited.

Interfacing logical and procedural behaviors. The interface between the two types of behavior components is essentially a form of procedure call. A logical behavior calls a procedural behavior by means of an evaluable predicate that is syntactically identified. The primary difference in the semantics is that the solution of a logical subgoal by a procedural behavior may succeed more than once. In this sense procedural behaviors behave more like normal subgoal solutions than like conventional Prolog built-in predicates. Subsequent calls to the procedural component will result in *next* messages that will retrieve the next stream element.

A procedural behavior may call a logical behavior by sending it a *solve* message containing a goal and a world in which that goal is to be solved. Calling the continuation of the logical behavior will retrieve the next element in its solution stream.

Resource allocation. OIL programs are compiled into object code on a host Lisp Machine. The host then downloads the object code onto the FAIM-1 for execution. Critical decisions about where to load individual objects are made during a phase of compilation called *resource allocation*. The resource allocation process involves balancing the use of the parallel execution hardware with the cost of runtime communication overhead. The resource allocation phase permits the writing of programs, even if the writer lacks a detailed understanding of the hardware, interconnection structure, or communications costs.

In general, there are three basic approaches to resource allocation:

- *Programmer-defined allocation*, which can be implemented either as part of the programming language or by an alternative description, places responsibility for making all resource allocation decisions on the programmer.
- In *dynamic allocation*, the overall processor utilization must be measured at runtime by the system. By means of this analysis, the workload is adjusted during program execution.
- *Static allocation* involves analyzing the source program and partitioning it into a set of allocatable tasks in a way that maximizes processing concurrency while minimizing overhead from interprocessor communication.

The complexity of resource allocation for large programs makes it unlikely that programmer-defined allocation will be a viable long-term solution. Dynamic allocation inherently implies significant levels of runtime overhead. Hence, the primary focus for FAIM-1 is on static methods. The OIL programmer can influence the static allocator by special annotations (see the discussion of pragmas, above), and some simple dynamic load balancing can be performed when runtime conditions indicate that it is necessary.

Hardware structure

The FAIM-1 architecture consists of a number of independent processing elements, called *Hectogons*, interconnected in a hexagonal mesh. The following sections describe the overall structure of the machine and provide insight into some of the design decisions made in creating it.

Communication topology. Many possible multiprocessor interconnection schemes are currently being investigated. See, for example, the current research efforts on the Cosmic Cube,[6] Butterfly,[7] and DADO.[8] Desirable topology characteristics include high performance, reduced wiring complexity, flexibility, fault tolerance, and simplicity of implementation. Furthermore, a goal is that these properties remain attractive under scaling.

The topology used in the FAIM-1 is a hexagonal mesh. Processing elements communicate directly with six neighbors; the processing elements themselves are organized into hexagonal *surfaces* that are combined in a similar six-neighbor fashion.

When wires leave a processing surface through the processing elements at the periphery, they are folded back onto the surface in a three-axis variant of a twisted torus. In Figure 1, the basic topology is illustrated, along with the wrap lines and switches that complete the interconnect structure. For purposes of illustration, only a single, wrapped axis is shown; in the complete topology, all edge ports are connected, requiring two additional sets of wraps like the one shown in the diagram.

This particular wrapping scheme results in a simple routing algorithm and provides a minimal switching diameter for a hexagonal mesh. All PEs are viewed as if they were the center PE of the surface, and routing decisions are based on three-axis relative coordinates. This simple algorithm is implemented in custom hardware for performance reasons.

Each peripheral port communicates with an off-surface device, as well as being wrapped back to the opposite edge of the surface. These off-surface connections permit communications with I/O devices and with other surfaces. The external connections are made by introducing a simple three-way switch, which is shown in Figure 1. Communication with other processors on the surface is via the three-way switch, which routes signals back to the other edge of the surface. The switches have three ports:

- internal (for local surface messages),
- external (for adjacent surfaces, I/O devices, or the host), and
- wrap (for the local surface via the wrap line).

Switching decisions are based on which of the three ports a message arrives on and the destination contained in the message header.

The size of a surface is defined by the number of processors n on each edge of the hexagonal surface. The surface is referred to as an *E-n* surface; the number of processors in a surface scales as $3n(n-1) + 1$. For example, an *E-3* processor surface has three processors on each of the six edges and contains a total of 19 processors. For surface sizes between *E-1* and *E-7* inclusive, the number of PEs is a prime number, which is advantageous from the standpoint of fault tolerance and initialization.

A hierarchical instance of a FAIM-1 processor is built by tessellating multiple hexagonal surfaces. Locality among groups of processors is increased and the communication diameter of the system is decreased as compared with a single-surface instance that has the same number of processors. These properties can be demonstrated by the following example. An *E-7* surface contains 127 processors and has a diameter of six, while seven *E-3* surfaces can be tiled to form an *S-2 E-3* machine (shown in Figure 2) that contains 133 processors with a diameter of five. The switching diameter improves dramatically as the processor count is increased. A 58,381-processor *E-140* has a diameter of 139, while a 58,807-processor *S-9 E-10* FAIM instance results in a diameter of 89, a full 50 hops better (worst case) than the single *E-140* surface.

The probability of component failure in a system statistically increases as more components are added to the system,[9] making fault tolerance an important aspect of highly replicated architectures. Fortuitously, distributed ensemble architectures intrinsically contain redundant elements that can be used to support fault-tolerant behavior. Koren[10] has shown that hexagonal meshes are particularly attractive fault-tolerant topologies. In addition, fault-tolerant message routing is possible because of the multiplicity of paths over which a message may be routed to its destination.

In the FAIM-1 machine, all of the topology-dependent hardware is contained in a single subsystem called the *Post Office*, which is described in detail in a later section. The remainder of the machine is independent of connection topology, and could be utilized in other connection schemes.

