LISP MACHINES COME OUT OF THE LAB

No longer restricted to the research lab, artificial intelligence is becoming increasingly attractive to commercial users thanks to computer architectures designed to support the Lisp language.

by Mache Creeger

Artificial intelligence, defined as the science of enabling machines to reason, make judgments, and even learn, is often seen as a field whose practical benefits will be realized only at some future date. This is not entirely true. Artificial intelligence researchers have already contributed to the development of such techniques as timesharing, networking, and window systems—a part of the commercial computing world. However, a powerful software tool developed by the artificial intelligence community, the Lisp language, is just beginning to make an impact outside of the research labs.

The Lisp language deals with the complex and unpredictable data characteristic of the artificial intelligence (AI) field. It permits large, powerful programs that traditional programming techniques cannot handle to be written, tested, and modified. However, traditional computers (such as Digital Equipment Corp's system10 or VAX, and similar machines), cannot support Lisp so that it can become an efficient tool for commercial use. As a result, the power and productivity of the language have remained in the research lab, where functionality, rather than speed, is the major consideration.

An efficient Lisp implementation requires an architecture optimized for the structure of the language, as well as very high storage capacity. In addition, Lisp must be integrated with other computing environments to eliminate the need for an all-at-once changeover from present software and/or hardware. Such a machine offers a programming environment that provides substantial productivity increases for the development of a range of software systems, as well as an evolutionary means of bringing "intelligence" to existing computer systems.

Lisp and the commercial world

While this potential should make it attractive to software developers and system integrators in the commercial world, the lack of appropriate hardware for the language has kept it in the research labs. Before the development of the first Lisp machines at the Massachusetts Institute of Technology (MIT) in the 1970s, Lisp implementations ran on mainframes. Since the architecture of these machines was optimized for numeric languages such as Fortran, much of the Lisp environment was in software, thereby imposing substantial overhead on program execution. The applicative and recursive nature of Lisp requires an environment that efficiently supports stack computations and function calling.

In addition, Lisp's memory requirements exceeded even the capacities of these large

Mache Creeger is director of marketing at LISP Machine Inc, 3916 S Sepulveda Blvd, Culver City, CA 90230, where he is responsible for sales and marketing of the company's Lambda machine. Mr Creeger holds a BS and MS from the University of Maryland.

