Lisp Hardware Architecture: The Explorer II and Beyond

Patrick H. Dussud
Texas Instruments Incorporated
Data Systems Group
Austin, Texas

January 15, 1987

1 Motivations for Lisp Hardware

1.1 Language Requirements

Common Lisp, as described in [1] has several characteristics that make it harder to implement as efficiently as conventional computing languages on ordinary architectures. Those characteristics motivate the design of specialized hardware optimized for Lisp execution.

Dynamic Typing and Object Reference

Lisp is a run-time typed language. Variables do not have a type; the type of a variable depends on the object it refers to. The type information has to be kept along with the object reference or within the object itself. During execution, the machine must maintain this typing information. There can be multiple references to an object but there is only one stored representation of this object. Typically, Lisp execution requires more references movement than data movement.

Functional Language

Most operations in Lisp are implemented as function calls. Traditional Lisp programs make heavy use of recursion, so fast execution of function calling and return is essential for good performance.

Generic operations

In Common Lisp, many of its functions are not specific to a type of object, but to a collection of types. There is only one ADD function: +. It works on every valid representation of a number (fixnum, float, ratio...). It is also desirable to detect invalid operations (such as adding a list to a floating point number).

Object-oriented Support

Several implementations of Common Lisp are extended by a Object-oriented system. A standard object system for Common Lisp , *CLOS* [2], is being developed. CLOS extends the concept of generic functions. Programmers can define their own types and write type-specific code (methods) for generic functions. The implementation, typically at runtime, selects the code to execute the generic function based on the type of its arguments.

Memory Management

Lisp frees the programmer from memory management tasks such as deallocating data storage and memory compaction. Conceptually, Lisp executes in a large, single address space. Lisp memory management systems include a garbage collector, whose task is to get rid of the objects no longer accessible by the program and optimize the placement of usable objects in order to maximize the locality of reference for memory intensive programs. This is seen as having a large impact on performance for nontrivial Lisp programs.

1.2 Program Development Requirements

Lisp hardware machines traditionally were program development oriented. Lisp program development stresses some other features of a Lisp system. It is worth mentionning that often, Lisp hardware machines are configured as Lisp development workstations.

Machine Reactivity

Developing a Lisp program is largely an interactive task. It is desirable to work in an integrated environment where the editor, compiler, debugger and program can access the same data. This requires a large address space. 100 MB of virtual space is not unusual for a high end Lisp workstation.

Another key feature is the reactivity of the machine. During interactive edit and debug sessions, the machine response time must be kept under one second or the programmer will get frustrated. This puts severe constraints on the memory management system.

Compiler Performance and debugging

A fast compiler is an important feature on a Lisp workstation, where compilation is done frequently. If the debugger supports symbolic debugging of compiled code, programmers will tend to debug their code in compiled form which will maximize performance and let the programmer get to the next bug quickly. Another reason for compiled code debugging is that a bug in a program can manifest itself inside of a system library function call, implemented as a Lisp compiled function. The programmer will get some clues if he can examine the state of the system right inside the library function call.

Code engineering

In order to maximize programmer productivity, it is desirable to let him concentrate on the program semantics instead of seeding the code with type declarations and compiler directives in order to boost the performance. The programmer's time should be spent getting the program right, and the implementation should make it run fast.

Code Safety

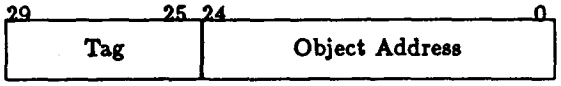
Complex programs are not immune to bugs, even in their production state. The error should be detected as soon as possible in order to maintain the integrity of the memory system and allow the programmer to take corrective action easily. This means that the system should perform runtime type checking on object access, even if the compiler could infer the object type.

