Reconfigurable, Retargetable Bignums:

A Case Study in Efficient, Portable Lisp System Building

Jon L White

Lucid, Inc.
707 Laurel Street
Menlo Park, CA 94025

NOTATION

In Lucid Common Lisp, there are internal names used in the development of the system itself for all the standard simple arithmetic functions that imply fixnum-only arithmetic; e.g., `(logior& x y)` is basically the same as

```
(the fixnum (logior (the fixnum x)
                    (the fixnum y)))
```

In the paper, use of names like '+&' and '*&' will imply these semantics, and furthermore, the Lucid Common Lisp compiler ought to emit the most efficient, fixnum-only arithmetic instructions for such operators. In the explanations below, note that '|' used as a operator does not mean 'logical or,' as in the C language, but rather 'juxtaposition,' as in mathmetical logic. The sense of the 'juxtaposition' is base arithmetic; for example, when the base is 10, then 3|5 means just 35. In general, when dealing with "bigits"—bignum digits—$x|y$ means `(+ (ash x bits-per-bigit) y)`.

1. INTRODUCTION

Bignums—indefinitely large integers—have been a part of Lisp for a long time. They seldom figure into the more prominent AI applications—in fact, Interlisp (and its predecessors) existed from the early 1960's until recent times without them—but in the area of symbolic algebra they are critical. Indeed, no modern Lisp could be considered complete unless it offered a smooth, user-invisible transition between efficient machine numbers—'fixnums'—and those of larger size. Since they are so rarely used in common operations, their efficiency is of less concern than, say, arithmetic in general; yet one one would not want to have to fear them as a system component of totally unknown performance capability.

Even in languages that have long been standardized, such as Fortran II, there are subtle variances between one hardware implementation and another, and programmers have been known to latch onto these variations and even build dependencies on them in their programs [Professor W. Kahan of UCBerkeley has a number of interesting tales to tell on this score!]. Yet the appearance of machine-dependencies in the user-level language would have little impact on the system implementor *unless* the implementation is being done in that very language itself. The rise of Lisp-in-Lisp systems brings this issue to the fore. The Lisp machine developments on special-purpose hardware were among the first to champion this approach (see: [Deutsch 1973]; [Greenblatt 1977]), but significant trends were already occuring along these lines for "stock" hardware (see: [Moore 1976]; [White 1979]; [Griss 1980], [Griss 1981], and [Gabriel 1983]). In fact, Lucid Common Lisp is a just such Lisp-in-Lisp system, targeted towards many different machine architectures.

In this paper, we shall investigate a technique for mediating between the extremes of, on the one hand a completely ad-hoc, machine language implementation for maximum speed, and on the other hand a purely Lisp-written implementation for transportability. The focus of our investigation is not on the rational algorithms themselves, which are essentially those termed "Classical" in [Knuth 1981], but rather on the classification of their coding into a purely machine-independent part, a purely machine-dependent part, and a part that can, electably, be placed in either category.

We shall present a set of new primitive arithmetic operations that focus on the substantial activity of bignums; these new operations may be coded either in Lisp, or

as machine language templates in the code-generator of the compiler. The bignum algorithms are coded in a machine- and implementation-independent fashion using these primitives, and are thus available without specific reference to the machine architecture at hand. These primitives have been implemented with variations in the target machines—the retargeting part—and variations in the degree of machine-dependence—the reconfigurable part.

The primitives are "highly-leveraged" in that a modest amount of effort spent reconfiguring them as described below will yield a large pay-off in overall efficiency. Since the overwhelming amount of bignum code (indeed, the overwhelming amount of Lucid Common Lisp itself!) is written in Lisp, we depend upon an optimizing compiler for a certain degree of efficiency (see [Steele 1982] and [Brooks 1986)] Performance increases are observed during the successive refinements of a configuration.

Finally, we present numerous benchmark data to gauge the performance of this design, both before and after the reconfiguring, and to compare its overall speed with that of some other commercially available Common Lisp implementations. [1]

1.1 *The Problems of 'Porting'*

A Lucid Common Lisp implementation consists of

- A very large base of Lisp code (on the order of 10^6 lines) that is machine- and implementation-independent;
- A machine-specific code generator, which is part of a dynamically-retargetable (and generally machine-independent) Lisp compiler [Brooks 1986];
- A table-driven assembler—LAP, the "Lisp Assembly Program"—which is easily tailorable to most machine architectures;
- Several thousand lines of LAP-coded (i.e., machine language) routines which support the virtual machine environment and some of the operating system interfaces;
- A kernel image loader that produces an executable format image for the target machine (the rest of the Lisp system, compiled for the target machine, is loaded into the kernel to build a full Common Lisp image).

[1] As this work has only recently been completed, some parts of the performance comparisons may be incomplete. Furthermore, the discussions of portings to a few of the more novel architectures has to be limited by Lucid's contractual obligation not to divulge what ports are being worked on until the hardware manufacturer—the OEM contractor for whom Lucid is doing the work— decides to announce the product. It is expected that several more more interesting ports will be free for public discussion by the time of the Lisp Conference in August 1986.

Since there are numerous ports of the system to many differing machines, it is advantageous to keep the amount of machine-specific code to a minimum.

The goal for bignums has been to strike a balance between the tensions mentioned above—that is, to place a good deal of "knowledge" about bignums in the machine-independent Lisp code, and thus amortize the cost of design, debugging, maintenance, etc. over the many ports, or implementations, in which very little, if any, bignum development would be done. Furthermore, the porters to "later" machines are generally not the same persons who wrote the first version of the machine-specific sections; so the larger these sections are, and the more intricate and delicate their design, the harder the porter's job becomes.

Because the special knowledge of how bignums work on a given target machine is tied up in the compiler's databases, and because the Lucid Common Lisp compiler is dynamically retargetable towards numerous machines all at the same time, we can call such an implementation 'retargetable bignums'. See the paper in these conference proceedings about Lucid's retargeting compiler [Brooks 1986].

Initially, a porter may elect to do no special work for bignums, prefering to concentrate his time on reaching a moderately large, fully correct Common Lisp kernel subset. Then as time permits, he may begin at the lower-levels of the new arithmetic primitives (introduced to aid in the decomposition of the bignum algorithms), and put in as much work as desirable; he may then expect to see the reward of an increase in performance proportional, in some degree, to the amount of work he has put in. Section §2.2, of this report describes the differing configurations based on two notions defined there: the machine's endian-ness, and the bigit sizes. The code-generator for the MC68000 implementations supports all the configurations described, although a porter's chief concern will be the switch to larger bigits than the initial implementation affords; this configuration change has been found to have the most payoff in performance.

1.2 *Performance Expectations*

The range of performance between a special-purpose, highly "tuned" machine language implementation, and a "vanilla", Lisp-coded one may be seen by comparing the port of PSL [Utah 1982] to the IBM/370 with IBM's YKTLisp (a commercially available Lisp dialect developed at the Thomas J. Watson Research Center in Yorktown Heights, New York). YKTLisp is descended from Lisp/370, which had a significant amount of its code written in 'best possible' machine language. Bignums were so coded, and were thus very dependent both on the 370 architecure and on the particular implementation details of Lisp/370. PSL is at the other extreme, with very little being machine specific; its bignums are Lisp-coded in

a "vanilla" style. Multiplication intensive benchmarks, such as computing x^{1000}, were generally about 300 times slower in PSL/370 than in YKTLisp; the person doing the port isolated a small subroutine in which the PSL version was spending most of its time, and by hand-coding that, the slowdown factor dropped from 300 to about 70 [McDonald 1985].

Of course, a benchmark like x^{1000} is not a very helpful one for most Lisp users, since it measures performance at or near some asymptotic limit; the relative behaviour of the algorithm may vary markedly in the 'smaller number' ranges, where most realistic encounters with bignums will occur. Nevertheless, our goal has been to be *not* two orders of magnitude slower than 'best possible', and *not* an order of magnitude slower, but something within the range of a half order of magnitude— something more akin to the relative discrepancy observed between highly "tuned" machine language and the output of a good compiler. We think it would be acceptable for this portion of a Common Lisp to come with a factor of 3 to 7 of 'best possible', given its relative significance in the overall application spectrum.

One very interesting observation is that the purely Lisp-written bignums come from Lisp implementations that were driven by symbolic algebra applications: PSL was driven by REDUCE [Hearn 1973], and NIL [White 1979] was driven partly by MACSYMA. Franz-Lisp [Foderaro 1982] was driven partly by VAXIMA, a derivative of MACSYMA, and, although not Lisp-written, its bignums were in C rather than in machine language (and as the timings tables show, the speed of the C-written version is in between the 'best possible' machine language and the purely Lisp-written versions). While AI research and applications do not, in general, seem to need much in the way of bignum support, it is counter-intuitive that the bignum-oriented applications would favor portability over speed. It seems to suggest that ever-increasing speed *in the asymptote* may be pursued more for the interesting technical challenge it poses than for the fulfillment of a user-driven needs. [However, questions about application needs nearly always have to be addressed on an individual basis.]

1.3 *Factoring In Low-Level Operations*

At some level, the machine dependencies can be encapsulated by the implementation within a language-specific feature, or module. One critical area implicit in the Lisp computation model is stack operations. Yet there are no primitives in Common Lisp for dealing with all the variations that occur in the notion of "stack". On some machines there are a wide variety of instructions and operand modes that make stack-frame usage a fundamental part of any compiled function, whereas on others, there is no support. For these reasons, it seems wise that that Common Lisp specification not include such primitives; yet the lowest-level aspects of compiled code must deal with them. For example, when a function definition specifies optional arguments ('&optional'), there must be runtime code that deals with extending the stack frame in much the same way that variable-binding contours extend it. These are among the kinds of operations, along with the function calling protocol, that are time-critical and that apparently do not decompose directly into existing Common Lisp primitives.