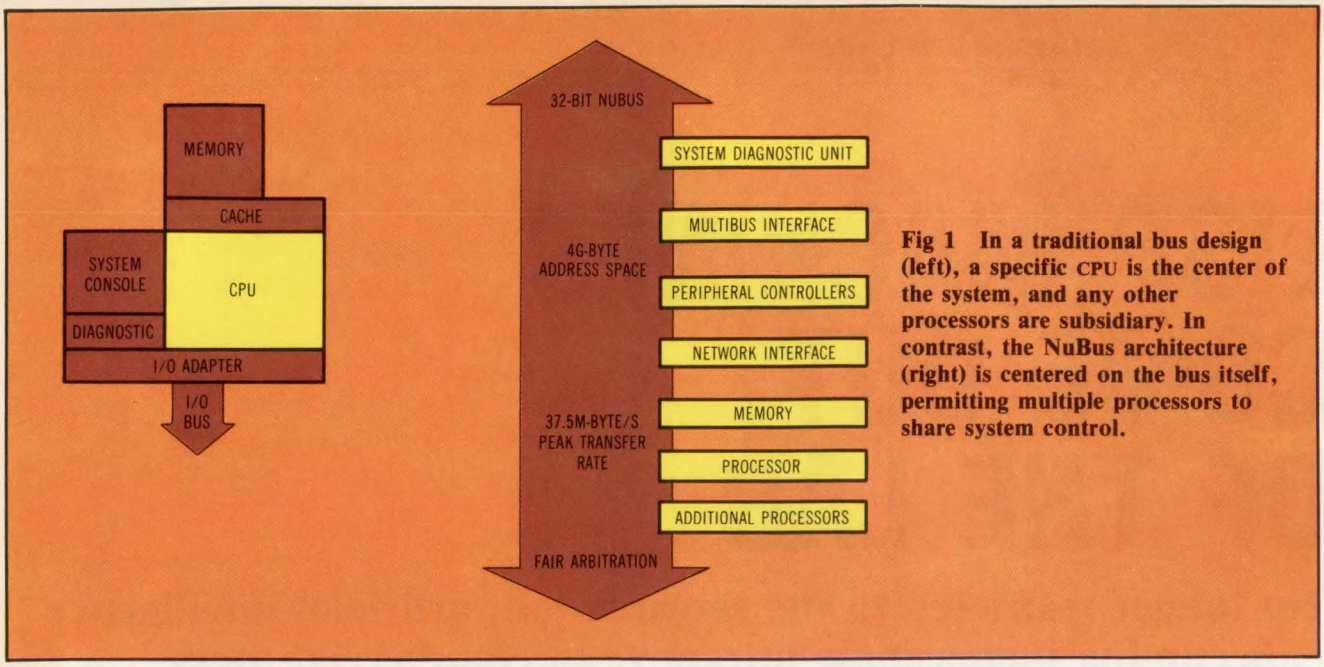


Fig 1 In a traditional bus design (left), a specific CPU is the center of the system, and any other processors are subsidiary. In contrast, the NuBus architecture (right) is centered on the bus itself, permitting multiple processors to share system control.

computers. Frequent stops for garbage collection made execution slow. Methods used to implement data-typing imposed another handicap. Lisp is a weakly typed language, meaning that functions can deal with a number of different data types (eg, fixed and floating point numbers) through a process called coercion, where a function recognizes the kind of data object it is dealing with and reacts accordingly. Traditional memory organizations require this process to be handled in a number of inefficient ways. Devoting fixed areas of memory to specified types causes memory fragmentation; other devices required extensive software overhead. Other problems were the language's poor arithmetic capabilities, since overcome by better compilers and hardware support, and its "standalone" nature. Because it was developed for use as a research tool by individuals or small groups, it did not integrate well into more traditional multiuser computing environments.

Lisp machine design

To illustrate how Lisp machine design overcomes these handicaps, consider the Lambda Machine from LISP Machine Inc. It is a lineal descendant of the original MIT Lisp machine, the CONS, which required mainframe support to operate. This machine was superseded by the CADR, which was later brought to the marketplace by LISP Machine Inc. The CADR was a personal, networked computer for programmers developing large, complex software systems. Drawing heavily on design experience gained from the CADR, the Lambda adds Lisp-oriented enhancements to the advanced high performance NuBus, also developed at MIT. Combining the Lambda processor with the NuBus architecture produces a modular, expandable Lisp machine with multiprocessor capabilities. The Lambda offers an integral Multibus, Ethernet-II networking, and the Lisp Machine Lisp/Zeta Lisp operating environment.

NuBus's device-independent architecture, originally developed at MIT's Laboratory for Computer Science and now supported by Texas Instruments, centers on a 32-bit bus with a 37.5M-byte/s peak transfer rate. Important aspects are its ability to support multiple processors and the architectural flexibility furnished by the system diagnostic unit (SDU). Both of these distinguish it from traditional architectures (Fig 1).

Traditional bus architectures center on a single processor, with major subsystems arrayed around a specific central processing unit (CPU). In contrast, the NuBus is a communication-centered design that allows rapid interchange of data between a variety of devices within a 4G-byte address space. Input/output (I/O), interrupt, and memory signals are initiated uniformly over the bus, and transactions are based on a "master/slave" concept: any given device may control the bus and address another device as a slave for that transaction. A simple handshake protocol used between master and slave enables modules with different speeds to communicate. This arrangement allows a variety of processor combinations to be used.

NuBus architecture handles five functional classes of signals. Four card-slot identification signals assign a unique physical location to each of 16 boards, so that any system module can occupy any board location; no dual-inline package switches, jumpers, or special backplane wiring are necessary. Six control signals—CLOCK, RESET, START, and ACKNOWLEDGE for data transfers, and two transfer mode (TM) signals for type of transfer—perform all control functions. Modes include 8-, 16-, and 32-bit (full-word) transfers as well as block transfers of up to

16 words. Thirty-two signals carry a 32-bit address at the beginning of each clock cycle, and 32 bits of data in the remainder of each cycle. Five signals control bus arbitration, and two indicate system parity and parity validity.