2 The Explorer Virtual Machine

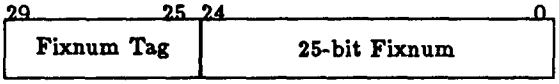
The Explorer Virtual Machine is an architecture model for the Explorer family. It describes the memory model, the instruction set, the calling discipline, and the storage management system. It is virtual in the sense that it can be emulated in software, or implemented in hardware. It is also concrete because it is what is presented to the programmer at the lowest level. On both members of the Explorer family (Explorer and Explorer II), the virtual machine is implemented partially by the microcode, partially directly by the hardware.

Memory Model

The memory is organized in 32 bit words and is *tagged*. The object type is stored along with the object reference itself. The tag is 5 bits wide, allowing 32 primitive representation types. More elaborate data types are built, but they are extension of those primitive ones and are stored along with the object. A reference to an object can be *by address* where the reference contains the address of the stored object it refers to. The address field is 25 bits wide, giving an effective address space of 128M Bytes. A reference can be *immediate*, where the object in contained in the reference. Immediate object reference is used for 25 bit fixnum, short-float and character types.



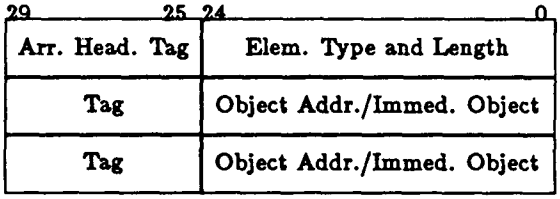
Object Reference By Address



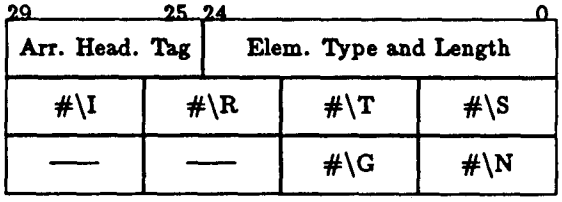
Immediate Object Reference

Stored Objects

Most stored objects begin with a tagged word called the *Header Word*. It further specifies the type and access mode of the object. There are two kinds of stored objects: the *boxed* object and *partially tagged* (or *unboxed*) object. In a *boxed* object, all the words of the object are tagged. In an *unboxed* object, some words are pure data. They don't contain a tag. This allows for storage of 32 bit integer arrays and a more compact representation of uniform data structure.



Boxed Array



Unboxed Array: "STRING"

Forwarding Pointers

Forwarding pointers redirect the reference to an object transparently. They are used internally by the system to grow arrays, rearrange lists, and by the incremental garbage collector.

Compact List Storage

The upper two bits of each word used for CONS storage make up the *cdr code* [5]. In traditonal Lisp systems, CONS objects have two words, one for the CAR of the CONS, one for the CDR. When the conses make up a list, half of the words point to the next CONS of the list. On

the Explorer, a list can be stored as one block of memory, using the *cdr code* to interpret the next word of the list. The *cdr code* can take four values:

- *Next:* The next word is the next element of the list.
- *End:* This is the last word of the list. Its CDR is NIL.
- *Cons:* The next word is the CDR.
- *Error:* This is the last word of a CONS. Its CDR is illegal.

A RPLACD on a cdr coded list will cause a forwarding pointer to redirect the rest of the list to the new CDR.

