UUCS-89-006

Cascade: A Hardware Alternative to Bignums

Tony M. Carter
University of Utah
Dept. of Computer Science
3190 Merrill Engg. Building
Salt Lake City, Utah 84112

13 April 1988

Cascade: A Hardware Alternative to Bignums

TONY M. CARTER* (carter@cs.utah.edu)
University of Utah
Dept. of Computer Science
3190 Merrill Engineering Building
Salt Lake City, Utah 84112

Keywords: Variable Precision Arithmetic, Object-Oriented , Computer Arithmetic, Computer Architecture, Signed-Digit Numbers

Abstract. The Cascade hardware architecture for high/variable precision arithmetic is described. It uses a radix-16 redundant signed-digit number representation and directly supports single or multiple precision addition, subtraction, multiplication, division, extraction of the square root and computation of the greatest common divisor. It is object-oriented and implements an abstract class of objects, variable precision integers. It provides a complete suite of memory management functions implemented in hardware, including a garbage collector. The Cascade hardware permits free tradeoffs of space versus time.

1 Introduction

Applications such as solid modeling of geometric objects [20], solving complex sets of equations using Gröbner bases [15], computer algebra [12] and encryption/decryption [3] often involve the use of very high precision arithmetic operations that generally must be implemented using the relatively low precision arithmetic units available in today's computers. In addition, the numbers used in such calculations are of variable length. For example, in Gröbner bases calculations, number lengths vary from a few to several hundred decimal digits. In Thomas' algorithm for combining b-spline surfaces [20], numbers vary in length from a few to over one thousand decimal digits. In encryption and decryption algorithms, the use of very large primes is desirable thereby requiring very high precision arithmetic.

In the world of symbolic computation, Lisp bignums [21] are frequently

*This research was supported by DARPA contract number DAAK11-84-K-0017. The author thanks H. Melenk and W. Neun of FB Mathematik und Informatik der Fernuniversität Hagen for running the bignum timing benchmarks on their Cray and Sun PSL implementations.

used in solutions to problems such as these. There are actually two time related problems encountered in such variable precision arithmetic software packages:

- digit-serial arithmetic based on limited precision arithmetic units, and
- significant memory management overhead.

Software implementations of Lisp bignums are necessarily slow because the limited precision, fixed word-widths of conventional processors require a digit-serial computational model. Digit-serial algorithms for addition and subtraction have $O(n)$ time complexity and those for multiplication and division have $O(n^2)$ complexity [13]. Secondly, storage management in most Lisp systems is a time-consuming operation. This includes both allocation of storage for variable precision numbers as well as increased garbage collection costs. Generally, neither of these storage management operations is directly supported in hardware.

Special hardware can be developed that will significantly accelerate algorithms that depend on the use of variable precision arithmetic. This hardware should permit ready expansion of the width of the arithmetic unit (and therefore the precision of arithmetic operands and results) to a size that more nearly matches the precisions normally required in these algorithms. The hardware should be linearly scalable in area with no time penalty paid for addition and subtraction. The algorithms mentioned above require hardware that is scalable from precisions of a few decimal digits to potentially thousands of digits. An ideal hardware solution would have $O(1)$ time complexity for addition and subtraction, $O(n)$ time complexity for multiplication and division-like operations with only $O(n)$ area cost. In other words, significant time reduction should be achievable without paying an inordinately high price in hardware.

Hardware solutions using conventional techniques such as the two's complement number representation and fast carry lookahead are not viable given such constraints. Full carry lookahead is impractical at large word widths since it has an area complexity of $O(n^2)$ and, in many technologies, actually exhibits a linear slowdown with the number of bits in the operand. Even with a very small constant on the $O(n)$ time complexity of full carry lookahead schemes, the area*time complexity of the full carry lookahead approach is unacceptable at $O(n^3)$. If block carry lookahead is used, the time complexity is reduced from $O(n)$ to $O(\log n)$, but the physical design is still complicated since a tree of block carry lookahead units must be used. In area, the block carry lookahead scheme has $O(n \log n)$ complexity which results in an area*time complexity of $O(n \log^2 n)$ [22] which is still unacceptable. An unconventional solution is required.

