The Architecture of the FAIM-1 Symbolic Multiprocessing System

A. L. Davis
S. V Robison

Artificial Intelligence Laboratory
Schlumberger Palo Alto Research
3340 Hillview Avenue
Palo Alto, CA 94304

ABSTRACT

The FAIM-1 is an ultra-concurrent symbolic multiprocessor which attempts to significantly improve the performance of AI systems. The system includes a language in which concurrent AI application programs can be written, a machine which provides direct hardware support for the language, and a resource allocation mechanism which maps programs onto the machine in order to exploit the program's concurrency in an efficient manner at run-time. The paper provides a brief synopsis of the nature of the language and resource allocation mechanism, but is primarily concerned with the description of the physical architecture of the machine. The architecture is consistent with high performance VLSI implementation and packaging technology, and is easily extended to include arbitrary numbers of processors.

I Introduction

The goal of the FAIM-1 machine is to provide a high performance symbolic multiprocessor, one hundred or more times faster than current machines in common use (e.g., a Symbolics 3670) to meet the voracious computational demands of future Artificial Intelligence applications. This implies a *real* machine — one that works, is affordable and that people can program. Such a machine should entice researchers into the area of distributed AI problem solving and encourage its widespread use in the research community. It is hoped that such usage will facilitate development of the necessary expertise to make sophisticated, cost-effective machine intelligence applications practical. The FAIM-1 is also an architecture which is conveniently extensible, both in terms of scale (number of processors) and for future improvements to incorporate the benefits of new technology and systems ideas.

The system's functionality is primarily motivated from the top by the needs of AI symbolic computation, but the system structure is also restricted by the need to produce a high-performance, cost-effective system in an available technology. We feel that it is necessary to provide a consistent system which is designed from first principles to meet the needs of AI applications rather than adopting an ad hoc combination of systems ideas and components that were developed for sequential, primarily numeric applications. Such a system must also be complete enough to permit viable use and evaluation. The FAIM-1 system therefore includes a language, programming environment, architecture (a hardware prototype is under development), and a resource allocation mechanism. The focus of this paper is to describe the physical architecture of the FAIM-1 system. A brief synopsis of the language and resource allocation strategy will be presented in order to provide some perspective for the architecture in the context of the overall system.

In order to achieve our goals for an appreciably higher performance generation of intelligent machine systems based on concurrent multiprocessing, it is necessary to make a significant break with conventional architectural principles. Some of the traditional mechanisms simply are not viable in a highly concurrent environment. On the other hand, a dramatic shift of computational base from sequential to concurrent processing will be difficult after 30 years of highly refined experience with uniprocessing. In practice programmers are not going to readily make the difficult shift if the new systems require a significant change in the style in which they solve problems, or if the speed of the target machine is too slow to motivate the effort. Our approach in satisfying these conflicting constraints is to provide a reasonably small shift in thinking at the programming language level in order to incorporate concurrency, while making major changes in the structure of both the system software and hardware architecture in order to achieve the necessary level of system performance.

The design of the FAIM-1 system is intended to exploit concurrency at all levels of the system, and to pursue technological performance mechanisms in the implementation of the prototype hardware.

II Language and Resource Allocation

A. The OIL Language

OIL *(Our Intermediate Language)* can be viewed both as the kernel of a high level, concurrent, AI symbolic programming language and as the machine language for the FAIM-1 multiprocessing system. The design of OIL was primarily influenced by current AI programming practices. Commonly used are languages for knowledge representation, logic programming, object-oriented programming, production rules, procedural code, etc. Future complex AI applications may require several (or all) of these programming styles. Emulating one programming style within another is inefficient, and therefore a need exists for a better linguistic mechanism that efficiently supports many of the major styles. OIL can be viewed as a blend of object-oriented, logic, and procedural programming semantics into a single and internally consistent linguistic framework. Effort has been made to retain as much as possible from existing and familiar AI programming languages. Some modifications to familiar mechanisms have been incorporated into OIL which were primarily induced by the need to provide concurrent semantics where possible.

An OIL program is a collection of objects that communicate by sending messages. Each object is viewed semantically as an independent and therefore potentially concurrent program module. The communication structure explicitly indicates the level of concurrency represented by the program. An OIL object consists of some local state information (local variables and data structures) and several *porta* 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 that is essentially a parallel version of Prolog [Clocksin and Mellish, 1981]. Numerous parallel logic programming semantics are being investigated within the research community. OIL logical behaviors are semantically OR-parallel, restricted AND-parallel [DeGroot, 1984] logic programs.

Procedural behaviors are written imperatively in a parallel lexically scoped dialect of LISP based on the language T [Rees *et a/.*, 1984]. Behaviors can be nested heterogeneously to form other objects to permit control to pass between declarative and imperative behaviors.

