A Lisp Oriented Architecture

by

John W.F. McClain

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Master of Science in Electrical Engineering and Computer Science

and

Bachelor of Science in Electrical Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 1994

Signature of Author ....
Department of Electrical Engineering and Computer Science
August 5th, 1994

[illegible]

Certified by ....[illegible].......................
Thomas F. Knight Jr.
Principal Research Scientist
Thesis Supervisor

[illegible]

Accepted by ....[illegible].......................
Frederic R. Morgenthaler
Chairman, Departmental Committee on Graduate Students

A Lisp Oriented Architecture

by

John W.F. McClain

Submitted to the Department of Electrical Engineering and Computer Science
on August 5th, 1994, in partial fulfillment of the
requirements for the degrees of
Master of Science in Electrical Engineering and Computer Science
and
Bachelor of Science in Electrical Engineering

Abstract

In this thesis I describe LOOP, a new architecture for the efficient execution of programs written in Lisp like languages. LOOP allows Lisp programs to run at high speed without sacrificing safety or ease of programming. LOOP is a 64 bit, long instruction word architecture with support for generic arithmetic, 64 bit tagged IEEE floats, low cost fine grained read and write barriers, and fast traps. I make estimates for how much these Lisp specific features cost and how much they may speed up the execution of programs written in Lisp.

Keywords: Lisp, LIW, computer architecture.

Thesis Supervisor: Thomas F. Knight Jr.
Title: Principal Research Scientist

Acknowledgments

First, I would like to thank those who shaped me long before knew Lisp: my parents John and Kay, who supported and nurtured me all these years. More recently, my wife, Jennifer, and her parents Ed and Paulette Schwartz have also helped me through the difficult parts of being a graduate student.

In addition to my family, I also thank my good friends Matt Calef, Alex Dogon, Mark Donelan, Andrew Draper, and John Huebner, who helped me through high school and in the world beyond. Also, thanks to Mark's wife Anne, and Maxwell and Natasha.

More recently, many people have had a direct influence on the thesis itself. I had many hours of of interesting computer architecture, operating system, and compiler discussions with André DeHon who also kept me from making LOOP the CISCest architecture since the Intel 432 or Burroughs' B6500. I will miss these discussions and 7th AI lunches the most.

Thanks also to Olin Shivers who pointed me towards the IBM VLIW research and wrote one of the more enjoyable PhD. theses [43] I read in the course of my research.

Robert MacLachlan shared two unpublished manuscripts that described some of his ideas on computer architectures for Lisp.

Thanks to Robert Thau for being the local heretic[1] and introducing the AI Lab to the World Wide Web which is destined to become one of the greatest research tools of all time. He also helped me figure out why the user cache was really not needed.

John Mallery had a long talk with me one night and gave me the push I needed to get out of my first set of thesis doldrums.

Moon [36] and Hauck and Dent's [19] articles helped me realize that computer architectures don't all need 32 registers and a pair of memory instructions and got me thinking about how a modern Lisp Machine might look in the first place.

Finally thanks to Tom Knight, my thesis advisor, who suggested the idea of turning the answerer to that question into a project in the first place.

[1]Which I mean in the best possible way.

Contents

List of Figures

Chapter 1

Introduction

This thesis explores the possibilities for the design of a new Lisp oriented architecture. One can get good Lisp performance on stock hardware but you often have to twist your programming style and/or disable safety to get this performance, eliminating much of the reason for programming in Lisp in the first place. The architecture described in this thesis attempts to let programmers get good performance without a lot of declarations or turning safety off. Although I have not hesitated to add features that only help Lisp the architecture is not that different from those found in modern main line micro-processors.

The question of what makes a processor good for Lisp is interesting because Lisp like languages are becoming more important as programs get more complex and other languages begin to look more and more like Lisp. Also there has been little research in the past seven years into what makes a good Lisp processor, but in that time the fields of VLSI chips design and computer architecture have undergone tremendous changes.

The proposed architecture, LOOP[1] is a four wide long instruction word (LIW) processor. It is not expected that implementations will do a lot of dynamic scheduling. Instead we rely on the compiler to find and exploit most of the instruction level parallelism. We also rely on the compiler for safety when its felt that this does not compromise speed.

1.1 How Lisp is Different

Lisp differs from C and FORTRAN in several important ways. It is these differences which make current main line computer architectures a somewhat poor target for Lisp.

1.1.1 Dynamic Typing

Variables in most Lisps are not statically typed. Instead of associating the type with the variable it is associated with the data value. As a result Lisp programs have

[1]Which stands for Lisp Oriented Processor.

to make many runtime type checks. This hits integers in Lisp systems particularly hard since in general a type check needs to be made before every integer operation, potentially doubling or tripling the time it takes to perform simple integer operations[2]. The type information also has to be stored somewhere. Floating point computations are hit very hard by this second problem, since the IEEE floating point format makes no allowance for type tags. Lisp implementations often have to resort to storing floats on the heap and referencing them by pointer instead. Both problems can be attenuated with declarations or by turning safety off, however, both options remove much of the benefit of for programing in Lisp in the first place. We address these problems in the traditional Lisp Machine [29, 36] way by adding hardware to make type checks in parallel with machine operations and by making the words wider than normal to accommodate type tags.

1.1.2 Memory Allocation

Lisp programs are often written in a functional style which results in more memory allocation than most C and FORTRAN programs. C and FORTRAN programs tend to use the same memory over and over again under explicit programmer control. Diwan, Tarditi, and Moss [14] find that allocating stores with subblock valid bits can significantly increase the effectiveness of data caches in the face of allocation intensive programs. We adopt their findings by giving LOOP both allocating stores and subblock valid bits.

1.1.3 Garbage Collection

Lisp programs can be written in a functional style because most Lisp systems have a garbage collector that periodically goes through memory and finds all data that can still be reached and separates it from the "dead" data. Basic garbage collectors have bad locality and/or fragment memory, cause frequent pauses in program execution which can be annoying to the user, and add significant runtime overhead. The runtime cost and locality concerns can be reduced by generational garbage collectors [35]. Unfortunately generational garbage collectors require associating a "barrier" with every non-allocating heap store. The pauses can be dealt with by making the collector "incremental" but this requires associating a barrier with every heap read and write, which adds substantial runtime overhead. LOOPs translation look aside buffer provides support for low cost, fine grained read and write barriers which can be used to implement generational and incremental garbage collectors at greatly reduced expense.

[2] Actually its not uncommon in unoptimized Lisp systems/programs to find a simple integer add taking many 10s of instructions.

1.1.4 Pointer Intensive Data Structures

Garbage collectors also make it easier for programers to use pointer intensive and dynamic data structures, resulting in an increased demand for memory bandwidth. LOOP provides wide loads and stores that can fetch or write 2 or 4 words at a time. Additionally I suggest a hardware feature that may interact with copying generational garbage collectors to increase the on chip hit rate and increase available memory bandwidth.

1.1.5 Higher Trap Frequency

Unlike many main line computer systems in which traps are only expected to handle infrequent and heavy weight events like page traps, or terminal events like overflows and memory access errors, many Lisp systems use traps to handle more frequent, light weight events. For example, in many Lisp systems when an integer overflows the capacity of the basic machine word the system replaces it with an arbitrary precision integer (aka "bignum"). The cost of creating such an integer is much less than taking a page trap. LOOP adopts many of the ideas from Johnson [24] to support fast trap handling and instruction emulation including shadow registers and user trap handlers. LOOP also provides multiple register contexts so registers don't have to be saved/restored on traps.

1.1.6 More Function Calls

Lisp programs generally have more function calls than C and FORTRAN programs. However, I have been reluctant to spend time trying to come up with hardware or architectural features to optimize procedures calls. Part of this reluctance come from the fact that Lisp procedure calling conventions and the algorithms that implement them can be very complex, and it is not at all clear which is the "best" one that deserves to be cast in silicon. Another problem is that different flavors of Lisp have very different features in there calling conventions. Common Lisp [46] is lexically scoped, has rest, optional, and keyword parameters. Scheme [8] is also lexically scoped but has only rest parameters. Yet Scheme also has call/cc which puts unique constraints on procedure calling convention. An architectural feature that supports one well is likely to be unhelpful for the other. And if we do find a feature that helps Common Lisp and Scheme it is unlikely to help GNU Emacs Lisp [31] which has rest and optional parameters but uses dynamic scoping.

1.1.7 Unpredictable Control Flow

Because most Lisps have first class procedures and/or generic function based object systems, control flow analysis is very difficult in Lisp programs. LOOP does very little to address this, in fact because its an LIW machine and statically scheduled the problem gets worse not better.

In addition to the features that help Lisp specifically LOOP has several features that should make programs faster in general. I have not emphasized these in the text but I will mention them briefly here. LOOP gives the program a lot of control over the cache allowing the program to use the cache in a pro-active manner. LOOP supports speculative execution through a mechanism that allows most exceptions to be handled by producing a specialty tagged object instead of passing control to an exception handler. LOOP also has several performance counters to support profiling.

In the rest of this chapter we will talk about past Lisp oriented architectures. In chapter 2 we describe LOOP's instruction set architecture. In chapter 3 we talk about how LOOP might be implemented and make some estimates of how big it might be in terms of chip area and how much the individual Lisp features cost. In chapter 4 we evaluate how much performance improvement these various Lisp features might buy us. Finally in chapter 5 we make some concluding remarks and suggest some avenues for future research.

1.2 Past Work

This first machines designed from the ground up to run Lisp were started in the early 1970s at MIT. The second of these machines, the CADR, inspired a number of commercial spin-offs by LISP Machines Inc., Texas Instruments, and Symbolics. All of these machines were stacked based, tagged architectures with support for generic operations. The implementations were heavily microcoded. Many of them used cdr coding, forwarding pointers, included hardware support for incremental garbage collection, and had built in array bounds checking [29, 36].

During the late 70s XEROX PARC developed the Dorado as a follow on to the their Alto computer [38]. It was heavily micro-pipelined, microcoded and had a 16 bit word size at the microcode level. It had a fixed number of microcode *tasks* each with a fixed priority. Each cycle the system would switch to the active microcode task with the highest priority. Most of the tasks were used to implement I/O device handlers with most devices having one task each. Most of the machine's registers were duplicated so each microcode task had its own copy, making it practical to switch tasks every cycle. The lowest priority microcode task implemented an emulator for language specific instruction sets. There were instruction sets for BCPL, Mesa, Interlisp, and Smalltalk. About halfway through the project a instruction fetch unit was added which significantly improved the speed of instruction set emulation.

Since these machines were developed very few projects have been started with the aim of developing a new Lisp oriented architecture. One of the most recent attempts was the SPUR [50] architecture developed at the University of California, Berkeley in the mid 80s. The SPUR is a typical early RISC architecture with some additional features to support Lisp. Like most RISC architectures the SPUR, is a load/store machine with a large register file and simple instructions that do not have to be microcoded, are easy to pipeline and mostly can complete in a single cycle. Unlike most RISCs it has tag bits with some support for generic operations; it also has register windows which are provided to make procedure calls faster. Notably the SPUR had a separate floating point unit with separate floating point registers making full generic arithmetic difficult.

I have also recently been informed of a Lisp computer architecture effort in the University of Indiana department of Computer Science, but no details are available at this time.

Chapter 2

Architecture

LOOP is a 64 bit, Long Instruction Word (LIW), load/store, tagged architecture. Each instruction can specify up to 4 operations to be executed in parallel. Each of these operations is similar in complexity to instructions found in current RISC microprocessors.

We use the word "trap" to mean traps, exceptions, or interrupts. We use the phrase an "exception is raised" to indicate an exceptional condition has been detected and one of two course of action will be taken, either control will pass to the handler for that exception or a specially tagged error object will be produced as the result of this operation. We use the phrase "an exception is taken" to indicate that an exceptional condition had been detected and control will pass to the handler for that exception.

An "instruction" in the LOOP architecture specifies several "operations" that will be executed in parallel and are all referred to by the same PC value. Because the different operations in an instruction are largely orthogonal with respect to each other we will often refer to them in the same way one might refer to instructions in a non-LIW architecture.

2.1 Architecture Data Types

The basic LOOP data object is the machine word. Each word has two parts, a 64 bit data field and a 5 bit tag field. The tag field's size is nearly invisible at the architectural level because it is not directly addressable (see section 2.2.2 for details of the memory model.) Figure 2-1 shows the format of the basic machine word.

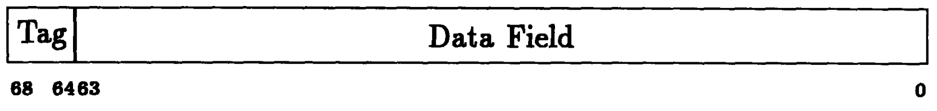

Figure 2-1: Basic Machine Word

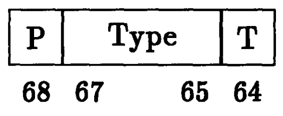

Figure 2-2: Detailed View of Tag Field

2.1.1 Tag Field

Four bits of the the tag field are used for the "type tag." They define how the data field is to be interpreted by the hardware. The hardware is responsible for checking if the types of an operation's operands are compatible with the operation and each other, raising an exception if they are not. The hardware is also responsible for generating the correct tag for the result.

The fifth bit is the "trap bit," most operations raise an exception when they operate on words which have their trap bit set. Only a WRTRAP operation will create a word with its trap bit set.

Figure 2-2 shows a detailed view of the tag field, P is the pointer bit, type is the remainder of the type field, and T is the trap bit.

2.1.2 Data Field

The architectural interpretation of the data field depends on the value of the type tag. Currently there are twelve hardware types defined. The lead bit of the type tag indicates if the word is pointer or non-pointer data.

Non-pointer Data

If the lead bit of the type tag is 0 then the word is non-pointer data. There are five types of architecturally defined non-pointer data.

Integer A type tag of 0 indicates the data field is a 64 bit twos complement integer. Predicate operations use integer one and zero to represent true and false respectively. When extracted from their words tags are represented as integers. Some instructions treat integers as unsigned.

IEEE Float A type tag of 1 indicates the data field is a 64 bit IEEE double precision floating point number.

Instruction A type tag of 2 indicates the data field holds part of an instruction. Instructions take two words (see section Section 2.3 Instruction Format for details.)

Untyped A type tag of 3 indicates the data field holds 64 bits of data that as yet does not have a type associated with it. Most operations raise an exception if performed on untyped words.

Error Object A type tag of 4 indicates the data field holds an error object. Error objects are created by the hardware when an exception is raised but not taken. Most operations raise an exception if they are performed on error objects. The error object type tag is also used for uninitialized words.