Cascade is a hardware architecture designed to accelerate operations on an opaque, abstract class of objects: variable precision integers. It uses, as described below, a redundant signed-digit number representation that eliminates carry propagation, giving O(1) time complexity for addition and subtraction. It is linearly scalable in space so that the arithmetic unit can, with reasonable cost, be expanded to the width required by the problems being solved. It also directly and automatically supports multiple-precision arithmetic operations when the physical word-width is inadequate to represent a number. Cascade is based on a design for a variable precision processor proposed by Chow in [9]. The radix-16 digit slice in Chow's processor has previously been designed and implemented in VLSI [7], [10], [18]. The arithmetic unit in Cascade differs from the one proposed by Chow in some simple, yet significant ways to better support division and extraction of the square root.

In profiling a complex software system (Alpha_1 [11]) written partially in Lisp, we discovered that memory management for objects often required more time than arithmetic (using double precision floating point). It is apparent that the cost of memory management frequently equals or exceeds the cost of arithmetic computation so the speed of memory management operations is at least as important as the speed of the arithmetic to the overall performance of the system. Cascade includes hardware which allocates storage for numbers and which collects garbage, thereby completely encapsulating variable precision integers.

2 Representing Numbers

Cascade uses only redundant, symmetric signed-digit numbers [2]; conversions to and from the two's complement number representation are performed as infrequently as possible and only when requested by an external agent. The representation of numbers is a very important consideration for both speed and circuit complexity.

As noted by Robertson [19], there are two critical parameters that describe a digit-set or the set of values that can be represented by a single digit. They are the diminished cardinality δ which is the number of distinct arithmetic values that a digit can represent minus one, and the offset ω which is the distance of the most negative value from zero. In this paper we denote digit-sets using the notation $< \delta.\omega >$. For example, normal unsigned binary digits are represented as $< 1.0 >= \{0, 1\}$. In particular and for reasons described by Chow in [9], Cascade uses radix-16 $< 20.10 >$ digits for which each digit can assume one of the 21 values in the following set ($\overline{X}$ means $-X$):

$$\{\overline{10}, \overline{9}, \overline{8}, \overline{7}, \overline{6}, \overline{5}, \overline{4}, \overline{3}, \overline{2}, \overline{1}, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}.$$

These $< 20.10 >$ digits are used in determining the architecture of the arithmetic unit with respect to addition, subtraction and multiplication.

Of particular importance is that, through the properties of set addition, digit sets may be represented as weighted sums of smaller digit-sets. For example, a four-bit two's complement number is represented as:

$$\begin{gathered} < 15.8 >= \\ 8 < 1.1 > +4 < 1.0 > +2 < 1.0 > + < 1.0 >= \\ \{\overline{8}, \overline{7}, \overline{6}, \overline{5}, \overline{4}, \overline{3}, \overline{2}, \overline{1}, 0, 1, 2, 3, 4, 5, 6, 7\} \end{gathered}$$

In Cascade, each $< 20.10 >$ digit may be represented as

$$\begin{gathered} < 20.10 >= \\ 4 < 4.2 > + < 4.2 >= \\ 4\{\overline{2}, \overline{1}, 0, 1, 2\} + \{\overline{2}, \overline{1}, 0, 1, 2\}. \end{gathered}$$

This makes it possible to model division using two limited-precision radix-4 steps rather than one radix-16 step [8]. Furthermore, each radix-4 $< 4.2 >$ digit is implemented as $2 < 1.1 > + < 2.0 >$ so that

$$\begin{gathered} < 20.10 >= \\ 4 < 4.2 > + < 4.2 >= \\ 8 < 1.1 > +4 < 2.0 > +2 < 1.1 > + < 2.0 >= \\ 8\{\overline{1}, 0\} + 4\{0, 1, 2\} + 2\{\overline{1}, 0\} + \{0, 1, 2\}. \end{gathered}$$

This makes the design of the Cascade arithmetic circuitry possible using Robertson's Theory of Decomposition [19], [17] and its physical counterpart, Structured Arithmetic Tiling [4], [5] which deal only with binary and ternary digit sets.