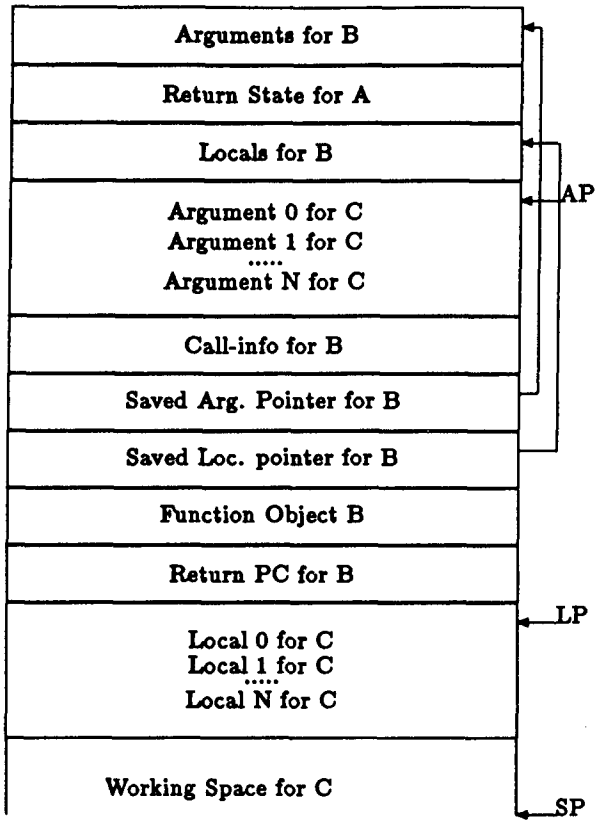
31 29	25 24	0
Next	Symbol Tag.	Addr. of FOO
Next	Fixnum Tag	34
End	Symbol Tag	Addr. of BAR

Cdr Coded (FOO 34 BAR)

The design of the memory model allows fast runtime type checking because of its large tag space. Since every stored object is tagged, the garbage collector has less context information to keep. It can be driven by the tag values. However, the presence of partially tagged objects complicates the garbage collector. Partially tagged object are used in order to achieve acceptable storage compactness on string and numeric arrays. The address space (128M Byte) is considered small by today's standards. This decision was made in order to use an industry standard bus for memory access (NuBus).

Execution Model

The machine uses a stack or *pdl* for its execution. There is no visible register. Each function call has a *frame* allocated. It contains the arguments, some return information, the local variables, and the temporary space of the current function. The following diagram represents the stack in a state where the current, executing function is C, which was called by B, itself called by A.



Call Frame Layout

Instructions are divided in two classes:

- *Mainops:* They can have 0 or 1 address. 1 address instructions include data movement (PUSH and POP). They compute the effective address of their operand by adding an offset to base address, which can be:
  - The beginning of the passed arguments (AP).
  - The beginning of the current local variables (LP).
  - The top of the stack (SP).
  - The current Function data constant portion.
  - a higher lexical context.
  - A Flavor Instance.
- *Auxops, Miscops:* These instructions behave like ordinary Lisp functions. They get their arguments on top of the stack and leave their result on top of the stack. These instructions include a number of list primitives, specially optimized for speed (MEMBER, ASSOC...).

The design of the instruction set allows for fast compilation because the instruction set is optimized for Lisp. It supports generic primitive functions. The compiler can ignore type declarations and does not have to generate instructions for runtime datatype checks.

The design of the instruction set allows for easy debugging of compiled code because most instructions map directly into Lisp functions. With a minimum of experience, programmers can read disassembled code.

Function Calling

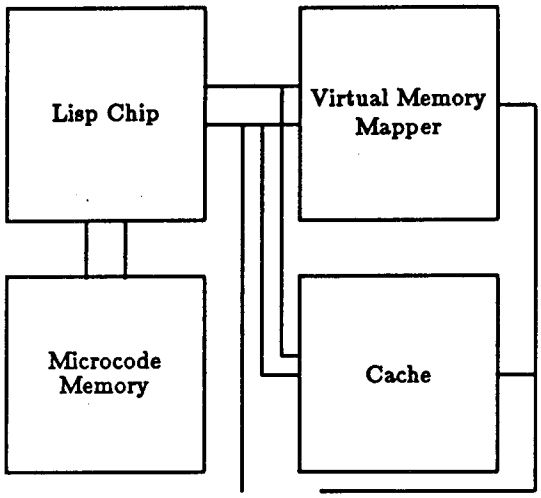
The virtual machine supports fast function calling in simple cases. It involves accessing the function argument-descriptor to check for the number of required arguments, taking care of the &REST [1] argument, pushing the saved context for the caller and branching to the first instruction of the callee. The optional argument defaulting and key argument decoding is done by some code generated in the called function. The RETURN handles the following: It makes sure that the returned values are not allocated on the stack as temporary structure, it pops off the arguments and the saved context, and pushes the returned value(s) on the stack. There are several CALL and RETURN instructions depending upon the complexity of the call and the return.