The programmer may also annotate the OIL code with *pragmas* which are used to provide information about the dynamic aspects of how the code may behave at run time. These programmer supplied *hints* are used by the static resource allocator in an attempt to properly partition the concurrency extant in the program onto the physical resources of the machine. The goal of this activity is to maximise the benefit of concurrent operation at run time. This pragma information gives the programmer optional control over some aspects of the allocation strategy. Pragmas, as the name implies, are purely pragmatic information and are therefore orthogonal to the functional correctness of the program itself. If no pragma information is supplied, the program will still run although perhaps not efficiently.

B. Resource Allocation

Given a program that is a collection of concurrent tasks and a machine composed of a number of processors on which to run the program, there is an inherent problem of how to allocate the tasks onto the physical resources in an efficient manner. In systems such as the Cosmic Cube [Seitz, 1984] the burden of task allocation is left for the programmer; for certain highly regular cases this is neither a complex nor difficult task. In general however, it is important that the task structure reflect the programmer's organization of the solution and be independent of the machine architecture. Efficient task allocation is a critical problem that must be solved if advanced, highly concurrent machine systems are to mature and be truly useful. Several mechanisms have been studied, but generally three main options exist:

1. Programmed resource allocation relies on a *smart programmer* to write specific load modules for each individual processing element (PE). The disadvantage is that the task may be complex and the solution non-intuitive. An additional problem is that such code does not port to new machine structures, and therefore effectively returns to the dark ages of machine dependent programming. The advantage is that in many cases the programmer knows the optimal allocation better than any automatic mechanism.
2. Dynamic resource allocation employs a *smart operating system* to observe how load is distributed in the system. If an inefficient allocation exists, then the operating system redistributes some of the load from busy processors to lightly loaded or idle processors. The advantage of this mechanism is that if the load changes rapidly, then neither the programmed nor the static mechanisms can adapt properly. The drawback is that the overhead of dynamic allocation must be paid at runtime and therefore can decrease system performance.
3. Static resource allocation relies on a *smart compiler* to analyse the program text and to partition the resulting object code into a set of cooperating, concurrent sub tasks that conform to the grain size of the PE's and their interconnection topology. The primary advantage of this mechanism is that it does not directly increase the programmer's burden and also does not diminish the run-time performance of the system. The primary disadvantages are that the compiler based resource allocator can be very complex, and if the program exhibits highly dynamic behavior, then the result may be far from optimal.

Since the major goal of the FAIM-1 system is very high performance, the focus for resource allocation strategies is on static methods. Aspects of both the programmed and dynamic methods exist in that pragmas are both a way in which the programmer can influence allocation decisions, and a mechanism by which predictions about dynamic run-time behavior can be specified. Due to the highly dynamic nature of many AI application programs, some form of additional dynamic load balancing support will also need to be provided by the run-time kernel. However, our present interest is to experiment with how far we can push static methods. The FAIM-1 static resource allocation decisions are made in a post-compiler process called the *Allocator.*

The intermediate compiled code and the associated pragmas are passed to the allocator by the OIL compiler. The allocator then performs a dataflow analysis on the procedural code, a communication connectivity analysis on the program objects, and an inference connectivity analysis on the logical behaviors to produce a directed-program graph. This graph is then manipulated into a more abstract graph form which encapsulates much of the program graph detail. This intermediate graph form is then embedded into the machine graph representing the physical machine resources via a simulated annealing process [Kirkpatrick *et al.,* 1984]. While optimal graph embedding is in general an NP hard problem, the method used here fortunately is not concerned with an optimal allocation and runs in polynomial time in order to find an adequate allocation partition.

III FAIM-1 System Architecture

The primary purpose of the FAIM-1 architecture is the efficient, high-performance execution of OIL programs. The objective is accomplished by supporting, directly in hardware, the computational model on which OIL is based.

The FAIM-1 is a multiprocessor system consisting of a number of identical processing elements, called *Hectogons* (the result of an inside joke with no other interesting meaning), interconnected by a communication network. Each Hectogon is a complete, self-timed computer capable of sequentially executing any compiled OIL program that can be stored in its local memory. Hectogons communicate with each other via messages which are sent through communication *ports.* Note the similarity between this model and the organization of an OIL program. Each Hectogon has 6 ports that may potentially be concurrently active. Communication lines run between ports on different Hectogons; the exact configuration of connections is called the *communication topology.*

A Hectogon is composed of 6 self-timed [Seitz, 1979] subsystems - named the *FRISC, ISM, CxAM, SPUN, SRAM,* and *Post Office.* Three of these subsystems (ISM, CxAM, SRAM) are specialised memory systems that provide *intelligent* storage, the Post Office supports inter-Hectogon communication, the FRISC element is the processor, and unification support is provided by the SPUN element. For the initial 19 processor FAIM-1 prototype, four of the subsystems are being implemented as custom CMOS VLSI components and the other two (SPUN and SRAM) subsystems are being constructed from commercially available components.