Extra Non-Pointer Type Tags Type tags 5, 6, and 7 are reserved for future use. Hardware should treat words with these tags as error objects.

Most of the possible uses for these tags are for variants of instructions. LIW processors have a problem with code density when the code has low instruction level parallelism. One of these tags could be allocated to be an instruction type whose individual operations would be executed serially, not in parallel.

MacLachlan [34] suggests the idea of "millicode" routines. One of these tags could be allocated to be an instruction type that is treated as a macro. When fetched it would be expanded into a sequence of 2 - 4 instructions, possibly replacing some or all of the source fields of the first instruction and some or all of the destination fields of the last instruction with bit fields from the original macro instruction. Branches would be allowed out of the macro, but not into it since conceptually the expanded instruction would all be at the same PC. Whether millicode would be useful would depend highly on the low level details of the compiler and runtime system. At least the first pass of a compiler and runtime system for LOOP would have to be written before this idea is fully flushed out. Knowing that such a millicode feature could be useful it should be possible to leave hooks for a "millicode expander" at the beginning of a design.

Pointer Types

If the leading bit of the type tag is 1 then the word holds a pointer to a word or double word. All memory reference instructions except STAx and LDAx (section 2.3.2 Memory Operations) mask out the low 3 bits of word pointers (low 4 bits for double word pointers) before adding them to the offset, which has its low 3 bits masked out, to form the effective address that is presented to the memory system. Operate operations (described in section 2.3.1) interpret all pointers as 64 bit byte pointers.

Basic Pointer A type tag of 8 indicates the data field is a word pointer.

Compound Number Pointer A type tag of 9 indicates the data field is a word pointer. The only difference from a basic pointer is compound number pointers raise a different exception when used as an operand for an arithmetic operation.

Pair Pointer A type tag of 10 indicates the data field is a double word pointer to a Lisp cons cell. Dereferencing a pair pointer with an offset after scaling that is not 0 or 8 bytes will raise an exception.

Small Compound Number Pointer A type tag of 11 indicates the data field is a double word pointer to a pair. The only difference from a pair pointer is small compound number pointers raise a different exception when used as an operand for an arithmetic operation.

Instruction Pointer A type tag of 12 indicates the data field is a double word pointer to a LOOP instruction. Only words of this type can be the destination of a jump, any other type will causes an exception to be taken. Memory references through instruction pointers go through the instruction cache if one is present.

Future Pointer A type tag of 13 indicates the data field is a word pointer. Any operate operation (section 2.3.1) can be performed on futures but memory operations on futures will raise an exception.

Physical Pointer A type tag of 14 indicates the data field is a word pointer to a physical location in memory. A dereference of this pointer will not go through the page mapping hardware. An exception will be taken if a user program tries to create a new physical pointer.

Extra Pointer Tags Type tag 15 is reserved for future use. One possible use would be error objects that contained a pointer, either to the instruction that raised the exception or to some block that held information on the exception. Another option would be a basic pointer type that is goes through a cache with a different policy, probably a different line size.

2.2 Storage Model

LOOP has three programmer visible storage areas: the register file, the main memory, and the special purpose registers.

2.2.1 Register File

There are 63 general purpose registers, R0 through R62. A 64th register, R63, is hardwired to integer 0. Each general purpose register holds one machine word, including tag bits. Additionally each register has carry and overflow condition code bits associated with it [12]. These bits get set/reset when the register is used as a destination for an instruction. If a register is written out to memory with any of its condition code bits set a user exception is taken.

2.2.2 Main Memory

Main memory is a flat, 2 exaword virtual address space. Memory can only be accessed through load/store operations, (discussed in section 2.3.2.) All memory access is on aligned word boundaries, but all address arithmetic is done in bytes with the tag field being invisible. Most memory reference operations mask out the low 3 bit of word pointers (low 4 bits of double word pointers) and the low 3 bits of the scaled offset before calculating the effective address. The combination of masking out the low bits and byte arithmetic leads to some interesting effects. For example: if R1 holds a word pointer to address A, LD(R1,1,R2) will load the word at A into R2. However, LD(R1,-1,R2) will load the the word before A. LD(R1,8,R2) will load the word after A, the tag is skipped.

Fine grained control of caches is provided to the programmer. Operations are provided to lock, unlock, flush, prefetch, and invalidate cache lines. The cache control instructions can be used with a write back cache to implement speculative stores. The lines you wish to do speculative stores to can be locked in, writes can then be done to these lines without destroying the copy in memory. The stores can be undone by invalidating the line and committed by unlocking the cache line.

Loads can be instructed to not cache the value being loaded. Variants of the basic store operation are provided that do not cache the store or do not load the rest of the line on a miss, instead just marking the other words in the line invalid. Non caching load and store operations provide a method of copying chunks of memory without polluting the cache. Not bringing the rest of the cache line in on stores has been shown by Diwan, Tarditi, and Moss [14] to significantly improve the performance of programs that are heap allocation intensive.

If an implementation uses an instruction cache, all references through instruction pointers must go through the instruction cache.

2.2.3 Special Purpose Registers

Values are moved between special purpose registers and the general purpose registers using special move operations. Because each operation field has one operand specifier that is 8 bits or larger accommodating 256 different special purpose register would not be difficult. Four different special purpose registers can be accessed each cycle. Access to some special purpose registers is privileged.

Writes to the special purpose registers are not visible until the instruction making the write has reached the commit stage of the pipe line.

Faculties accessed through the special purpose register are:

Processor Status Word (PSW) controls floating point rounding mode, which traps are enabled, which exception are taken, which produce error objects. The condition codes are *not* in the PSW.

Address of User Trap Vector

Address of System Trap Vector Writes of this register are privileged.

User Cycle Counter A 64 bit counter that increment once every 1 to 16 (implementation dependent) cycles while the machine is in user mode.

System Cycle Counter A 64 bit counter that increments once every 1 to 16 (implementation dependent) cycles.

Cache Miss Counters Various counters to count the number of cache misses. The policy implemented by these counters and the cycles counters may be implemented with programmable logic [10].

Page Table Root Register Points to root of virtual to physical page translation table. Writes of this register are privileged.

Page Privilege Table Root Register Points to root of virtual page privilege table. Writes of this register are privileged.

User Page Privilege Table Root Register Points to root of virtual user page privilege table.

Network Interface Registers Henry and Joerg [21] find that a hardware register based network interface can reduce much of the overhead of network communications. The exact details of such an interface, including how many registers are needed, would depend on the details of the network. One possibility is to provide this interface through programmable logic [11].

Jump Target Prediction Stack Writing pushes a new value on the stack. Reading pops the top of the stack into the specified register. Giving the user direct access to the jump prediction stack allow the user program to avoid the pipe line bubbles associated with calculated jumps if the target is known long enough before the jump is taken.

Trap Information Registers A set of 4 registers. When a trap is taken the PC of the trapping instruction, a word describing the trap and destination of the trapping operation, and if appropriate the operands of the operation that caused the trap are placed in these registers. See section 2.5 for details.

Overflow Bits Holds the overflow bits of all the general purpose registers, speeds up contexts switches. This register is only valid after a pipe drain instruction.

Carry Bits Holds the carry bits of all the general purpose registers, speeds up contexts switches. This register is only valid after a pipe drain instruction.

2.3 Instructions

Conceptually LOOP has 4 functional units, A1, A2, M, and C. A1, A2, and C each have two read ports from and one write port into the register file. M has 2 read ports and 1 read/write port. All four have one read/write port into the special purpose register file. All four functional units can perform the operations listed below in the section 2.3.1 Operates. A1 and A2 can also do multiplication, shifts, rotates. They can also do arithmetic operations on floating point data. M is connected to the memory system and can perform the load, store, cache control and operations described in Section 2.3.2 Memory Operations. C can read and write the PC to perform branches, jumps and conditional traps as described in section 2.3.3 Control Operations.

In the operation descriptions the first operand is called A, and comes from the functional unit's first register file read port. The second operand is called B and comes from the second read port or a sign extended immediate field encoded in the operation. A1, A2, and M each have an 8 bit immediate field, C has a 15 bit immediate field. The register being written, through the functional unit's write port, is called D.

Results are unspecified if more than one operation in an instruction tries to write the same register, unless that register is R63, in which case none of the writes have any effect.

If a functional unit is programmed to perform a operation it can not do (i.e. M doing an add with a floating point word) a user exception is raised.

Each instruction can specify one operation for each functional unit. Each functional unit can be programmed on a per operation basis to return an error object instead of taking an exception if an exception is raised (this option is not available for operations that do write a value into the register file or control operations.) The A1, A2 and M functional units can be programmed on a per operation basis to squash their result if C performs a control operation that goes in the non-predicted direction.

Some operations require more that one functional unit, and are known as "wide operations." They are described in section 2.3.4.

In general passing operands of the wrong type to an operation will result in an illegal type exception being raised.

2.3.1 Operates

Operates include arithmetic, logical, predicate, and move operations. All four functional units can perform all of these operations. All operations unless otherwise mentioned clear the condition bits and trap bit of D. All operations unless otherwise mentioned will raise a trap bit exception if the trap bit of either operand is set.

ADDx, SUBx Add or subtract B to/from A. A and B must be integers. If A or B are compound or small compound number pointers then a compound number exception will be raised instead of an illegal type exception. B will be scaled by x before the operation is performed. Valid values for x are 1,2,8 and 16 and is specified as part of the opcode. The result will be the same type as A. D's condition code bits will be set as appropriate. If an overflow occurs an overflow exception will be raised.

UADD, USUB Unsigned add or subtract. Identical to ADDx and SUBx, except there is no scale factor and will not raise an exception on overflow.

MODPRIME MODulo PRIME, D gets A mod 2^B- 1. A and B must be integers. The result will be an integer.

CMPEQ, CMPLT, CMPLTE Compare A, equal, less than, and less than or equal to B. A and B must be integers. If A or B are compound or small compound number pointers then a compound number exception will be raised instead of an illegal type exception. The result will be integer zero if the comparison is false and integer one otherwise.

UCMPLT, UCMPLTE Compare A, unsigned less than, and unsigned less than or equal to B. A and B must be integers. The result will be integer zero if the comparison is false and integer one otherwise.

FEQ, DEQ Compare A equal, including type tag, or data field only, to B. A and B can be any type except error object. The result will be integer zero if the comparison is false and integer one otherwise.

AND, OR, NAND, XOR, XNOR Bitwise and, or, nand, xor, xnor the data field of A with the data field of B. A and B can be any type except error object. The result will have the same type as A.

MOVEQ, MOVNEQ, MOVGT, MOVGTE, MOVLT, MOVLTE
Conditionally copy B, into D, based on A being equal, not equal, greater than, greater than or equal, less than, less than or equal to zero. A must be an integer or a float. B can have any type.

MOVS, MOVC Conditionally copy B, into D, based on the low bit of A being set, or clear. A must be an integer. B can have any type.

PADDx Pointer add A to B. A must be a pointer and B must be an integer. B will be scaled by x before the operation is performed. Valid values for x are 1,2,8 and 16 and is specified as part of the opcode. The result will be the same type as A. D's condition code bits will be set as appropriate. If an overflow occurs an overflow exception will be raised.

PSUBx Pointer subtract B from A. A must be a pointer and B must be an integer. B will be scaled by x before the operation is performed. Valid values for x are 1,2,8 and 16 and is specified as part of the opcode. The result will be the same type as A. D's condition code bits will be set as appropriate. If an overflow occurs an overflow exception will be raised.

PDIFF Pointer difference. A and B must be a pointers. The result will be an integer that represents the number of bytes between A and B. The result will be positive if A is larger than B. D's condition code bits will be set as appropriate. If an overflow occurs an overflow exception will be raised.

PCMPEQ, PCMPLT, PCMPLTE Pointer compare A, equal, less than, and less than or equal to B. A must be a pointer, B must be an integer or a pointer. The result will be integer zero if the comparison is false and integer one otherwise.

BEXL, BEXB Little endian and big endian extract byte B from A. Only the low 3 bits of B are used as an index into A. A can be any type except error object, B must be a pointer or integer. The extracted byte will be zero extended to 64 bits, given the same type tag as A and placed in D.

CBIT, SBIT Copy A into D replacing the Bth bit with 0 (CBIT) or 1 (SBIT.) Only the low 6 bits of B are used as the index. A can be any type except error object and B must be an integer.

RDBIT Write integer zero into D if the Bth bit of A is 0, otherwise write integer one. Only the low 6 bits of B are used as the index. A can be any type except error object and B must be an integer.

CPYTAG Copy A's data field into D's data field and B's type tag into D's type tag. A and B can be any type.

WRTAG Copy A's data field into D's data field and the low 4 bits of B into D's type tag field. A can be any type, B must be an integer.

RDTAG Zero extend the B's type tag to a 64 bit integer and place it in D. A is not used in this operation, B can have any type.

PTRP, NUMP Write integer zero into D if A is not a pointer (PTRP) or A is not an integer, float, compound number pointer, or small compounded number pointer (NUMP), otherwise write integer one. B is not used in this operation, A can have any type.

TAGEQ Write integer zero into D if the tag of A dose not equal the tag of B, otherwise write integer one. A and B can be any type.

CHKTAG Write integer zero into D if the type tag of A does not equal the low 4 bits of B, otherwise write integer one. A can be any type, B must be an integer.

WROV, WRCR Copy A into D, writing the low bit of B into D's overflow or carry flag. A can be any type except error object, B must be an integer.

RDOV, RDCR Write integer zero into D if A's overflow (RDOV) or carry (RDCR) flag is cleared otherwise write integer one. B is not used in this operation, A can not be an error object.

WRTRAP Copy A into D, writing the low bit of B into D's trap bit. A can be any type except error object, B must be an integer. This operation will not trap if A or D has its trap bit set.

RDTRAP Write integer zero into D if A's trap bit is clear other wise write integer one. B is not used in this operation, A can have any type. This operation will not trap if A has its trap bit set.

SPRWR Write A into the special purpose register specified by B. A can have any type, B must be an integer. D should be R63.

SPRRD Write the continents of special purpose register specified by B into D. A is not used in this operation, B must be an integer.

Additionally A1 and A2 can perform the following operations:

ADDx, SUBx, CMPEQ, CMPLT, CMPLTE Same as standard ADDx, SUBx, CMPEQ, CMPLT and CMPLTE except A and B can be floats as well as integers.

SMULT The low 64 bits of the product of A and B is placed in D. A and B must be integer, or floats. If A or B are compound or small compound number pointers then a compound number exception will be raised instead of a illegal type error. The result will have the same type as A. D's condition code bits will be set as appropriate. If an overflow occurs an overflow exception will be raised.