3 Cascade's Architecture

As mentioned above, the Cascade hardware is essentially an opaque physical implementation of an abstract class of objects, *variable precision integers*. (Some modifications to its control chip would permit it to operate on normalized fractions as well). It contains its own storage for both numbers and memory management information. The sole connection to the outside world is through a request/acknowledge message interface. Cascade normally returns *handles* to variable precision integers, although it can return the value of a variable precision integer in an extended two's complement form or in its internal number representation if necessary.

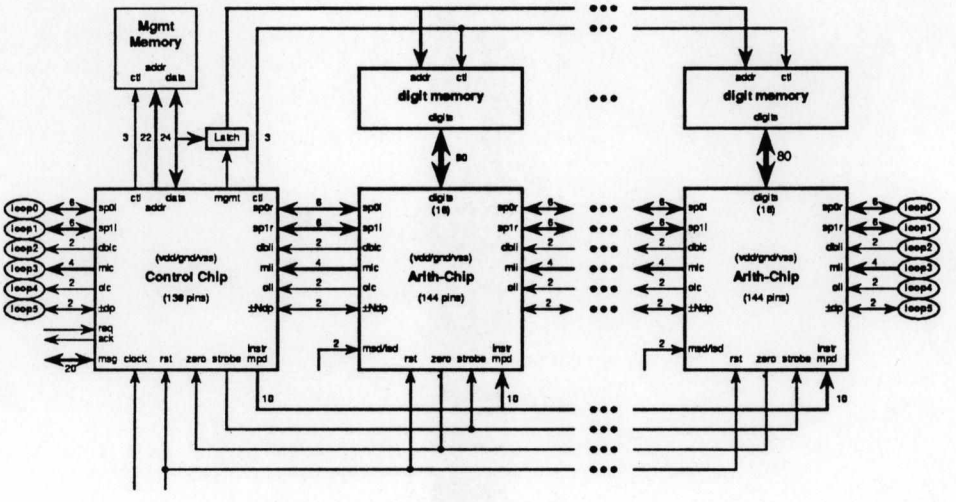

Figure 1: The Cascade Architecture

Cascade is composed of two distinct module types as shown in figure 1. The first is a single *control module* which contains a control chip and associated number management memory. The second is a set of 2^N ($N \geq 0$) *arithmetic modules* each of which contain a single *arithmetic chip* and associated digit memory. The control chip in the control module contains a multi-faceted controller that:

- interacts with external agents via the message port,
- manages the available digit memory through hardware-resident memory allocation and garbage collection algorithms,
- controls single and multiple precision arithmetic operations (+ , - , * , ÷ , $\sqrt{}$, and gcd) on variable precision integers, and
- optimizes both memory management and common arithmetic operations.

The control chip also contains model division hardware for the two-stage, radix-16 division algorithm described in [8].

Cascade can, at the option of an external agent, return a handle to a variable precision integer as soon as storage has been allocated but before its value has been computed. This capability, known as *arithmetic futures*,

permits external agents that use the Cascade hardware to proceed without having to wait for a value to be computed. Arithmetic futures are possible because the Cascade hardware completely encapsulates all aspects of variable precision integers; all references to these are via handles.

Figure 1 shows the architecture of Cascade. A detailed description of this architecture and the operation of the hardware is presented in [6]. The signal loops labelled *sp0* and *sp1* are shift paths, capable of shifting right or left by whole or half digits. The loops labelled *dbl*, *ml*, and *ol* are transfer digit paths (in signed-digit arithmetic the transfer digit is the analog of a carry). The bottom loop "±Ndp" is used for sign computation, normalization detection and for controlling the insertion of a root digit into an accumulating square root. At the top of each custom chip there is a memory interface. The control chip originates all memory control signals to both management memory and digit memory.

At the lower left of the control chip there is the message port which consists of request/acknowledge lines and a twenty-bit data bus through which handles, values and other information are passed between external agents and the Cascade hardware. The control module generates a ten-bit instruction word that is broadcast to the arithmetic modules where it is decoded and applied to control points within the arithmetic chips. The *sdv* signal is an open-drain bus driven by all the arithmetic chips. It is used by the control chip to detect when an arithmetic operation results in zero or other values that are represented as a single digit. The control chip can then optimize storage use and subsequent arithmetic operations that involve very common values such as 0, 1 and -1.