The Hectogon. The processors located at each node in the communication topology are called *Hectogons*. A Hectogon can be viewed as the homogeneously replicated processing element of the FAIM-1 architecture on one hand, and as a medium-grain, highly concurrent, heterogeneous, shared-memory multiprocessor on the other. Internally, each Hectogon is constructed of several subsystems or coprocessors, all of which may be active concurrently.

This double view is the result of the consistent exploitation of concurrency at all levels of the FAIM-1 system, and is motivated by our belief that the scalability of a multiprocessor architecture is of prime importance. Performance of an architecture as it is scaled up is critically affected by four factors:

- the performance of each individual processor,
- the average percentage of processors that are active,
- the efficiency of interprocessor communication, and
- the total number of processors in the aggregate machine.

The intent of the FAIM-1 project is to pursue aggressively each of the four aspects by designing a powerful processing element that permits high levels of replication.

Individual coprocessors directly support logic programming, parallel Lisp, and complex runtime system duties, such as task switching and scheduling. Within each coprocessor, other linguistic features are supported by specific aspects of the architecture. For example, the evaluation processor contains parallel tag hardware to support the polymorphic function-calling nature of Lisp. Rather than allocating a single task to each processing element and risking a low percentage of active processors, FAIM-1 allocates a number of parallel tasks to each processing element. Tasks can be complex program fragments requiring a wide range of computational support. This strategy improves processing element utility by increasing the probability that a runnable task will exist at any given time. Interprocess communication is facilitated by including a high-performance message-handling coprocessor (the Post Office). A large number of Hectogons can be tiled together (as described in the section on "Communication topology," above) to form a fully distributed (that is, with no shared memory or control) multiprocessor system.

The Hectogon's subsystems are connected by an asynchronous System Bus (SBus), and by custom interfaces in some cases. Subsystems communicate with each other by means of a flexible, speed-independent signalling protocol. The self-timed behavior of the subsystems allows them to be independently tuned in terms of both performance and function without impacting the designs of the other subsystems. While each of the six subsystems is a reasonably general system-level component for distributed ensemble architectures, the particular instantiation of each has been tailored with a specific view of the Hectogon in mind. A Hectogon's subsystems and their interconnection are shown in Figure 3.

The six subsystems connected by the SBus are

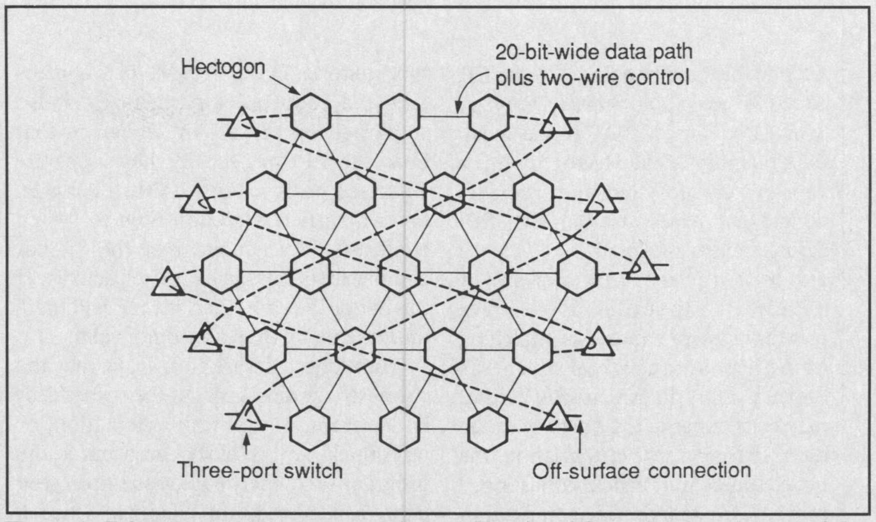


Figure 1. *E-3* surface with three-way switches. (Reprinted from the *Proceedings of the Sixth International Conference on Distributed Computing Systems* © IEEE.)

- *Evaluation processor* (*EP*). A streamlined, non-microcoded, stack-based processor responsible for evaluating machine instructions.
- *Switching processor* (*SP*). A small context-switching processor that is responsible for interpreting the runlist in data memory (the Scratch RAM) and moving blocked contexts out of processor registers and into SRAM process-control blocks, and then loading a new, runnable context into the processor registers.
- *Instruction stream memory* (*ISM*). A specialized instruction memory that not only stores the instructions, but is also responsible for finding the appropriate instruction, partially

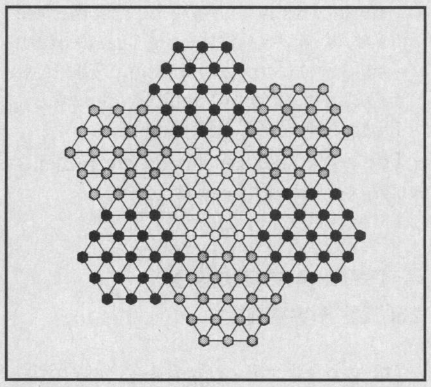


Figure 2. *S-2* tessellation of *E-3* surfaces. (Source: *Proceedings of the 1985 International Joint Conference on Artificial Intelligence*, 1985. Used by the courtesy of Morgan Kaufman Publishers.)