AS, LS, ROT Arithmetic shift, logical shift, or rotate, A B bits. A can be any type except error object, B must be an integer. If B is positive the shift/rotate will be left, otherwise to the right. The result will have the same type as A.

FF Write the position of the first 1 bit in B. The result is tagged as an integer. A is not used in this operation, B can not be an error object.

BMXC Select out the bits of A corresponding to the ones in B, then use them as the low order bits for the result, giving the same type tag as A. A and B can be any type except error object.

M8ADD, M8SUB Multi-gauge add and subtract. A and B are treated as 8, 8 bit twos compliment integers. They are added pair wise. A and B must be integers, the result will be an integer. If any of the adds overflow an exception will be raised. The carry and overflow bits of D will be written with the logical or of the overflow and carry bits of the eight adds.

M8UADD, M8USUB Multi-gauge unsigned add and subtract. Identical to M8ADD and M8SUB except will not raise an exception on overflow.

M8SADD, M8SSUB Multi-gauge saturating add and subtract. A and B are treated as 8, 8 bit twos complement integers. They are added pair wise but saturate at +127 and -128. A and B must be integers, D will be an integer.

M8CMPEQ, M8CMPLT, M8CMPLTE, M8UCMPLT, M8UCMPLTE Multi-gauge compare equal, less than, less than or equal, unsigned less than, unsigned less than or equal. If the i th byte of A and B have the appropriate relation then the i th byte of D is a one, otherwise its a zero. A and B must be integers, D will be an integer.

M8AS, M8LS, M8ROT Multi-gauge arithmetic shift, logical shift, and rotate. If j is equal to the i th byte of B, then the i th byte of D is the i byte of A shifted or rotated j bits. The shift/rotation is to the left if j is positive and to the right otherwise. A and B must be integers, D will be an integer.

Additionally A1 can perform the following operation:

IMASK Concatenate A and B and bitwise OR the result with the data fields of the next two instruction words as they enter the decode stage. Results are undefined if the instruction containing the IMASK operation or the instruction being masked takes an exception. Interrupts are disabled in such a way that if the instruction containing the IMASK operation completes then the next instruction will complete as well. The operation has no effect on the copy of the masked instruction in memory or in the caches. A and B can be any type. D should be R63.

2.3.2 Memory Operations

Memory operations include loading and storing words from memory, cache control and prefetching. These operations can only be used with the M functional unit. All memory operations use A as the base. For all memory operations except LDTAGx the base must a be pointer. With LDTAGx base can be any type of word. All memory operations use B as an offset which must be a signed integer representing a number of bytes. If the base and offset are not the correct type an illegal type exception is raised.

Effective virtual addresses are formed by adding the scaled offset and base. All memory operations except LDAx and STAx mask out the low 3 bits of the offset, and the low 3 bits (or 4 bits if it is a double word pointer) of the base before performing the add. All memory operations raise an exception if the low 3 bits of the virtual address are not zero.

If the base is an instruction pointer the operation goes through/affects the instruction cache if present.

Load Operations

All load operations have a scale factor, x, for the offset and a cache control flag, both are specified as part of the opcode. The scale factor can be 1,2,8 or 16 and is applied before the low bits of the offset are masked. The cache control flags determines if the reference should be cached. If the flag is set then the cache line of the word addressed will not be fetched into the cache.

Load operations with pair or small compound number bases will raise an exception if the scaled offset is not 8 or 0.

Load operations will raise a trap bit exception if the base or offset has its trap bit set.

All load operation which have a future as a base will raise a future exception.

Loads clear the condition code bits of D.

LDx LDx forms the effective address, fetches the corresponding word from memory and places it in D. A trap bit exception will be raised if the word fetched from memory has its trap bit set. This exception is "imprecise," other instructions after the load that do not depended on it may have completed before this exception is raised.

LDTAGx Identical to LDx unless base is a pair, small compound number, or not a pointer in which case base's type tag is padded with zeros to form a 64 bit integer and written to D.

LDAx Identical to LDx except the low bits of the base and offset are not masked. As a result if the low 3 bits of the sum of the base and scaled offset are not zero an unaligned accesses exception will be raised. This, along with STAx bellow, allows byte pointers to be implemented in a manor similar to the DEC Alpha architecture [13].

LDNTx Identical to LDx except it will not raise an exception if the word being fetched has its trap bit set.

CRx Identical to LDx except it will raise an exception if base is not a pair or small compound number pointer.

LDLx Identical to LDx except the address accessed is recorded in the locked-address register and the locked flag is set. The lock flag will be cleared if any stores are made to the location in the locked-address register. LDLx can be used with the BLL and store operations to implement atomic test and set operations. This facility is closely modeled on load locked/store conditional facility provided by the DEC Alpha architecture [13].

Store Operations

All store operations have a scale factor, x, for the offset and come in three versions that affect the cache in various ways, the scale factor and variants are all specified as part of the opcode. The scale factor can be 1,2,8 or 16 and is applied before the low bits of the offset are dropped.

The first variant is a standard store operation that will fetch the appropriate cache line into the cache as part of the store. The second variant will only write word into the cache if appropriate cache line is already there, it will not displace any line from the cache, this store is the counterpart to a load with the no-cache flag set. The final variant will allocate space in the cache for the word's cache line but will not fetch the rest of the line, instead the other words in the line will be marked invalid.

Store operations with pair or small compound number bases will take an exception if the scaled offset is not 8 or 0.

All Store operations will take a trap bit exception if the base or offset has its trap bit set.

All store operation which have a future as a base will take a future exception.

If the register being stored has any of its condition code bits set an exception will be taken.

STx STx forms the effective address and writes the continents of D to memory. A trap bit exception will be taken if the word being written has its trap bit set.

STAx Identical to STx except the low bits of the base and offset are not masked. As a result if the low 3 bits of the sum of the base and scaled offset are not zero a unaligned accesses exception will be taken. This, along with LDAx above, allows byte pointers to be implemented in a manor similar to the DEC Alpha architecture [13]

STNTx Identical to STx except it will not take an exception if the word being written has its trap bit set.

RPLACx Identical to STx except it will take an exception if base is not a pair or small compound number.

Cache Line Control

LOOP provides a number of cache line control operations. In all cases the cache line affected is the line containing the effective address. These operation only use base and offset, D should be R63. These operations do not have a scale factor for the offset.

FLUSH Will flush the appropriate cache line, writing it out to main memory if necessary. This operation has no effect if the line is not in cache.

INVAL Will invalidate the appropriate cache line, clearing any dirty bits. The line will not be written out to memory. This operation has no effect if the line is not in the cache.

FETCH Will move the cache line closer to the processor. This operation will never raise an exception.

LOCK Will lock the cache line into the cache, fetching it if it not already there. The cache line will not be replaced until it is unlocked. Behavior is undefined if it is attempted to lock two lines into the cache that map to the same cache location.

UNLOCK Will unlock a cache line that has been locked, allowing it to be bumped out of the cache.

Memory Map Test

LOOP provides instructions to test the validity of a virtual address without actually performing a memory operation or raising any exceptions. These operations do not have a scale factor for the offset.

MMPLD Runs the effective address through the memory mapping hardware as if it was a load. Returns integer 0 if no exceptions would be raised if an actual load was performed of that effective address. If an exception would be raised an integer bit map is returned that indicates which exceptions would occur. Exceptions caused by the loading of words with their trap bit's set are not checked for.

MMPST Runs the effective address through the memory mapping hardware as if it was a store. Returns integer 0 if no exceptions would be raised if an actual store was performed to that effective address. If an exception would be raised an integer bit map is returned that indicates which exceptions would occur. Exceptions caused by the writing of words with their trap bit's set are not checked for.

2.3.3 Control Operation

Control operations include branches, jumps, and conditional trap. They can only be executed by the C functional unit. All control operations write an instruction pointer to the instruction following the control operation to D, whether or not the branch, jump, or trap is taken.

Branches

All branches perform some test on A, if the test is true the branch is taken. B is used as branch offset and is interpreted as a signed number double words. Each branch has two flags specified in the opcode, I and P. If set the I flag will invert the sense of the test. If the P flag is set the branch will be predicted as taken, otherwise as not taken.

D's condition code bits are overwritten. Unless otherwise specified if A has its trap bit set an exception is taken.

BEQ Branch if A is zero. A must be an integer or a float, if A is a compound or small compound number pointer a compound number trap will be taken instead of an illegal type exception.

BLT Branch if A is less than 0. A must be an integer or a float, otherwise an incorrect type trap will be taken, if A is a compound or small compound number pointer a compound number trap will be taken instead of an illegal type exception.

BEQL Branch if A is less than or equal to 0. A must be an integer or a float, otherwise an incorrect type trap will be taken, if A is a compound or small compound number pointer a compound number trap will be taken instead of an illegal type exception.

BLOW Branch if the low bit of A's data field is one. A can be any type but error object.

BTRAP Branch if A's trap bit is set. This instruction will not trap if it A's trap bit is set. A can be any type but error object.

BNUM, BPTR, BPAIR Branch if A is a integer, float, compound, or small compound number (BNUM), or A is a pointer (BPTR), or A is a small compound or pair (BPAIR). A can have any type.

BOV, BCR Branch if A's overflow (BOV) or carry (BCR) is set. A can be any type but error object.

BLL Branch if the locked flag is not set and clear the lock flag. If in a multiprocessor system where more than one processor is executing a BLL operation with respect to the same value in their lock-address registers only one of the BLL operations can go in their predicted directions. BLL can be used with a store that has its squish bit set and a previous LDLx operations to implement atomic test and set operations. This facility is closely modeled on Load locked/store conditional facility provided by the DEC Alpha architecture [13].

BU Branch unconditionally. This branch does not have the P and I flags of the other branches. Instead it has the same stack, pop, and push flags that are described for jumps. A is not used.

Jumps

The destination of jumps are specified by B. If B is not an instruction pointer an incorrect type trap is taken. All jumps have three flags, stack, pop, and push, encoded in the opcode that determine how the jump's target should be predicted and the jump prediction stack should be modified. If stack is set the top of the jump prediction stack is used to predict the jump destination. If stack is not set the branch target cache[1] is used for prediction instead. If pop is set the top of the jump prediction stack is popped after it is used for any prediction. If push is set a pointer to the next instruction is pushed on the jump prediction stack after any reading or popping takes place. If B has it trap bit set a trap will be taken.

Conditional jumps have two more flags, I and P. I inverts the sense of the test. If P is set the jump is predicted taken. The test is performed on A. If A has its trap bit set a trap will be taken.

JUMP Unconditional jump.

JMPEQ Branch if A is zero. A must be an integer or a float, if A is a compound or small compound number pointer a compound number trap will be taken instead of an illegal type trap.

Conditional Traps

Conditional traps perform a test on A and B. Conditional traps are alway predicted not taken. Each conditional trap operation has its own entry in the user trap vector. Each conditional trap has an I flag that inverts the test. If A or B have the trap bits set a trap bit exception will be taken.

CTEQG Conditional trap if A is greater than or equal to B. A and B must be integers.

[1]Branch target caches are explained in section 3.1.1

CTTAGEQ Condtionl trap if A and B's type tags are equal. A and B can be any type.

CTFEQ, CTDEQ Trap if A is equal, including type tag, or data field only to B. If A and B can be any type but error object.

CTCHKTAG Trap if A's type tag is equal to the low 4 bits of B. A can be any type, B must be an integer.

CTPAIR Trap if A is a pair or small compound number. A can have any type, B is not used.

CTNUM Trap if A is a integer, float, compound, or small compound number pointer. A can have any type, B is not used.

CTPTR Trap if A is a pointer. A can have any type, B is not used.

2.3.4 Wide Operations

The functional units can also be used together in a wide operation. A wide operation is specified by a special opcode for A1 or A2. Wide operations use 2 or 4 of the functional units. Wide operations trap if any of their operands have there trap bit set unless otherwise specified.

In the description bellow A1.A is functional unit A1's A operand, A1.B is its B operand, etc.

DMUL Double multiply. Uses A1 and A2, takes A1.A and A1.B and writes the low 64 bits of their product to A1.D and the high 64 bits to A2.D. A1.A and A1.B must be integers. The condition code bits of A1.D and A2.D are cleared.

UDMUL Same as DMUL but the multiplication is performed unsigned.

DMULAC Double multiply and accumulate. Uses A1 and A2. A1.A and A1.B are multiplied together to form a signed 128 bit product. The product is added to the number formed by concatenating A2.A and A2.B (with A2.B being the high order bits). If A2.A is a float then only it is used. The high order bits of the sum is written to A1.D, the low order bits to A2.D, unless A2.A was a float, then the sum is a float and only goes to A2.D. The condition code bits are set as appropriate. All the operands must be integers or floats, a compound number trap is taken if they are compound or small compound numbers instead of a illegal type trap.

MS Multi-word shift. Uses A1 and A2. A1.A and A1.B can be any type but error object. A2.B must be an integer. A1.A is shifted A2.B bits and written to A1.D. Instead of shifting in zeros the high order A2.B bits of A1.B will be shifted in for left shifts and low order bits for right shifts. The bits that get push out the end will be zero extended to 64 bits, given the same tag as A1.A and written to A2.D. If A2.B is positive the shift will be to the left, otherwise to the right. Shifts of 64 or greater will copy A1.A into A2.D and A1.B into A1.D.

BINL, BINB Little and big endian byte insert. Uses A1 and A2. The low order byte of A1.B is overwrites the A2.Bth byte of A1.A. The result is written to A1.D. A1.A and A1.B can be any type but error object. A2.B can be an integer or a pointer, only its low order 3 bits are used to form the index. A1.D will have the same type as A1.A.

Double Loads Uses A2 and M. Same options and variants as single word loads. Base is M.A, offset is M.B. Word at effective address is loaded into M.D, word at effective address + 1 is loaded into A2.D. If base is a pair or small compound number offset must be 0 or an exception will be raised.

Double Stores Uses A2 and M. Same options and variants as single word stores. Base is M.A, offset is M.B. M.D is written into memory at the effective address. A2.B is written into memory at effective address + 1. If base is a pair or small compound number offset must be 0 or an exception will be taken.

Quad Loads Uses all four functional units. Same options and variants as single word loads. Base is M.A, offset is M.B. Word at effective address is loaded into M.D, word at effective address + 1 is loaded into A2.D, word at effective address + 2 is loaded into A1.D, and word at effective address + 3 is loaded into C.D. An exception will be raised if base is a pair or small compound number.

Quad Stores Uses all four functional units. Same options and variants as single word stores. Base is M.A, offset is M.B. M.D is written into memory at the effective address. A2.B is written into memory at effective address + 1, A1.B at effective address + 2, and C.B as effective address + 3. An exception will be taken if base is a pair or small compound number.