4 Arithmetic Modules

Each arithmetic module consists of a 16-digit (80-bit) wide digit memory and a custom arithmetic chip containing a 16-digit slice of the arithmetic datapath (roughly equivalent to 64-bits). Figure 2 shows the structure of the arithmetic chip. The XL box encodes each six-signal $< 20.10 >$ digit as a five-signal $< 31.10 >$ digit for storage in digit memory. The LX box is the inverse of this operation. Thus the storage overhead in Cascade over what would be required in a normal two's complement system is only 25%.

The shift paths described above interface directly to four 16-digit (96-bit) registers enabling any of these to be shifted. Each of these registers can serve as input to either port of the arithmetic unit and can latch the output of the arithmetic unit. Under control of the *root digit position register*, any given digit in a register can store the value of the current root digit during square root extraction. This permits the unknown root to be accumulated in position, left to right.

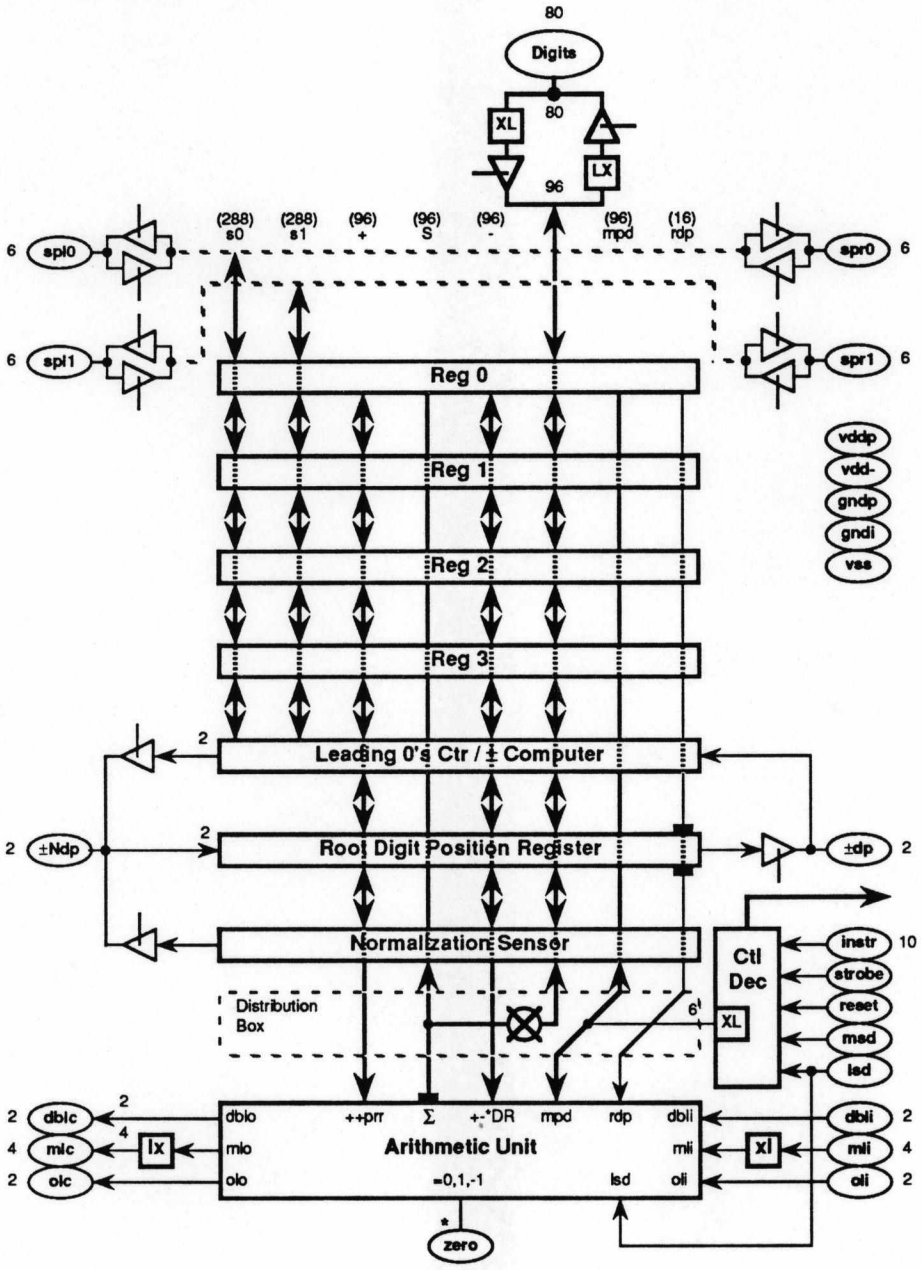

Figure 2: Cascade's Arithmetic Chip

The addition of two large numbers with opposite sign may result in a number of much smaller magnitude. The size of this result cannot be predicted before the operation, so the number of leading zeros must be computed following each addition or subtraction. This calculation is done in the *sign computer/leading zeros counter* by having each arithmetic chip count the number of leading zeros in the segment of a number that it contains. This will be a number between 0 and 16. The collection of arithmetic chips then shifts these counts to the left using a shift path and the control chip accumulates the number of leading zeros until a count of less than 16 is encountered.

In a signed-digit number, the sign of a number is determined by the sign of its most significant non-zero digit. The *sign computer* uses a priority encoding scheme to find the sign of the most significant non-zero digit and to report it to the control chip so that it can be cached in the descriptor for that number. This is critical in accelerating comparison operations; if the sign of two numbers is known as well as the number of digits in each, many comparisons may be computed without a subtraction when the number of digits differs.

The *normalization sensor* examines the three most significant digits of a number to see if any more normalization operations are required. Under control of an instruction received from the control module, the arithmetic chip can detect radix-16, radix-4 or radix-2 normalization.