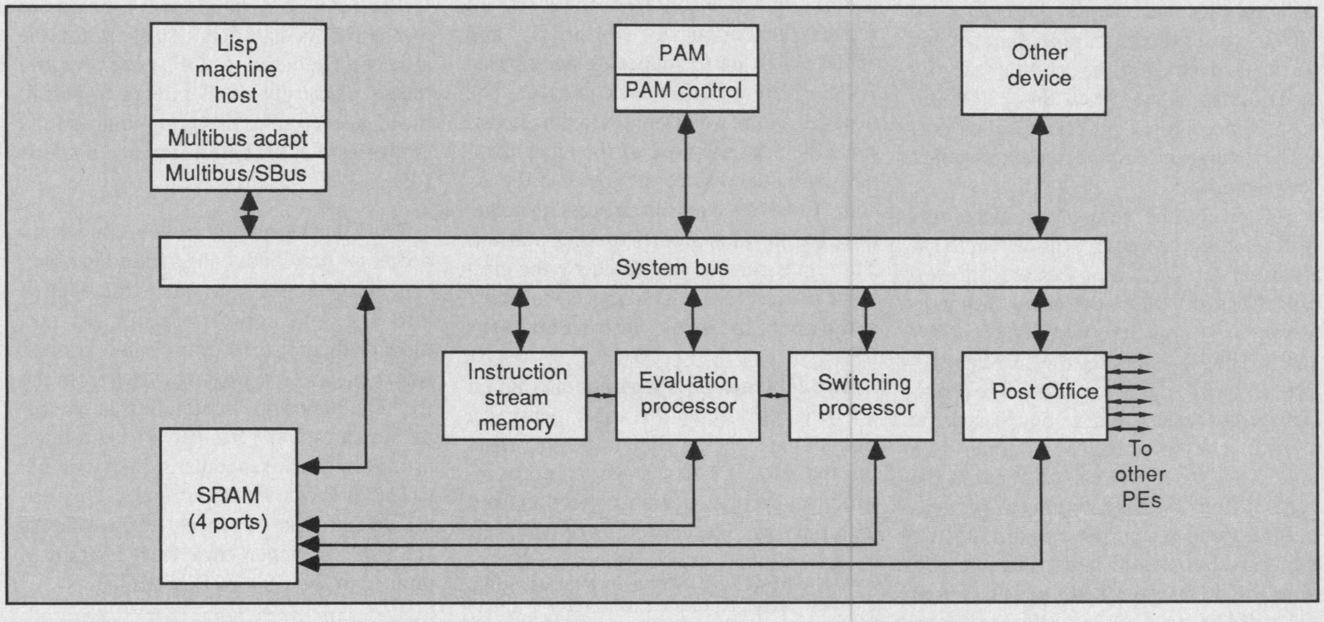


Figure 3. Block diagram of a Hectogon.

decoding it, and handing it to the EP.

- *Scratch random access memory (SRAM)*. The SRAM is the local-data memory of the Hectogon. It is a four-ported subsystem that provides concurrent access to the EP, SP, SBus, and Post Office.
- *Pattern-addressable memory (PAM)*. A parallel associative memory system capable of matching complex structures, such as S-expressions, in which *don't cares* may exist either in the query or in the data. In the present instantiation, the match function is "mock unification," that is, the extent to which unification can be done without dereferencing bound logical variable pointers.
- *Post Office.* An autonomous communications subsystem responsible for all aspects of the physical delivery of inter-Hectogon messages. The Post Office is the only topology-dependent coprocessor in a Hectogon.

The remainder of this article presents each coprocessor in some detail.

A two-processor partnership

The processor is designed to be a high-speed evaluator of OIL programs. As such, it contains specific features that provide efficient support mechanisms for object-oriented, procedural, and logic programming. Processor utilization on the FAIM-1 is improved if a small number of independent, and therefore concurrent, tasks are allocated onto a single Hectogon. This increases the probability that for any particular Hectogon, a runnable task will be available at any given time. This imposes a need for a processor to support multitasking and rapid context switches between tasks.

To achieve the goal of rapid context switching, the processor is implemented as a partnership of two processors; the two processors are called the *evaluation processor (EP)* and the *switching processor (SP)*. The EP evaluates the currently active task, while the SP sets up the next runnable task. Tasks run in a context-specific register set; the processor contains two context-register sets. Each set contains the stack buffers, processor status word, general-purpose registers, and so on. One set is marked *active* and is used by the EP, while the other is marked *idle* and is used by the SP to save the old context and load the next context from process-control blocks stored in the SRAM. This concurrent and cooperative partnership is also supported by the SRAM design in that both the EP and the SP have separate dedicated ports to the SRAM. It typically takes a single instruction time to switch contexts. This assumes that the SP has completed its next-task prefetch activity. If this is not the case, then the EP will halt* until the new context becomes valid.

Normally there are four ways that the currently running task can be suspended: an interrupt, a trap, task termination, or task block. A task block can occur at the program's request, or when an unforeseen delay is incurred, for example, when a message sent to a remote processor requires a reply before the sending task can continue.

In FAIM-1, the same context sets and switching mechanism are used for all four cases. There is no special supervisor or kernel context. In both task termination and task blocking, the next reasonable action is to run another task, as nothing more can be done on the evaluation of the current task. Since traps usually require information from the current context (for example, the arguments to the trapped instruction require modification by the service routine), trap service routines are run in the current context in a manner similar to that of a simple function call. Interrupt-driven tasks inherently start and terminate cleanly, and therefore can also be evaluated in the current context.

The EP. The EP executes instructions that it receives from the ISM by making use of the active-context register set. To reduce the complexity of the EP and achieve high performance, a stack-based RISC[11] organization was chosen. The need for a long word length does not exist, since the address space of the local memory is small and since the focus of the architecture is on symbolic processing rather than large-value numeric computation. The EP is further simplified by the existence of the ISM, which selects, formats, and delivers the proper instruction to the EP.

The existence of a separate instruction delivery subsystem (the ISM) permits a simple two-stage pipeline to be employed for the EP. The first stage supports instruction decode, operand selection, and modification. The second stage performs the ALU operation, modifies the result if necessary, and stores the result. The ISM runs concurrently with the EP and effectively contains another two pipeline stages that perform effective-address translation, instruction fetch, and partial instruction decode.

*Actually, the EP is not explicitly aware that the SP is not ready. The self-timed interface between the EP and SP will automatically and transparently inhibit the EP from evaluating an invalid context.

The EP datapath is 28 bits wide. Eight bits are reserved for tags that are processed in parallel by separate tag-manipulation hardware. The tags are analyzed concurrently with the processing of the 20-bit data field, and may generate a trap when the operand tags are inappropriate for a particular operation.