One of the key features of this design is to experiment with the utility of specialised storage subsystems to relieve the processor from much of the storage management duties which are typically very time consuming in AI applications. These *intelligent* memory components permit a higher level of processor memory interaction, which inherently alleviates the classic memory bottleneck of conventional Von Neumann architectures.

Each of the 6 subsystems can be be viewed as concurrent system modules, and as an ensemble they support a considerable level of concurrent activity within the confines of a single processing element. Larger grain concurrency, corresponding to parallel program tasks, is exploited by distributing the tasks across multiple processing elements.

A. Communication Topology

The FAIM-1 communication topology is divided into two levels. At the bottom level, Hectogon elements are wired together to form a *processing surface.* At the top level, an arbitrary number of processing surfaces are connected together to produce a multiple surface instance of a FAIM-1 system. A processing surface is a planar hexagonal mesh structure and Hectogons which are on the edges of the plane are called *peripheral.* When a connection leaves the surface from a peripheral Hectogon's ports, it is routed to a simple 3-ported switch. One of the remaining ports of the switch is used for connection to an adjacent surface while the other is *wrapped* around to a switch on the opposite edge of the same hexagonal surface. Figure 1 shows a 19 processor surface, with the switches and wrap lines on one axis only. The switches and wrap lines have been omitted on the other two axes for clarity. This wrapped, hexagonal mesh is a 3-axis variant of a twisted toroidal topology [Martin, 1981]. This particular variant has a communication diameter of N-1 where N is the number of processors on an edge of the surface and each wrap is twisted N-l increments on each axis. The N-l twist creates a provably optimal switching diameter and can be supported by a rather simple routing algorithm which scales to permit multiple surfaces to be interconnected as well. This N-l twist topology can also be viewed as a uniform hexagonal mesh of processors which covers the surface of a sphere.

Hex Display with one Wrapped Axis

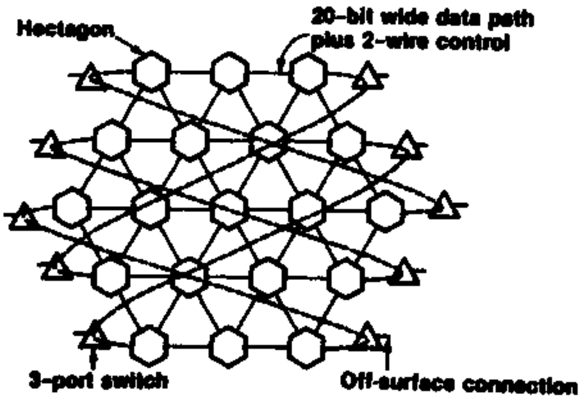

Figure 1: An E-3 Processing Surface

The wrap lines reduce the maximum communication distance between elements on the surface. It is hoped that the resource allocation strategy will be able to allocate tasks onto this surface such that a high degree of communication locality will be achieved. However, it is unlikely that strict locality can be efficiently achieved for highly dynamic AI programs. Therefore reducing worst-case communication times between non-local processing elements (by reducing communication distance) is also important.

Multiple FAIM-1 surfaces can be arbitrarily tessellated in a planar array, by connecting the adjacent Hectogon off-surface switch wires. Figure 2 illustrates this multi-surface interconnection plan. In this figure, Hectogons on the same surface have similar textures. The switches and wrap lines have been omitted for clarity. The tiling of a plane with several processor surfaces produces a large number of peripheral ports that can be attached to non-Hectogon devices. This is useful for I/O purposes, providing a large number of connections to secondary storage units or a host processor. By varying the surface size and the number of surfaces that are connected together it is possible to produce a system containing any desired number of processors.

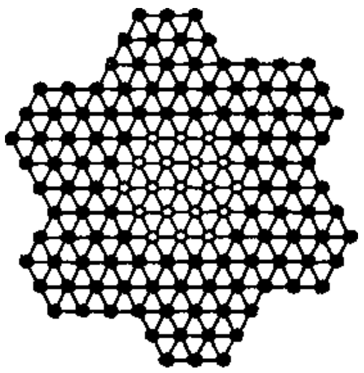

Figure 2: The Tiling of Multiple Surfaces

The primary advantage in using a hex communication topology is that it is easily extensible. The periphery of a processing surface forms a regular hexagon. In the 19 processor instance, each peripheral edge contains 3 processing elements. This particular configuration is therefore called an £-3 surface. An interesting artifact of this surface configuration, is that up to an £-7 surface, the number of processing elements on a surface is a prime number. This can be exploited during surface initialization to concurrently determine individual Hectogon status and identity.