Arithmetic, however, encompases many Common Lisp primitives. While advanced mathmetics may play a part in the implementation of functions such as, say, ATANH, a number of clever but simple tricks in the compiler's code generator are very important for the basic rational operations. For example, many implementors have independently discovered a suitable choice of tag layout in a 'tagged pointer' Lisp implementation so that generic + may be compiled in such a way as to impose very little overhead when the arguments are fixnums. (Researchers at IBM's Yorktown Research Center used such a trick in the early 1970s in LISP/370—[White 1978]) In a typical port of Lucid Common Lisp, three very fast non-memory instructions—a logical 'or', a bit test, and a conditional branch—suffice to certify that the arguments are fixnums. Admittedly this is slower than having just the one 'machine add' instruction (which could be emitted when appropriate compiler declarations are in effect) but there are many places where this difference just isn't important; especially in the RISC-type machines, the extra register-to-register instructions begin to resemble microcode for the special-purpose Lisp machines. Of course, if one or more of the arguments to the function aren't fixnums, then some slower route will be entered; the bignum algorithms will be invoked by just such a route.

2. Format of Bignums in Lucid Common Lisp

Bignums are stored-memory objects with a header word containing the length in bits. They are of a different data type than any array, but in other respects they resemble simple bit vectors. There are low-level, but machine-independent, primitives to access 16-bit chunks, to access the length field, to allocate memory for one of a given size, and to test the type (i.e., BIGNUMP). There are also access/update primitives for 8-bit chunks, but either one of the 16-bit or 8-bit primitives could be derived rather simply from the other given efficient fixnum versions of LDB and DPB.

The interpretation of the bits of a bignum is as the two's complement representation of a number, with the higher-order accessor indices holding the more significant bits (i.e., the accessor primitives are 'little-endian'; if it

were 'big-endian', then the lowest-valued accessor index, rather than the highest, would designate the most significant chunk of bits). It appears as though bignum implementations prior to the advent of Common Lisp used signed-magnitude representations, although there is very little, if any, documentation of those efforts. We chose two's complement format because there seems to be a feeling among many other Common Lisp implementors that LDB should execute in constant time; and if LDB is applied to a negative number in signed-magnitude format, then to be consistent with fixnum representation (which is two's complement on all the machines currently supporting Lucid Common Lisp) the bignum would have to be negated, or at least partially negated in order to fetch the proper bits. Due to carries, negating a bignum might involve accessing every word up to and including the byte of interest. We note also that Symbolics Common Lisp, VAX LISP and VAX/NIL have also gone for two's complement representation.

The particular algorithms used are simply the so-called Classical Algorithms described in section 4.3.1 of [Knuth 1981]. Although we have investigated some more complex algorithms that do show a significant improvement *in the asymptotic behaviours*, we are also concerned with the behaviour on relatively small bignums; a rule of thumb for programs that are typically written in Lisp is:

- almost all numbers are fixnums;
- almost all integers that aren't simply fixnums are smaller than a '2-word' bignum;
- almost all bignums that are bigger than a '2-word' bignum are still smaller than a '4-word' bignum;
- and so on.

This may be characterized as a kind of 'Zipf's Law'—exponentially diminishing probability of occurance as the numbers get larger. The author, while at MIT, experimentally verified this rule by observation of various technical and engineering applications in MACSYMA (MACSYMA, a symbolic algebra 'expert system', is currently available through Symbolics, Inc., Cambridge, Massachusetts).

2.1 *Operations on 'Words'*

We use the term 'word' to mean a chunk of storage large enough to hold a basic Lisp pointer; we also use it to mean the size of arithmetic that the machine's ALU supports. Normally this is the same size—32-bits for machines like the MC68020—but there are a few 'rough' spots in the MC68010 (the ALU does 32-bit additions, but only 16-bit multiplications and subtractions). When word-sized operations are available, it makes sense to extend one or more of the bigit sizes in order to operate upon word-sized chunks. For example, a machine might not efficiently support byte or half-word addressing; it could then be more than twice as expensive to fetch each half-word separately and operate upon the half words, than to fetch and operate upon the whole word.

Bignums are allocated in units of at least one 32-bit word (a consequence of the memory manager for Lucid Common Lisp). Thus sign-extension will be to the 32-bit word, and this is required if any of the machine's 'word-sized' arithmetic instructions are to be used.

We seek operations that can work on 'words' at a time, especially those for which the underlying primitive can be constructed with minimal effort. A good candidate is BIGNUM-WORD-REPLACE; many machines have a very short, fast instruction sequence to do a block transfer of words (or of bytes—it doesn't matter here), and these sequences would be the target of a primitive that copied the bits from one or more words of one bignum into another. Since it is desirable not to allocate new storage for intermediate operations, there are a number of places in our encoding of the 'Classical' algorithms where we do BIGNUM-WORD-REPLACE, or BIGNUM-WORD-NEGATE-INTO-BIGNUM (a variant that 'replaces' with the negation of the source) into resource-managed, temporary bignums. The section on comparative performance shows that even these copying steps are visible in benchmarks such as FRPOLY.

While the compiler's code generators may contain primitives that handle 32-bit chunks (or 'machine words'), such is not the case with Lucid Common Lisp's fixnums which, in the current design, are limited to 30 bits. This is the main reason why some primitives must be implemented directly by the code generators—it is not so much that the compiler would produce terrible code for the Lisp-written version, but rather that when the chunk size for the algorithm is increased to a certain size, a few of the intermediate calculation values may exceed the fixnum capacity. We note that a 'machine word' need never be represented by a Lisp quantity; only the location of the 'word' in question must be specifiable. We refer to 'words' as a normal object pointer and a word-sized index offset from that pointer. Thus we haven't taken the step of creating a separate systems-implementation language, like SYSLISP [Benson 1981] or LIL (an undocumented 'Lispy Implementation Language' variously developed at MIT and Symbolics during the early 1980's)[1], but rather have chosen to extend the compiler in a very straightforward way.

Nevertheless, we do have to arrange the bignum algorithms so that they can take advantage of the speedier code-generator templates when available. A typical sort of approach is (1) do some primitive operation a bigit at a time until a word boundary is reached, then (2) 'whiz

[1] One might view these languages as a kind of BCPL embedded in Lisp

along' using a different primitive that operates a word at a time, as long as a full word's worth of bits remain, and then (3) 'wind down' by going a bigit at a time, until the final end point is reached. We refer to this strategy as 'Whiz-Along-By-Words'. The carry-propogation part of the addition algorithm is a prime candidate susceptible to this kind of breakdown.

Of course, if a 'bigit' is at least as large as a 'word', then there is not nearly so much advantage to 'Whizzing-Along-By-Words', especially if the Lisp compiler can approach a good hand coder in efficiency. This isn't the case in most ports of Lucid Common Lisp now, because the fixnum size is generally smaller than the word size.

2.2 *The Problem of 'Bigit' Size*

2.2.1 *What is a 'Bigit'*

A 'bigit' is a 'bignum digit', and is thus an integer between 0 and $R-1$ for some positive radix R; the size of a bigit, in terms of number of bits, will vary from implementation to implementation *as well as from algorithm to algorithm.* In other words, there is a different bigit size for the addition algorithm than for the multiplication algorithm. We use the term 'abigit' to mean a digit in the range suitable for the additive algorithms (addition and subtraction); we use the term 'mbigit' to denote a digit in the range suitable for the multiplication algorithm; and we use the term 'dbigit' to denote a digit in the range suitable for the division algorithm.

The option of non-uniform radices is exercised to achieve the highest efficiency within the other constraints of design. Sometimes these constraints are due to the hardware [the MC68010 hardware can only do a 16-bit multiply with 32-bit result, whereas other machines can do a 32-bit multiply with 64-bit result] and sometimes they are due to the amount code involved in the algorithm. For example, bignum-by-bignum division is a rather complex algorithm, and writing it in such a way that intermediate quantities are not held in Lisp variables would involve putting more into machine language than could perhaps be tolerated by the goal of 'easy portability'.

2.2.2 *Bigit Accessors*

All Lucid Common Lisp implementations support 8- and 16-bit accessors as primitives, which are respectively named `BNREF-8BIT` and `BNREF-16BIT`. The existence of machine instructions like **MOVB** make it an easy task for the porter to build these access/update primitives and to support the 'arithmetic' accessors described below by macroexpanding them into `BNREF-8BIT` and `BNREF-16BIT`. For abigits, mbigits, and dbigits, the access functions are named respectively `ABIGIT-REF`, `MBIGIT-REF`, and `DBIGIT-REF`. Each one merely fetches a contiguous sequence of bits, and returns them as a digit in the radix of that type. For example, if the dbigit radix is 256 ($= 2^8$), then the i'th zero-origin dbigit of `x`, `(abigit-ref x i)`, can be computed by

```
(mod (/ x (expt 256 i)) 256).
```

Since the radix is a power of two, this could also be expressed as

```
(ldb (byte 8 (ash i 3)) x)
```

In addition to the bigit sizes introduced for the four basic rational operations, these routines deal with "xbigits", with accessor function named `XBIGIT-REF`. Xbigits are merely guaranteed to be a fixnum, regardless of any constraints or extensions in the implementation. When writing Lisp code to copy one bignum to another, for example, there is no particular reason to "pick up" and "put down" abigits, or mbigits—and in fact it would be impossible should they be extended to 32 bits. Furthermore, the implementation of functions like `LOGXOR` should avoid dependencies on the particular coding developed for bignum addition; having an abstract, fixnum-sized bigit ensures that there will be no conflict in the logical operators when bignums are "reconfigured" for more speed from the rational operators.