Memory Management

The garbage collector is incremental [9] and generational [6]. This neccessitates two runtime mechanisms called the *Read Barrier* and the *Write barrier*. When the machine reads an object that needs to be relocated, an exception is raised by the machine and when the access is resumed, the object has been relocated and a forwarding pointer is stored in its place. A special forwarding pointer is inserted during an object write access, when an object of a higher generation refers to an object of a younger generation. On generation collection, the generation roots are contained in these forwarding pointers. The system optimizes the placement of objects based on their inactivity. Objects that have not been accessed recently are moved to inactive regions.

3 The Explorer II CPU

The Explorer II CPU hardware consists of a Explorer Lisp Microprocessor [8], an external writable microcode memory, a virtual memory mapper and a high speed memory cache.



Explorer II CPU

The memory mapper is reponsible for implementing the page fault and read/write barrier exceptions. The *auto transporter* is able to follow forwarding pointers without CPU intervention .This makes the incremental garbage collector very efficient.

Since the Explorer Lisp Microprocessor is too fast for the NuBus memory, a cache has been added. The cache and mapper are accessed in parallel. On read, when the reference is in the cache, the memory mapping operation is aborted. For a cache miss, the mapper will access the memory over the NuBus. On memory write, the data is always written over the NuBus, even for a cache hit.

The Explorer Lisp Microprocessor

The Explorer Lisp Microprocessor has been optimized in order to implement the Explorer virtual machine.

Since the Virtual Machine executes mainly on the stack, the top 1K words are cached on chip. Overflow and underflow of the cache are checked and handled during CALL and RETURN.

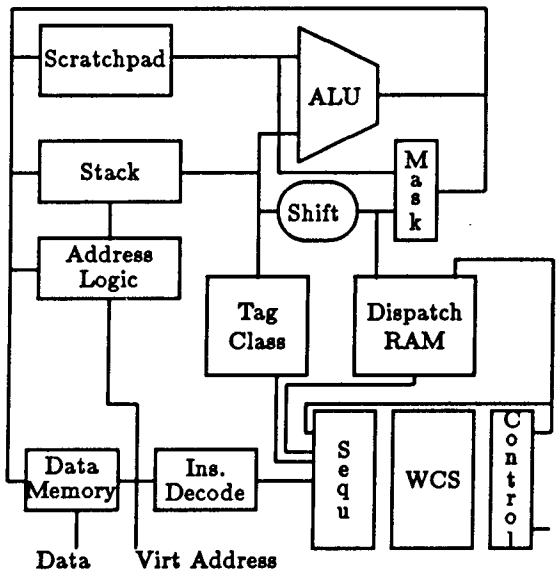
The Address bases of the instruction are materialized by on chip registers and the effective address computation is done in hardware, with prefetching on external memory access.

The Explorer Lisp Microprocessor has extensive support for tag oriented computation:

- The 32 bit ALU is conventional. It can operate in 32 bit mode, or in a mode where the cdr code and tag bits are ignored for the operation, and generated in the result in the same clock cycle.
- The ALU is complemented by a masker/barrel shifter able to deposit any number of contiguous bits in a 32 bit word taken from a 64 bit source. This masker/shifter is used for tag extraction as well as bit oriented and graphics instructions.

- Multiway (up to 128) branch is accomplished in one cycle using a large dispatch memory. This is used for generic operation dispatch.
- For runtime type checking, a *tag class* memory checks the tag portion of the operands during ALU instructions and can generate a jump if the operands don't belong to a specific tag class. This is done during the ALU cycle.

Measurements of execution show that about 50% of the executed instructions are one clock cycle instructions.