Multiprocessor operations

Two elements of this high speed bus design are particularly important for supporting multiprocessor operations: the memory mapped interrupt scheme, and the distributed bus arbitration logic that governs the master/slave relationships among devices. There are no interrupt lines on the NuBus. Instead, interrupts are accomplished by write transactions into memory addresses monitored by the interrupted processor. Any memory location may be specified as an interrupt address for any processor. This technique specifies interrupt priorities in software by memory mapping the priority level of each interrupt, thus eliminating the difficulties otherwise encountered in systems using multiple processors with differing interrupt schemes.

Arbitration occurs each time control is transferred between bus masters, and is independent of data transfers. The winner of the arbitration controls the bus until an arbitration is won by another device, but control is not transferred until the losing bus master completes any current data transfer. The distributed bus arbitration logic provides fair bandwidth sharing between processors by organizing devices on the bus into logical groups. When several devices simultaneously request the bus, the highest priority device gains control, but no device can initiate new bus requests until all devices in the group have acquired the bus. This prevents high priority processors from starving those with lower priority.

A bus master that acquires the bus is automatically the highest priority device within its group; thus it can accomplish an undivided set of data transfers by continually arbitrating for, and winning, the bus. If no other processor requests the bus, the current bus master may continually initiate data transfers without rearbitrating for the bus each time. This scheme speeds up processing by relieving a bus master of unnecessary arbitration overhead.

The NuBus's modularity and device independence comes from the SDU. This 8088-based board serves both as an architectural supervisor and as a smart diagnostic front end. Upon power-up, the SDU verifies bus integrity, identifies boards in the system from the contents of a small read only memory (ROM) on each board, and configures the system accordingly. It tests each board, signals the presence of any defective modules, and then boots the system. The SDU stores system configuration information in a nonvolatile battery-backed complementary metal oxide semiconductor random access memory (CMOS RAM) and can dynamically change the system configuration on command. Two RS-232 serial ports serve either for remote diagnostics or as general purpose serial ports. The SDU is also the system clock source. Fig 2, a block diagram of the Lambda machine, illustrates the SDU's importance in system control and configuration.

The SDU also serves as the NuBus interface with the Multibus. The Multibus allows the Lambda to interface with numerous peripherals and board-level products. The two buses operate independently except during bus conversions, which are accomplished through a hardware mapping scheme that requires no participation by the 8088 processor. The Multibus's entire 1M-byte address space appears as one continuous block in the 4G-byte NuBus address space. Conversion from NuBus to Multibus is transparent; a NuBus processor can access data or execute a program from Multibus memory. Conversion from Multibus to NuBus is accomplished by a page-mapping scheme that uses the upper 10 bits of the Multibus address to reference a page-mapping table.

The 22-bit page-frame number obtained from the map is concatenated with the lower 10 bits of the Multibus address to yield a 32-bit NuBus address. Interrupts originating in the Multibus are mapped into NuBus interrupt addresses by the 8088 processor; interrupts from NuBus to Multibus are written by the NuBus to an addressable latch on the