DPIPE Drain pipeline. Uses all four functional units. Lets all operation currently being executed complete before executing next instruction.

Non-Instruction Tag If an instruction is fetch and ether word is not tagged as an instruction the word with the lower address gets written into R62, the other word gets written to R61. Alternatively the processor can be set to take a user exception if a non-instruction word is fetched.

Notes on the Instruction Set

Most of LOOP's operations would be at home in any modern RISC processor's instruction set. The uses of some the other operations may be non-obvious. Also the interactions between tags and some instructions may appear to be inconsistent with other aspects of the architecture.

Pointer Integrity The architecture does not make any serious attempt to enforce pointer integrity. This is really a job for the compiler. As a result the bitwise logic operations, shifts, and other bit twiddling operations all accept pointers as valid operands. This is done partly because systems will some times have to twiddle the bits of pointers. When a system does do this it will probably know what it is doing so putting type checking in its way seems counter productive.

The other problem with providing credible pointer integrity at the architectural level is that it is very expensive. In general it would require making each pointer a three word triplet of base, offset, and bound. The operations on these pointers would be long and complicated, probably requiring multiple cycles. Most pointers don't need this protection, especially in Lisp where the programer usually can't get his or her hands on a raw pointer the way they can in C. Software can provide this kind of protection if it is needed [3]. Implementing safe pointers in software has the added advantage that the compiler has more flexibility in scheduling the subparts of safe pointer operations.

LOOP does allow for single cycle loads and stores with bounds checking when it is needed. The CTEQG operation can be placed in the same instruction with a load or store to perform the bound check. This can be done safely by setting the memory operation's squish bit.

The one problem with this is that it requires the bound to be in a register. LOOP used to provide a variant of a vector length cache [30] in the form of a small user associative memory and a special conditional trap operation to get around this problem. The conditional trap would get a word out of the associative memory and compare it with an integer, trapping if the integer was larger than the value from the memory. By storing array bounds in the associative memory with the array's base as the key bounds checking could be implemented by issuing the conditional trap with the base and offset as the operands in the same cycle as the memory operation to be protected. This idea was dropped because its felt that most of time the compiler will have enough information to simply cache the array bound in a register. The only time a problem arises is in code like this[2]:

```
(define (get-element i)
  (vector-ref some-vector i))

(map get-element some-list-of-numbers)
```

In this example it would be unreasonable to expect the compiler to be smart enough to cache the bound of `some-vector` in a register unless both `vector-ref` and `get-element` were inlined. However, if the calls are not inlined their cost is likely to dominate. The cost of an extra load is not significant.

The primary reason why some operations don't accept pointers is not pointer integrity but to make low cost generic arithmetic posable. We did not provide no pointer versions of the various bit twiddling operations because it is felt that Lisp programs don't do much bit twiddling. If a lot of bit twiddling is done it will probably be on fixed sized arrays of bits. In this case the compiler will probably be able to safely elide the type checks.

[2]This example is in Scheme

Byte Operations The multi-gauge operations are provided so byte operations can be efficiently handled by our word addressed machine. Its often claimed that word addressed machines can't handle byte operations efficiently. If there are going to be a lot of byte operations the bytes are going to be in some sort of array, which can be packed efficiently in our 64 bit words and and quickly operated on in groups of eight bytes by the multi-gauge operations. It would not be surprising if LOOP is faster at many byte manipulation operations than a similar byte addressed machine without multi-gauge operations. The multi-gauge operations are not expected to be hard to implement. Most involve optionally blocking some of the carries in the adder. Another alternative would be just to add eight 8 bit ripple carry adders which may well be fast enough and would be very small. Eliminating byte addressability greatly simplifies the memory system and can allow the whole chip to be much faster.

Generic Function Calls The LDTAGx and MODPRIME operations were suggested by G.J. Rozas as a way to make generic function dispatch fast. With an appropriate software convention LDTAGx allows the type of an object to obtained with a single operation. MODPRIME can then be used as a hash function so the appropriate method can be looked up in a hash table.

2.3.5 Instruction Format

Each instruction fills the data field of two words, we will refer to the word with the higher address as the high word, and the one with the lower address as the low word.

There are four sections in each instruction, one for each functional unit. Each section has an opcode field, an A field, a B field, a D field and a no-exception flag. Additionally the sections for A1, A2, and M each have a squish flag. The sections for M and C each have an extra bit of opcode to differentiate between memory/control operations and operate operations.

The A field is 6 bits and specifies the register to get the A operand from.

The B field is 9 bits for A1, A2, and M, 16 bits for C. If the high bit of this field is 0 then the low 6 bits specify the register to get the B operand from, otherwise the low 8 (or 15) bits are sign extended, tagged as an integer and used as the B operand. Note branch operations always use the B field of C as an immediate and use the immediate/register select bit and no-exception flag to form a 17 bit immediate.

The D field is 6 bits and specifies what register the result will be written to. Store operation use this filed to specify what register should be written out to memory.

If the no-exceptions flag [10, 16] is set and the operation raises an exception it will be dealt with by writing an error object to D, not by passing control to an exception handler. This includes TLB misses on loads. Results are unpredictable if the bit is set on operations that do not return a value or on control operations. In the figures bellow the no-exception flag is indicated by an E.

If the squish flag is set and there is a control operation in the instruction the functional unit will squish it results (including writes to the memory system) if the control operation goes in the non-predicted direction. In the figures bellow the squish flag is indicated by an S.

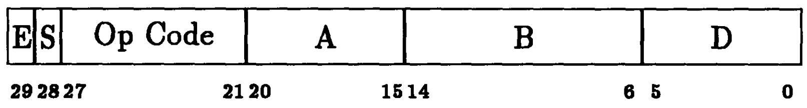

Figure 2-3: Format of A1 and A2 Operations

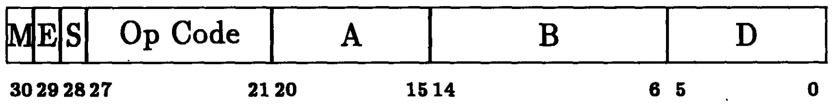

Figure 2-4: Format of M Operation

Format for A1 and A2 operation

A1's section goes in the top 30 bits of the high word's data field, A2's section goes in the next 30 bits.

Format for M operations

M's section takes up the low 4 bits of the high word's data filed and the high 27 bits of the low word's data field. In figure 2-4 the M flag is set to one for memory operations and to zero otherwise.

Format for C operations

C's section uses the remaining 37 bits in the low word's data field. In the figure 2-5 the C flag is set to one for control operations and to zero otherwise.

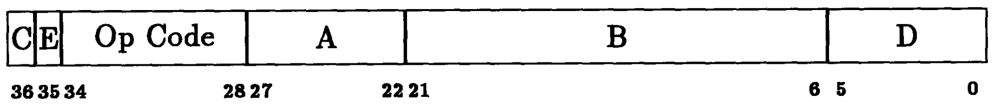

Figure 2-5: Format of C Operation

2.4 Memory Management

LOOP supports the standard paged virtual memory abstraction with a few refinements. Protection information is handled independently from the address translation information, allowing multiple contexts to have one address space with different access privileges. Wulf's WM also has this feature [52], Chase, Levy, Backer-Harvey, and Lazowska [7] suggest that this would be useful in the type of single address space operating systems that may be practical with the very large address spaces 60+ address bits provide.

The LOOP memory management unit supports read and write barriers with a granularity of 8 to 256 words. This is done by making part of the TLB out of programmable logic and adding a few dozen extra bits to each TLB entry. The programmable logic in the TLB can be programmed to use the extra bits as dirty, write protect, or access protect bits for small sets of words.

2.5 Traps

LOOP supports fast trap handling by providing multiple contexts, user level traps, and direct support for instruction emulation.

LOOP has multiple "contexts" on chip. Each context consists of one complete register file and a set of pipe line registers. When a trap is taken the processor is switched to a clean context, as a result none of the state in the register file needs to be written out to memory, nor does the pipe line have to be drained.

We would expect to have at least two contexts, one for normal operations and one for the trap handler, although providing 1 or 2 others could be useful. A third context allows system traps to be active during user level trap handlers, freeing the user level trap handlers from knowing about details of operating system. A fourth context could allow the garbage collector traps to be active during other trap handler, though having the trap handlers make in-line GC checks would probably also be acceptable.

Many of LOOPs traps are "user traps," which don't perform a costly address context switch (they do perform the register/pipe line context switch described above.) Many traps, such as type, garbage collection, overflow, floating point, and unaligned address traps are easier to service in the addressing context of the task that trapped since that is where the necessary information usually is.

Johnson [24] suggests a nice mechanism for emulation that LOOP adopts in a slightly modified form. When an exception occurs in addition to the PC of the trapping operation being provided to the trap handler the operands and a word describing the exception and the destination of the operation are also provided in special purpose registers[3]. Operations are provided to read and write the registers of the task that trapped, and to restart the trapped task while squashing and optionally replacing the result of the operation that trapped, or restarting the instruction.

[3]Johnson also suggest providing a copy of the trapping instruction as well since it may be difficult to get, primarily because it probably wont be in the data cache. This is not as much of a problem in LOOP because instruction pointer references are satisfied by the instruction cache.

Chapter 3

Notes on Implementation

By its traditional definition an "Instruction Set Architecture" gives no hint of how it might be implemented, however, only the foolish architect designs an architecture without considering how it may be implemented. Also RISC advocates have traditionally pushed for the barrier between "architecture" and "implementation" to be made porous. It can also be argued that VLIW machines make no distinction at all between the two. In this chapter we discuss the costs of some LOOP's lisp oriented features, how we think LOOP should be implemented, and some features which can not be properly termed "architectural" features.

3.1 Area Estimates

We base our area estimates on the DEC 21064 [1] [15], the first commercially available implementation of DEC's Alpha architecture [13]. The 21064 is a good chip to base our estimates on because it also implements a 64 bit architecture with a similar set of operations. It is also fast with a 5ns cycle time in a process with a 0.5μ effective minimum channel width.

Neither the Alpha architecture nor the 21064 directly support tags, fine grained memory barriers, or multiple on chip contexts. First we will present area estimates for a LOOP implementation without Lisp support based on data from the 21064. Then we will make estimates on how tags, memory barriers, and contexts impact the speed and size of a LOOP implementation. This will highlight the cost of Lisp support at the hardware level. In the next chapter we will look at what these features might buy us.

In the estimates, instead of using μ or mills we will use, λ as our unit of measurement. λ is 1/2 the minimum feature size in a process and gives us a process independent way to measure areas. For reference the 21064 is 1.67 $G\lambda^2$. Dally [9] suggests that by 1997 chip designers will have 10 $G\lambda^2$to work with. The estimates for low level components such as memory cells also come from [9].

[1]These estimates are from Prof. William Dally, he extracted this information the ISSCC 92 presentation on the 21064 and from talking to various people who worked on the chip.

3.1.1 LOOP Without Lisp Support

A LOOP implementation is likely to have 8 major components. The A1, A2, M and C function units, the register file, the instruction and data caches, and write buffer.

A1 and A2 functional Units

The A1 and A2 functional units are 64 bit integer/floating point units with barrel shifter and 64 bit multipliers. The 21064's integer unit is $72\text{M}\lambda^2$, this includes a non-pipelined 64 bit multiplier and barrel shifter.

A number for the 21064's floating point units is a little more difficult to get. The floating point unit is $175\text{M}\lambda^2$, but this includes a 5 port register file. Using the integer unit's 16.8 $\text{M}\lambda^2$ 6 port register file as a model we estimate the floating point unit's register file is $16.8M\lambda^2 \times 5^2/6^2 = 11.7M\lambda^2$, suggesting that the floating point unit without register file is $163\text{M}\lambda^2$.

Taken together these estimates suggest the A1 and A2 functional units would be around $235\text{M}\lambda^2$each.

Register File

LOOP needs a register file with 8 read ports, 3 write ports, and 1 read/write port. One way to provide this is with two register files with 8 ports each. Since the size of a register file goes up with the square of the number of ports, the pair of smaller files is smaller than one big one. To get an estimate for an 8 port LOOP register file from the 21064's $16.8\text{M}\lambda^2$ 6 port register file we multiply its area by $8^2/6^2$ to account for the extra ports, then by 2, because LOOP has 64 registers not 32. Using these number each 8 port file is $60\text{M}\lambda^2$.

M functional Unit

The M unit performs many of the same functions as the A units, but it does not do multiplies, shifts or floating point operations. It does need a TLB. The integer part of the A units is $72\text{M}\lambda^2$, but this includes a 64 bit multiplier and barrel shifter. Dally [9] estimates that a 64 bit multiplier would be $15\text{M}\lambda^2$, and a barrel shifter would be $5\text{M}\lambda^2$. Assuming a 512 KWord page size, TLB entries would need 52 bit keys, 61 bit physical addresses, 4 flag bits (valid, dirty, no-cache, and read-only). SRAM cells are around $1\text{K}\lambda^2$, content addressable memory cells are about $1.5\text{K}\lambda^2$, making each entry $143\text{K}\lambda^2$. A 32 entry TLB would be $4.6\text{M}\lambda^2$. This does not include the hardware for fine grain memory protection, we will add that later. Using these estimates the M unit should be about $57\text{M}\lambda^2$.

C functional Unit

The C unit is similar to the M unit but it does not have a TLB, suggesting the C unit is around $52\text{M}\lambda^2$.

Write Buffer

The 21064 has a 4 entry by 32 byte write buffer that is $36M\lambda^2$.

Caches

The 21064 data cache and instruction caches are $1.3K\lambda^2$and $1.75K\lambda^2$per bit respectively. These figures include the overhead for tags, valid bits etc. If we had 64 K byte (8 K words) data and instruction caches they would take up $696M\lambda^2$and $928M\lambda^2$respectively.

Branch Address Table and Jump Prediction Stack

To operate effectively pipelined processors need some way to predict changes in control flow. For conditional branches and jumps LOOP uses a taken/not taken bit in the instruction word, but it still needs a way to predict the address, preferably in the instruction fetch stage.

The address for jumps can be predicted using a jump prediction stack [13], subroutine calls push the return address on the stack and returns pop the top element and use it as a predicted address. Such a stack would be a 60 bit wide SRAM. A 32 entry stack, would be $2M\lambda^2$.

For branches we use a branch target cache, this is a degenerate "branch target buffer" [49] that only records the branch target address and it is not used to predict if the branch is taken or not. The branch target cache is accessed in parallel with the instruction cache. If the branch target cache hits and the fetched instruction has a predicted taken branch the result of the lookup for the branch target cache is used as the next PC. If a predicted taken branch is fetched and it does not have an entry in the branch target cache or its entry is incorrect a prediction is placed in the branch target cache after the its target is calculated.