Lisp Chip Diagram

4 The Ivory Processor

The Symbolics Ivory Chip [10] is a VLSI custom microprocessor which implements a complete Lisp CPU. It contains the microcode ROM, a virtual memory mapper and a memory controller. Its 40 bit word (2 cdr code bits , 6 tag bits and 32 bit address) allows for a much larger address space than the Explorer Lisp Microprocessor. Industry standard 32 bit address and floating point format are contained inside the reference field. A co-processor handles the floating point cases of generic instructions. The I-machine virtual machine [11] is derived from the 3600 machine architecture [7]. Ivory is on an evolution path where two trends can be seen:

- Shrinkage of the microcode over time, and migration of low level functions into Lisp. The 3600 implements the device drivers, page faults and some other low level operating system functions in Lisp. The Ivory goes beyond and implements some complex instructions (BITBLT, CONS) in Lisp using special primitives.
- Migration of functions from separate hardware into the processor. The virtual address mapping, memory control and memory error correction are implemented on chip.

5 Future directions

The future of specialized hardware lies in its ability to keep pace with the development of conventional processor technology. For this, some issues are of primary importance.

- As the IC technology evolves, the VSLI design of the processor must be adapted to new VSLI processes. In addition to having an adaptable design, this issue requires powerful design tools which allows for quick target technology adaptation.
- The architecture of the processor must support the gain in speed due to IC technology improvement. The critical data paths of the processor must stay on chip, because off chip communication is going to scale at a slower pace than the internal circuitry. The microcode program memory is on one of these critical paths. It will be essential to implement it on chip or get rid of it. Some other areas of integration are the data cache and the floating point hardware.
- The issue of integration with conventional languages and industry standard operating systems will be more and more important. This can be addressed by having the Lisp processor run conventional languages, or by using a co-processor approach.

6 Summary

This paper has presented some motivation for specialized Lisp CPU, described the Explorer virtual machine, the Explorer II CPU, and the Symbolics Ivory CPU. Some issues critical to the future of Lisp specialized processors are listed.

References

[1] Guy L. Steele Jr. *Common Lisp,* Digital Press, 1984.

[2] *CLOS* ANSI X3J13 Standing Document 87-002.

[3] Linda G. DeMichiel, Richard P. Gabriel *The Common Lisp Object System: An Overview,* ECOOP'87 AFCET BIGRE + GLOBULE n° 54.

[4] Richard P. Gabriel *Performance and Evaluation of Lisp Systems* MIT Press 1985.

[5] D. G. Bobrow and D. W. Clark, "Compact Encoding of List Structures", *ACM Transactions on Programming Languages and Systems,* pp 266-286.

[6] David A. Moon "Garbage Collection in Large Lisp System", *1984 ACM Symposium on Lisp and Functional Programming,* pp 235-246.

[7] David A. Moon "Architecture of the Symbolics 3600", *Proceedings of the 12th Symposium on Computer Architecture, June 1985, pp 76-83*

[8] Patrick Bosshart "A 553K-Transistor LISP Processor Chip", *IEEE Journal of Solid-State Circuits, VOL sc-22, nr 5, October 1987.*

[9] H. G. Baker, "List Processing in Real Time on a Serial Computer", *Communications of the ACM 21, 4(April 1978) 280-294.*

[10] Clark Baker, David Chan, Jim Cherry, Alan Corry, Greg Efland,Bruce Edwards, Mark Matson, Henry Minsky, Eric Nestler, Kalman Reti, David Sarrazin, Charles Sommer, David Tan and Neil Weste, "The Symbolics Ivory Processor", *IEEE International Conference on Computer Design '87*

[11] Bruce Edwards, Greg Efland, and Neil Weiste "The Symbolics I-Machine Architecture", *IEEE International Conference on Computer Design '87*

[12] *Explorer System Design Notes* TI PN 2243028-001A.

QUESTION:
What is the difference between a squeamish housekeeper and the CLOS metalanguage protocol?

ANSWER:
One really objects to finding dead flies on the window sill after using the Flit sprayer a bit.

The other is silly -- it redefines dead window objects on the fly, but then BITBLT hasn't a prayer.

Richard P. Gabriel
and Guy L. Steele Jr.
March 16, 1988