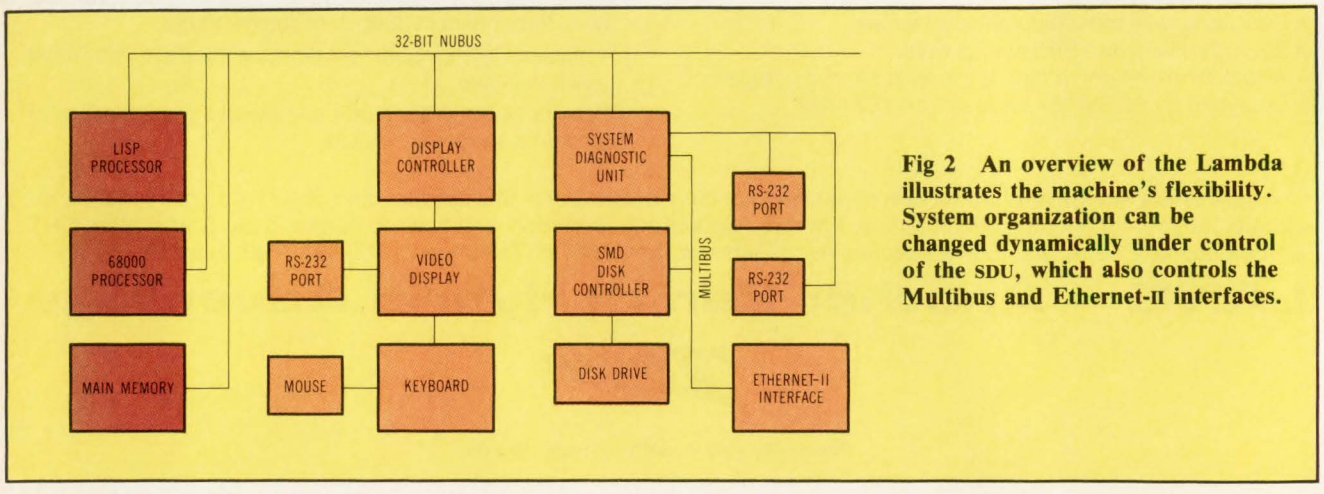


Fig 2 An overview of the Lambda illustrates the machine's flexibility. System organization can be changed dynamically under control of the SDU, which also controls the Multibus and Ethernet-II interfaces.

