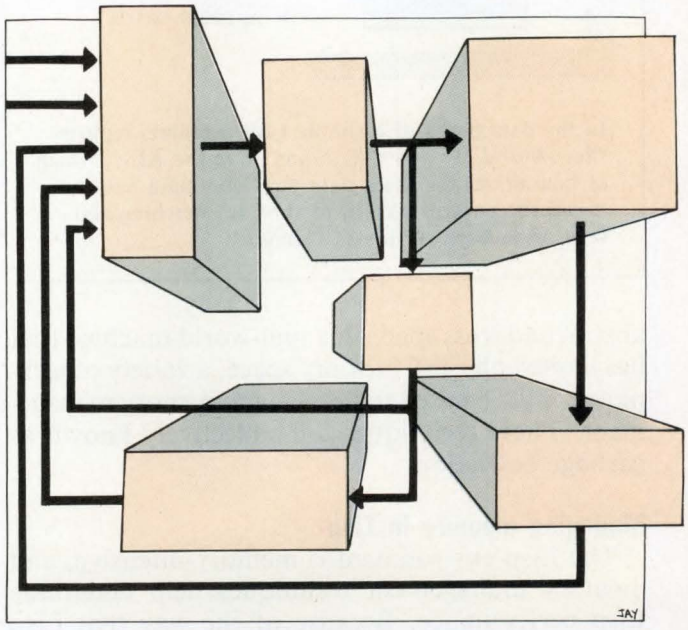
Matching hardware to Lisp yields peak performance

The Lisp environment differs significantly from conventional computing. Understanding how the software features of Lisp affect the operation of Lisp machines, gives designers an edge in choosing a high-performance workstation.

The system designer or software developer moving into symbolic processing soon finds that choosing an intelligent workstation requires a knowledge of the unique features of Lisp architecture. These features include the extensive use of multiple variables and data types, the use of garbage collectors to control data traffic, a linear address space that uses a large number of data structures, and an extended data-tagging system field.

A Lisp machine that effectively implements these features in hardware takes advantage of the idiosyncrasies of the Lisp environment. To implement the internal Lisp features, the Lisp machine must have a very large microcode memory. It must also have a variety of extra internal registers to take care of data tagging. Finally, the machine must have a dual-pointer stack to track data movement within the large abstract address.

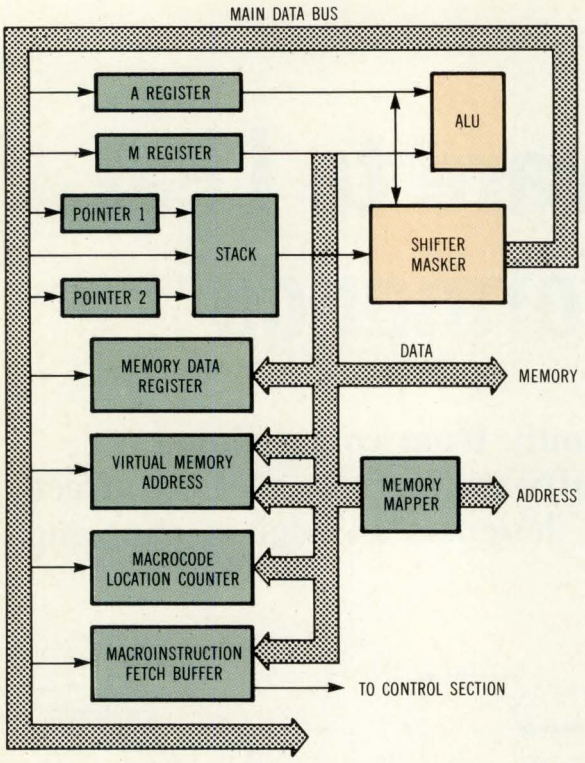
In Fortran or Pascal, a variable can hold only a single data type. In Lisp, many basic operations work for several types of data, and any variable can hold any type of data. For example, there is only one add operation in Lisp (called "PLUS" or "+"), which works for all valid numerical representations, such as fixed and floating point. Attempts to operate on invalid data types are detected and aborted with an error message. One example of an invalid operation is adding a list to an integer. Since this operation is done at run time, rather than during compilation, the hardware architecture must provide a method of carrying the data type along with the data value. This provision is not necessary for conventional architectures.



Because the programmer works symbolically in a large address space, there is less inherent concern for memory management issues in the Lisp environment. But since memory management is still required to implement the programmer's large

Gene Matthews, Glenn Manuel and Steven Krueger

Matthews is director of the Symbolic Computing Lab for Texas Instruments (Dallas, TX). Manuel is a technical contributer also for TI. Krueger is a senior member of TI's technical staff.



In the data paths of a simple Lisp machine, register files (A and M) and a data bus drive the ALU, which in turn drives the main data bus. The main data bus drives the control section of the Lisp machine and feeds data back to the ALU inputs.

abstract address space in a real-world machine that has limited physical memory space, a variety of techniques must be used to implement memory management. These techniques are collectively known as garbage collection.