A secondary advantage of the surface topology is that the on-surface wiring scheme is strictly planar and therefore amenable to wafer-scale packaging, either as a hybrid or a full wafer-scale integration design. We do not anticipate using wafer-scale integration in the near term due to the inherent problems associated with yield induced component failures. However, immersion cooled, hybrid wafer-scale packaging (Stopper, 1985] is an attractive option. In general it will be important for future high-performance architectures to be amenable to fabrication in modern circuit and packaging technology. Namely it does not make sense for multiprocessor architectures which exploit concurrency for increased performance to sacrifice an order of magnitude in speed due to a poor choice of implementation technology.

Fault tolerance is an important aspect of any highly replicated multiprocessor architecture, since the probability of at least one processor being down increases with the number of processors in the system. Homogeneous multiprocessor architectures intrinsically contain redundant elements which could be used to support fault tolerant behavior. Unfortunately most of these architectures to date have not utilised this feature, and have by default become *fault intolerant.* Koren [Gordon *et* a/., 1985] has shown that hexagonal mesh structures are particularly attractive fault tolerant topologies. In addition, communication fault tolerance is enhanced in a hex mesh topology because each element has a number of paths by which it may send messages to any particular destination.

B. Pott Office

Hectogons communicate with each other by sending messages. The Post Office subsystem is a highly concurrent, autonomous communications controller. It is responsible for the physical delivery of messages which are sent between communicating Hectogons. Initially messages are created by the FRISC and linked into the DELIVERABLE-MESSAGES list stored in the SRAM. The Post OfBce can independently access this structure and route individual messages to their intended destinations.

Messages are variable length structures and contain two subfields: a list of destinations, and the message body. A destination is a relative address which indicates the physical offset of the receiving Hectogon in the hexagonal mesh. Physically, FAIM-1 Post Offices form an independent packet routing network on the hexagonal mesh topology. Messages are therefore decomposed by the PoBt Office into a series of fixed length packets. Each packet contains a destination and a packet body. Routing of individual packets is done separately and recomposition of a message from a collection of packets is done by the receiving Post Office. This implies that packet arrival at the destination Post Office is inherently unordered, since several physical routing paths may be used for member packets of a particular message. While this increases the level of responsibility in the receiving Post Office, it permits congestion delays to be avoided dynamically in the switching topology.

Routing decisions are made algorithmically in the Post Office by a finite state machine as follows:

- If the destination is one of the neighboring Hectogons, then the packet is sent out on the appropriate port if it is available. If the port is not available, then the Post Office waits until it is. This model assumes that ports do not fail and can be easily generalised to permit a more lenient failure model.
- If the destination is not a local neighbor then the packet is sent out on any port which reduces the Manhattan distance to the destination. If no such port is available after a time specified by a CRITICAL-TIME parameter, then the packet is sent out to the first available port. This randomised routing after a critical time permits messages to be routed around areas of congestion. Simulation results demonstrate that this mechanism permits congested areas to be gradually dissipated by spreading the communication load over a wider spectrum of the physical communication resources.

Upon receipt of a packet, the receiving Post Office determines if the packet has arrived at its destination. If it has, the packet is stored into the appropriate location in the SRAM. If this packet completes the receipt of a message then the FRISC is interrupted by the Post Office to handle the received message. If the local Hectogon is not the final destination then the packet is forwarded using the routing algorithm described above. The final word of each packet contains a CRC which is monitored by the receiving Post Office to validate packet integrity during transmission. If the CRC check indicates that a transmission error has occurred, the receiving Post Office signals the sender to retransmit the last packet.

Physically the Post Office is composed of the following components:

- The Packet Buffer Pool is a multiported storage module consisting of a set of packet buffers.
- The External Controller is a finite state machine which makes routing decisions, manages the packet buffer pool, and connects port controllers to packet buffers.
- The Port Controllers (there are six - one for each Hectogon port), are responsible for receiving a packet and placing it in the assigned packet buffer (vice versa for output). They also perform bus master nomination duties and control the asynchronous handshaking of the bus wires used to asynchronously control word transmissions during packet transfer. The port controllers are also responsible for checking the CRC codes and signalling retransmission when necessary.
- The Internal Controller performs DMA accesses between the SRAM and the packet buffer pool. It is responsible for packetizing and depacketizing messages. The internal controller interrupts the FRISC when a complete message has been received and has been composed in the SRAM. The internal controller contains a MAIL-TABLE which is used to store target addresses in the SRAM for incoming messages, and to indicate which packets of a particular message have been received. A similar table is maintained for STREAM communications which require that the order in which messages are sent will be the order in which they are handled by the receiving program object.