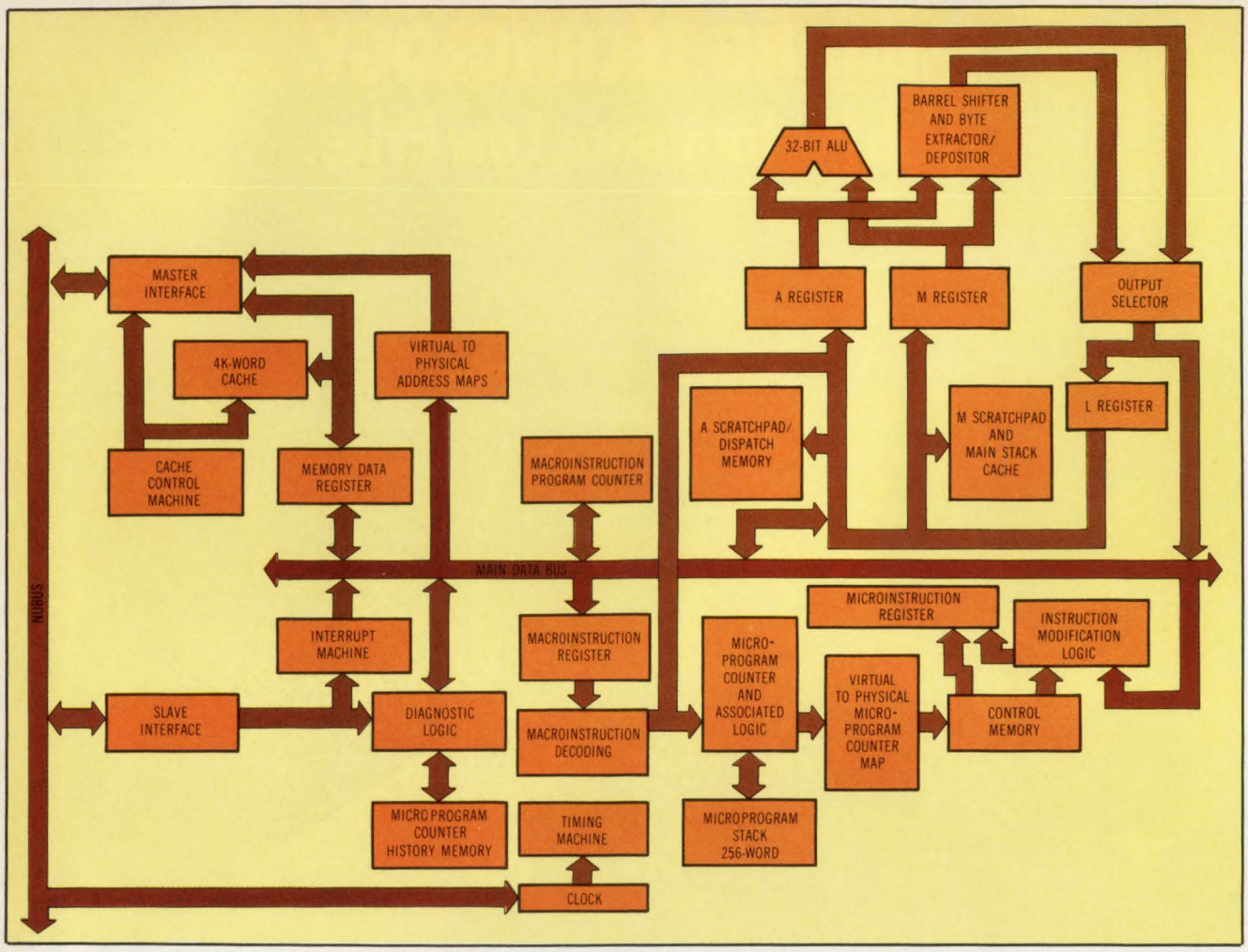


Fig 3 The Lambda's Lisp processor consists of four boards (not outlined in this diagram). Clockwise from the top left are the memory interface board, which interacts with the NuBus; the data paths board, where data manipulation takes place; the control memory board, which contains the microcode functions; and the random gates board, which includes several control and diagnostic functions.

SDU, which creates the appropriate Multibus interrupt. If both buses request each other simultaneously, the SDU prevents deadlock by giving priority to the slower Multibus-to-NuBus transfer, since NuBus-to-Multibus transfers can be rescheduled faster.

Another important aspect is Lambda's Ethernet-II interface, which executes the Advanced Research Projects Agency network (ARPANET) Transmission Control Protocol/Internet Protocol (TCP/IP). This interface facilitates resource sharing and interuser communication, easing the task of integrating a Lisp machine into an existing system. The Ethernet-II interface is controlled by another 8088-based board that provides the hardware interface and handles all network control protocols (NCPs). This frees the processor(s) from the overhead usually associated with NCPs, making protocol updates simpler and less time consuming.

Other hardware elements enhance NuBus operation. Memory boards in the NuBus system are self-contained memory controllers that support block transfers and error correction and logging. The memory holds 39-bit words: 32 bits of data and 7 bits of error-correction code. The video display system supports a high resolution (800 x 1024) display with two 1M-bit video buffers (useful when screen updates should not be seen), onboard logical functions, and a keyboard-mouse interface. Rounding out the generic aspects of the architecture are a disk controller that can handle four storage module device (SMD) drives, a 470M-byte Winchester disk drive, and a card cage with 21 slots: 13 for NuBus, 5 for Multibus, and 3 for either.

The Lisp processor

Four boards, utilizing high speed Schottky transistor-transistor logic (TTL) devices and communicating through a private bus on the backplane, constitute the Lisp processor. This general purpose 32-bit microprogrammable processor provides efficient pipelined execution of complex order codes. Although its optimal function is interpreting the Lisp compiler's bit-efficient 16-bit order code, the processor easily adapts to high level language execution as well as to specific

The Lisp language

The Lisp (List processing) language deals with arbitrary symbols—that can represent any concept—rather than numbers. The basic Lisp data structures are the atom and the CONS node. An atom, as its name indicates, is a data object that cannot be further broken down. A CONS node is a data structure that consists of two fields, each holding a pointer to another Lisp data object, which in turn may be an atom, another CONS node, or any other Lisp object, such as a string or an array. Any number of CONS nodes may be linked together to form data structures of arbitrary size and complexity; such structures prove ideal for handling unpredictable data such as natural-language representations. The list is one of the most important forms these complex data structures can take—hence Lisp's name.

Each Lisp atom has associated with it a property list, which gives additional information about the atom, including the atom's value (if it is a variable), its print name (a pointer to its character representation in memory), or any other property the programmer assigns. For instance, an atom that is an English word might have as a property its part of speech, its phonetic representation, or even its connotations.