Thus, with only 16-bit abigits, 8-bit mbigits, and 8-bit dbigits, and 16-bit xbigits, all the bignum algorithms are functional, being written entirely in Lisp using fixnum arithmetic only. A port will have a working bignum implementation merely for the cost of duplicating some simple-bit-array primitives. The port may *later* be reconfigured to use larger bigit sizes, which will necessitate some amount of "machine language" for the code generator, in order to gain speed. Improvements in speed might be sought by 'inching' a 16-bit bigit up to a non-word-aligned 24- or 28-bit abigit, but the introduction of primitives that can take advantage of the full 32-bit word operations provided by virtually all these machines is certainly preferable.

2.2.3 *Unexplored Alternatives*

It is convenient to focus on bigit sizes of 8, 16, and 32 (or possibly even 64), because of the existence of memory access instructions that operate in these units. Some machines have additional hardware or firmware to support other sizes; e.g., the VAX instruction **EXTZV** should make it very easy to code accessors for any bigit size from 1 to 29, although it would very likely be slower than **MOVB**, **MOVW**, or **MOVL** when the bigit sizes are respectively 8, 16, or 32.

A question for further research would be whether it is profitable to use LDB to acquire a 14-bit bigit for multiplication and/or division in the "vanilla" case (instead of an 8-bit bigit); the attraction would be possibly a quadratic speed-up *in the asymptote* due to the increased bigit size, but this would have to be measured against

the slowdown of units conversions and bigit access, especially on machines like the MC68000 series. However, this issue is not very pressing, because we expect all Lucid implementations will sooner or later make the step to "machine language" versions of the new primitives.

Another possibililty is to use only 14 out of every 16 bits in memory; the bigit access function is merely BNREF-16BIT, but it would guarantee that the bigit would be less than 2^{14}. This would break the similarity between bignums and machine arithmetic, and would complicate the LDB function somewhat, although maybe not as much as signed-magnitude representation does; but the real objections are the same as in the preceeding paragraph—it is a side-track away from the road to real "machine language" level arithmetic.

2.3 *Bigits for Additive Algorithms*

Abigits are non-negative integers less than $2^{\text{abigit-size}}$. This says that we need an (unsigned) adder that is one bit wider than the abigit size. Most machines manage to achieve an abigit size of exactly one word by keeping a carry bit around in the process status register, thereby extending the adder width by one bit. For example, if the abigit size—the number of bits needed to hold an abigit—is N, then the intermedite step of the addition algorithm needs to construct a number that is the the sum of two abigits, and possibly a carry from the previous position; each abigit is strictly less than 2^N, so the maximum value this sum may have is $(2^N-1)+(2^N-1)+1$, or $2*2^N-1$, which is strictly less than 2^{N+1}. Thus one more bit of adder width is satisfactory.

A primitive addition facility could be this 3-input, 2-output function:

```
(defun primitive-abigit-add (carry-in x y)
  ;; Adds 'x' to 'y', plus the 'carry-in'
  ;; Returns two values: the sum, modulo the
  ;;  bigit size, and the carry-out
  (let ((sum (+& carry-in x y)))
    (values (ldb (byte abigit-size 0) sum)
            (if (>=& sum (expt 2 abigit-size))
                1
                0))))
```

Note that all intermediate results are strictly less than $2^{1+\text{abigit-size}}$; thus the additive algorithms can be written in Lisp providing only that fixnums are at least abigit-size + 2 bits in width (remember: fixnums are twos's complement format, which still requires an extra bit for the sign). This means that 16-bit abigits would be workable for Lucid Common Lisp, or even up to 28-bit abigits; but 29-bit abigits would not work because fixnum overflow would produce a wrong value for the calculation of sum above, and the '>= &' test would give the wrong result.

If this definition is taken for the lowest-level addition primitive (and if the target machine can ignore overflows from its ALU), then it can be implemented in the code-generator as just a couple of machine language instructions, providing only that

abigit-size < bits-per-word,

On the MC68010, and machines like it, the code pattern would simply be something like

```
'((add ,carry ,x)
  (add ,x ,y)
   ;; Sum produced in register 'y';
  (clr ,carry)
   ;; carry-out in register 'carry'
  (bclr ,(+ abigit-size offset ) ,y)
   ;; branch if no bit in the carry-out posit:
  ((bcc ne) ,done-adding)
  (move '1 ,carry)
 ,done-adding ...
  )
```

where *offset* is an artifact of the pointer encoding scheme for fixnums. The code-generator interface for this kind of function specifies that arguments 'x', 'y', and 'carry' must be located in data registers, and it also specifies that the two results are located respectively in the registers where 'y' and 'carry' were received as input. For more details on the code generator features, see [Brooks 1986].

However this definition is a little too bare; further analysis will show that, for addition, the more useful 'lower-level' operation is defined as 'Add the i'th bigit of the bignum x, plus the carry-in, into the j'th bigit of bignum y, and return the 'carry-out'. With the latter definition, the Lisp-written additive algorithms will be dealing only with 0's and 1's for the carry-out/carry-in arguments, and various indices into the bignums. Thus the additive algorithms can be written in such a way as to be unaffected by the bigit size. In a section below, we will exhibit a MC68000 version of this expanded definition, called 'primitive-abigit-ref-add-into-abigit-ref' to show that it isn't any more difficult to put this one into the code generator than it is to put the simpler definition above in.

Regardless of the primitive used, the 'Classical' algorithm calls for adding two abigits together and propogating the carries; if one argument is shorter than the other, then a separate piece of code may continue the carry propogation. An interesting problem for the two's complement format is to know when overflow has occured. The steps of addition, for two's complement format, are exactly the same regardless of whether one or both arguments are negative, but the detection of overflow—when the result bignum has to be 1 bit larger than the larger of the two arguments—is not as obvious as with signed-magnitude representations. A carry out of the high-order

bit of the accumulation *does not necessarily* imply overflow; only if the carry out of that bit position does not equal the carry in to it. See, for example, the expressions denoting how the MC68020 calculates separate Overflow and Carry bits for its two's complement addition, found on page A-3 of [Motorola 1985]; a "vanilla", Lisp-only version of the addition algorithm will emulate this 'hardware' test one way or another.

In the description of the primitives for addition and subtraction, 'carry-in' and 'carry-out' as well as 'borrow' and 'borrow-back' all refer to a 1-bit quantity—a 0 or a 1.

2.4 *Bigits for Multiplication Algorithm*

The 'bigit' situation is not quite so pleasant for the multiplication algorithm as it was for addition. When coded in Lisp, there are many lines of code that reference intermediate quantities as large as a full mbigit. Worse yet, when coded in Lisp, the lowest-level, 3-input-2-output multiplier shows that the mbigit size must be less than half the maximum fixnum size.

```
(defun primitive-mbigit-multiply (carry x y)
  ;; Multiplies 'x' times 'y', adding in
  ;;  the 'carry';
  ;; Returns two values: the low half of
  ;;  the result, and the high half.
  (let ((product (+& carry (*& x y))))
    (values (ldb (byte abigit-size 0)
                 product)
            (ldb (byte abigit-size abigit-size)
                 product))))
```

When 'x' and 'y' take on values of the largest bigit, the intermediate product will be twice as long. This implies that the mbigit size must be less than half the fixnum size—30 bits in Lucid Common Lisp—or at most 14 bits. Because 14 is an unwieldy access size, an mbigit size of 8 bits was chosen, *for the Lisp-only development* (but see §2.2.3 for a discussion of 14-bit alternatives).

If x and y are integers, how big, then, can the product $x * y$ be? Theorem 1 below gives the answer, which is of use when allocating memory space for the result of a bignum-by-bignum multiplication. We don't want the multiplication routine to allocate extra space needlessly, nor do we want it to be caught short by one bit at the very end!

DEFINITION: *The integer-length of an integer x is the non-negative integer n such that*

$$\begin{aligned} 2^{n-1} &\le x < 2^n && \text{if } x > 0 \\ -2^{n-1} &> x \ge -2^n && \text{if } x < 0 \\ 0 & && \text{if } x = 0 \end{aligned}$$

This particular definition focuses on the notion that inspired the name, i.e., the length of the bit field capable of holding the number in binary notation. Its format is particularly useful in proving Theorem 1 below. (It should be trivial to prove this definition equivalent to the one found in [Steele 1984], which is based on a form like $\lceil \log_2 \ldots \rceil$.) The following observation immediately falls out from this definition.

COROLLARY: *The number of bits needed to represent an integer x in two's complement format is integer-length(x) + 1.*

THEOREM 1. *Let x and y be non-zero integers, with integer-lengths respectively of n and m; then either*

$$\text{integer-length}(x * y) = n + m$$

or

$$\text{integer-length}(x * y) = n + m - 1$$

except when both x and y are a negative power of two, in which case

$$\text{integer-length}(x * y) = n + m + 1$$

First, assume that both x and y are positive. By the definition of integer-length, we have

$$2^{n-1} \le x < 2^n \qquad (1)$$
$$2^{m-1} \le y < 2^m \qquad (2)$$

and by multiplying these inequalities together, we obtain the following bounds on integer-length($x * y$):

$$2^{n+m-2} \le x * y < 2^{n+m} \qquad (3)$$

Splitting this interval in half about the point 2^{n+m-1}, it is clear that one and only one of the following holds true:

$$2^{n+m-2} \le x * y < 2^{n+m-1} \qquad (3\downarrow)$$
$$2^{n+m-1} \le x * y < 2^{n+m} \qquad (3\uparrow)$$

Equation (3 ↓) is equivalent to saying that integer-length($x * y$) = $n + m - 1$, and equation (3 ↑) is equivalent to integer-length($x * y$) = $n + m$.