Potentially the Post Office can keep all 6 external Hectogon ports active concurrently, freeing the FRISC element from using its cycles on communication overhead. This is a significant improvement over designs such as the Transputer [INMOS, 1984] which steals processor cycles to drive the ports and where only one of the four ports can be active at any particular time. In addition the hardware support for the routing algorithm is integrated into the architecture and does not have to be part of the run-time kernel.

C. Instruction Stream Memory

The Instruction Stream Memory (ISM) is a specialized high speed instruction delivery subsystem. In the FAIM-1, the locus of control for instruction delivery resides solely in the ISM. The ISM provides storage for the FRISC object code, and provides the control to decide what instruction should be executed next and delivers it to the FRISC.

The instruction stream can be viewed as a sequence of instructions broken by calls or jumps, both of which may be conditional. Modern programming practices tend to produce relatively short instruction sequences that correspond to small procedures. The role of either a jump or call is to select a next instruction that is not in lexical order. Since the jump and call instructions are seen by the ISM before they are evaluated, the ISM can predict when one will appear and plan for it in advance. The use of specialised hardware to enhance instruction delivery is certainly not new; branch prediction and instruction prefetch have been used to improve performance in many conventional architectures. Most of these systems (for example the scoreboards in the IBM System 360/91 [Bell and Newell, 1971], translation look aside buffers, and instruction caches [Smith, 1982)) increase speed by interposing a piece of specialised hardware between the memory and the processor.

The ISM takes this approach a step further. It provides a specialised instruction memory rather than merely placing a specialised interface in front of a conventional memory. The obvious disadvantage is that the ISM is only useful for storing instructions, and therefore the ISM cannot serve multiple storage roles as in conventional systems. The advantages, however are numerous:

- The ISM can be tuned for its sole function of high-speed instruction delivery.
- The processor complexity can be reduced.
- Code density can be increased since jumps and calls are removed from the code stream.
- A separate parallel data path for instruction delivery can be used.
- Operational concurrency is increased, since the ISM and FRISC can operate as concurrently cooperating partners.
- A more flexible interrupt and trap structure is permitted.

The code density issue is an important one since the power of the FAIM-1 machine is derived primarily from the high level of replication of processing elements that the architecture supports, and not from the power of the individual processor design. In order to permit high replication levels it is necessary that the size of the individual processor be small. Therefore the amount of storage available to each Hectogon is much less than that typically associated with conventional main frame architectures. The reduced storage sise provides significant motivation to any mechanism which can improve code density, hence the removal of branch instructions and the use of a stack based micro-architecture for the FRISC element are steps in the right direction.

The current ISM design attempts to improve performance by capitalising on the fact that instruction access patterns are not random. Hence the use of RAM memory for instruction storage is both slow and unnecessarily complex. The ISM essentially performs instruction prefetch of all instruction paths that can be taken with the exception of interrupt and trap sequences, and has them ready for delivery at the time they are needed.

Physically the ISM organises its storage into a set of *tracks*, each of which contains 16 instruction packets. An individual packet is 6 bits long. The FRISC instruction format permits two types of instruction lengths. Short instructions are a single packet and long instructions are formed as a pair of packets. The ISM decodes the instructions and sends properly formatted instructions over the 12 bit wide instruction bus. Hence it dynamically delivers a code stream of the right length instructions from a packed array of stored instruction packets. This reduces the amount of dead storage caused by fragmentation that might be incurred by permitting multiple instruction lengths. A track is viewed as a linear sequence of instructions which terminate by a jump or call instruction, or by the physical end of the track. If sequential object code segments are longer than one track, they are continued on the next track. If they terminate by a call or jump then each track header contains a tag indicating where the call or jump target is located. This tag replaces what would normally be jump and call opcodes in the instruction stream. Since 2-way conditional call/jump structures are supported, two possible targets exist. One will always be located on the *next* track and the other will point to what is termed the *remote* track.

Three track buffers are connected to the instruction output bus, they are the current, remote, and next track buffers. Instructions are delivered from the current track buffer. Concurrent with the delivery of the first instruction, the current track buffer tag is examined to determine the address of the appropriate remote track. The next and remote track buffers are then loaded. Since the machine is a completely asynchronous system, there is no way to determine the exact synchronization of these activities. However, in normal operation (no trap or interrupt occurs), two instructions are delivered from the current track buffer while the remote and next track buffers are loaded. This corresponds to full prefetch of both possible branch targets. If the branch does not occur in the first two instruction times of the current track then no delay will occur on either a call or a jump.