Data from Kaeli and Emma [26] and Perleberg and Smith [37] suggests[2] that making the BTB larger than 512 entries does not help much. For LOOP each entry would need a 60 bit tag, a 60 bit target address and a 1 bit valid flag, for a total of $161K\lambda^2$. A 512 entry branch target cache would be $82M\lambda^2$.

What Is Missing

I have not including estimates for the clock driver, pads, or control logic. On the 21064 these components take up $730M\lambda^2$, but it is not reasonable to assume that these figures would carry over to a LOOP implementation as the size of these components are likely to be very implementation dependent. We have also not included estimates for the special purpose register since their number and size are likely to depend on implementation details. There are also some ALU functions that LOOP has, that the 21064 does not, such a bit extract and compact, multiply-accumulate, and byte adds,

[2]It should be noted that both studies used C, Pascal and/or FORTRAN programs and the Kaeli and Emma were using a fully associative cache

however these functions are not likely to be area intensive or use mostly components already present.

Total Estimated Area without Lisp Support

The total estimated area with out tags, tag checking logic, multiple contexts, or fine grained memory barrier support, is 2,448Mλ^2. This includes A1, A2, M, and C functional units, instruction and data caches, instruction[3] and data TLBs, a write buffer, jump prediction stack, and branch address table. It does not include pads, control logic, or clock drivers.

3.1.2 Cost of Lisp Support On Chip

In this section we make estimates for how much chip area tags and tag checking, multiple contexts, and fine grained memory barrier support use. We will also discuss how likely these features are to slow the chip down.

Tag Support

Tag Storage Not all of the components get bigger when tag support is added. The address in the branch target caches and jump prediction stacks hold only instruction pointers. Their types can be checked when they are inserted and if necessary reattached when removed. The logic to perform these operations would take up only a few Kλ^2, a trivial amount of area.

Similarly for the TLBs, the keys are pointer whose exact types are not really relevant (although they would probably be made simple data pointers), and their data are physical pointers which don't need to have their tags stored.

Each word in the data and instruction caches, and the write buffer would need another 5 bits to store the tag. If we simply increase the estimates for the data and instruction cache, and write buffer by 5/64th we get increases of 54Mλ^2, 72Mλ^2, and 3Mλ^2respectively. In practice the increase in size would be a bit less since the various address tags and house keeping bits in each of these components would not need the addition of tag bits.

Both of the register files need to be made 5 bits wider, increasing their size by 5Mλ^2each.

The total additional on chip area for tag storage is 139Mλ^2.

Making the caches wider will probably slow them down a little, however this effect should be negligible. The accesses time of a cache is dominated by the time it takes the bit cells to drive the bit lines. Making the cache wider has no effect on this time.

Tag Checking and Generation To estimate the cost of the tag checking logic I synthesized some PLAs that performed the appropriate logic functions. I did not include the logic for checking the trap bits, however, the size of this logic should be

[3]The instruction TLB is estimated to be the same size as the data TLB, about 5Mλ^2.

trivial. For each functional unit there are four PLAs, one to classify the opcode, one for each of the A and B operands to decode their type, and one larger one that uses the outputs of the other three and signals errors and control the type of result. In additional to the PLAs a 4 wide mux is need to route the correct type tag to the result. I eliminated columns in the input plane that were not connected to the output plane but did not do any folding.

To implement the PLAs for the A1 and A2 functional units 1398 PLA cells are required. If the 4 bit wide, 4 input mux was built out of full CMOS using pass gates and inverters would be about $20K\lambda^2$.[4] Assuming each PLA cell is $100\lambda^2$the total area for the tag checking/generation logic for the A1 and A2 units is $160K\lambda^2$each, this is an incredibly small percentage of an A unit's area, but does not account for the extra wire channels that have to be laid down next to the functional unit to carry the tag info. Each wiring channel is 8λ wide, and 5 would be need to carry the tag bits, so the total tag channel would be 40λ wide. The integer and functional units on the 21064 are 6000λ wide. Assuming the A units would have the same width this would make the A units $39K\lambda$ long, so the total area of the tag routing channel would be $1.6M\lambda^2$. Total area of the wiring channel and tag check/generation logic is $1.8M\lambda^2$for each A unit.

For the M unit the tag checking PLAs need 2109 cells, that along with a similar mux is $231K\lambda^2$. Assuming the M unit has a similar width to the A units the tag wiring channel is $380K\lambda^2$. Total area cost for tag support in the M unit is $611K\lambda^2$.

For the C unit the tag checking PLAs need 2245 cells, together with a 5 input mux totals to $250K\lambda^2$. Assuming the C unit is 6000λ wide the tag wiring channel for the C unit is $347K\lambda^2$. Total area cost for tag support in the C unit is $597K\lambda^2$.

The tag checking and generation is unlikely to add to the cycle time of the chip. They can be done in parallel with the actual operations (in fact the classification of the opcode can be done in the instruction decode stage) and are unlikely to amount to more than a few gate delays.

Taken altogether the total area overhead to support tags storage, checking, and generation on chip is less than $144M\lambda^2$, less than 1.5% of the chip area that we expect to have available in 1997. Most of this area, $139M\lambda^2$, is for tag storage, the tag checking logic adds less than $5M\lambda^2$.

Support for Fine Grained Memory Barriers

LOOP supports fine grained memory barriers by providing extra bits in each TLB entry, and programmable logic to control them. We envision extra bits being used for fine grained dirty or protections bits to support generational and concurrent garbage collection, shared virtual memory, concurrent checkpointing, persistent store, and extended addressability. These are all applications that Appel and Li [2] suggest could use the virtual memory mapping hardware and need smaller than page size granularity. There are other applications though, for example the extra bits could

[4]This is based on an estimate from [9] that datapath logic is $500\lambda^2$to $800\lambda^2$per a device, we have assumed the low end of the range here

be used to implement a facility similar to the "address space identifier" found in the MIPS architecture, which tags each TLB entry with a process ID that must match the current process for a TLB hit to occur.

Assuming we have 512 Word virtual memory pages and we wanted to be able to provide memory barriers for individual groups of 8 words we would need 64 extra bits per a TLB entry[5].

For the programmable logic we need a block of logic for each column of extra bits. Additionally a decoder for the top 6 bits of the page offset would be needed to make the extra bits useful for the implementation of a fine grained memory barriers. Each logic block could get one line from the decoder. You would also want to make the decoder programmable so it would decode fewer bits in which case it would assert more of its decode lines (i.e. if it was set to decode 5 bits it would assert 2 adjacent lines, 4 would assert 4 adjacent lines, etc.) In addition to the input from the decoder the logic blocks would need an input indicating the type of operation (read or write), and an input that is the bit in their column that was in the line that hit. Generational garbage collectors could also benefit on stores from having an input indicating whether or not the word being written is a pointer and a line indicating if the page maps a first generation page[6]. The logic blocks would need to be able to set the extra bit in their column that was in the line that hit, or raise an exception. Presumably logic blocks that read their extra bit could only raise exceptions, while one that set their bit could not read it.

For the applications we have in mind no logic block would need more than four inputs. The configurable logic blocks (CLBs) of the Xilinx FPGA architecture [53], can implement any two 4 input logic functions. Additionally they have two registers which would be necessary for implementing facilities like the address space identifier. A single Xilinx CLB makes a good model for the amount of logic required to implement two of our logic blocks. A single Xilinx CLB is 556Kλ^2[41]. The decoder could be easily implemented by running 25 wiring channels over the top of the logic blocks. The total area of 32 Xilinx CLBs with the wiring channels is 22Mλ^2. Adding 64 bits to each TLB entry is another 2Mλ^2. This makes each TLB [7] 29Mλ^2, 6.5 times increase in their size, however making these changes to both TLBs adds less than 2.5% to the total chip area *before* we added tag support.

Even if we increase the number of TLB entries the size is unlikely to go up significantly since the logic blocks dominate the cost of the TLBs and number of logic blocks is $O(1)$ in the number of entries. For example doubling the number of TLB entries to 64 would make each TLB 36Mλ^2.

The extra facilities are unlikely to add to the cycle time since they can work in parallel with the memory access.

[5]In practice we might want a few more, to provide something like the address space identifier or generation id bits, but 64 bits is a good ball park number.

[6]This last piece of data would probably be kept track of by another extra bit

[7]Chances are the instruction TLB does not really need memory barriers at this level of granularity.

Support for For Extra Contexts

To support four contexts in the register file one has to add 3 extra storage cells in each bit of the file and provide some way to access them. Because the area of the register file is dominated by wiring, the extra storage cells themselves are unlikely to add much to the area. In fact they may be able to be placed underneath the wires. We will ignore their contribution to the area.

The extra lines to control which context is accessed is another matter. If we allowed fully general access so any particular operation could access any context independent of the other operations in its instruction we could easily increase the size of the register file by a factor of four if we wanted four contexts. A more reasonable option would be to place the restriction that for a given register all reads from or writes to a given register in a particular cycle will take place with respect to the same context. With this option we only need to run one additional line along with the register selects into each register for each context. With four contexts this would be four lines in addition to the eight register select lines, so the overhead would be 50%.

LOOP also has contexts for the pipe line state so the pipe line does not have to be flushed before or refilled after a trap. Adding the extra bits to the pipeline register should not have a significant impact on the area of the design. It will probably result in an additional level of logic in each pipe line stage which will almost surely lengthen the critical path. Modern pipelined microprocessors have 10-20 levels of logic in each stage so adding another level could be expected to increase the chip's cycle time by 5% to 10%.

The total area cost of the adding multiple on chip contexts is $30M\lambda^2$, or 1.2% before adding tag or fine grained memory barrier support. The time penalty may be as much as 5% to 10%.

Total Cost of Lisp Support on Chip

The total area cost for all Lisp features is $255M\lambda^2$, an increase of just over 10.4%. This would be just over 2.5% of the total chip area we expect to have. If we filled the remaining area with tagged memory then somewhere between 8% and 9% of the area would be going to Lisp support features[8]. The only Lisp feature that is expected to make the chip significantly slower is multiple pipe line contexts which is expected to slow down the chip 5% to 10%.

3.2 Pipe Line

Like early VLIW machines [32] implementations of LOOP are not expected to reorder instructions. On possible exception to this policy is loads. It may be reasonable to allow instructions that are after a load and independent of its result to complete before the load does. This is useful because compilers have a hard time predicting

[8]We can't really fill all the remaining area with memory because we still need control logic, pads, etc. but this gives us a nice upper bound on how much area Lisp features could take up.

load latency in the presence of caches and this allows the compiler to schedule a the load as far in advance as possible without stalling independent instructions that come after it. Providing this limited form of instruction reordering should not be too difficult since most traps a load might cause can be detected in or before the TLB lookup stage which be easily finished before any following instructions reach the commit/write back stage. The one load trap that may happen after instructions following the load complete are traps caused by loading words with their trap bit set. Making these traps imprecise should not cause many problems for software.

Unlike early VLIW machines LOOP implementations are expected to have register interlocks and bypass paths. Fisher [17] points out that interlocks and bypass paths allow the compiler to be optimistic with respect to the latency of operations and may allow for a limited form of binary compatibility. Because of generic arithmetic features interlocks and by-pass are especially important for LOOP. Without them all adds and subtracts that could not be proved to be integer operation would have to be scheduled as floating point operations. Interlocks may also increase the code density by reducing the number of NOPs needed.

There are no operate instructions in the LOOP architecture that are intrinsically lengthy. There is evidence to show that even multiply and accumulate can be done quickly. Hiker, Phan, and Rainey describe in [22] a 3.4ns 53x53b multiplier to be used in a 3 stage 200Mhz multiply-adder circuit.

In the following sections we describe how the traditional fetch, decode, execute, and write back stages would work in LOOP. An aggressive implementation may subdivide some of the stages.

3.2.1 Fetch Stage

The fetch stage would fetch the pair of words out of the first level I-cache that was pointed to by the PC. At the same time the new PC would come from one of 4 sources, the PC plus 2 words, the jump prediction stack, branch table, or the from the control logic. The PC form the control logic is used when there was a mispredicted branch or jump, or if there was a trap. The branch table is used when the instruction holds a branch that is predicted taken, or a jump that is predicted taken and is not using the prediction stack. The prediction stack is used when the instruction holds a jump that is predicted taken and has its use flag set.

3.2.2 Decode

The decode stage decodes the instruction and fetches operands from the register file or forms them by sign extending the B fields. 10 of the 12 operand fields in a LOOP instruction can be interpreted without referencing the opcode fields. The D field of the M unit can ether be a register read for a store operation, or a write port for all others. The same bits of the instruction are used to name the register in ether case. Branches use use an extended immediate field, and the decision to use the extra bits must be put off until its determined a branch is being performed. The first part of the tag check (classifying the operation) can also be performed in this stage, making it unlikely that tag checks will affect the critical path.

3.2.3 Execute

In most implementations this would be more than one stage. The actual operations are performed in these stages but no changes to the state of the machine are made, although results do get passed back via by-pass paths. All operations go thorough all stages of the execute pipe line, even if they are not doing work after the first stage.

3.2.4 Write Back

None of the instructions permanently change the machine state before this stage. The register file is updated this stage. Instructions that write the special purpose register file wait unit this stage to make the write.

3.3 Memory

3.3.1 Tags

Architecturally LOOP's memory is tagged. There are several ways to implement this. The memory can be made out of actual 69 bit words. The tags can be allocated in a separate array and the special hardware can be added to fetch a word's tag when it is fetched. If the memory is byte oriented, such as Rambus [39] the memory interface can simply be programmed to fetch an extra byte for every word. Of course different levels of the memory hierarchy can use different methods to implement tags.

Allocating Tags Separately

Several people, including Kieburtz [27] and Shivers [44], have suggest allocating the tags in a separate array. For example one page may hold the tags for 8 other pages, the relation could ether be kept in the page table/TLB, or it could be a static mapping (i.e. physical pages x through x + 7, have their tags on page y, pages x+8 to x+15 on page y+1...). When a cache line is brought in from main memory, another memory reference would be initiated to bring in the word that held the tags for that line.

This method has the advantage that you can use standard mass market parts for the memory system. Additionally programs that don't need tags (i.e. scalar programs written in C, FORTRAN, etc) could tell the system not to load the tags[9], or possibly even allocate them. This way only programs that needed tags would not have to pay for them.