The EP also contains a full 20-bit wide ALU that supports logical operators, integer arithmetic, and shifting of instructions.

The EP instructions combine the small instruction size of stack-organized processors and the streamlined style of the RISC methodology. This is achieved by trading register-based arithmetic and logical operations for stack-based operations. In addition, the instruction set supports a simple load-and-store model, which may also be indexed from any of the general-purpose index registers.

The instruction set is tuned so that most instructions run in a single cycle. The only instructions that consume additional cycles explicitly are instructions that generate memory references, and interface instructions to the other subsystems. However, any instruction may trap and thus effectively consume additional cycles. In this case, the instruction is completed whenever the target trap routine returns.

The result is a simple RISC-processor that provides significant support for the efficient evaluation of OIL programs and that is physically small enough to permit the high levels of replication required for a cost-effective, high-performance FAIM-1 PE.

The SP. The context switcher is a small processor that loads values found in a process control block stored in the SRAM into the idle context-register set and unloads them, too; it thus permits context switching without stealing cycles from the EP. The main data structure that the SP manipulates is the *run-list*. The run-list is produced by the scheduler, which is an EP task that is run when necessary. The run-list is a linked list of process-control blocks (PCBs)** in which each PCB contains a pointer to the next PCB in the list.

**The actual implementation is more complex than this, but additional detail is omitted here for clarity.

In blocking, a process calls a system routine that requests that the SP perform a context swap and save the process's state. (Depending on why the process is blocking, it may be moved to the blocked-list or may remain on the run-list.) The SP immediately switches to the other set of context registers, and the EP can execute instructions from the new context. The SP then moves the contents of the now idle context register set to the appropriate task PCB in SRAM, examines the run-list to find the next runnable task, moves its context from the PCB to the idle context register set, and sets the proper context validity flag. The SP will halt at this point and remain halted until another context switch is signalled.

Arriving message traffic from other Hectogons may cause idle processes to be rescheduled as runnable tasks. The message handler and scheduler support this by linking arriving messages with their target tasks and moving them to the run-list. The scheduler can therefore be invoked by a "message-delivered" interrupt from the Post Office or host machine.

The instruction stream memory

The instruction stream memory (ISM) is one of the specialized "smart memories" that are employed in the FAIM-1 processing element, and is designed to deliver instructions to the processor at high speed. To accomplish this task efficiently, the ISM performs several functions that are usually performed by the processor in a conventional system. These include calculation of instruction addresses, processing of branches and subroutine calls, and handling of traps and interrupts.

Implementing a separate instruction delivery module can have several advantages, as is demonstrated in the Dorado architecture.[12] Since only instructions are stored in the ISM, it can be optimized for its sole function of instruction delivery. Conventional solutions place a specialized intermediate piece of hardware between the processor and main memory, such as an instruction cache or translation lookaside buffer. This implies more complicated control and requires that relatively slow off-chip signals be driven twice. The ISM organization capitalizes on the coplacement of both the instruction memory and the instruction-delivery-control logic on the same VLSI device. Since on-chip bandwidth is usually an order of magnitude faster than off-chip bandwidth, a significant level of manipulation can be performed concurrently with processing activity in the EP. In addition, very wide bus widths are practical on-chip but impractical at chip boundaries, thus increasing the effective on-chip bandwidth.

In addition to the advantages of more efficient access to instructions, the design of the EP is simplified. The instruction-fetch stage of the pipeline has effectively been transferred to the ISM, along with the program counter. The resulting shorter pipeline in the EP increases the throughput of the processor and simplifies exception-handling duties.

The ISM capitalizes on the fact that instructions are not randomly accessed. In most programs, code is naturally partitioned into small sequences of linearly ordered instructions that terminate in branches or subroutine calls. These sequences are formalized in the FAIM-1 machine and are called *tracks*. As one intuits, instruction tracks vary in length; however, the ISM's instruction storage maps them onto a set of fixed-length physical tracks. Linkage information corresponding to control instructions is associated with the physical tracks in a header field.

Control point information is maintained in a *current track-address register* (*CTR*), which is similar in function to the program counter of a conventional system. An individual instruction address consists of a track number and a track offset.

Branch processing. Execution of branch-type instructions is quite complex in most large machines, owing to the large number of instructions that can cause a branch and the wide variety of addressing modes allowed. In keeping with the streamlined RISC theme, the FAIM-1 uses only one jump format, which is flexible enough to support conditional and unconditional branches, calls, and returns. Branch-addressing modes that require multiple accesses to memory (for example, indirect addressing) or branches requiring additional arithmetic operations (for example, indexed or decrement-and-skip-if-zero) are not implemented. Instruction decoding becomes simple both for the EP and ISM. The ISM examines each instruction as the instruction is prepared for delivery to the EP. When a jump is detected, the ISM autonomously and concurrently processes that instruction, continuing delivery with the first instruction from the target stream.

Conditional branches are a canonical problem, and delay the delivery of source code in pipelined program execution. They are usually the main bottleneck in lookahead and prefetch strategies. In conventional systems, the problem is that by the time the branch is executed and the correct path is resolved, some succeeding instructions have already entered the pipeline. If the branch is taken, the current contents of the pipeline must be discarded, incurring a delay while the pipe is filled again. The standard method of keeping the pipeline full is to use a delayed branch instruction, as is done in the MIPS architecture.[11]

In FAIM-1, the ISM decodes jump instructions before they even enter the pipeline, so a pipeline flush is not necessary and the delayed branch strategy is not applicable. This is because the required continuation of the instruction stream is always correctly delivered. There is, however, an analogous dependence problem in that the outcome of a conditional jump may depend on the result of a previous instruction that has not yet finished executing. To ensure that the branch condition has become valid by the time the ISM detects the jump, an *advance condition code set* method is used.