Now, assume that x is negative, but y positive. Then we obtain ($\bar{1}$) by the definition of integer-length for negative integers:

$$-2^{n-1} > x \ge -2^n \qquad (\bar{1})$$
$$2^{m-1} \le y < 2^m \qquad (2)$$

Multiplying together the inequalities again, $x * y$ will be bounded as follows:

$$-2^{n+m-2} > x * y > -2^{n+m} \qquad (\bar{3})$$

and by the same reasoning as above that led to the split of equation (3) into (3 ↓) and (3 ↑), we have the result that integer-length($x * y$) is either $n + m$, or $n + m - 1$, but by reference to the definition for negative numbers.

A completely parallel case holds when y is positive, but x negative.

Now assume that both x and y are negative; then from the definition

$$-2^{n-1} > x \geq -2^n \quad (1)$$
$$-2^{m-1} > y \geq -2^m \quad (2)$$

and again, by multiplying inequalities, we have

$$2^{n+m-2} < x * y \leq 2^{n+m} \quad (3')$$

which can be factored into the following two equations, one of which, equation (3), we've already covered:

$$2^{n+m-2} \leq x * y < 2^{n+m} \quad (3)$$

$$x * y = 2^{n+m} \quad (3\dagger)$$

That is, equation (3') implies that one, and only one, of equations (3) and (3†) is true. Equation (3) covers the case when integer-length($x * y$) is either $m + n$ or $m + n - 1$. The only way that equation (3†) can be true is if $x = -2^n$ and $y = -2^m$, i.e., both x and y are a negative power of two.

Q.E.D.

In fact, a case to watch out for is when either x or y is a negative power of 2; whereas integer-length$[2^n] = n+1$, we have, oddly enough, integer-length$[-2^n] = n$ However, integer-length$[-2^n + k] = n + 1$ for $0 < k < 2^n$. Thus there are many places in the coding of these bignum algorithms that special-case the computations when one or more of the arguments is a negative power of two—in particular when estimating the length required to hold an additive operation that may cause some argument to be negated (i.e., converting -2^N into 2^N increases it's integer length by one). Note also that a minimal field width to hold a integer representation in two's complement format is exactly one greater than the integer-length of that integer.

The primitive operation actually used in the multiplication algorithm is the basic component of the "Classical" multiplication method, and is a bit more complex than the 3-input, 2-output multiplier shown above.

```
(defun primitive-mbigit-multiply-add
       (carry multiplier multiplicand addend)
  ;; Basic 4-input, 2-output unsigned
  ;;  multiplier, with additive carry in.
  ;; Returns two values: low half of the
  ;;  product/sum, and the high-half
  (let ((accumulation
         (+& carry
              (*& multiplier multiplicand)
              addend)))
    (values
      (logand& accumulation mbigit-mask)
      (ashr& accumulation mbigit-size))))
```

Note that the carry mbigit may be any value between 0 and $2^{\text{mbigit-size}} - 1$ inclusive. The resultant combination, the product and sum, is

multiplier * multiplicand + carry + addend

Let m = mbigit-size be the number of bits per mbigit; then the maximum value obtained by the combination above will be

$$\begin{aligned} & (2^m - 1) * (2^m - 1) + 2 * (2^m - 1) \\ = \; & 2^{2m} - 2 * 2^m + 1 + 2 * 2^m - 2 \\ = \; & 2^{2m} - 1 \end{aligned}$$

The result will thus be representable in two mbigits, and the two values returned will each be an mbigit; one will be stored as part of the partial product accumulation, and the other will be used as the carry to the next `primitive-mbigit-multiply-add` call. See the discussion of 'Algorithm M' in [Knuth 1981].

There are separate functions for multiplying two bignums together, and for multiplying a fixnum by a bignum. In the latter case, there are a number of opportunities for optimization, primarily when the multiplier is merely one mbigit, and thus there need be no loop overhead to cycle through the mbigits of a bignum multiplier; also, the cases of multiplying by -1, 0, or 1 will come to this function, and can be dispensed with quickly there. Additionally, the lowest-level step is actually producing the bigits of the product, rather than producing intermediate bigits which would subsequently have to be added into the final product.

If the abigit size is increased, one would expect a linear speed-up for the additive operations; but many calls to the multiplication and division routines will exhibit a quadratic speed-up. For suppose x and y are bignums of about n mbigits; then the "Classical" multiplication algorithm takes n^2 primitive multiplications and $2n^2$ primitive additions also (adding in the carry is counted separately from adding in the low-order part of the result to the accumulating partial product). So when the mbigit size is doubled, the number of mbigits *in each operand* is decreased by a factor of 2, and the number of primitive multiplications subsequently required becomes $(n/2)^2 = (n^2)/4$, for a speed-up of a factor of 4.

2.5 *Bigit Size for Division Algorithm*

The division algorithm is essentially 'Algorithm D' as found on page 257 of "The Art of Computer Programming, Volume 2, Seminumerical Algorithms" by Knuth [Knuth 1981]; but with minor variations. In particular, the normalization used is that most obviously implied by Knuth's "Theorem B" on page 257—namely the divisor is shifted until the hi-order bigit is greater than or equal

to $2^{\text{dbigit-length}}/2$; this is equivalent to saying that the leading divisor digit in decimal notation is 5 or greater. The dividend is moved into a resource-allocated temporary bignum, and the bigits of both the quotient and remainder are produced 'in place'; of course, the dividend must be shifted by the same amount corresponding to the divisor normalization.

2.6 *Endian-ness:*

The primitive access to bignums is from the little-endian point of view; namely, increasing the "index of access" will increase the significance of the bit fields involved. Unfortunately, the MC68000 series of machines is fully big-endian—it is byte-addressable, and when bytes are packed into a half-word (into a 'word', in Motorola's terminology) they are packed with the higher byte address being the bits of lesser significance. Even the main addressing mode of the **ADDX** instruction, memory-to-memory with auto-decrement, is biased toward big-endian for full words (i.e., larger word address mean lesser bit significance). We prefer little-endian since that is the protocol parallel to vector accessing and to the arguments of the LDB function; also, it seems to be the protocol that more modern computers accept. Thus we have the incongruity between the machine's preference for word format, and that imposed by the little-endian approach. Since our approach is only visible when coding the lowest level access and update primitives, it is perfectly acceptable to pack and de-pack according to any pattern of scrambling a word of bits into a machine full word [but, we cannot, at this level compensate for the ordering of full words]. So we are concerned with (1) which format the machine uses—big- or little-endian, (2) which way we want bytes to pack into a half-word, and (3) which way we want half-words to pack into full words. One reason why (2) may not be the same as (3) is that the relative cost of compensating for one may be much worse than compensating for the other; for example, on the MC68000 series, (3) may be compensated by a **SWAP** instruction, which is among the fastest, whereas (2) would, in a 'worse' case, be compenstated by two **SWAP**s and two **ROLW**s, with the **ROLW**s being relatively expensive.

The table labelled "Memory-access Modifications for Big-Endian Host" indicates how the memory access must be modified to maintain the illusion of little-endian in a big-endian host such as the MC68000. The leftmost column of the table indicates how one may chose to pack the bits of a bignum into the computer's memory (i.e., how bytes and half-words are combined into full word chunks); the phrase "*8bit/16bit*" where *8bit* and *16bit* are each either "Big" or "Little", indicates that the "*8bit*" endian mode is used for packing bytes into a half-word, and that the "*16bit*" endian mode is used for packing half-words into full words. The remaining columns of each line list the modifications necessary to the primitive accessors for that particular layout. Two modifications potentially exist for each accessor: one to modify the index of access, and the other to permute the bits of the unit accessed. A form like "#bxxxx" means "change the low-order bits of the index by xor'ing them with the bit pattern 'xxxx'"; the phrase "SwapB" means to swap the bytes in each half-word of data; the phrase "SwapH" means to swap half-words within a full (32-bit) word of data; and finally "↔" means "do nothing".

Since Big/Big is the preferred ordering of the MC68000, then only the choice of Big/Big packing will permit straightforward 32-bit access. For a little-endian machine such as the VAX, the entire last line of the table would be "do nothing" entries (and there would be no other "do nothing" entries).

MEMORY-ACCESS MODIFICATIONS FOR BIG-ENDIAN HOST

Packing	8-Bit Access	16-bit Access	32-bit Access
Big/Big	#b11	#b10	↔
Big/Little	#b01	↔	SwapH
Little/Big	#b10	#b10,SwapB	SwapB
Little/Little	↔	SwapB	SwapH,SwapB

It would appear that one should choose a representation that permits the most common access to be done with no "patch ups". For example, if 16-bit access is to be favored, then choosing big-endian for bytes within the half-word, but little-endian for halfwords within the (long) word is the best choice; indeed, the "vanilla" Lisp-coded additive algorithms would wind up stressing the 16-bit access. However, it is most likely that at some point in development, a porter will want to have at least some of the additive primops "in machine language" (in the code generator), and thus the porter should bias his choice towards the eventual preference of 32-bit access; that is, he would favor the case that *could possibly* lead to the most efficient machine language instructions being used in the inner loop.

3. New Arithmetic Primitives

Lucid's bignum functions are written under the assumption that about two dozen or so primitive functions exist. Not all of these are truly primitive, in that they are simply macros (or compile-time macros—a Lucid extension) which expand into simple lisp code using the 'true' primitives. In the interest of conservation of printed space, only the 'true' primitives will be listed and explained here; these are the ones that have been put into the code generators of one or more of the compiler's target machines, or for which there might be some additional speed-up to be taken if they were so implemented.