In the *distribution box*, the output of the arithmetic unit can be directly connected to the digit-memory bus so the result of an arithmetic operation can be stored to memory without first being moved into a register.

At the bottom is the arithmetic unit. It is composed of sixteen identical radix-16 digit slices (described in section 4.1). The arithmetic unit contains zero detecting circuits at all digital positions. It also contains a small *single digit value* detecting circuit at the least significant digital position to enable the detection of results that are represented as a single digit. This permits the control module to detect values like 1, 0 and -1 as the result of an arithmetic operation. When such values can be detected and when an arithmetic future is not requested, storage need not be used for them and operations such as multiplication by zero, one, two or four can be dynamically optimized in the hardware.

4.1 The Radix-16 Digit Slice

The heart of the arithmetic chip is the radix-16 digit slice pictured in figure 3. It has a conditional *doubling circuit* (an adder plus a multiplexor) used during extraction of the square root. During square root extraction

using completion of the square, for which the recursion equation is

$$p_{j+1} = rp_j - q_{j+1}r^{-(j+1)}(2Q_j + q_{j+1}r^{-(j+1)}),$$

all root digits are doubled except the most recently generated, as indicated by the root digit position register. This doubling circuit is not present in the digit-slice proposed by Chow in [9].

A $< 20.10 >$ digit is broadcast to all digital positions of the arithmetic unit for use during multiplication, division and square-root extraction. There, an *elementary multiplier* performs the radix-16 multiplication of a multiplicand digit by the multiplier digit. The output of the *elementary multiplier* is sent to the *m0 adder* which transforms the result into a $16 < 12.6 >$ transfer digit, a $< 8.4 >$ sum digit that is recombined in the *m1 adder* with the incoming $< 12.6 >$ transfer digit to form a $< 20.10 >$ digit, and a $4 < 2.1 >$ digit that is passed on to the normal addition circuitry. Just below the *m1 adder* is a pair of multiplexors. During multiplication, division and extraction of the square root these multiplexors pass on the output of the multiplication circuitry.

There is a pair of conditional complementing circuits just below these multiplexors to permit subtraction by addition of the complement, assisting in division by permitting the recurrence equation $p_{j+1} = rp_j - q_{j+1}d$ to be computed in a single step. The location of these conditional complementing has been changed from Chow's proposed digit slice. Below the conditional complementers is the two-level signed-digit addition circuitry, the *a0 adder* and the *a1 adder*.