Subroutine calls and returns. A subroutine, or function call, in the OIL language compiles to a simple jump instruction and is processed by the ISM in almost exactly the same manner as a normal jump. The jump instruction format contains a "save" bit that is set for a "calling" jump and indicates that the current contents of the "program counter" (the CTR) should be saved in a special register called the *jump track-address register* (*JTR*) before the jump is executed. A return from subroutine consists of a Jump instruction specifying the current contents of the JTR as the target address. If the called subroutine needs to call other subroutines, then it must explicitly save and restore the JTR by making use of the control stack.

Exception handling. Traps and interrupts are additional factors that cause a break in the instruction stream. As such, they have an impact on the operation of the ISM. Traps correspond to error conditions arising during the course of executing an instruction, while interrupts are generated by some external device and are completely asynchronous with the program being run. In both cases, the current code stream must be broken to permit the execution of the trap or interrupt-service routine. This is accomplished in a manner similar to that of a subroutine call.

ISM utility. In a conventional processing system, there is a large amount of undesirable traffic between the processor and main memory. Not only useful data, but information on where to find the data, and even information needed to compute this information, must be sent to and from memory.

In FAIM-1, one major class of memory references is associated with the access of the machine code instruction stream. Studies indicate that instruction fetches can account for 50 percent of all memory references on computers such as a VAX or IBM S/370. The single-chip organization of the ISM greatly reduces the severity of this bottleneck. By reflecting the structure of instruction tracks in the structure of the memory and providing integrated logic to perform in an autonomous manner common functions, such as branching, prefetching and subroutine calls, the design of the processor is simplified and new levels of performance are made possible.

The proposed design is simple enough to allow single-chip implementation; multiple ISM chips can be cascaded to provide a maximum-size instruction bank of one million instructions. The EP organization permits an arbitrary number of banks to be used in a single PE; however, the first prototype will contain only a single maximum-size bank per Hectogon.

The Scratch RAM

The SRAM is a four-ported, random-access memory that is the local-data memory system of the Hectogon. An SRAM contains 1 Meg of 28-bit data words. Each data word is further divided into eight tag bits and 20 data bits. The SRAM allows access to memory through any of its four ports, which may all be active concurrently. The ports are connected to the SBus, the evaluation processor, the switching processor, and the Post Office.

The Post Office

Operational responsibilities of the Post Office. The Post Office[13] is an autonomous coprocessor of messages that is capable of delivering messages concurrently with program execution in the processor. The FAIM-1 Post Office contains seven data ports: six to Post Office communication processors in topologically adjacent Hectogons, plus an internal port to the SRAM of the local Hectogon. There is also a system bus link to control communication with the EP and other devices in the Hectogon. All message-delivery control is handled autonomously by the Post Office, freeing the local Hectogon to process tasks rather than stealing EP cycles to support inter-Hectogon message traffic.

The Post Office is responsible for the physical delivery of messages across the communication topology. Communication is initialized by the EP, which generates a message header for and pointer to the variable-length message body and places them in the SRAM. The Post Office extracts the message from the SRAM and delivers it by means of *virtual cut-through*[14] over the communication network. The receiving Post Office places the message in the destination Hectogon's SRAM and notifies its EP that a new message has arrived. Since messages may vary in length, the Post Office may break larger messages up into a series of fixed-length packets that are physically transmitted across the topology. These packets are delivered to their intended destination individually.

The normal mode of delivery is to route a message to its destination by means of a hardware routing algorithm. In richly connected topologies, such as the hexagonal FAIM-1 topology, it is possible for a packet to take multiple paths to a nonlocal destination. The routing algorithm calculates the shortest paths and secondary paths to the destination. General communication efficiency can be significantly enhanced with a routing mechanism capable of detecting congestion and of routing packets around such congested areas in the communications network. All routes for packets in the Post Office are chosen dynamically at delivery time by prioritizing the paths and selecting a path according to the state of the buffers and the network. This mechanism can also be extended to introduce fault tolerance, since messages can be routed around nonfunctional nodes and ports.

The capability of dynamically routing packets around congestion and failed components implies that the order of packet arrival may vary from the order in which the packets were sent. The capability of reordering packets at their final destination is essential to ensure both deterministic behavior and to permit proper reassembly of multiple packet messages. Although this increases the amount of work required at the receiving Post Office, it reduces congestion and failure-related message delays.

From the EP's point of view, messages are assumed to be delivered error-free. This assumption places the burden on the Post Office to verify checksums for each packet, organize retransmission when an error is detected, and synchronize packet-copy destruction in a way that ensures that at least one correct copy of the packet exists in some Hectogon. Positive and negative acknowledgment messages can be automatically echoed back to the sender from the destination Post Office upon receipt of a message. To avoid deadlock, the Post Office prevents unrestricted incremental resource claiming, thereby ensuring that each Post Office will not be congested indefinitely.

Post Office components. The Post Office is constructed from three types of basic components—*port controllers, buffer controllers,* and a *buffer pool*—as shown in Figure 4.

Most of the functions of the Post Office are performed by independent port controllers to enhance concurrency. The *packet ports* transmit data to an adjacent Hectogon across the topology, while the *PE port* stores and retrieves data from the local SRAM. The buffer pool acts as a temporary storage site and queue for messages that have arrived at the destination or require intermediate buffering between the source and destination nodes.

All seven ports share the same internal buffer pool. The function of the buffer controllers is to arbitrate and control access to this limited shared resource. The controllers are responsible for freeing space in the buffers by matching buffered packets with free ports on the path to the packets' respective destinations. The dynamic behavior of packet delivery is further enhanced by the buffer controllers, which access the buffers in a fair but unordered fashion.