When a conditional branch is taken, due to the two stage pipeline of the processor, the condition line will not be valid until two stage times after the branch instruction. The strategy used is similar to the delayed branch technique used in the MIPS machine [Prsybylski *tt* a/., 1984], i.e. to insert non-condition modifying instructions in the code stream in order to prevent dead-time in the evaluation pipeline. Since most of the FRISC instructions do not modify the condition flags, this is typically fairly easy.

The communication between the FRISC and ISM units is self-timed, and controlled by a 4 cycle request/acknowledge protocol. Typically the ISM is ready with the next instruction long before the FRISC has completed evaluation of the previous one. The ISM also fields interrupts and traps and provides the proper instruction streams in these cases. In the case of a trap or interrupt, a 2 stage delay occurs since this is the only case when the ISM cannot predict where the desired instruction will be in advance and prefetching is therefore not possible.

In cases where a context switch between user processes is necessary the ISM can predict this case and prefetch the desired code as well as preset the appropriate status bits in the FRISC to create the context switch. It is noteworthy that not only does the ISM execute certain control instructions such as branch and call, but it also makes up instructions that are not in the code stream in order to initialise the processor status word when context switching is performed. For testability, it is possible for the FRISC to read or write any packet in the ISM.

While implementation details have been suppressed here for brevity, it is interesting to note that the ISM is a significant performance enhancement over traditional instruction delivery mechanisms. It also uses instruction bus bandwidth more efficiently, and is a design that would not be economical if built from commercially available components. The ISM is being fabricated as a 64 pin custom CMOS component, where each part contains storage for 2K instruction packets. These parts can be cascaded with no performance penalty into a group of up to 32 ISM chips to form the instruction storage and delivery subsystem for a particular Hectogon. The maximum amount of code storage per Hectogon is therefore 64K short instructions.

D. The FRISC and SRAM

The FRISC (for Fanatically Reduced Instruction Set Computer) is a specialised processor which supports the operations of OIL and coordinates the other Hectogon subsystems. It is also designed to be an efficient multitasking evaluation engine. Internally the FRISC contains a 20 bit wide data-path (4 tag bits, and 16 data bits), and is essentially a stack machine. Parallel tag hardware permits tag based traps to be used in order to optimise main line instruction streams for the common case. This is similar to the mechanism used in other high speed symbolic uniprocessor architectures such as the Symbolics 3600 [Symbolics, 1984]. This approach makes complex microprogramming support for the most general case unnecessary, and removes a level of control indirection which improves the overall performance of the FRISC element [Patterson and Sequin, 1981].

The instruction set is tailored to support OIL user programs. Altogether, the FRISC supports 64 basic instructions, corresponding to OIL functions such as UNIFY, CAR, etc. Integer arithmetic is supported by the ALU, but multiplication and division are performed iteratively using MULTIPLY-STEP and DIVIDE-STEP instructions. It is also interesting to examine the functions that are not in the FRISC instruction set. Jump and Call opcodes are absent since they are handled directly by the ISM. Also missing are complex string search instructions since they are supported directly by the CxAM.

The FRISC is actually composed of two processors, the evaluation or E processor and the switching or S processor. The S processor is responsible for manipulating the run-list and loading/unloading shadow registers to permit rapid context switching. This considerably increases the physical complexity of the FRISC element but provides high performance direct hardware support for multitasking. The goal is to keep as much of the FAIM-I's physical resources active as the process structure warrants. For example if a task initiates a CxAM search it will block and another task can be executed in overlap fashion pending arrival of the answers from the CxAM. The S processor is actually a small finite state machine with a couple of index registers which permit it to drive the data paths in the E processor and access the SRAM.

In the E processor, all of the primary registers in the data-path exist in triplicate, one register for each of the three contexts that the hardware supports: USERO, USER1, and KERNEL. The status word indicates which set of shadow registers are active for the current process. The primary registers are the Index Register, Frame Pointer, Stack Pointer, Stack Top, Stack Next, and the Stack Buffer. The stack buffer acts as a stack cache, and is actually 16 words deep. The stack buffer has its own memory controller which attempts to keep the buffer half full. This implies that a sequence of pops or pushes will not typically need immediate access to the SRAM memory bus. Non-shadowed registers include the Q register used during multiply and divide step loops, and the memory data and address registers. The ALU takes sources from two internal busses, one of which can be shifted prior to the ALU. Results from the ALU pass through a barrel shifter and are posted on a result bus which is used to deliver the result to the selected target. This shift arrangement facilitates the arithmetic and bit field manipulation instructions of the FRISC instruction set.