For example, `abigit-carry-propogate-thru-bn` is a function which propogates an additive carry one abigit at a time; it uses one of the lower-level primitives listed below, and is merely a short, Lisp-coded loop. It is very unlikely that putting it into machine language will speed up any interesting benchmark, because even if its time were to go down to zero seconds per loop iteration, it just doesn't do that many iterations. Carries don't propogate far, on the average. Thus our main concern is that the short, simple `primitive-add-into-abigit-ref` be as efficient as possible.

```
(defun abigit-carry-propogate-thru-bn
        (bn astarti aendi)
  (loop
    (when (=& astarti aendi) (return t))
    (unless (increment-abigit-ref 1
                (ABIGIT-REF bn astarti))
      (return nil)) ; 'carry' stops here!
    (incf& astarti)))
```

where `increment-abigit-ref` is a macro:

```
(defmacro increment-abigit-ref
            (amount (ignore bignum index))
  `(primitive-add-into-abigit-ref
       ,amount  ,bignum  ,index))
```

However, `abigit-carry-propogate-thru-bn` is compiled 'in-line' in the few places where it is called. A complete definition of 'primitive-add-into-abigit-ref' follows below.

The "new primitive" function names are listed immediately below, and following that is an interface description of their functional behaviour. The subcategories of division are labelled with the names of the steps in 'Algorithm D' ([Knuth 1981], page 257) wherein they are used.

3.1 *New Primitive Names*

Addition/Subtraction into Abigits
- primitive-add-into-abigit-ref
- primitive-sub-fromoutof-abigit-ref
- primitive-abigit-ref-add-into-abigit-ref
- primitive-abigit-ref-sub-fromoutof-abigit-ref

Basic Multiplication Step
- primitive-mbigit-multiply-add

Division:
- D3: primitive-dbigit-divide
- D3: dbigit-trial-quotient-toobigp
- D4: primitive-dbigit-multiply-sub
- D6: primitive-add-into-dbigit-ref

Word-at-a-time Copying
- bignum-word-replace
- bignum-word-negate-into-bignum
- bignum-word-zero

Word-at-a-time Comparisons
- bignum-word-comparison

3.2 *Interface Specifications*

3.2.1 *Addition/Subtraction into Abigits*

---

fn: **primitive-add-into-abigit-ref**
args: *carry-in x bn i*
fn: **primitive-sub-fromoutof-abigit-ref**
args: *borrow-in x bn i*

Adds (or subtracts) a fixnum into the *i*'th abigit of bignum *bn*, modifying it in place. In the addition case, the fixnum added is x + *carry-in*; in the subtraction case, the fixnum subtracted is x + *borrow-in*. In both cases, the carrys and borrows are either 0 or 1.

Returns the generated carry-out (or borrow-back) as a fixnum 0 or 1.

---

fn: **primitive-abigit-ref-add-into-abigit-ref**
args: *carry-in src i dst j*
fn: **primitive-abigit-ref-sub-fromoutof-abigit-ref**
args: *borrow-in src i dst j*

Adds (or subtracts) the *i*'th abigit of bignum *src*, into the *j*'th abigit of bignum *dst*, modifying *dst* in place. The *carry-in* (or *borrow-in*) is treated exactly as in the function 'primitive-add-into-abigit-ref' (or 'primitive-sub-fromoutof-abigit-ref') described above.

Returns the generated carry-out (or borrow-back) as a fixnum 0 or 1.

3.2.2 *Basic Multiplication Step*

---

fn: **primitive-mbigit-multiply-add**
args: *carry-in multiplier multiplicand addend*

Computes *addend* + *multiplier* * *multiplicand* + *carry-in*, where all arguments are mbigits.

Returns two values, the low-half of the result and the high-half, as mbigits.

3.2.3 *Division*

---

fn: **primitive-dbigit-divide**
args: *divisor dividend-low-half dividend-high-half*

The three arguments are dbigits. *divisor* is divided into *dividend-low-half* | *dividend-high-half*.

Returns two results, the quotient and the remainder of that division, as dbigits. [Remember that "|" here means "juxtaposition"]

---

fn: **dbigit-trial-quotient-toobigp**
args: *Vhi-1 $\hat{q}$ Uj-1 r*

Arguments are the intermediate quantities of 'Algorithm D', step D3, of [Knuth 1981], where an estimation of the next quotient digit, $\hat{q}$, is being made.

Returns non-NIL iff $Vhi\text{-}1 * \hat{q} > r \mid Uj\text{-}1$. [Remember that "|" here means "juxtaposition"]

This test will be true for all cases where the trial quotient digit, $\hat{q}$, is 2 too large, and will be true for almost all cases where $\hat{q}$ is 1 too large. The division algorithm simply decrements $\hat{q}$ until this test is passed; then, on the average, two out of $2^{dbigit\text{-}size}$ trials will pass with $\hat{q}$ still too large by 1, and step D6 of 'Algorithm D' will be necessary.

---

fn: **primitive-dbigit-multiply-sub**
args: *borrow-in multiplier multiplicand minuend*

Used at step D4 of 'Algorithm D' of [Knuth 1981]. This function is almost exactly like primitive-mbigit-multiply-add except that (1) the two additive operations are subtractions instead of additions, and (2) the arguments are dbigit-sized instead of mbigit-sized [but probably dbigit-size is the same as mbigit-size anyway]. Computes a result

*minuend − multiplier * multiplicand − borrow-in.*

Returns two values, the low-order dbigit of the result in two's complement form, and the borrow-back generated, which is non-zero only when the result is negative.

Let the two's complement form of the result be $x \mid y$. The first return value is just y, which can be computed as

```
(ldb (byte dbigit-size 0) result)
```

but in the case when x is non-zero (i.e., when the result is negative), the generated borrow-back is $2^{\text{dbigit-size}} - x$, which can be computed as

```
(ldb (byte dbigit-size 0)
     (- (ldb (byte dbigit-size dbigit-size)
             result)))
```

---

fn: **primitive-add-into-dbigit-ref**
args: *carry-in x bn i*

Used at step D6 of 'Algorithm D' of [Knuth 1981]. This function is exactly the same as 'primitive-add-into-abigit-ref', except that dbigits are used rather than abigits. Note, that even when the dbigit size is extended (to 16, from 8), a single dbigit itself will fit within a fixnum, and this operation only needs 1 bit more than a dbigit-size for intermediate calculations.

Returns the generated carry-out as a fixnum 0 or 1.

3.2.4 *Word-at-a-time Copying*

---

fn: **bignum-word-replace**
args: *dst src dst-starti dst-endi src-starti src-endi*

This primitive is used in numerous functions, and is somewhat akin to the `replace` function of Common Lisp ([Steele 1984], page 252). The words of bignum *dst* are replaced with those of bignum *src*, beginning at word index *dst-starti* of *dst* and word index *src-starti* of *src*; words are 'replaced' up to, but not including, the end indices. If the subsequence intervals specified are not of the same length, then the length of the shorter of the two is taken; if either end index argument is nil, then it is defaulted to the bignum-length of the corresponding bignum.

Returns *dst*.

---

fn: **bignum-word-negate-into-bignum**
args: *borrow dst src dst-index src-index count*

Used in unary minus on bignums, and when converting negative arguments to positive format for multiplication or division. *borrow* must be 0 or 1; if it is 0, then the words of bignum *dst* are replaced with those of the negation of bignum *src*, beginning at word index *dst-index* of *dst* and word index *src-index* of *src*, for a total of of *count* words; if *borrow* is 1, then *dst* is replaced with the complement of *src* rather than the negation. This definition suggests an implementation strategy whereby words are succesively subtracted from 0, with a borrow being propogated.

Returns the final borrow-back. If the argument *borrow* is 0, then there will be a final borrow at the end if and only if the bignum segment of *src* is all zeros.

The algorithms for multiplication and division are actually carried out in signed-magnitude form. Thus negative arguments to these algorithms must first be copied and negated (into a temporarily-allocated bignum of sufficient size); the result, if negative, must also be negated (in place) before returning it. The author has investigated multiplication algorithms which will work

with the two's complement bigits, just as the additive algorithms will so work; but it appears as though the implicit carry-propogations in these algorithms will, on the average, be more costly than a quick copy-and-negate. No such effort has been expended to try to find a division algorithm that will work on complemented bigits; likely there is very little to be gained, since the copy/negate time is so small in comparison to the total divison time.

Negative powers of two, in two's complement form, have all their non-sign bits zero; the only case, then, when 'bignum-word-negate-into-bignum' ought to return a non-zero carry is when the number being negated is a negative power of $2^{word\text{-}size}$. This is the case where 2^N actually needs one more word to be represented than does -2^N (memory allocation is rounded up to words).

---

fn: **bignum-word-zero**
args: *bn nwords*

Used in ASH on bignums. Zeros out the words of bignum *bn*, from word indices 0 through *nwords* − 1.

Returns *bn*.

A function bignum-replace supports the Common Lisp function ASH for bignums, as well as some internal data movement. Whereas 'bignum-word-replace' described above is 'replace' on word boundaries, bignum-replace is 'replace' on bit boundaries.

3.2.5 *Word-at-a-time Comparisons*

---

fn: **bignum-word-comparison**
args: *bn1 bn2 wlength*

Used in function 'bignum-bignum-compare', which supports equality and inequality comparisons.