Performance. Each of the seven ports of a Post Office node can potentially be transmitting data simultaneously. A pessimistic estimate of 20 MHz cycle time per port yields a burst communication bandwidth of 2¼ gigabits per second per processor-Post Office pair. An *E-3* FAIM instance with 19 Hectogons will have a total peak communication bandwidth of approximately 25 gigabits per second. The

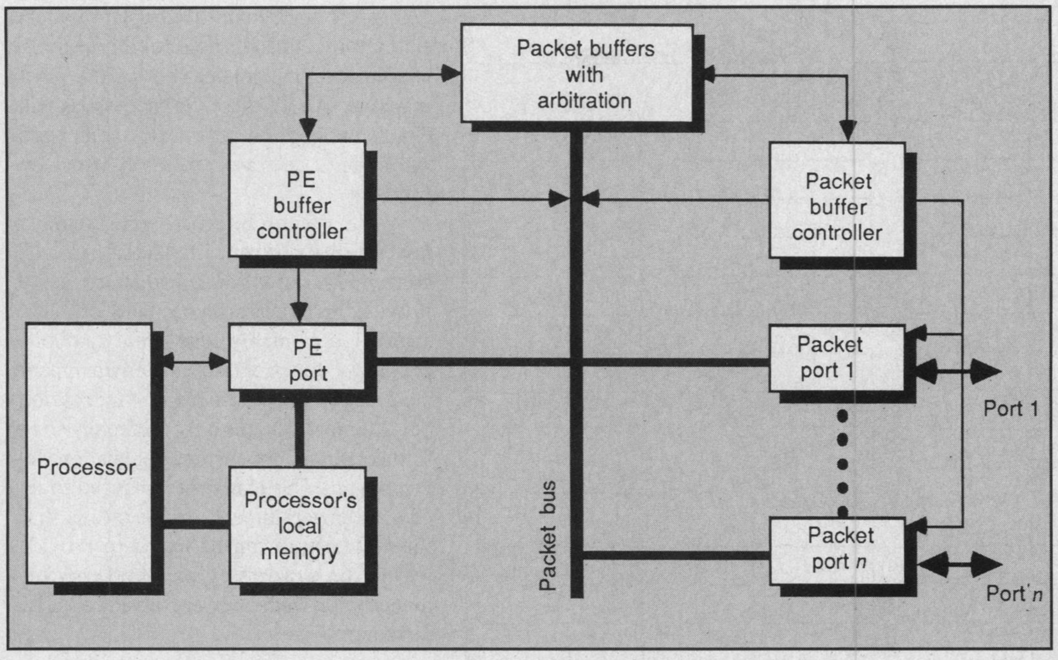


Figure 4. Block diagram of the Post Office.

speed-independent nature of the communication links automatically avoids synchronization and timing problems when peak demands are greater than buffering capacity or are owing to resource conflict. Such a high bandwidth is desirable for experimenting with machine scalability and supporting large processor counts.

In general, it is our belief that direct support for communications will be required in MIMD multiprocessors with many processing sites. This requires a significant reallocation of the transistor budget to achieve the proper balance between processing and communication performance. The Post Office is an experiment that couples a scalable topology and architecture with a communication coprocessor that is completely responsible for the physical delivery of messages. By operating concurrently with program execution and allowing all ports to transfer data simultaneously, the performance of the Post Office should be sufficient even for large-scale networks of processors.

The PAM

Pattern matching and logic programming. One of the most common functions used in AI programming is pattern matching (that is, matching on the LHS of rules, database searches, and so on). We feel that special pattern-matching support is necessary to achieve high-performance symbolic processing. As elsewhere in the Hectogon, this support takes the form of a specialized memory component, here called the *pattern-addressable memory* (or *PAM*). The PAM is designed to store and match on S-expression structures of symbols and words, in much the same way that a traditional content-addressable memory (CAM) deals with single words. Like a CAM, the PAM does its matching in parallel over the whole of its stored contents.

The logic-programming part of OIL is based on Prolog, in which the execution mechanism, unification, is a special form of pattern matching. The PAM can therefore be used as a high-performance subsystem to support the evaluation of logical OIL behaviors. In a wider sense, the declarative semantics of Prolog provides a way of indicating data to be pattern matched and a means to control the application of that pattern matching.

PAM operation. The PAM consists of a number of PAM chips and the PAM controller, the latter providing the interface with the rest of the Hectogon. The PAM is used to store the heads of logic-program clauses as a linear array of symbols. Each clause head ends in a pointer to the compiled code area for that clause. When a logic program is evaluated by the EP, it sets up the goal to be solved in the SRAM; makes a call to the PAM controller; and then blocks, awaiting the results. The PAM controller, accessing SRAM over the SBus, fetches the goal and enters it into the PAM chips. The goal is entered symbol by symbol and the match computed for all the clause heads stored "on the fly." The PAM contains circuitry to detect the absence of any matches as soon as that condition occurs. In that case, matching stops and control of the process returns to the EP.

Otherwise, the controller puts a list of the code pointers from the matching expressions in the SRAM, and reports to the EP that it has completed the task. The EP then uses the code indicated to complete the unification with the clause head. If successful, it then processes the clause body.

The most obvious function of the PAM is, therefore, as a clause-indexing device. Pattern matching is, however, more than just a useful means of clause selection; it is also a significant part of the unification algorithm (*unification = pattern matching + the logical variable*[15]). The matching that takes place in the PAM amounts to full unification for all those terms not involving variables. Recording and checking variable bindings is, however, outside of the PAM's scope. Such "bookkeeping" is therefore left to the EP, which acts as a unification post-processor.