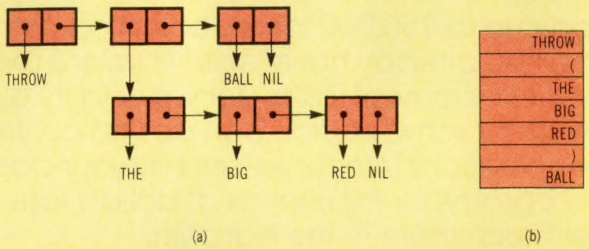
Part (a) of the Figure illustrates Lisp's data structure. It represents a simple list—(THROW (THE BIG RED) BALL)—made up of six linked CONS nodes (the double squares) and seven atoms (each word in the list). NIL is a special atom used to mark the end of a list or sublist. This Lisp construct contrasts with a Fortran array shown in (b) containing the same list. In the Fortran array, parentheses indicate the beginning and end of sublists.

The Lisp list structure offers several advantages. For example, if a single sublist—such as "THE BIG RED"—appears in many lists, it need be represented in memory only once, and pointers in each main list can reference it. Moreover, the elements of a list need not be adjacent to each other within memory, allowing efficient use of storage space. Furthermore, elements can be easily added to or deleted from a list without affecting other data elements—only pointers need be changed. In contrast, many elements in a Fortran array must be moved up or down when one is inserted or deleted. In addition, Lisp allows sublists to be skipped during searches of main lists; as a result, Lisp can process large lists much faster and more efficiently than Fortran.

Productivity advantages stem from Lisp's functionally based programming style, its runtime nature, and its extensive editing and debugging facilities. In contrast to traditional programming, where a great deal of time is spent merely specifying application parameters that may not be fully known until the program is finished, a Lisp programmer can have a program up and running very quickly. It then can be modified to suit the needs of end users based on their actual experience with the software. As an example of Lisp's productivity enhancements, LISP Machine Inc devised a sophisticated Lisp-based CAD system that it in turn used to design its Lambda machine. The CAD system was completed in less than two man-years; if designed with traditional programming methods, completion would have required an estimated 50 to 100 man-years.

Lisp programs consist of a group of functions, in contrast to traditional languages, which consist largely of sequential instructions and attendant subroutines. Lisp is thus a naturally modular language, and programmers can readily break down a function into many easily handled subtasks, or smaller functions. Lisp is highly recursive, allowing a function to call itself. This is a useful feature when a subtask is identical to the main task.

The language can either run interpretively or be compiled. In the interpretive mode, Lisp functions and data have the same structure; therefore, functions can manipulate or even create other functions. In modern Lisp machines, every bit of software, from the operating system to the editing and debugging utilities, is written in Lisp and can thus be easily customized to suit a programmer's needs. For example, a programmer can design applications software that creates a function in data-structure form, submits that structure to the system Lisp compiler (which is itself a Lisp program), and then automatically executes the resulting compiled program as part of the applications software.



(a) (b)

Lisp's runtime nature, which stems from its dynamic storage allocation and link-edit features, eases program generation by allowing programmers to defer decisions regarding the form of the final program. In contrast to traditional programming languages, Lisp does not require a declaration of required storage prior to writing the program. New storage is allocated during program execution as the program requires it. When the system senses that an area of memory can no longer be accessed by a program (eg, when a sublist's pointers are deleted from the main list), the inaccessible storage is automatically reclaimed and made available for new allocation through a process called garbage collection.

In addition, Lisp programs do not require a separate link-editing phase during compilation. Instead, functions are linked at run time and therefore can be easily changed even after compilation. Program modifications involve editing only those functions affected by the change and recompiling them—there is no need to recompile and link-edit the entire program.

Since Lisp has been the language of choice in the artificial intelligence field for many years, a powerful set of editing and debugging tools have been developed for it. Using such tools, a programmer can, for example, concurrently observe program source and execution, retrieve and modify any function, and recompile the modified function back into the program with very few keystrokes. Furthermore, because all of the programming utilities are written in Lisp, they can be easily incorporated into an applications program through Lisp's dynamic-linking capability. For example, the Lambda's Lisp Machine Lisp/Zeta Lisp environment includes an extensive window system, implemented by a message-passing feature called Flavors. This window system can be easily modified to serve as a user friendly interface for an applications program.

applications that rely on certain macroinstructions, which can be microcoded for faster execution. Main data paths of the processor are shown in Fig 3.