Compares the two bignums *bn1* and *bn2*, from the word at index *wlength* − 1 down to word 0, and returns

(i) returns 0 if args are equal;
(ii) returns +1 if *bn1* > *bn2*;
(iii) returns −1 if *bn1* > *bn2*;

3.3 *Addition, Multiplication: Easy Examples*

The inner part of the addition and subtraction loop looks something like the code which follows. The smaller of two addends is called 'addend' and is being added into the other one, called 'sum'. [The coding has been changed slightly to facilitate presentation; among other things, we have abbreviated 'primitive-abigit-ref-sub-fromoutof-abigit-ref' by 'parsfar'].

```
(let ((carry 0))
 (dotimes (i (abigit-length addend))
  (setq carry (parsfar carry addend i sum i)))
 carry)
```

After this loop has finished, if carry is non-zero and if the length of sum is greater than that of addend, then a carry-propogation step will take place. [Note: abigit-length is the number of abigits in the bignum]. See the introduction to §3.0, where a sample definition of 'abigit-carry-propogate-thru-bn' is presented.

Virtually all the speed-up on the factorial benchmark is due to increasing the mbigit size from 8 to 16. In order to deal with the intermediate 32-bit result, 'primitive-mbigit-multiply-add' must be coded in machine language; see §2.4 for a Lisp-written version of the primitive. Here is the version for the MC68000 series, when the abigit size is 16:

```
(defprimop primitive-mbigit-multiply-add 4 2
 ;; Basic 4-input, 2-output unsigned
 ;;   multiplier, with additive carry-in
 ;; Returns two values: the low half of
 ;;    the product/sum, and the high-half
 :args ((carry-in dreg)      ;All arguments
        (multiplier dreg)    ; are passed in
        (multiplicand dreg) ; thru data
        (addend dreg))       ; registers
 :effs nil
 :result-loc '(VALUES ,multiplier ,addend)
 :code '((fixnum-to-machine  ,multiplier)
         ;; LAP macro: typically 1 real inst.
         (fixnum-to-machine  ,multiplicand)
         (mulu ,multiplier ,multiplicand)
         (add ,carry-in ,accumulation)
         (fixnum-to-machine  ,accumulation)
          ;; add in the product to the
          ;;  accumulation
         (add ,multiplicand ,accumulation)
         (clr ,multiplier)
          ;; strip out the low-half of the
          ;;  accumulated product
         (movew ,accumulation ,multiplier)
         (machine-to-fixnum  ,multiplier)
          ;; strip out the high-half too
         ((clr w) ,accumulation)
         (swap ,accumulation)
         (machine-to-fixnum ,accumulation)))
```

3.4 *Division is Even Easier*

The paradigm above for 'primitive-mbigit-multiply-add' is, in fact, exactly the same required for 'primitive-dbigit-multiply-sub', which is the important part of the "Classical" division algorithm—step D4 on page 258 of [Knuth 1982]. A little analysis reveals that when dividing a 2n-dbigit number by an n-dbigit number, there will be about n invocations of 'primitive-dbigit-divide'—the various estimations of the trial quotient bigits, which are almost always right (according to a variation of Theorem B on page 257 of [Knuth 1981]). But that division will also require on the order of n^2 invocations of 'primitive-dbigit-multiply-sub'. Thus the "Classical" bignum division algorithm actually trades primitive division steps for primitive multiplication and addition steps.

Let us trace the calculations of 'primitive-dbigit-multiply-sub' when bits-per-dbigit = 4 and the arguments are 3, 2, 5, and 15.

```
(primitive-dbigit-multiply-sub  3 2 5 #xF)
  =>  (- (- #xF (* 2 5)) 3)
  =>  2
```

and thus the low-order dbigit is 2, and the generated borrow-back is 0. But in this example

```
(primitive-dbigit-multiply-sub  6 8 7 9)
  =>  (- (- 9 (* 8 7)) 6)
  =>  -53  =  #x-35
```

the result is negative. Now since
#x-35 = -(16*16) + #xCB, then #xCB is the 2's complement format of #x-35, expressed as two hex digits. Since

```
\#x35 = -1*(16*16) + \#xC*(16) + \#xB(1)
```

thus the low-order dbigit is 11 (or #xB), and the generated borrow-back is $2^4 - \#xC = 16 - 12 = 4$.

The primitive 'dbigit-trial-quotient-toobigp' is the code about which Knuth says on page 258 of [Knuth 1981] "The ... test determines at high speed most of the cases in which the trial $\hat{q}$ is one too large, and it eliminates *all* cases where $\hat{q}$ is two too large." This primitive would be called n times in the example of the previous paragraph, so its time performance may not be critical; but both it and primitive-dbigit-divide would have to go into machine language when the dbigit size is about half the fixnum size or greater.

4. Comparative Performances

A number of timing comparisons are exhibited below, to demonstrate two things: (1) that placing the seven prototype new primitive operations in the compiler's code generator will yield speed-ups of factors from three to seven on some common bignum benchmarks compared to using the purely Lisp-written implementation, and (2) that this minimally-extended Lisp implementation will perform within a factor of three to a factor of seven of the "best possible" machine language implementations [i.e., will not be one or two orders of magnitude slower].

4.1 *Some Typical Bignum Benchmarks*

The factorial function is one of the easiest Lisp programs to remember; inevitably, someone who walks up to a Lisp system will type it in and time it. We use

```
(defun fact (n)
  (if (< n 2) 1 (* n (fact (1- n)))))
```

as the definition of factorial, and time it at n = 1000, not because this is a particularly revealing benchmark, but because it may be very common—people will frequently type it in by hand since it is so easy to do. Then we list four more "micro-benchmarks" which test, respectively (2) bignum-by-bignum division, (3) bignum printout in base 10, (4) bignum-by-bignum addition, and (4) multiplication of a small fixnum by a "small" bignum [to ascertain whether the achievement of speed in the asymptotic limit case has degraded performance on the smaller, common cases].

```
f1000   =  (fact 1000)
f1%f9   =  (truncate f1000 f900),
               where f900 = (fact 900)
Pf1000  =  (print f1000)
+f1000  =  (dotimes (i 1000) (+ f1000 f1000))
20f19   =  (dotimes (i 10000) (* 20 f19)),
               where f19 = (fact 19)
```

Note that one often must use some "sleight of hand" to prevent a compiler from optimizing away the entire DOTIMES expressions in the latter two lines.

Another prominent bignum-oriented benchmark is FRPOLY ([Gabriel 1985], section 3.20) when run on the *r2* polynomial. We timed the 5'th degree case, the 10'th degree case, and 15'th degree case, and we present the results respectively as FR2-5, FR2-10, and FR2-15. The 5'th degree case does not use much bignum arithmetic—161 additions and 260 multiplications of numbers smaller than 2^{128}—these runs were done primarily for comparisons, to see how the amount of bignum arithmetic in this benchmark is increased as the degree of the polynomial increases. FR2-10 does 2424 additions of positive bignums in the range $[2^{128}, 2^{256})$, and 74 smaller additions; it does 2993 bignum-by-bignum multiplications with arguments about equally distributed over the interval $[2^{32}, 2^{256})$. The actual distribution over that interval is probably more like some bell shaped exponential, but the variation between high and low points doesn't appear to be more than about 50%.

FR2-15 is more bignum-intensive. It spends almost all its time in the bignum routines for addition and multiplication, with the time spent in multiplication being about three times greater than that spent in addition [were our multiplication algorithm a faster operation, i.e., closer to the ideal machine-language implementation, this ratio might be closer to two times greater instead].

Here is a histogram of argument sizes for those operations:

Argument Size	Additions	Multiplications
$[2^{32}, 2^{64})$	56	76
$[2^{64}, 2^{128})$	1290	940
$[2^{128}, 2^{256})$	2666	22950
$[2^{256}, 2^{512})$	37422	19800
Totals	41434	43766

Note: every call supplies two arguments; also, the summations for + are not exactly the same as twice the number of calls because there were two arguments in the interval $[2^{30}, 2^{32})$, which is not shown.

4.2 *Timings of Lucid Common Lisp*

The timings of Lucid Common Lisp implementations were run on a Sun-2/160 workstation, a Sun-3/160 workstation, and on a workstation built on a RISC-type machine. The results are listed in Table I. To see the effect of the endian-ness of representation, compare the several trials run in a little-endian format implementation with the later big-endian format for the RISC-type machine, whose natural format is big-endian. In Table I,the column headed "*Impl*" identifies the implementation by its machine name and by the configuration of endian-ness used in the implementation (see §2.6). The column headed *Bigit Sizes* identifies the sizes of the three major bigit categories—an n-tuple $\langle a, m, d \rangle$ means an abigit size of 'a', an mbigit size of 'm', and a dbigit size of 'd'. A double-dagger after the ntuple, like $\langle a, m, d \rangle\ddagger$, means that the `BIGNUM-WORD-REPLACE` primitives were also coded in machine language.

Here is a summary of the meanings for the *Impl* column of Table I, and of Table II:

Impl Entry Name	*Workstation* Machine	*8bit/16bit* Endian-ness
68010L	Sun-2/MC68010	Little/Little
68010B	Sun-2/MC68010	Big/Big
68020L	Sun-3/MC68020	Little/Little
68020B	Sun-3/MC68020	Big/Big
RiscL	RISC-type	Little/Little
RiscB	RISC-type	Big/Big
uVAX-II	MicroVAX/II	Little/Little

The RISC-type machine does memory fetch/store operations in big-endian format, just like the MC68000 series; the VAX does memory operations in little-endian format. Thus the preferred format for the MC68000 and the RISC-type machines is Big/Big; that for the VAX is Little/Little. To see the separate effects of a mismatch between the machine's preferred format and the implementation format, a couple of runs have been made with only this factor varying; in addition, in order to see the incremental effects of increasing the mbigit size, the dbigit size, and the abigit size, there are a series of runs made with just these variations.