The PAM memory chip. Each PAM chip consists of a number of blocks, each of which in turn consists of storage and logic parts. The storage part contains the symbol name and special tag and status bits used in the matching operation. In the current implementation, 10 such words

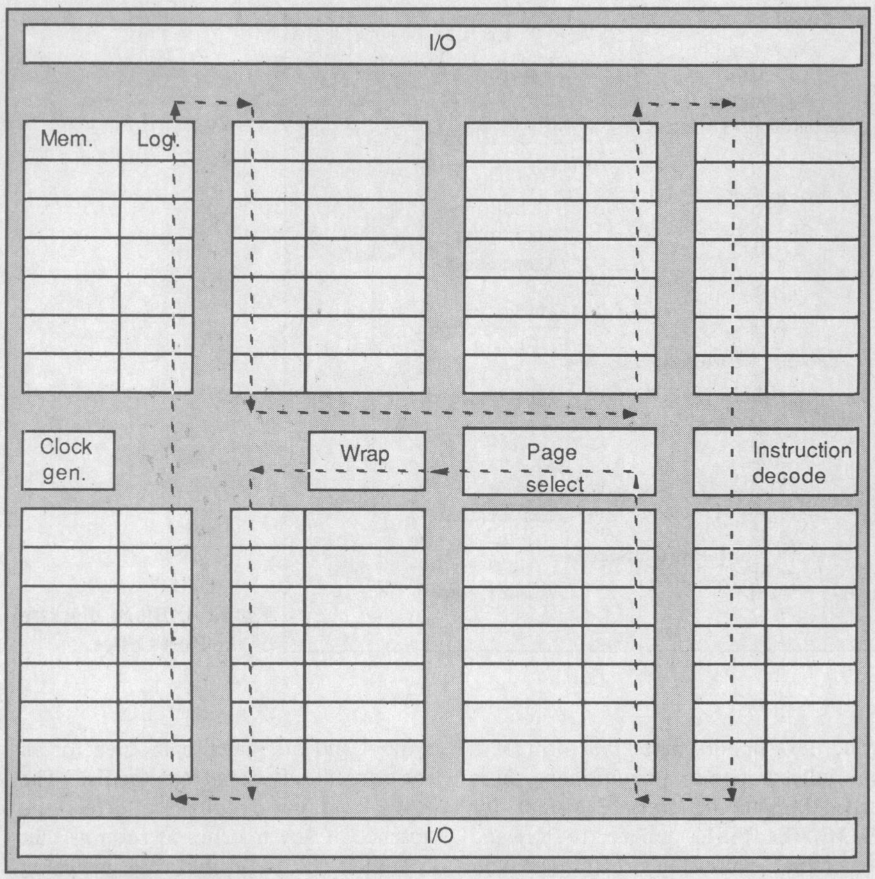


Figure 5. PAM floor plan.

are stored for every "match" section of logic. In effect, this logic is multiplexed over the storage so as to achieve a workable memory density.

The logic part consists of comparators for the name fields and small finite-state machines operating on the tags to perform the matching function. Fifty-six of these blocks are included on each PAM chip, as shown in Figure 5.

The chip contains 560 words of 16-bit symbols. If an average clause head has a size of seven PAM words, one chip can store 80 such expressions. The PAM chip is designed so that the controller can run a number of them together and in parallel without the need for "glue" logic. In this way, the total capacity of the PAM is only limited by the number of chips used. Owing to its parallel operation, the speed of the matching function is independent of the PAM's size.

We have presented an overview of the physical architecture of a novel, high-performance, symbolic multiprocessing system known as FAIM-1. A small-scale prototype of FAIM-1 that consists of a single *E-3* surface (19 PEs) is under construction. The primary performance mechanism is the exploitation of concurrency at all levels of the system. From the top, the FAIM-1 appears as a medium-grain, homogeneous multiprocessor. The topology scales to permit an arbitrary number of processing elements to be interconnected. Each processing element can be viewed from the bottom as a heterogeneous, shared-memory multiprocessor containing servers that are specialized to perform message delivery, mock-unification, data storage and delivery, instruction storage and delivery, instruction evaluation, and rapid context switching.

The architecture takes advantage of advanced circuit and packaging technology as a secondary performance enhancement mechanism. The present small-scale prototype is being implemented with custom CMOS VLSI circuits (the Post Office, PAM, and ISM subsystems) and commercially available components (the processor and SRAM). The 19-PE initial prototype will be attached to a host Symbolics Lisp Machine. The programming environment, OIL compiler, resource allocator, debugging tools, and a complete system simulator that can be used for initial application development all run on the Host machine. All of these characteristics will enable the FAIM-1 prototype to serve as a symbolic acceleratoı to the host Lisp Machine.

While the architecture represents a novel and significant reallocation of the conventional transistor budget so as to provide high performance and efficient evaluation of highly concurrent symbolic programs, the programming environment departs from conventional AI program development systems only minimally so as to incorporate concurrent application development. This eliminates the need to reeducate programmers, and it means that the architecture can be scaled to provide two to three orders of magnitude performance improvement over conventional AI machines. □

References

1. C. L. Seitz, *Introduction to VLSI Systems*, "System Timing" (Chapter 7), McGraw-Hill, New York, 1979.
2. W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, Springer-Verlag, New York, 1981.
3. J. A. Rees, N. I. Adams, and J. R. Meehan, "The T Manual," technical report, 4th ed., Yale University, Computer Science Dept., New Haven, Conn., 1984.
4. J. S. Conery and D. F. Kibler, "Parallel Interpretation of Logic Programs," in *Proc. Conf. Functional Programming Languages and Computer Architectures*, ACM, Oct. 1981, pp. 163-170.
5. D. DeGroot, "Restricted AND-Parallelism," *Proc. Int'l Conf. on Fifth-Generation Computer Systems*, Nov. 1984, pp. 471-478.
6. C. L. Seitz, "The Cosmic Cube," *Comm. ACM*, Vol. 28, No. 1, Jan. 1984, pp. 22-33.
7. R. Gurwitz, "The Butterfly Multiprocessor," talk presented at the 1984 ACM National Convention, San Francisco, Oct. 1984; also, "Butterfly Parallel Processor Overview," BBN Laboratories, Inc., 10 Moulton St., Cambridge, MA 02238, Dec. 18, 1985.
8. S. J. Stolfo, D. Miranker, and D. E. Shaw, "Architecture and Applications of DADO, A Large-Scale Parallel Computer for Artificial Intelligence," *Proc. Eighth Int'l Joint Conf. Artificial Intelligence*, Karlsruhe, West Germany, Aug. 1983, pp. 850-854.
9. D. P. Siewiorek and R. S. Swarz, *The Theory and Practice of Reliable System Design*, Digital Press, Bedford, Mass., 1982.
10. D. Gordon, I. Koren, and G. M. Silberman, "Fault-Tolerance in VLSI Hexagonal Arrays," technical report, Dept. of Electrical Engineering, Computer Science Technion, Haifa 32000, Israel, 1984.
11. John L. Hennessy, "VLSI Processor Architecture," *IEEE Trans. Computers*, Vol. C-33, No. 12, Dec. 1984, pp. 1221-1246.
12. B. W. Lampson, G. McDaniel, and S. M. Ornstein, "An Instruction Fetch Unit for a High-Performance Personal Computer," *IEEE Trans. Computers,* Vol. C-33, No. 8, Aug. 1984, pp. 712-730.