Managing memory in Lisp

The Lisp environment is memory-intensive, and memory management techniques help determine Lisp performance. Because of the way that Lisp manipulates lists, for example, many small chunks of memory are used for short periods of time. A large amount of list handling can quickly fragment the Lisp machine's memory and exhaust its memory space.

To ease the handling of large lists, a Lisp system has a built-in garbage collector. Periodically, the garbage collector reclaims chunks of memory that are no longer used. In addition, most garbage collectors perform compaction. This process rearranges the blocks of data in memory, removes fragmentation and restores memory to a linear, contiguous order.

To maintain the integrity of the data structures during the compaction process, garbage collectors need to keep track of data movement so that pointers to the data can be updated. One method of tracking data is to place a pointer's new location in its old location when the pointer is moved. The old location is marked as a forwarding pointer, rather than a normal pointer. As a protective feature, any attempt to use the old location is automatically redirected to the location to which the data has been moved.

Tracking the data

When this technique is used, the pointer must be identified as a normal or forwarding pointer. In addition, the presence of forwarding pointers complicates memory access. Each time a pointer is accessed, it must be checked to see if it is a normal or forwarding pointer. Since the pointer can be either a forwarding or normal pointer, the checking process must be repeated.

If it is a forwarding pointer, the pointer must be followed to where it points. When the end of the forwarding pointer chain is reached, the original pointer which started the memory access must be updated with the final pointer value. This enables subsequent memory access to go directly to the new location, instead of going through the chain of forwarding pointers.

Since the process of checking for forwarding pointers adds to the overhead of each memory access, it must be as efficient as possible. A Lisp hardware architecture must provide support for forwarding pointers.

A linear address space is the critical feature of an efficient Lisp architecture. To provide the best possible implementation of the Lisp address space, logical memory should be as large, linear and uniform as possible. Memory address spaces with partitions imposed by logical address limitations (such as base registers and segments) complicate efficient memory management for a variety of reasons.

Overcoming management obstacles

First, because Lisp pointers are actually memory addresses, they must be extended to include the memory partition. Unfortunately, this procedure uses memory inefficiently, since pointers are longer. It also slows execution because of a more complex pointer format. The base register, for example, must be checked. It may even have to be changed for every access.

Second, symbolic processing programs tend to use large numbers of data structures. If the size of

a data structure exceeds the size of a memory partition, the data structure access will be very inefficient. Finally, since garbage collection must be done on the entire address space, the garbage collector itself can add significant overhead to program execution (depending upon the algorithm used). This problem is also aggravated by the base register changes that are required as the garbage collector moves through memory.

Tagging the data

Because of the need to reorganize memory via garbage collection, and because Lisp contains both pointers and data, data must be tagged. In a tagged data architecture, each memory word contains both the data and a tag which indicates the data type. In addition to indicating the data type, tags differentiate between pointers and actual data. Special bits in the tag may also be required for memory management to provide forwarding pointers and garbage collector status bits. These status bits indicate whether or not a word can be reclaimed.

Another requirement for Lisp implementation is a flexible architecture. Since Lisp is an interactive and dynamic language, the architecture must be easy to modify, so that it can track and support changes and extensions in the language. The key to this flexible architecture is effectively modifying the architecture in accordance with the microcode and microprogramming.

A simple Lisp machine

A Lisp machine has a very large microcode memory (typically 16 kbytes × 50 to 60 bits). In such a machine, Lisp source code is not executed directly, but is compiled down to a virtual machine code (macrocode). The microcode then interprets the macrocode. To allow interpreted execution of Lisp source code, a compiled Lisp interpreter program is always resident in the system. Many of the internal Lisp features, such as the virtual memory management and garbage collection, are implemented in microcode. Since the microcode is resident in the system, changes, extensions and modifications can be easily made.

In the data paths of a simple Lisp machine, a register file drives one input of the ALU, and the other ALU input is driven by a bus. On this bus is a second register file, stack cache, virtual memory address register, memory data register, Lisp macrocode location counter and macrocode instruction buffer. The ALU output drives the machine's main data bus. This bus feeds data back to the data path sources and to parts of the control section.

In parallel with the ALU is a shifter/masker which also drives the main data bus. Each machine instruction uses the ALU or the shifter/masker to perform its operation. To eliminate a separate tag processor, tags are implemented as the top few bits of the data word and the normal data paths are used for tag processing. The shifter/masker is added to make tag processing, which involves many bit manipulations, more efficient.