As of the writing of this paper, Lucid does not have a complete implementation for the VAX; thus we have timed the various benchmarks on the "vanilla" implementation, which required only the 8-bit and 16-bit fetch-and-store operations to be implemented (the rest being in Lisp). Since there is a rather regular speed-up observed on the other implementations when going from the "vanilla" implementation to an "extended bigit" implementation, we have projected *possible* timings for an "extended bigit" VAX implementation of Lucid Common Lisp in the uVAX-II row with an asterisk.

The timings are measured over numerous runs, and a *reproducible* minimum time is taken. Generally this meant that any run with significant disk-swapping time had to be discarded (but see exception below for the TI Explorer). All of these implementations were done under some form of Unix; therefore, the coarse granularity of the Unix runtime metering must be considered. But still, after discounting disk loading, most of these runs would not vary more than about 2% or 3% greater than the times shown.

TABLE I: Timings of Lucid Common Lisp Implementations

All times are in seconds. Bigit Sizes: $\langle abigit, mbigit, dbigit \rangle$

Impl	Bigit Sizes	f1000	f1%f9	Pf1000	+f1000	20f19	FR2-5	FR2-10	FR2-15
68010L	⟨16, 8, 8⟩	80.	9.78	65.6	53.1	13.4	1.18	26.3	460
68010B	⟨32,16,16⟩†	17.12	2.20	11.52	11.8	10.0	.776	12.32	143.4
68020L	⟨16, 8, 8⟩	20.8	2.48	20.9	14.1	4.44	.386	7.64	125.6
68020B	⟨32,16,16⟩†	4.60	.586	3.0	4.40	3.60	.280	4.14	45.2
RiscL	⟨16, 8, 8⟩	59.2	7.5	46.8	20.7	8.6	.70	17.3	320
RiscB	⟨32,16,16⟩†	10.5	1.30	7.20	5.1	5.1	.40	6.4	77.9
uVAX-II*	⟨16, 8, 8⟩	56.5	7.06	35.2	28.1	10.44			
uVAX-II*	⟨32,16,16⟩†	12.5	1.6	6.2	7.	7.8			

f1000 computes 1000!
f1%f9 truncates 1000! by 900!
Pf1000 prints 1000!
+f1000 adds 1000! to itself 1000 times
20f19 multiplies 20 times 19! 10000 times
FR2 is the Bignum-oriented FRPOLY Benchmark

4.2.1 *Comparison for Specific Effects*

By comparing several stages of implementation on the RISC-type machine, we can note the incremental effect of, say, increasing the dbigit size and bringing the several division-related primitive operations into the code-generator.

These runs were compiled with an earlier version of Lucid's compiler, which produced slightly less efficient code than that used for the runs in Table I above; consequently, they will not be directly comparable. Furthermore, only a couple of runs were made on each benchmark, so the variance of times will be higher (e.g., the difference between 11.2 and 11.4 is not significant, and may not even be reproducible).

All times are in seconds.

Bigit Sizes: ⟨*abigit*, *mbigit*, *dbigit*⟩

Impl	Bigit Sizes	f1000	Pf1000
RiscL	⟨16, 8, 8⟩	59.1	47.1
RiscB	⟨16, 8, 8⟩	62.1	50.2
RiscB	⟨16,16, 8⟩	12.4	46.4
RiscB	⟨16,16,16⟩	11.2	8.5
RiscB	⟨32,16,16⟩	11.4	7.4
RiscB	⟨32,16,16⟩‡	11.3	7.3

TABLE IIA

The following effects are noticeable:

- the time to adjust for endian mismatch in accessing causes a 5% to 7% slowdown, as seen by comparing RiscL⟨16,8,8⟩ with RiscB⟨16,8,8⟩;
- when the mbigit size is doubled (from 8 to 16), the speed of computing large factorials is increased by a factor of 5. In general, bignum-by-bignum multiplication would show a quadratic speed-up—doubling the mbigit size would quadruple the speed—and that effect is visible in f1000 because in all multiplications after the 255'th the multiplier is a double digit bignum rather than a fixnum (because a mbigit size of 8 implies that 255 is the largest fixnum representable in 1 mbigit);
- when the dbigit size is doubled (from 8 to 16), the binary-to-decimal conversion is speeded up by a factor of 5 1/2; the conversion algorithm is not just repeated divisions by 10, but is also subject to the quadratic speed-up mentioned for multiplication.

Results from FRPOLY show

- when the "fast copy" routine `BIGNUM-WORD-REPLACE` is put into machine language, a realistic benchmark—FRPOLY—is speeded up by over 4%. Before dismissing this factor of 1.04 as being of little or no value, one should try to recall how often a couple of straightforward lines of code have yielded a 5% speed-up on a non-trivial benchmark. As the saying goes, "nickles and dimes eventually mount up into dollars."[1]

All times are in seconds.

Bigit Sizes: ⟨*abigit*, *mbigit*, *dbigit*⟩

Impl	Bigit Sizes	FR2-10	FR2-15
RiscL	⟨16, 8, 8⟩	18.1	334.3
RiscB	⟨32,16,16⟩	7.3	87.4
RiscB	⟨32,16,16⟩‡	7.1	83.8

TABLE IIB

Table II does not contain sufficient information to gauge the effect of doubling the abigit size; but Table I is revealing on this matter, when comparing the +f1000 times on the two runs RiscL⟨16,8,8⟩ and RiscB⟨32,16,16⟩‡.

- when the abigit size is doubled, the additive algorithm is immediately speeded-up by a factor of two; in addition, another factor of two appears to come from the fact that the code-generator template for the new primitive operation is "hand tailored" and probably better than the Lisp compiler could do by itself. Net result: a quadrupling in speed. [However, the ratio for the Sun-3/160, a MC68020 processor, was more like a factor of three than a factor of four speed-up; this effect has not been satisfactorily explained yet.]

4.3 *Some Other Common Lisp Implementations*

The same micro-benchmarks were run on several other Common Lisp vendors' implementations, as well as a couple of predecessors to Common Lisp; these results are presented in Table III. PSL, Franz, and NIL, in the latter part of Table III are not Common Lisps, but are in the same family of MacLisp descendents that led to Common Lisp; timings from [Gabriel 1985] for the FRPOLY benchmarks are included for comparison.

The timings for FranzLisp are the cases designated in [Gabriel 1985] as "TrlOn & LclfNo"—tail-recursion elimination permitted, but conversion to "local" functions calls not permitted. This is the configuration which most accurately matches the Common Lisps which were timed. PSL was just barely implemented on the MC68000 architecture when Gabriel's timings were compiled, and it may not have been able to run FR2-15 then; NIL's bignums were put more into machine language sometime after the Gabriel numbers were taken.

---

[1] A cautionary word to the wise: it says "nickles and dimes", not "pennies". Before dickering around with speed-ups below the 1% level, one would do well to reflect upon Masinter's Dictum: "Premature optimisation is The Source of all bugs."—an aphorism variously attributed to Larry Masinter of Xerox Palo Alto Research Center.

TABLE III: Timings of Other Lisp Implementations

All times are in seconds. Numbers marked by asterisk are from [Gabriel 1985]

Lisp/Machine	f1000	f1%f9	Pf1000	+f1000	20f19	FR2-5	FR2-10	FR2-15
Symbolics 3600	3.0	.29	3.16	5.2	4.1	.21	3.0	27.
Symbolics 3600†	2.78	.276	3.03	3.88	3.94	.196	2.93	21.8
Symbolics 3645†	1.86	.184	2.50	2.6	2.90	.147	2.45	15.6
TI-Explorer	.91	.124	1.94	4.23	1.11	.134	2.2	17.8
FranzCL/68010	4.08	.500	15.1	3.23	6.55	.5	5.8	53.
FranzCL/68020	1.22	.150	3.57	.933	1.30	.183	2.1	16.9
VaxLisp/uVAX-II	2.57	7.9	12.8	3.7	3.5			41.3
VaxLisp/VAX-750						.60*	7.25*	57*
NIL/750						2.15*	38.7*	479
PSL/VAX-750						1.58*	27.7*	394
Franz/750						1.18*	8.87*	155
Franz/68010						.85*	16.53	188
PSL/68010						2.04*	37.6*	⟨fail?⟩

f1000 computes 1000!
Pf1000 prints 1000!
20f19 multiplies 20 times 19! 10000 times

f1%f9 truncates 1000! by 900!
+f1000 adds 1000! to itself 1000 times
FR2 is the Bignum-oriented FRPOLY Benchmark

Designation	Machine	Implementor(s) and Trade name
VaxLisp/uVAX-II	MicroVAX/II	Digital Equipment Corp., "VAX LISP"
VaxLisp/VAX-750	VAX-11/750	Digital Equipment Corp., "VAX LISP"
FranzCL/68010	MC68010 (Sun2)	Franz, Inc., "Franz Common Lisp"
FranzCL/68020	MC68020	Franz, Inc., "Franz Common Lisp"
Symbolics 3600	Symbolics 3600	Symbolics, Inc., "Symbolics Common Lisp"
Symbolics 3645	Symbolics 3645	Symbolics, Inc., "Symbolics Common Lisp"
TI-Explorer	TI-Explorer	M.I.T. and Texas Instruments
PSL/VAX-750	VAX-11/750	Univ. of Utah, "Portable Standard Lisp"
Franz/750	VAX-11/750	Franz, Inc., "FranzLisp"
Franz/68010	MC68010	Franz, Inc., "FranzLisp"
PSL/68010	MC68010 (Sun2)	Univ. of Utah, "Portable Standard Lisp"
NIL/750	VAX-11/750	M.I.T. ("NIL": New Implementation of Lisp)

4.3.1 *Benchmarking Technique Tidbits*

The timings on the Symbolics 3600 series appear to be subject to much more variation than those of the Unix-based machines above—timings exceeding more than 50% of those reported above were quite common. In order to get reproducible results, the trials had to be run underneath a `WITHOUT-INTERRUPTS` form. During normal usage, shutting off interrupts would be an untenable situation; but in this study, the focus is on bignum implementation strategies and not on overall machine performance, so its use may be excused. A dagger (†) by the machine name are for those runs in which interrupts were turned off; the other trials were taken normally.