On the down side this method is likely to be slow. Modern memory system are oriented towards burst accesses. The second memory access to read or write the word

[9]A special "sticky tag" mode or load operation would have to be provided where loads would not overwrite the tag field in the destination register.

with the tags could easily add 20% to 40% to the time the memory accesses takes[10]. For this reason you would probably not want to build the on chip memories this way.

Building a "Custom" Memory

Many of the machines Symbolics made were built with the memory wide enough to accommodate tags. The custom memory approach does not to suffer the speed penalty which separate allocation does.

The traditional criticism of this approach is that it makes the machine incompatible with industry standard buses and memory boards. For example you can't plug a VME memory board into a Symbolics 3600. This is less of an issue than it used to be. Manufacturers often do not use I/O buses for memory anymore. Instead the memory comes on SIMMs or SIPs (small PC boards with just memory on them) and sits on its own private memory bus. One could probably get way with building standard IBM-PC SIMMs[11]. Until recently the standard width for these SIMMs was 8 or 9 bits. To design in the extra bits one needs for tags you would just need to put another SIMM socket for each bank of memory on the mother board. Only 5% of the memory would be wasted. Now 16 and 18 bit SIMMs are becoming more common, but even this should not be a real problem when you consider that ECC/parity bits will tend to drive a 69 bit machine word close to 80 bits, again requiring just an extra SIMM socket.

Rambus

Rambus is a high speed (500 MB/s peak) byte oriented DRAM technology developed by Rambus Inc [39]. The bus master sets up a transfer by broadcasting an header that is made up of several 9 bit bytes and includes address information. After a delay up to 256 bytes are transmitted, one every 2ns. Such a technology can be easily adapted to a tagged memory architecture by simply getting an extra byte for each word.

Using the Rambus 16 Mbit RDRAM [40] as a model we can estimate the overhead of tags when using a Rambus based memory system. Assuming a 4 word cache lines a processor could read a cache line without tags as fast as 104ns, and write it in as little as 80ns. The tagged read and writing the tagged versions would take 112ns and 88ns respectively, an overhead of 7.7% and 10% respectively.

Since tags are non-addressable in the LOOP architecture, are present as addressable bytes in the Rambus memory, translating from LOOP addresses to Rambus address requires an add. Assuming it can't be hidden (done in parallel with looking

[10]We are using a simple bus based model here. Assuming a burst mode bus fetching four words would take 5 cycles, 1 to put the address on the bus, and then one cycle to read each of the four words. To get the tags it would take additional 2 cycles, 1 to put a new address on the bus, and one to read the word of tags. If instead we fetched 8 words at a time we would be at the lower end of the over head range.

[11]There is nothing in the computer world that is more of a commodity item than standard PC SIMMs

up the word in the cache for example) we might expect this add to lengthen the time it took to transfer a cache line by 4ns. In this case the total time overhead for tags would be 12% and 15% for reads and writes respectively.

3.3.2 Knapsacks

Knapsacks are an idea suggested by Austin, Vijaykumar, and Sohi [4]. Knapsacks are on-chip memories where each location maps to exactly one address in the address space. Like caches they are invisible at the architectural level, but because they are a one to one mapping they can be smaller because they don't need tags, and faster because they can check for a hit in parallel with the lookup.

In [4] Austin, Vijaykumar, and Sohi simulate a processor with a knapsack on 15 programs from the SPEC '92 benchmark suite. They found that a 16KB knapsack was able to field 13% to 97% of all the dynamic references in the 9 floating point benchmarks they used and 17% to 76% in the 6 integer benchmarks they used.

Unfortunately to get these hit rates with the knapsack you need to do control and data-flow analysis, which is only just beginning to be practical in Lisp programs [43]. A generational garbage collector may be able to put such a feature to good use. By allocating the first generation or allocation area of a generational garbage collector into the knapsack, many of the heap references would end up going to the knapsack. In [33] MacLachlan does some limited experiments with this idea and found that storing the first two generations in an on chip memory could reduce off chip reference by 40% to 60% over systems that just had a cache.

Zorn [56] gives us some idea of how many heap references might go to a knapsack that was able to hold the entire first generation. For one of the programs he profiles, a microcode compiler, he finds that 71.5% of heap references are pointer loads, 0.4% are non-pointer loads, 9.0% are pointer stores, 0.1 are non-pointer stores, and 18.9% are initializing store (both pointers and non-pointers). Later he indicates that in the same application 66.6% of the pointer stores are to the first generation. Combined with the fact that initializing stores are always to the first generation this suggests that for this one program more than 89% of the heap stores would be to the first generation.

More importantly knapsacks have the potential to dramatically increase the memory bandwidth available to programs. Knapsacks should be easier to multiport than caches. Even without multi-porting if one has multiple knapsacks on chip, mapping non-overlapping portions of the address space, you don't have to worry about keeping them consistent, eliminating the need for extra write ports. The same applies to the data cache if care is taken to make sure that it won't cache locations covered by knapsacks. By giving each knapsack its own TLB entry you can avoid multi-porting the TLB as well.

Of course to exploit the extra memory bandwidth available, some the functional units would need to have memory interfaces added although the full functionality offered by the M unit would probably not be necessary. The cache control operations would not be necessary. It also not clear if the non-caching load and non-allocating store instructions would be valuable for accessing the knapsacks.

Another possibility is to use a few small knapsacks (1K to 4K each) to form a top of stack cache. The knapsacks would be set up initially to map the bottom of the stack area, and when the stack ran off the last knapsack the first knapsack would be flushed and remapped to cover the new area. The same thing would happen in reverse when you ran off the bottom. The stack buffers on the Symbolics 3600 [36] and Ivory [28] processors worked in a similar manner. One nice feature of this set up is that the compiler should be able to differentiate between stack and heap access easily so it could schedule multiple memory access in one cycle (one heap, one stack) if the implementation supported parallel reads from different knapsacks.

3.3.3 Wide Loads and Stores

Currently the double and quad word load and stores only need to be word aligned. This can cause problems for an implementation since one memory operation could access more than one page or cache line. Probably the best way to handle in hardware is to stall the pipe line when a wide load that cross a cache line is detected, and then do the load or store in two parts.

Another option is to force double and quad word memory operations to be aligned and leave it up to the software to follow a convention that will guaranty this. Guaranteeing that all objects are quadword aligned in a Lisp system could be problematic since pairs are only two words long. One option would be to have different heaps for objects with different alignments. This questions needs further investigation.

Chapter 4

Evaluation

In this chapter we evaluate the effectiveness of the Lisp support features in the LOOP architecture versus their cost. Unfortunately without a compiler and implementation of the architecture it is hard to get solid answers to many of the relevant questions.

4.1 Tag Support

4.1.1 Tag Checking and Generation

The tag checking/generation logic adds $5M\lambda^2$to the total area of the chip. This is less than 0.05% of the total chip area we expect to have to work with in 1997. It is less than 0.25% of the area the functional units and caches are estimated to consume. Because the checks require relatively little logic and can be performed in parallel with the main operations they are not expected to directly slow the chip down at all.

How much does tag checking and generation buy us? Steenkiste and Hennessy [48] suggest that with a simple software implementation for tags the cost of run time tag handling can be as high as 25%. With a more intelligent tag implementation that uses the low bits of the word this overhead drops to a little over 20%. Adding full hardware support similar to what is found in LOOP drops the overhead to just under 3%[1]. They also suggest that better compilers can get rid of many of the type checks. One problem with Steenkiste's and Hennessy's work is that their benchmark programs were all small, and none of them were numerically intensive or used any floating point. Only one of them invokes the garbage collector.

A more recent paper by MacLachlan [34] is interesting because its findings are based on the profiling of 4 large Common Lisp programs 3 of which are real applications in every day use. It also interesting because the programs were compiled with the CMU Common Lisp compiler which is one of the better compilers at type inference. MacLachlan finds that Common Lisp programs that are poorly declared can take a 10% to 80% speed hit. Inlining these tag checks can cut the penalty in half but results in significant code growth. In programs run with safety on another significant cost can be checking for uninitialized references to global variables. In one

[1]The remaining 3% is due to programmer specified tag checks such as consp

program MacLachlan found this increased runtime by 15%. LOOP's type checking hardware can save this overhead if memory is initialized with error objects.

Type Checking, Type Inference, & Compilers

Many people have questioned the utility of type checking/generation hardware. The first major argument against it is that compilers can do a good enough job at type inference to make most runtime checks unnecessary. This statement appears to be untrue. There are a few Lisp compilers than can emit efficient floating point code for main line processors but it appears to be very expensive.

At this point in time it does not appear to be practical for compilers to eliminate runtime checks for bignums [43]. To do so requires a level of compile time range analysis which does not seem practical at this time. This is so big a problem that some members of the Lisp community have started to avocate the elimination of bignums in future Lisp like languages.

Separating floating point and integer values at compile time can be done, at the cost of possibly altering one's programming style and using a more complicated compiler. The cost in the compiler does appear to be significant. Many Lisps, including some commercial Common Lisp implementations, do not even try to provide credible floating point support. Some do, but it is interesting to note that CMU Common Lisp, which is generally considered the best Common Lisp for floating point work, is a slower compiler than most and is reported to generate very poor symbolic code.

Of course if the hardware provides parallel type checking, much of this work goes away and the compiler writer is free to put his or her energy into other parts of the compiler. Furthermore it seems that adding type checking to the architecture is fairly cheap and does not greatly complicate the job of the architect or chip designer.

This is really the whole idea behind architecture: to determine at level a task is hard and if possible move it to a level where it is easier or more makes the system more efficient. For the past 15 years as part of the RISC movement we have been moving tasks into the compiler writer's domain, and for the most part this has made sense. But this does not mean that some tasks shouldn't be moved in the other direction.

The other major argument against type checking/generation hardware is that the idea of dynamic typing as presented by Lisp is flawed. It is claimed that type systems like the one used in ML are much better because they allow many type errors to be caught at compile time. Which way you like to program seems largely a question of personal preference, however, the ML style type checking can still benefit from hardware type checks of the sort LOOP provides. While compile time checking can be very helpful, it does not eliminate the desirability of an abstraction that captures the idea of a number. Hardware of the sort LOOP provides allows one to efficiently implement such an abstraction.

4.1.2 Extra Bits for Tag Field

Providing extra bits in each word adds 143M$\lambda^2$to the total area of the chip. This is less than 6% of the area of the functional units and caches. It is less than 1.5% of total area we expect to have. Although if we filled our extra on chip area with tagged memory the percentage of on chip area consumed by type tags would go up to a little over 7%. Making room for the tags on chip is not expected to slow the chip down. If a "custom" memory system is not designed to go with the chip it may be slower than a standard memory system, by up to 40%. However, this would only affect memory references that missed the cache.

The benefits from having extra bits in each word for tags shows up in two places. Probably the most important advantage is that the extra bits allow floats to be unboxed which reduces consing, makes generation GC more efficient, and makes the the compilers job easier. In his study MacLachlan [34] finds that when run with safety on the one floating point intensive program in his study spent 10 times as long in garbage collection because of the heap allocation of floats. This resulted in a 50% increase in run time. Wilson [51] cites small heap allocated objects as problems for generational garbage collectors and boxed floats being a prime example of these objects. He does not give an indicating of how much trouble this can cause though. Finally unboxed floats make the compiler's life easier because it makes it easier for the compiler to pass and return floats to/from procedures in registers as opposed on the stack. It appears this can be a real problem. At least some major commercial implementation of Common Lisp it is impossible to pass or return unboxed floating point numbers to/from functions [45, 1].

The other major advantage is compatibility. Having the data portion of the word being 64 bits makes it much easier to store and share data from/with the rest of the computing world. Again floating point numbers are prime example. Lisp implementation on stock hardware can avoid a lot of the problems described above it they use short floats, but truncating the floats changes behavior of floating point math and makes it difficult to share data between a Lisp program and a C or FORTRAN program. This comes up even with non-floating point data. With true 64 bit integers one does not have to worry about a C function that passes integers to a Lisp routine not passing fixnums. Also as 64 bit architectures become common, various standards (i.e. networking, data interchange, etc.) will probably be tuned to work best with 64 bit words.

It should be remembered that tags also contain the trap bit which can be very useful for parallel programs written in languages that normally don't need tags. Yeung and Agarwal [54] found that the application they studied was sped up by 40% to 30% when they had full/empty bits in hardware, one type of facility LOOP's trap bit can be emulate.

We can also evaluate the impact of tags in off chip memory system. To do this we need a model of how memory accesses contribute to the clocks per instruction (CPI). One such model is:

$$\begin{aligned} CPI &= C_{non-load}(1 - f_{load}) + \\ &\quad f_{load}(C_{first}h_{first} + (1 - h_{first})(C_{second}h_{second} + C_{main}(1 - h_{second}))) \end{aligned}$$

Where:

f_{load} is the frequency of instructions containing load operations.

$C_{non-load}$ is the average cost in cycles of instructions that do not contain load operations.

h_{first}, h_{second} is the probability of a hit in the first and second level caches respectively.

C_{first}, C_{second}, C_{main} The average number of cycles it takes to satisfy a first level cache hit, second level cache hit, or main memory access respectively.

Steenkiste [47] reports that in the Lisp programs he profiles on the MIPS-X, memory references instructions account for about 40% of the instructions, with loads outnumbering stores 3 to 1. Since each LOOP instruction holds up to four operations it's reasonable to assume almost all instruction will have a memory operation in them. Taking into account the load/store ratio 0.75 seems like a reasonable value for f_{load}.

We will assume $C_{non-load}$ is 1 cycle, this may be somewhat optimistic but most LOOP operations have a latency of once cycle and it seems probable that those that don't can be scheduled so their latency is hidden.

For C_{first}, C_{second}, and C_{main} we will assume values of 1, 5 and 20 cycles respectively. This assumes most of the loads were scheduled so the first instruction after the load does not need the load's result. It also assumes that all three levels of the memory system are built with tags in mind.

Zorn [55] studied garbage collectors using four Lisp programs and reports that the miss rate with a 64KB direct-mapped caches using a generational stop and copy collector is between 7.5% and 1.4%. The low end of this range is from the Boyer benchmark and is probably not representative. Throwing out the Boyer benchmark the lowest miss rate is 4%. Using this data we will assume h_{first} is 94%. To estimate h_{second} we assume the hit rate of a second level cache is equal to the difference between the hit rate of a first level cache of the same size as the proposed second level cache and h_{first} divided by $1 - h_{first}$. Assuming the second level cache was 1MB and again using numbers from Zorn we estimate h_{second} to be 66%.