The FRISC views most data structures as objects; a conventional paged memory with a small finite-state machine attached to each page (collectively called the SRAM) provides an object-oriented memory system for the FRISC. The SRAM (Structure RAM) stores all procedural data structures, as well as logical variable bindings and the bodies of logical rules in list form. The SRAM's atomic addressable entity is a word, which is composed of two portions: a 4-bit tag and a 16-bit data field. Multiple-word objects (e.g. simple vectors, bignums, or continuations) are represented as a structure containing one or more header words followed by indexable data fields. The tag bits support the common dynamic data types allowed in many AI languages. Using the data tag bits, the SRAM can (concurrently with other FRISC computation) follow a pointer chain to retrieve an object requested by the FRISC.

The close connectivity between the FRISC and its small SRAM removes the usual performance gap between registers and primary memory. In our case, registers have at most a 2:1 speed advantage over memory, so the complexity of a general register architecture is not easily justified. Stack architectures are a more natural fit as a compiler target, providing improved instruction code density, reduced data path complexity, and faster context switches. The resulting simple data path and simple instruction set is a candidate for straightforward control implementation.

E. Context Addressable Memory

The Context Addressable Memory (CxAM) is a highly parallel associative storage subsystem capable of searching for and retrieving structured data. Pattern matching and associative storage accesses are common operations in many AI applications. The CxAM provides direct hardware support for this important activity. Rule headers for logical OIL behaviors, and procedural data structures which are associatively accessed are stored in the CxAM.

In previous systems, a variety of hashing schemes have been used in lieu of CAM components. This choice makes sense in traditional architectures where the "smart processor with big, dumb memory" partition is cast in concrete. The typical CAM does not provide sufficient associative support for AI match functions, since they match either tag bits or single word contents. In FA1M-1, the CxAM can match structures as well as slot contents.

The structure of both entries and queries in the CxAM is a LISP S-expression. Each slot therefore can either be a structure or an atom. Atoms can be symbols, numbers, variables, or *don't cares.* Semantically, variables are treated as *don't cares* by the CxAM. The inclusion of variables as atom types for the CxAM is based on FAIM-I's support of logic programming. The retrieval and setting of logical variable bindings is supported elsewhere in the Hectogon.

The CxAM responds to four commands: *Find Match, Give Match, Delete Structure,* and *Add Structure.* The Find and Give functions are optimized for speed, while the Delete and Add functions are implemented with more concern for minimizing circuit area than performance. The frequency of Find and Give is much higher during program execution than that for Delete and Add. The CxAM also manages its own free space and removes garbage automatically, thereby freeing the FRISC element to process user instructions rather than manage storage.

Physically the CxAM consists of a storage area and a number of parallel search engines which share a multiported query buffer which contains the pattern to be matched. Each search engine concurrently searches a subset of the storage area. A complete and very detailed exposition of this device, implemented as a custom CMOS component, can be found in [Brunvand, 1984].

F. Streamed Pipeline Unifier

The Streamed Pipelined UNifier (SPUN) provides direct hardware support for unification of logical OIL behaviors. The CxAM can be used to find the next rule or set of rules to be tried, but the CxAM does not perform full unification since its match function does not consider variable bindings. The SPUN unit takes the query and the streamed set of matched structures from the CxAM, detects which variable bindings still need to be matched, fetches bindings in the current context from the SRAM, and completes the unification. This may entail binding a variable, in which case the SPUN unit must post this binding back in the SRAM. It may also entail starting another subgoal unification, in which case the present state must be stacked, and a new query must be presented to the CxAM.

IV Conclusions

In this paper, we have presented an architectural overview for the design of a highly parallel symbolic processor known as

FAIM-1. In general there have been two approaches taken in the design of similar systems. The first is to build concurrent processing ensembles out of conventional processor and memory components as has been done for the Cosmic Cube [Seits, 1984], Butterfly[Gurwits, 1984], and DADO (Stolfo, 1983] systems. In general we feel that to truly achieve a new generation of viable symbolic processors which are a major performance improvement over existing systems, it will be necessary to significantly reallocate the transistor budget to support tasks which are specific to the domain of symbolic processing. This is not possible by merely assembling old components in new ways. The other approach is to experiment with radical new models of computation which are inherently highly parallel as is the case with the Connection [Hillis, 1981] and Boltsmann[Hinton *et al.,* 1984] machines. The problem with this approach is that the ways in which we solve problems must change radically as well, and the incorporation of 30 years of expertise in programming is all but impossible in the short term. We feel that both approaches are viable: the first in the short term and the second in the long term. The FAIM-1 design attempts to fill the gap by providing a rather different but specialised architecture for performance, while requiring only a small change in programming practice to incorporate concurrency.