The particular Symbolics 3645 machine used had a faster disk than the other 3600 type machines, and it also had an IFU where the other did not.

The +f1000 run on the TI-Explorer could not be made without significant disk overhead, and the reported time includes that amount; likely, that is why the Explorer makes such a relatively poor showing on this task, whereas generally its microcoded bignums are significantly faster than any other implementation.

It is noteworthy that when the Symbolics (ephemeral) garbage collector is turned off, the 3645 numbers grow significantly worse: the 2.6 seconds of +f1000 becomes 4.5, and the 15.6 seconds of FR2-15 becomes 21. For many bignum-oriented applications, the 'construct-

ing' up of bignums as intermediate returned values, and the subsequent reclamation thereof, places a memory-management load on the system that can dominate the cost of the numerical algorithms. In this case, it was probably the disk swap time to make room for the ever-increasing address space; a properly working incremental garbage collector will tend to re-use recently abandoned cells before resorting to paging activity.

4.3.2 *Comparisons with Symbolics 3600*

Interestingly enough, Symbolics bignums are basically written in Lisp; however, they have a few extra microcoded operations to prevent 'consing' during the low-level arithmetic steps. For example, they have a 2-input, 2-output 32-bit multiply so that their mbigit size is 32, which surely helps explain the 3600's overall good performance for a Lisp-written implementation. By way of comparison, that is very roughly twice as fast as Lucid's implementation on the MC68020, which currently uses an mbigit size of 16 bits; but it is roughly twice as slow as Franz Inc's implementation on the MC68020, which also has an mbigit size of 16 bits, but which has a 'best possible' machine language strategy. [Both Lucid's and Franz Inc's MC68000 series implementations are positioned to run on the MC68010, and do not yet take advantage of any of the extensions available on the MC68020.]

Symbolic's 3600 is descended from the MIT Lisp machine, which had bignums in micro-code. TI's Explorer is also descended from the MIT machine, but is much closer to it than the 3600 is; in fact, the Explorer's bignums are in micro-code. Hence we can understand why the Explorer, which clocks in at about 3/4 the speed of a 3600 on many other benchmarks, surpasses the 3600 by factors up to three and a half on these micro-benchmarks.

4.4 *Comparison of New Primitives with "Best Possible"*

The only implementations directly comparable to the publicly available Lucid Common Lisps are those produced by Franz, Inc. on the MC68000 series of machines. Franz Inc.'s Common Lisp bignums run several of the micro-benchmarks noticeably faster than the other MC68000 implementations; ineed, Franz describes their algorithms as being "in machine language" all the way. So we use them as a comparison point for the Lisp-written ones.

Table IV simply re-lists information tabulated elsewhere, but make it easier to compare the relevant parts.

If the figures postulated for a Lucid Common Lisp on the micro-VAX/II are anywhere near correct, then we could compare them with corresponding times for VAX LISP. An interesting point of departure is that while addition and multiplication are coded in a machine language "best possible" way for VAX LISP, the division algorithm is still in Lisp. This would be noticed by VAX LISP having a factor of four or so edge in the addition and multiplication micro-benchmarks, but in Lucid Common Lisp having a factor of four or so edge in the division micro-benchmark.

5. Conclusions

Several prototype, simple primitive operations have been identified which, when put in the compiler as templates for the code generator, will yield striking improvements in performance for the Lisp-written bignum implementation. Speed-up factors of three to five on realistic benchmarks have been observed. Furthermore, the speeded-up versions were well within the goal of half an order of magnitude of best possible in that the FRPOLY benchmarks in a comparable environment, but with "best possible" machine language coding of bignums, were only from 1.5 to 2.7 times faster; similarly the micro-benchmarks for addition, multiplication, and division were only three or four times faster in the "best possible" version.

These new primitive operations are indeed simple to implement; those for the RISC-type machine mentioned above were programmed and fully debugged in well under four man days by a Lucid employee who knew nothing about the bignum algorithms themselves, and very little

TABLE IV: Timings of Implementations on MC68000 Series

All times are in seconds.

Lisp/Machine	f1000	f1%f9	Pf1000	+f1000	20f19	FR2-5	FR2-10	FR2-15
PSL/68010						2.04*	37.6*	(fail?)
Franz/68010						.85*	16.53	188
Lucid/68010	17.12	2.20	11.52	11.8	10.0	.776	12.32	143.4
FranzCL/68010	4.08	.500	15.1	3.23	6.55	.5	5.8	53.
Lucid/68020	4.60	.586	3.0	4.40	3.60	.280	4.14	45.2
FranzCL/68020	1.22	.150	3.57	.933	1.30	.183	2.1	16.9

about the the structure of the compiler's code generator. Another Lucid employee, working on a port to another kind of machine, coded up the primitives, and their auxiliary LAP macros, in only a couple of hours (but that port has not yet reached the operational stage yet, so it is not possible to debug them yet—we eagerly await his results). This supports the hypothesis that the new primitives are easy for a porter to handle.

Starting from a Lisp-written base is a viable strategy for bignums, and Lucid is not alone in trying it. Apparently declining the option of re-working the MIT Lisp Machine microcode for the 3600, Symbolics opted for a Lisp-written version *even though* they are not in the business of porting their system to a variety of other hardware.

6. Acknowledgements

Thanks to Walter van Roggen of Digital Equipment Corporation for measuring VAX LISP, to Richard Acuff of the Knowledge Systems Laboratory at Stanford University for measuring the TI-Explorer and the Symbolics 3645, and to John Foderaro of Franz Inc. for measuring their Common Lisp. Thanks to Leonard Zubkoff of Lucid who implemented the new primitives for the RISC-type machine, and who compiled Table II above; to Jim McDonald of Lucid who researched the differences between PSL/370 and YKT-Lisp bignums; and to Jim Boyce of Lucid who type-set the tables of §4. Finally, thanks to Jim Boyce and Frank Yellin for help in type-setting, and to Jim McDonald, Harlan Sexton, Robert Stetak, and Leonard Zubkoff for help in proofreading.

References

[Benson 1981] Benson, E. and Griss, M., *SYSLISP: A Portable LISP Based Systems Implementation Language*, Technical Report UCP-81, University of Utah, February 1981

[Brooks 1986] Brooks, R., Posner, D, et. al., *Design of an Optimizing, Dynamically Retargetable Compiler*, Proceedings of the 1986 LISP and Functional Programming Conference, 1986

[Deutsch 1973] Deutsch, L. P., *A Lisp Machine With Very Compact Programs*, Proceedings of the Third International Joint Conference on Artificial Intelligence, Stanford 1973

[Foderaro 1982] Foderaro, J., Sklower, K. **The FRANZ Lisp Manual**, University of California, Berkeley, April 1982

[Gabriel 1983] Brooks, R., Gabriel, R, and Steele, G., *Lisp-in-Lisp: High Performance and Portability*, Proceedings of the Eight International Joint Conference on Artificial Intelligence, Karlsruhe, West Germany, August 1983

[Greenblatt 1977] Bawden, A., Greenblatt R., et. al., *LISP Machine Progress Report*, Technical Report AIM-444, MIT Artificial Intelligence Laboratory, 1977; also published as AD-A062-178 through National Technical Information Service, US Dept. of Commerce, Reports Division, Springfield VA

[Griss 1980] Griss, M. L., *A Portable Implementation of Standard Lisp*, USCG-Operating Note #47, University of Utah, August 1980

[Griss 1981] Griss, M. L., and Hearn, A. C., *A Portable LISP Compiler*, Software Practice and Experience 11:541-605, June 1981

[Hearn 1973] Hearn, A. C., *REDUCE 2 Users Manual*, Technical Report UCP-19, Utah Symbolic Computation Group, University of Utah, 1973

[Knuth 1981] Knuth, D., **The Art of Computer Programming, Vol. II**, Second Edition, Addison-Wesley, 1981

[McDonald 1985] Private communications from Jim McDonald, who did the port of PSL to the IBM/370 at Stanford University's Institute for Mathmatical Studies in the Social Sciences, in 1984.

[Motorola 1985] Motorola, Inc., **MC68020 32-bit Microprocessor User's Manual**, Prentice-Hall, 1985

[Moore 1976] Moore, J Strother, **The Interlisp Virtual Machine Specification**, Technical Report CSL-76-5, Xerox Palo Alto Research Center, September 1976

[Steele 1982] Brooks, R., Gabriel, R, and Steele, G., *An Optimizing Compiler for Lexically Scoped LISP*, Proceedings of of the SIGPLAN '82 Symposium on Compiler Construction, June, 1977

[Steele 1984] Steele, G., **Common Lisp the Language**, Digital Press, 1984

[Utah 1982] The Utah Symbolic Computation Group. *The Portable Standard LISP Users Manual*, Technical Report, Department of Computer Science, University of Utah, January 1982

[White 1979] White, Jon L, *NIL - A Perspective*, Proceedings of the 1979 MACSYMA Users' Conference, 190-199, MIT 1979

[White 1978] White, Jon L, *LISP/370: A Short Technical Description of the Implementation*, SIGSAM Bull. 12,4 November 1978