Four boards centralize related areas of the Lisp environment. Briefly, the data paths (DP) board contains the arithmetic logic unit (ALU), dispatch logic, scratchpad memories, and associated registers. The control memory (CM) board incorporates microcode functions and associated logic, and the microinstruction stack. The memory interface (MI) board, a NuBus master, contains cache, cache state machine, location counter, and diagnostic logic. Responsible for relations between the Lisp processor and the NuBus system, its operation is especially important for multiprocessor applications. Finally, the random gates (RG) board holds the macroinstruction decoder, statistics counter, history RAM, clock, matrix multiplier, and a slave NuBus diagnostic interface.

The Lisp processor's data paths are 32 bits wide: 24 bits for data and 8 bits for data-typing and other operations, giving the Lambda 67M bytes of address space (2^{24} 4-byte words). A 40-bit enhancement (planned for early next year) will expand the address space to 2l.5G bytes. Since the number of bits used for data-typing will remain the same, little or no reprogramming will be necessary.

In addition to its large address space, the Lambda uses a technique known as CDR-coding to reduce storage demands of list structures by almost 50%, making the address space seem even larger. CONS (constructor) nodes (see the Panel) cannot be inserted into a list compressed by this technique. However, since the processor automatically reexpands the CDR-coding when the list is accessed for modification, the technique is transparent to the user.

Processor design aspects

Data are passed to and from the Lisp processor under control of the cache state machine, a specialized high speed processor. Using NuBus block-transfer capability, the cache state machine manages memory accesses in a look-ahead/look-behind mode based on the principle of set-local operations, or locality. Presuming that the next word to be accessed is nearby the last word requested, the machine transfers an entire block (up to l6 words) centered on the requested word into the cache.

To avoid the problem of one processor interfering with another's data and resulting inaccuracy of the data buffered in a processor's cache, the NuBus is monitored continuously. The cache state machine, in combination with the master interface, constantly checks the NuBus to determine whether any other processor is writing into a location represented in the cache. If so, it invalidates that location, thus both assuring reliable data and avoiding the need for cache sweeping—a fragile and unreliable method of cache verification.

Memory access occurs through a 2-level virtual paging system that employs three virtual-to-physical address maps to map 24-bit virtual addresses into addresses within physical memory in the NuBus. This paging implementation also supports an efficient garbage-collection algorithm, which reclaims static-memory areas less often than more volatile areas, thereby consuming less processing time.

The system's vectored interrupt system gives each device an address space in the interrupt slot, and assigns an address in software to each type of interrupt. Each interrupt's status is stored in a RAM, which is scanned to see if any device has requested an interrupt. When an interrupt request is found, scanning stops, thus preventing interrupts from being lost (new interrupts are still stored). No other interrupt is noted until the current one is serviced and cleared. Provision for both fast and slow interrupts provides a flexible interrupt environment.

Associated with the interrupt machine is the slave interface to the NuBus, used largely for diagnostic purposes. It communicates with the diagnostic logic. This logic includes a 4K x l6-bit microprogram-history RAM that holds the control memory addresses of the last 4096 microinstructions executed. For debugging, the system can be manually halted. It can also be programmed to halt in the event of a specified error or other condition, such as the execution of a given instruction a set number of times. At the time of the halt, the system's state is saved with no loss of information: the machine state is exactly as it was during the execution of the instruction that initiated the halt.

Diagnostic logic can then be used for unclocked transactions to trace machine state, or can single-step the system with user-generated clocks to track down possible timing problems. In combination with software debugging facilities, these diagnostic capabilities enable programmers to easily pinpoint and correct problems from the largest programs to the user-written microcode. The RG board also contains a high speed 16 x 16 matrix multiplier. This greatly speeds array referencing as well as simple arithmetic.

A macroinstruction program counter holds the address of the next macroinstruction to be executed. Since two consecutive instructions are usually used, two macroinstructions are fetched concurrently, packed into a single 32-bit word, and placed into the macroinstruction register. Macroinstruction decoding hardware allows a transfer to the appropriate microcode subroutine in a single operation, saving a significant amount of processing time.

Processor pipelining and virtual control store operation are governed by the microprogram counter and its associated logic. The logic tracks the options available when making or returning from a microinstruction subroutine call. This tracking prepares the machine for rapid execution of the next