13. K. S. Stevens, S. V. Robison, and A. L. Davis, "The Post Office—Communication Support for Distributed Ensemble Architectures," *Proc. Sixth Int'l Conf. Distributed Computing Systems,* May 1986, pp. 160-167.
14. P. Kermani and L. Kleinrock, "Virtual Cut-Through: A New Computer Communication Switching Technique," *Computer Networks,* Vol. 3, 1979, pp. 267-286.
15. D. Warren, "Implementing Prolog," Technical Report 39, Edinburgh University, May 1977.

Selected bibliography

Davis, A. L., and S. V Robison, "The Architecture of the FAIM-1 Symbolic Multiprocessing System,"*Proc. Ninth Int'l Joint Conf. Artificial Intelligence,* Aug. 1985, pp. 32-38.

Goldberg, A., and D. Robson, *Smalltalk-80: The Language and its Implementation,* Addison-Wesley, Reading, Mass., May 1983.

Kluge, W. E., and K. Lautenbauch, "The Orderly Resolution of Memory Access Conflicts Among Competing Channel Processes," *IEEE Trans. Computers,* Vol C-31, No. 3, Mar. 1982, pp. 194-207.

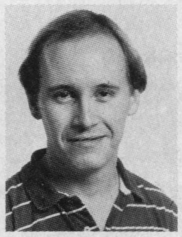
Lee, J. K. F., and A. J. Smith, "Branch Prediction Strategies and Branch Target Buffer Design," *Computer,* Vol. 17, No. 1, Jan. 1984, pp. 6-22.

Lyon, R. F., "Single Instruction Stream, Multiple Data Stream, Multiple Port Pipelined Processor," Schlumberger internal report, Schlumberger Palo Alto Research, 3340 Hillview Ave., Palo Alto, CA 94304, Sept. 1985.

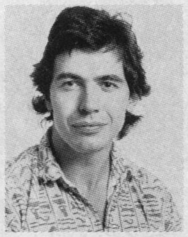
Martin, A. J., "The Torus: An Exercise in Constructing a Surface," *Proc. Second Caltech Conf. VLSI,* Jan. 1981, pp. 527-538.

Przybylski, S. A., et al., "Organization and VLSI Implementation of MIPS," *J. VLSI and Computing Systems,* Vol. 1, No. 3, Fall 1984, pp. 170-208.

Transputer INMOS Limited, "IMS T424 Transputer Reference Manual," Whitefriars, Lewins Mead, Bristol, England, UK, 1984.

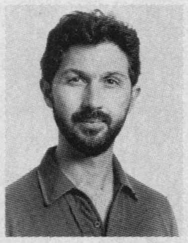


William S. Coates joined the AI Architectures Group at Schlumberger Palo Alto Research in 1984. He received his BS in computer science in 1980, and an MS in computer science in 1985, both from the University of Calgary, Alberta, Canada.

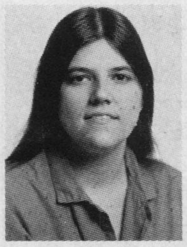


Ian N. Robinson is a member of the AI Architectures Group at Schlumberger's Palo Alto Research Center. He received a BSc degree in physics with electronics from the University of Sussex, England, in 1979.

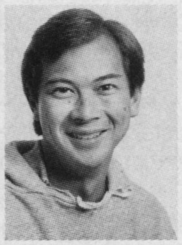
A. L. Davis received a BS in electrical engineering from MIT in 1969, and a PhD in electrical engineering and computer science from the University of Utah in 1972. He is currently a member of the AI Architectures Group at Schlumberger Palo Alto Research. He is a member of IEEE.



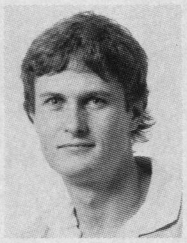
Shane V. Robison is a member of the AI Architectures Group at Schlumberger's Palo Alto Research Center. He received his BS and MS degrees in computer science from the University of Utah, Salt Lake City, Utah, in 1980 and 1983, respectively. He is a member of IEEE, ACM, and AAAI.



Judy Anderson is a member of the research staff of the AI Architectures Group at Schlumberger Palo Alto Research. She received a BA in philosophy in 1983 and an MS in computer science in 1984, both from Stanford University.



Robert W. Hon is a member of the AI Architectures Group at Schlumberger Palo Alto Research. Prior to joining Schlumberger, he was an assistant professor of computer science at Columbia University. He holds a BS in electrical engineering from Yale University and an MS and PhD in computer science from Carnegie Mellon University. He is a member of IEEE and ACM.



Kenneth S. Stevens is a member of the research staff of the AI Architectures Group at Schlumberger Palo Alto Research. He received a BA degree in biology in 1982, and BS and MS degrees in computer science in 1982 and 1984 from the University of Utah. He is a member of IEEE.

Readers may write to Shane V. Robison at Schlumberger Palo Alto Research, 3340 Hillview Ave., Palo Alto, CA 94304.