The shifter/masker consists of a barrel shifter and a full-width programmable AND gate. The barrel shifter shifts a data word any number of bits in one operation, and the masker allows any number of bits in the shifted word to be masked off. It then combines the masked word with a back-

Real world implementations

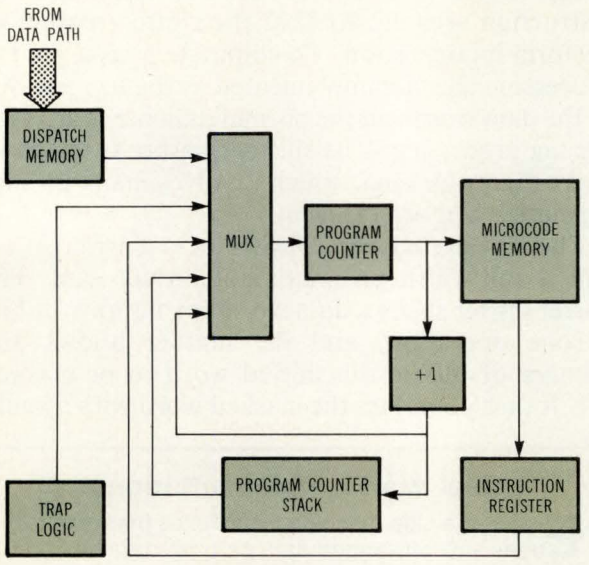
Explorer, a Lisp machine from Texas Instruments, uses a 32-bit tagged data path (25 data bits and 7 tag bits). The 25-bit pointers provide a 128-Mbyte virtual address space. Memory accesses, however, can bypass the virtual address mapping hardware, which allows full use of the 32-bit (4-Gbyte) logical address space.

The Explorer data paths use two register files, an ALU, a shifter/masker and 1-kbyte words of stack cache that include two pointers, the Lisp macrocode location counter, the memory data register, the virtual memory address register and the macrocode instruction buffer. Additional hardware improves tag processing. A tag comparator across the inputs of the ALU implements a tag equal check on the two operands of an instruction. In addition, a tag classifier RAM (instead of a dispatch memory) handles generic tag checking.

The control section features a 16-kbyte × 56-bit microcode memory, a program counter stack with a depth of 64 bits and a 4-kword dispatch memory. A 2-kword microcode PROM is used for booting and self-test. Circuitry is included to modify the instruction register of the next instruction. This design provides for dynamic instruction modification, such as one-instruction calculation and the ability to set up the shifter/masker control bits for the next instruction.

Designers investigating Lisp machines for symbolic processing will see a Lisp architecture in VLSI. Texas Instruments is developing a 2-micron CMOS generic Lisp processor architecture called Compact Lisp Machine. Although this chip will not implement the microcode memory and the memory mapper, it will include some features not in the generic architecture, such as a normalizer that improves floating-point performance.

Since Compact Lisp Machine is too fast (40-MHz clock) to be supported by standard DRAM, the data cache is accessed in parallel with the mapper. In this design, for a cache hit, the data is returned immediately, with the memory operation aborted. For a miss, the memory operation in progress is completed and memory is accessed over the Lisp machine system bus (NuBus).



The decoded outputs of the instruction register are the control signals that bring about the execution of each instruction. The program counter is selected from the previous counter (incremented), a field of the instruction register, a program counter stack, trap logic or an output of dispatch memory.

ground word. The shifter and masker perform the load byte operation, the deposit byte operation and the selective deposit operation. Load byte, which is useful for extracting a tag from a data word, replaces a specific number of the lowest bits of a word with a field of the same length. (This field can be located anywhere within another word.) Deposit byte uses this same number of lowest bits to replace the same number of bits in another word. This is useful for storing a tag with a data word.

Selective deposit replaces a field of length n in one word with the bits in the same location of another word. This is useful for copying tags from one word to another. Finally, the shifter/masker is useful in processing the data portion of the word. The result is greatly improved performance for many data operations, especially the extensive bit manipulations needed for bit-mapped graphics.

Stack aids performance

The Lisp machine is a stack machine—each call to a function causes a frame to be added to the stack. The frame contains information and storage space for that invocation of the function, including storage of local variables. When a function is executed, its frame is removed from the stack and replaced by the frame of the next function. To improve the performance of stack operations, a hardware stack cache is provided.

Two pointers are used to index the stack. One always points to the top of the stack for conventional push and pop operations. The other can point anywhere, and is useful for accessing local variables within the stack. This hardware stack is used as the top of a large memory-resident stack that is managed by the microcode.

The abstract address space of the Lisp machine is implemented as demand-paged virtual memory. Memory mapping hardware translates the CPU's virtual addresses into the memory's physical addresses. Memory access is performed independent of the CPU. This independent operation lets the CPU start a read request before it actually needs the data, nearly eliminating wait states. Lisp macrocode is contained in the virtual address space.

Maintaining control

In the control section of the basic Lisp machine, microcode memory is addressed by the program counter, and its output is latched in the instruction register. The decoded outputs of the instruction register are the control signals, which cause execution of each instruction. The program counter is selected from either the previous counter, a field of the instruction register (for jump instructions), a program counter stack (for subroutine calls or returns), a trap address generator or the output of dispatch memory.

Dispatch memory is used for macroinstruction decoding and generic operations. It contains the starting addresses of specific microcode routines. The dispatch memory address is selected from one of two sources. For example, using the macrocode instruction buffer as the source, the next macroinstruction is decoded by jumping to the microcode routine which implements it. Using the shifter/masker output as the source, the tag can cause a jump to the microcode routine for handling a particular data type (this function is used for generic operations). The shifter/masker output can also be used for other more specialized applications. **CD**