V Acknowledgments

The FAIM-1 system is the result of a wide variety of ideas from a number of project members, and does not represent the sole effort of the authors. Key contributions have also been made by: Judy Anderson, Allan Schiffman, Shimon Cohen, Ken Stevens, Ian Robinson, Mike Deering, Marty Tenenbaum, Dick Lyon, Erik Brunvand, Bill Coates, Bill Athas, Dan Carnese, Barak Pearlmutter, Bob Hon, John Conery, Alvin DeSpain, and Gary Lindstrom.

References

[Bell and Newell, 1971] C. G. Bell and A. Newell. *Computer Structures: Readings and Examples.* McGraw-Hill, 1971.

[Brunvand, 1984] £. Brunvand. *Context Addressable Memory for Symbolic Processing Systems.* Masters Thesis, University of Utah, Dept. of Computer Science, August 1984.

[Clocksin and Mellish, 1981] W. F. Clocksin and C. S. Mellish. *Programming in Prolog.* Springer- Verlag, 1981.

[Conery and Kibler, 1981] J. S. Conery, D. F. Kibler. *Parallel Interpretation of Logic Programs.* Functional Programming Languages and Computer Architecture, October 1981, 163-171.

[DeGroot, 1984] D. DeGroot. *Restricted AND-Parallelism.* Proceedings of the International Conference on Fifth Generation Computer Systems 1984, Institute for New Generation Computer Technology, 1984, pp. 471-478.

[Gurwits, 1984] R. Gurwitz. *The Butterfly Multiprocessor.* Talk presented at the 1984 ACM National Convention, San Francisco, October, 1984.

[Hillis, 1981] D. Hillis. *"The Connection Machine[9] (Computer Architecture for the New Wave).* AI Memo 646, M.I.T. Artificial Intelligence Laboratory, 1981.

[Hinton *et* al., 1984] G. Hinton, T J. Sejnowski and D. H. Ackley. *Boltxmann Machines: Constraint Satisfaction Networks that Learn.* CMU-CS-84-119, Carnegie Mellon University, May 84.

[INMOS, 1984] INMOS Limited. *IMS T424 Transputer Reference Manual.* INMOS Limited, 1984.

[Kirkpatrick *et* a/., 1984] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. *Optimization by Simulated Annealing.* Science 220, 1984, pp. 671-680.

[Gordon *et al.,* 1985] D. Gordon, I. Koren and G. M. Silberman. *Fault-Tolerance in VLSI Hexagonal Arrays.* Preprint.

[Martin, 1981] A. J. Martin. *The Torus: An Exercise in Constructing a Surface.* Proceedings of the Second Caltech Conference on VLSI, 1981, pp. 527-536.

[Patterson and Sequin, 19881] D. A. Patterson, C. H. Seguin. *RISC 1: A Reduced Instruction Set Computer.* Proceedings of the Eighth International Symposium on Computer Architecture, 1981, pp. 443-458.

[Prsybylski *et* a/., 1984] S. A. Pryzybylski, T. R. Gross, J. L. Hennessy, N. P. Jouppi, C. Rowen. *Organization and VLSI Implementation of MIPS.* Journal of VLSI and Computer Systems, Vol. 1, Number 2, 1984, pp. 170-208.

[Rees *et al.,* 1984] J. A. Rees, N. I. Adams, J. R. Meehan *The T Manual.* Yale University, Fourth Edition, 1984.

[Seits, 1984] C. L. Seitx. *The Cosmic Cube.* To appear CACM.

[Seitz, 1979] C. L. Seitz. *System Timing.* In Introduction to VLSI Systems, Chapter 7, McGraw-Hill, 1979.

[Shapiro, 1983] E. Shapiro. *A Subset of Concurrent Prolog and its Interpreter.* TR-003, Institute for New Generation Computer Technology, , 1983.

[Smith, 1982] A. J. Smith. *Cache Memories.* Computing Surveys 14, 3, 1982, pp. 473-530.

[Stolfo, 1983] S. J. Stolfo, *et al. Architecture and Applications of DADO, A Large-Scale Parallel Computer for Artificial Intelligence.* Proceedings of the Eighth International Joint Conference on Artificial Intelligence, Karlsruhe, West Germany, August, 1983, pp. 850-854.

[Stopper, 1985] H. Stopper. *A Wafer with Electrically Programmable Interconnections.* Proceedings of the 1985 IEEE International Solid-State Circuits Conference, 1985, pp. 268-269.

[Symbolics, 1984] Symbolics, Inc. it Symbolics 3600 Technical Summary. Symbolics, 1984.