Using these numbers the CPI is around 1.41. We can model the effect of having a memory system that does not directly support tags by increasing C_{main}. We estimated in section 3.3.1 that allocating the tags separately would increase the memory access time by 40% to 30%, which would change C_{main} to 28 or 26 cycles and would increase the CPI to 1.53 to 1.50, a 8.5% to 6.4% slow down. We estimated 15% overhead for implementing tags with a Rambus main memory system which would change C_{main} to 23 cycles resulting in a CPI of 1.45, a slow down of less than 3%.

4.2 Fine Grained Memory Barrier Support

Providing 64 extra bits in every TLB entry and programmable logic to control them adds $48M\lambda^2$if the modifications are made to both instruction and data TLBs. This is less than 0.5% of the chip area we expect to have and less than 2% of the area taken up by the functional units and caches. It is not expected to slow the chip down since there only needs to be one stage of programmable logic and it is expected it can do its work during the cache access (for trap generation) or in parallel with the TLB lookup (for dirty bits).

Zorn [56] estimates inline write barriers add 2% to 6% to the run time of systems using a generation garbage collector not including the cost of the garbage collection itself[2]. A generation garbage collector using LOOP's extra TLB bits to implement card marking would suffer no run time overhead. It may suffer additional overhead during the actual garbage collection because the remembered set would not be as precise as one using in line checks could be. While it would be easy to allow the programmable TLB logic to filter out writes of non-pointers it probably would not be possible for it to filter out writes of pointers that were not inter-generational. However, this may not be significant. Hosking, Moss, and Stefanović [23] in their comparisons of various write barriers for Smalltalk tried card marking system with and without filtering of inter-generational references and found little impact on the cost of root processing.

In [56] Zorn also estimates the cost of inline read barriers required for incremental garbage collection to be 10% to 20%, again not including the cost of the actual garbage collection. There is also a factor of 2 growth in code size. There will be some overhead from entering and exiting the trap handler but if multiple pipeline contexts are provided this should only be a few cycles more than what is required by the jump to and return from the transport routine required by the inline check. Even if multiple pipeline contexts are not provided the trap handler will not need to spill registers to memory to do its work as the transport routine for the inline case probably will be required to do.

It is not enough to simply use the conventional page mapping hardware with fast traps for these tasks because the granularity provided by normal pages is much to large. Hosking, Moss, and Stefanović [23] found that when using a card marking scheme the root processing time could grow by as much as a factor of two when a 4KB sized cards were used versus cards that were 256 or 512 bytes.

A non-lisp application for fine grained memory barriers is distributed virtual memory. Bolosky [6] uses page protection hardware to implement a coherent multiprocessor system. He finds that two key requirements are small page sizes and a fast trap mechanism.

[2]Zorn estimates that the cost of garbage collection itself can be as high as 20%.

4.3 Multiple Contexts to Support Fast Traps

The multiple register contexts add $65M\lambda^2$to the total area of the chip after the register file has been enlarged to make room for tags. This is 0.65% of the total area we expect to have to work with and 2.7% of the area of the functional units, register files and caches before adding the special Lisp support features. Because register files are rarely in the critical path, the multiple register file contexts are not expected slow down the cycle time of the chip. The multiple pipe line contexts are expected to require negligible area but will slow the cycle time of the chip down by 5% to 10%.

To evaluate how much multiple contexts help we need to know the frequency of traps, average length of trap handlers, and how much time the multiple contexts save. Unfortunately information on trap frequency is very hard to get. Taylor, Hilfinger, Larus, Patterson and Zorn [50] do offer some information on generation and type trap frequency, however a LOOP biased system can use the programmable TLB to do card marking and does not need to take generation traps, and most of the SPUR type traps were to handle floating point operands which LOOP handles in hardware. One of the Gabriel [18] benchmarks, frpoly(bignum), used in [50] does take bignum traps, but it not clear if this represents how bignums would be used in real world applications.

The other major type of trap we would like to handle efficiently is incremental garbage collection transport traps. In general people have not been been building incremental collected system because the overhead is viewed as being too high if hardware support is unavailable. Johnson [24] suggests that the frequency of transport traps would be comparable to the frequency of write barrier traps however it is not clear if this is a reasonable assumption.

4.4 Allocating Stores with Subblock Valid Bits

Although we did not discuss their implementation costs (which are expected to be minimal) in the previous chapter, LOOP's allocating stores with subblock valid bits can greatly increase the performance of allocation intensive programs. When subblock valid bits are used, the line that contains the word being written is not fetched into the cache, instead the other words in the line are marked as invalid. Subblock valid bits makes consing much cheaper because words are not brought into the cache that are about to be overwritten. Diwan, Tarditi, and Moss [14] look at how allocating stores with subblock valid bits can affect performance. They profiled how eight Standard ML programs ran using various memory system when compiled with the SML/NJ compiler. The programs probably use more heap allocation than most Lisp programs because ML programs are usually mostly functional. The SML/NJ compiler does not use a stack, so all dynamic data structures are allocated on the heap. The results are still interesting since many Lisp programs do a lot of heap allocation. They conclude that when the cache is 64KB or larger the overhead from cache misses drops from greater than 50% without subblock valid bits down to less than 9% with subblock valid bits.

4.5 What Else Can the Area be Used For?

It is not enough to show that a feature will help more than it slows a chip down, you also have to show the area it uses is not better used for some other feature.

Increasing the number of branch table, TLB, or write buffer entries, or the depth of the jump prediction stack is unlikely to increase performance. Perleberg and Smith [37] found that going from 512 entries to 1024 entries increase there BTB hit rate by .04%[3].

For the utility of increasing the depth of the jump prediction stack we can look at Shaw's [42] study of four Lisp programs. Shaw gives us data on the length of uninterrupted strings of control stacks pops and pushes and on the distribution of control stack depths at function entry. More than 50% of the time the programs he profiled did not make more than one push or pop in a row, and fewer than 1% of the push or pop chains were greater than 6 pushes or pops. One of the programs never had a control stack deeper than 23 frames. Another program had a maximum control stack depth of 46 but most of the procedure calls (90%) started with more than 10 frames on the stack suggesting that a 32 prediction stack would rarely overflow. The other two programs he looked had had a maximum stack depth in excess of 200 frames but only 10% of the functions started with stack depths greater the 84 frames deep. Overall Shaw concludes that control stack depth does not "wander radically throughout its range," suggesting that increasing the depth of the control stack is not useful.

As well as looking at allocating stores and subblock valid bits, Diwan, Tarditi, and Moss [14] looked at how TLB size and write buffer depth affected performance. They found that if main memory is built out of page mode DRAMs the differences between a 6 deep and 4 deep write buffer was small very small, less than a 0.01 difference in the CPI (which was never less than 1) if the caches were 64KB. The found with a unified 32 entry TLB that contribution of TLB misses to CPI was less than 0.1 for all their programs, and less than .01 for half of them. When they simulated a 64 entry unified TLB the contribution of TLB misses to the CPI was less than 0.01 for all programs. It seems unlikely that increasing the number of TLB entries or write buffer depth with significantly improve the performance of a LOOP implementation.

4.5.1 More Cache

Unlike the TLBs, write buffer, jump prediction stack and branch table, increasing the size of the cache or adding an on chip second level cache could significantly increase the performance a LOOP implementation, but by how much?

The largest single Lisp feature is extra bits in each word to store tags, this requires 143Mλ^2which is equivalent to less than 14KB of data cache. Zorn [55] says for direct mapped caches the miss rate decreases by about 1% each time you double the size of the cache until the cache is big enough to hold the newspace of the first generation.

[3]It should be remembered that they were profiling C, FORTRAN and Pascal programs, not Lisp programs.

This suggests the relationship between cache size and hit rate is logarithmic. Thus in going from a 64KB data cache to a 78KB data cache we decrease the miss rate by 0.29%. Using the same model we used to evaluate the cost of various main memory schemes a, 0.28% increase in h_{first} changes the CPI from 1.41 to 1.39, a less than 1.5% increase[4]. We have seen that floating point programs can benefit greatly from having tags supported by the hardware. How much advantage integer or symbolic programs gain from it is unclear but it would not be surprising if it was greater than 1.5%.

The programmable TLBs, register contexts, and tag checking hardware individually take up much less area than the extra bits for tags do, so replacing them with cache would be even less profitable.

This also ignores the fact that after we layout our functional units, caches, TLBs, and add the Lisp support we have only used 2,703Mλ^2. Even if we assume the pads, control logic, clock drivers doubled the size of the layout we would still have over 4.5G$\lambda^2$of on chip area to work with. If we decided to fill this area with memory the performance lost by replacing on chip memory with Lisp support features would go down since it appears the benefits of adding cache go up logarithmically, but the area lost to Lisp support goes up linearly with respect to on chip memory. For example if we have 512KB of first level cache on chip we could make it 40KB bigger if we did not have tags. Assuming we increased the size of the second level cache by enough so it kept its 66% hit rate the CPI decrease in going from a 512KB cache to a 540KB would be a just over 0.6%.

4.5.2 Extra Functional Units

The last major way to use area to speed up an implementation would be to add functional units. Adding a fifth simple functional unit, without floating point, a multiplier, or barrel shifter to an implementation without the Lisp support features would add 97Mλ^2, this includes the extra register ports that would be needed. To add a sixth would require another 157Mλ^2.

How much speed up would we get in return? Potentially the speed up for adding a 5th functional unit could be 25% and an additional 20% for a 6th. In practice the improvement is much less. In his extensive study of superscalar processor, design Johnson [25] finds that going from a two issue machine to a four issue machine resulted in a projected speed up of less than 20%. If the speedup provided by a 5th and 6th functional unit is degraded by a similar proportion (and the degradation is likely to be greater, not less) we might expect speedups of 5% from the 5th functional unit and 4% from the 6th.

The programmable TLB and tag checking/generation logic both take up significantly less area (53Mλ^2together) than than a 5th functional unit and together are projected to provide a speed up of at least 7%, so they should win over extra functional units. Multiple register contexts are a bit more expensive, 65Mλ^2, but still

[4]In practice the improvement will probably be less because we have not accounted for the fact that h_{second} will probably drop if h_{first} goes up.

much less than a 5th functional unit. For systems with incremental garbage collectors the fast traps are likely to save nearly 10% to 20%, again more than the speed up another functional unit is likely to add.

The hardware tag bits cost almost 50% more than a 5th functional unit. If your programs are floating point intensive or use fine grain synchronization it is very likely that the tag bits will save you more (possibly much more) than 5%, making them much more valuable than a 5th functional unit in those cases. If you need to interoperate with systems that really want to deal with 64 bit quantity you are again likely to win with tags over an extra functional unit, though it is not as clear by how much. If you just doing scalar, integer (where the integers are less than 62 bits) or symbolic computations you may still benefit from hardware tags but it not clear if it would be more than the 5% the 5th functional unit *might* provide.

Chapter 5

Conclusion

I have proposed a four wide LIW architecture with features to support Lisp like languages. These features are estimated to add just over over 10.4% to the basic layout. This percentage drops to bellow 9% if we fill out our chip with tagged memory, either in the form of more cache or something like knapsacks. The only feature which is expected to slow the chip down directly is the multiple pipe line contexts which may add 5% to 10% to the basic cycle time.

There may be some indirect slow down due to the fact the area used by the Lisp features could have gone to more functional units or larger caches. In the case of caches the loss is estimated to be less than 1.5% with the 64KB data cache provided. The performace loss would be less with bigger caches. It is less clear how much speed up extra functional units would give, but a 5th functional units is projected to improve performance by less than 5%. All of the Lisp specfic features with the exception of the tag field are significantly smaller than a fifth functional unit.

It appears tagged memory can slow down execution by as much as 8.5% if the main memory does not implement tags directly, however, the penalty drops to less than 3% if Rambus style memory is used. The performance impact of having a tagged main memory is expected to drop to 0 if main memory is built wide enough to include tags directly. This may have been commercially impractical in the past but I believe it to be a viable alternative today.

The tag checking logic is estimated to save between 5% and 40% over inline checks. The programable TLB is expected to eliminate all of the runtime overhead associated with generation garbage collectors (but not the cost of the collection itself) which can amount to 2% to 6%. With fast traps it is expected the programable TLB can eliminate most of the runtime overhead associated with incremental garbage collection (again not including the cost of collection) which is estimated to be between 10% and 20% making incremtal garbage collection practical. It is unclear how much the tags at the architectural and hardware levels help symbolic and integer programs, however, it appears they can substantially speed up floating point programs, especially when they are run with safety on. In one example we saw the runtime was increased by 50% because floats had to be heap allocated.

It is unclear that multiple pipe line contexts help more than they hurt so we will assume that they are not used. We will assume multiple register contexts are used since they are relatively small and don't appear to directly slow the chip down.

We would expect integer and symbolic Lisp programs using a generation collector to run at least 7%, and maybe as much as 50% faster than on a main line architecture with the same sized cache and number of functional units. If an incremental collector is used we would expect the improvement to be 15% to 60%. It does not seem unreasonable to expect floating point intensive programs to run up to 70% faster. We might also expect to get substantial speed up from using allocating stores with subblock valid bits, although some main line architectures, like the R3000, have this feature also.

5.1 Future Work

What needs to be done most is to write a simulator, Lisp compiler, and runtime system for LOOP so some simulations can be run. The simulator, compiler, and runtime system should all be written so the various Lisp support features can be independently added and deleted. Also one needs to be able to vary different architectural parameters such as the number and type of functional units and the number of registers.

There are a few areas where I fear LOOP may be lacking and should be looked at especially. It seems likely that one memory operation/instruction is too few. The wide loads and stores should mitigate this somewhat. If knapsacks our found to work for Lisp they should make the job of adding memory capability to another functional unit easier.

It would also be a good idea to question the number of control units LOOP has, although this worries me less than the number of memory units. Adding conditional traps to the other functional units probably would not be hard. Most of the logic is already there and since traps are always predicted not taken they would not put additional strains on the control prediction resources.

It also worth asking if LOOP is too conservative in the number of functional units it has overall. Making it wider would definitely require adding more ports into the memory system and adding more control units. Although it is hard to imagine the semantics of an instruction with more than one branch field, the IBM VLIW [16] project from the late 80's has some nice ideas about how to do this, though it may involve rethinking how condition codes and predicate operations are handled.

Knapsacks and copying garbage collection deserve a lot of attention. Not only do they have the potential to significantly increase the memory bandwidth available to Lisp programs but they also seem like a feature that might find there way into main line architectures and already appear in many processors intended for DSP or embedded applications. A good place to start would be to simply get a break down of how many references go to each generation in a generation garbage collector. This should give one a first order idea if knapsacks are practical for Lisp. After that a compiler has to be modified to generate multiple memory references per a cycle, possibly trying to distribute them across generations or between the heap and the stack.

But My Boss Won't Let Me Build a Lisp Machine!