5 Control Module

As mentioned above, the control module contains memory for managing variable precision integers, a message port for interfacing with external agents, a state machine that sequences operations for single and multiple precision arithmetic operations, and a model division that generates radix-16 quotient digits as described in [8]. The following sections describe the message interface and the hardware-resident memory management scheme.

5.1 External Message Port

The interface to the Cascade hardware can be summarized by the set of variable precision integer manager methods included as figure 4, most of which relate directly to variable precision integers although some are required to initialize and inquire about the status of the Cascade hardware. In figure 4, "mvr" is shorthand for "multiple-value-return" indicating that more than one cycle of the message port will be required to transfer the

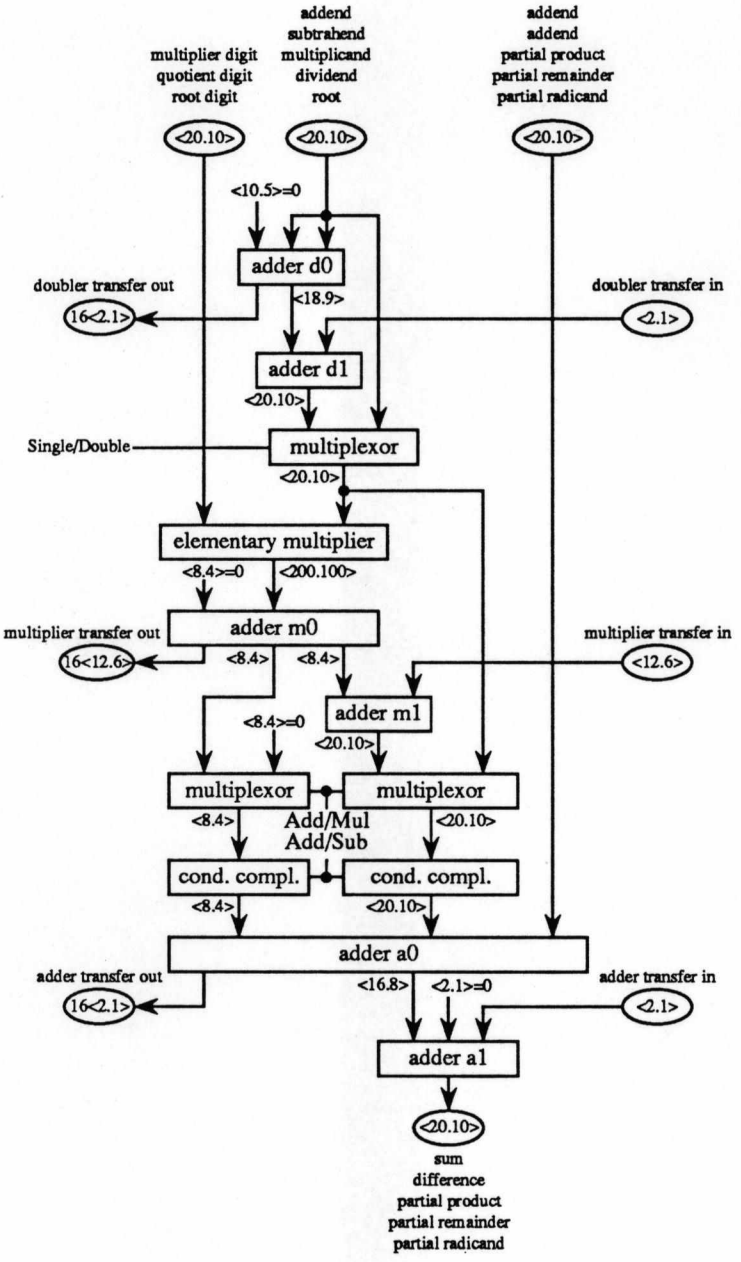

Figure 3: Cascade's Digit Slice

```
(defun vpimgr::create    (ms-16-bits ls-16-bits)      (return Handle))
(defun vpimgr::destroy   (Handle)                     (return nil))
(defun vpimgr::assim     (Handle)                     (mvr
                                                       #16B-chunks chunks))
(defun vpimgr::save      (Handle)                     (mvr desc0 desc1
                                                       #4D-chunks chunks))
(defun vpimgr::restore   (desc0 desc1 #4D-chunks      (return Handle))
                          &rest chunks)
(defun vpimgr::neg       (Handle &optional f d i)     (return Handle))
(defun vpimgr::add       (Handle-add Handle-add       (return Handle-to-sum))
                          &optional f d)
(defun vpimgr::sub       (Handle-add Handle-sub       (return Handle-to-diff))
                          &optional f d)
(defun vpimgr::mul       (Handle-mpy Handle-mcd       (return Handle-to-prod))
                          &optional f d)
(defun vpimgr::div       (Handle-num Handle-den       (return Handle-to-quot))
                          &optional f d)
(defun vpimgr::sqrt      (Handle-rad &optional f d)   (return Handle-to-root))
(defun vpimgr::rem                                    (return Handle-to-rem))
(defun vpimgr::gcd       (Handle-u Handle-v           (return Handle-to-gcd))
                          &optional f d)
(defun vpimgr::cmp       (Handle-add Handle-sub       (return comparison))
                          &optional d)
(defun vpimgr::sign      (Handle &optional d)         (return sign-indicator))
(defun vpimgr::digits    (Handle &optional d)         (mvr ms-ndig ls-ndig))
(defun vpimgr::setreg    (Register Bit-Pattern)       (return nil))
(defun vpimgr::getreg    (Register)                   (return Bit-Pattern))
(defun vpimgr::gc        nil                          (return nil))
```

Figure 4: Cascade's Message Formats

results of the operation back to the host. The optional f, d and i parameters indicate, respectively, that a future is desired, that the arguments (one or both) are to be destroyed following the operation and that the operation should be performed in place. These messages are received and replied to through the external message port.