Another interesting question is what features one would add to an architecture if one wanted good Lisp performance but did not want to make a "Lisp Machine." In such a framework one probably could not justify extra bits in each word for tags. One might be able to get away with parallel type checking/generation logic since it appears to add very little to the over all chip area. However, without explicit tag bits in the architecture it becomes more difficult to decide exactly which bits to use for the tag and how to interpret them. Its much more likely that some Lisp researcher will come up with a better way to represent tags that will be incompatible with your well crafted logic. Implementing this logic out of programable logic is one possibility but that could dramatically increase its area and delay, although this may not be an issue considering how small it is to begin with. One possibility would be include tags in the registers and use a BiBoP scheme. Each TLB entry could have a field that held the page's type, so when a word was loaded its type could be place in the register type field.

If you are dead set against extra bits for tags any where in the architecture, Steenkiste and Hennessy [48] have some ideas for special compares and arithmetic operations that would mask their operands with a pattern from a special register. Such a register would not be hard to implement and would work with a variety of tag implementations. In conjunction with these masked operations the number of branches or conditional traps that can be issued in a cycle should be increased since more control operations will be required.

Floating point performance will be hit the hardest by the lack of extra bits for tags. One possibility is to make it easy for Lisp implementations to uses "short floats" not in the traditional sense of a single word, but by having a floating point format that was just a little shorter. It appears that best software tag implementation uses the low bits so operations on a short floats would ignore the low 3 bits. This of course changes the characteristics of the floating point arithmetic. One way to get around this might be replace the short float with a full float when there is an underflow, similarly to the way fixnums are replaced by bignums when they overflow.

Henry Baker [5] has suggested another way for Lisp systems on 64 bit architectures to get good floating point performs. He suggests using floating point NaNs to represent non-floating point lisp objects. This allows full sized 64 IEEE floats without boxing or a non-standard word size. However, it seems likely this will only pay off if most of the data is floating point numbers or integers shorter that 53 bits.

The programable TLB is probably much easier to justify. Garbage collection is becoming more common out side of the Lisp community. The programable TLB is also helpful in implementing shared virtual memory, concurrent check point, persistent store, and extended addressability many of which are useful to non-lisp systems as well.

Allocating stores with subblock valid bits should be easy to place in almost any architecture and we have seen they can provide a dramatic speed up for programs do a lot of heap allocation. In general the memory system deserves lots of attention since many Lisp programs depend on it heavily [34].

Traps have to be made much faster than current practice in main line architectures today if you are going to get full use out of the programable TLB or tag checking logic. Fortunately this is probably not very hard. Providing user traps that don't make addressing context switches is likely to be an effective technique and probably won't cost much. Providing multiple registers contexts does not appear immediately prohibitive either. Providing multiple pipeline contexts may not be practical but much of the need for multiple contexts can be elevated if the parts of the pipe that take a long time to drain, such as long latency floating point and memory operations don't have to finish before the trap is taken. User traps could make this much more practical. Because it would not be expected that a process switch would occur in a user trap, the long latency operations in flight when the trap occurs could be left running and held in special registers (or maybe even placed in their destination register if there are multiple contexts) if the trap handler has not finished yet. Johnson's [24] ideas for supporting emulation are also probably appropriate for a main line architecture that wants to support Lisp.

Bibliography

[1] Kenneth R. Anderson. Courage in profiles. FTPed from ftp://wheaton.bbn.com/pub/profile/profile.ps, June 1994.

[2] Andrew W. Appel and Kai Li. Virtual memory primitives for user programs. Technical Report CS-TR-276-90, Princeton University, Department of Computer Science, July 1990.

[3] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. Technical Report TR1197, Computer Sciences Department University of Wisconsin-Madison, 1210 W. Dayton Street, Madison, WI 53706, December 1993.

[4] Todd M. Austin, T.N. Vijaykumar, and Gurindar S. Sohi. Knapsack: A zero-cycle memory hierarchy component. Technical Report TR1189, Computer Sciences Department, University of Wisconsin-Madison, 1210 W. Dayton Street, Madison, WI 53706, November 1993.

[5] Henry G. Baker. Private communications, 1994.

[6] William Joseph Bolosky. *Software Coherence in Multiprocessor Memory Systems*. PhD thesis, College of Arts and Science, University of Rochester, Department of Computer Science, College of Arts and Science, University of Rochester, Rochester, New York, 1993. Also TR456.

[7] Jeffrey S. Chase, Henry M. Levy, Miche Baker-Harvey, and Edward D. Lazowska. How to use a 64-bit virtual address space. Technical Report 92-03-02, Department of Computer Science and Engineering, University of Washington, FR-35, University of Washington. Seattle, WA 98195, 92.

[8] William Clinger, Jonathan Rees, (Editors), H. Abelson, N. I. Adams IV, D. H. Bartley, G. Brooks, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, G. L. Steele Jr., G. J. Sussman, and M. Wand. Revised^4 report on the algorithmic language Scheme. AI Memo 848b, MIT Artificial Intelligence Laboratory, 545 Technology Sq. Cambridge MA, 02139, November 1991. Dedicated to the Memory of ALGOL 60.

[9] William Dally. Parallel processing: VLSI and microarchitecture lecture slides. Lecture slides, 1993.

[10] André DeHon. Private communications, 1993.

[11] André DeHon. Transit note #100, dpga-coupled microprocessors: Commodity ics for the early 21st century. Transit Note 100, MIT Artificial Intelligence Laboratory, 545 Technology Sq. Cambridge, MA 02139, January 1994. Anonymous FTP transit.ai.mit.edu:transit-notes/tn100.ps.Z.

[12] Tom DeHon, André Knight. Transit note #11, Early Processor Ideas. Transit Note 11, MIT Artificial Intelligence Laboratory, 545 Technology Sq. Cambridge, MA 02139, April 1991. Anonymous FTP transit.ai.mit.edu:transit-notes/tn11.ps.Z.

[13] Digital Equipment Corporation, One Burlington Woods Drive, Burlington, MA 01803. *Alpha Architecture Reference Manual*, 1992.

[14] Amer Diwan, David Tarditi, and Eliot Moss. Memory subsystem performance of programs with intensive heap allocation, 1994. A paper containing some of the results presented in this paper will appear in *21st Annual Symposium on Principles of Programming Languages.* Also CMU CS technical report CMU-CS-93-227.

[15] Daniel Dobberpuhl, Richard Witek, Randy Allmon, Robert Anglin, Sharon Britton, Linda Chao, Robert Conrad, Daniel Dever, Bruce Gieseke, Gregory Hoeppner, John Kowaleski, Kathryn Kuchler, Maureen Ladd, Michael Leary, Liam Madden, Edward McLellan, Derrick Meyer, James Montanaro, Donald Priore, Vidya Rajagophalan, Sridhar Samudrala, and Sribalan Santhanam. A 200mhz 64b dual-issue CMOS microprocessor. In *1992 IEEE International Solid-State Circuits Conference*, pages 106–106, 256, Castine, ME 04421, February 1992. IEEE, John H. Wuorinen.

[16] Kemal Ebcioglu. Some Design Ideas for a VLIW Architecture for Sequential-Natured Software. In *To appear in the Proceedings of the IFIP Working Conference on Parallel Processing*, April 1988.

[17] Josh Fisher. Where is instruction-level parallelism heading? Talk given at Harvard, November 1993.

[18] Richard P. Gabriel. *Performance and Evaluation of Lisp Systems.* MIT Press, Cambridge MA, 02139, 1985.

[19] E. A. Hauck and B. A. Dent. Burroughs' B6500/B7500 stack mechanism. In *Spring Joint Computer Conference*, pages 245–251, 1968.

[20] John L. Hennessy and David A. Patterson. *Computer Architecture a Quantitative Approach.* Morgan Kaufmann Publishers, Inc., 2929 Campus Drive. San Mateo, CA 94403, 1990.

[21] Dana S. Henry and Christopher F. Joerg. A Tightly-Coupled Processor-Network Interface. In *ASPLOS V*, pages 111–122, 1992.

[22] Scott Hilker, Nghia Phan, and Rainey Dan. A 3.4ns 0.8μm BiCMOS 53x53b multiplier tree. In John H. Wuorinen, editor, *1994 IEEE International Solid-State Circuits Conference Digest of Technical Papers*, pages 292–293, 355, Castine, ME 04421, February 1994. IEEE, John H. Wuorinen.

[23] Antony L. Hosking, J. Eliot B. Moss, and Stefanović. A comparative performance evaluation of write barrier implementations. In *Proceedings ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 92–109. ACM, October 1992.

[24] Douglas Johnson. Trap Architectures for Lisp Systems. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 79–86, PO BOX 64145, Baltimore, MD 21264, June 1990. ACM, acm press.

[25] Mike Johnson. *Superscalar Microprocessor Design*. Prentice Hall Series in Innovative Technology. Prentice Hall, Englewood Cliffs, New Jersey 07632, 1991.

[26] David R. Kaeli and Philip G. Emma. Branch history table prediction of moving target branches due to subroutine returns. In *The 18th Annual International Symposium on Computer Architecture*, pages 34–41, New York, New York, May 1991. ACM, ACM.

[27] Richard B. Kieburtz. A RISC architecture for symbolic computation. In *Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 146–155, 1730 Massachusetts Ave. NW, Washington DC 20036-1903, September 1987. IEEE, Computer Society Press of IEEE.

[28] Thomas F. Knight, Jr. Private communications, 1993-1994.

[29] Peter M. Kogge. *The Architecture of Symbolic Computers*. McGraw-Hill Series in Supercomputing and Parallel Processing. McGraw-Hill, New York, 1991.

[30] K. Koniaris and G.J. Rozas. Vector Length Caching: A means for Fast and Safe Array Access. to be ai memo, Massachusetts Institute of Technology Artificial Intelligence Laboratory, Cambridge, MA, April 1993.

[31] Robert Krawitz, Bil Lewis, Dan LaLiberte, and Richard M. Stallman. GNU Emacs lisp. GNU Emacs info file.

[32] P. Geoffrey Lowney, Stefan M. Freudenberger, Thomas J. Karzes, W.D. Lichtenstein, Robert P. Nix, John S. O'Donnell, and John G. Ruttenberg. The multiflow trace scheduling compiler. *The Journal of Supercomputing*, 7(1/2):51–142, May 1993.

[33] A. MacLachlan, Robert. Memory architectures for lisp. Unpublished manuscript obtained from author, 1987.

[34] Robert A. MacLachlan. Lisp v.s. RISC or What Common Lisp Implementors Really Want. FTPed from CMU, January 1993.

[35] David A. Moon. Garbage collection in a large lisp system. In *Proceedings of 1984 ACM Symposium on Lisp and Functional Programming*, pages 235–246. ACM, August 1984.

[36] David A. Moon. Symbolics Architecture. *IEEE Computer*, pages 43–52, January 1987.

[37] Chris H. Perleberg and Alan Jay Smith. Branch target buffer design and optimization. *IEEE Transactions on Computers*, 42(4):396–412, April 1993.

[38] Kenneth A. Pier. A retrospective on the dorado, a high-performance personal computer. Technical Report ISL-83-1, Xerox Corporation Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, California 94304, August 1983.

[39] Rambus Inc., 2465 Latham Street, Mountain View, California USA 94040. *Architectural Overview*, 1993.

[40] Rambus Inc., 2465 Latham Street, Mountain View, California USA 94040. *Rambus Product Catalog*, 1993.

[41] Soon Ong Seo. A high speed field-programmable gate array using programmable minitiles. Master's thesis, University of Toronto, Toronto, Ontario, Canada, 1994.

[42] Robert A. Shaw. *Empirical Analysis of A Lisp System.* PhD thesis, Stanford University, Computer Systems Laboratory, Departments of Electrical Engineering and Computer Science, Stanford University, Stanford, CA 94305-4055, February 1988. Also Computer Systems Laboratory Technical Report: CSL-TR-88-351.

[43] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda.* PhD thesis, School of Computer Science Carnegie Mellon University, Pittsburgh, PA 15213, May 1991. Also TR:CMU-CS-91-145.

[44] Olin Shivers. Private communications, 1993.

[45] Jeffrey Mark Siskind. Re: Floating point. Posting to USENET news group comp.lang.lisp on or around March 7th 1994.

[46] Guy L. Steele Jr. *Common Lisp, the Language.* Digital Press, second edition, 1990.

[47] Peter Steenkiste. *Lisp on a Reduced-Instruction-Set Processor: Characterization and Optimization.* PhD thesis, Stanford University, Computer Systems Laboratory, Departments of Electrical Engineering and Computer Science, Stanford University, Stanford, CA 94305-4055, March 1987. Also Computer Systems Laboratory Technical Report: CSL-TR-87-324.

[48] Peter Steenkiste and John Hennessy. Tags and Type Checking in LISP: Hardware and Software Approaches. In *Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 50–59, 1730 Massachusetts Ave. NW, Washington DC 20036-1903, September 1987. IEEE, Computer Society Press of IEEE.

[49] Carl O. Stjernfeldt, Edward W. Czeck, and David R. Kaeli. Survey of branch prediction strategies. Technical Report CE-TR-93-05, Northeastern U., July 1993.

[50] George S. Taylor, Paul N. Hilfinger, James R. Larus, David A. Patterson, and Benjamin G. Zorn. Evaluation of the SPUR Lisp architecture. In *Proceedings 13th International Symposium on Computer Architecture*, pages 444–452, 1730 Massachusetts Avenue, N.W. Washington DC 20036-1903, June 1986. IEEE Computer Society Press.

[51] Paul R. Wilson. Uniprocessor garbage collection techniques. FTPed from cs.utexas.edu:/pub/garbage/bigsurv.ps, January 1994. Draft – Comments Welcome.

[52] Wm. A. Wulf. The WM Computer Architecture Definition and Rationale Version 1. Technical Report TR-88-22, Department of Computer Science University of Virginia, Charlottesville, Virginia 22903, October 1988.

[53] Xilinx Inc., 2100 Logic Drive, San Jose, CA 95124. *The Programmable Gate Array Data Book*, 1991.

[54] Donald Yeung and Anant Agarwal. Experience with fine-grain synchronization in MIMD machines for preconditioned conjugate gradient. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 187–197. ACM, May 1993.

[55] Benjamin Zorn. *Comparative Performance Evaluation of Garbage Collection Algorithms.* PhD thesis, University of California at Berkeley, Berkeley, CA 94720, December 1989.

[56] Benjamin Zorn. Barrier methods for garbage collection. Technical Report CU-CS-494-90, University of Colorado at Boulder Department of Computer Science, Department of Computer Science, Campus Box 430, University of Colorado, Boulder, Colorado 80309, November 1990.