With the exception of the **gc** message, the transmission of a message to Cascade and the return of a result always requires multiple cycles of the message port. A single data transfer cycle of the message port suffices to transmit the message name and any control flags (f, d and/or i). Most messages also require one or two additional cycles to transmit the operands to Cascade and one cycle for Cascade to return the result. Messages which transfer values of variable precision integers (**assim, save, restore**) re-

quire a variable number of cycles.

The **create** message takes a single 32-bit two's complement value as an argument (passed in two 16-bit halves) and returns a handle to a new variable precision integer whose value is equivalent to the argument. The **destroy** message simply marks a variable precision integer as garbage. The garbage collection and memory allocation algorithms will subsequently reclaim and reuse the storage allocated to the destroyed number. The **assim** message is essentially an inverse of **create**. It returns, as an extended (long) two's complement number, the value of a variable precision integer.

The **save** message returns three things, a 40-bit pseudo-descriptor containing the sign and number of digits in a variable precision integer, a count of the number of transfers required to transmit the value of a variable precision integer (in internal format) in 4-digit chunks, and a series of 4-digit chunks. The **restore** message takes this information and rebuilds a variable precision integer from it, returning a handle.

The **neg**, **add**, **sub**, **mul**, **div**, **rem**, **sqrt** and **gcd** messages take handles as their arguments and return handles as a result. The **cmp** message returns a comparison indicator (one of $<, >, =$). The **sign** message returns the sign of a number (either $+$ or $-$) and the **digits** message returns the number of digits in a variable precision integer.

The **getreg** and **setreg** messages are used to read and write setup registers internal to the control chip. The **gc** message invokes Cascade's garbage collector and returns immediately (garbage collection is a *future* operation — a big performance win). The receipt of subsequent messages is blocked until the garbage collection has been completed. The registers can then be read to find out how much free memory is available.

The external message port consists of a request/acknowledge signal pair and a twenty-bit data bus. When an external agent wishes to send a message to the Cascade hardware, it first asserts its data and then raises the request line. When the Cascade hardware is ready, it latches the data and interprets it. When done, it raises the acknowledge line until the external agent lowers the request line. The number of request/acknowledge cycles of the external message port required to fully transmit a message and receive the reply varies as described above. This interface permits the Cascade hardware to be interfaced to any host with a minimal amount of extra hardware to move data from the host's bus to the Cascade hardware.

5.2 Memory Management

Since the Cascade hardware is an opaque implementation of variable precision integers and since variable precision integers are dynamically created and destroyed, Cascade must perform its own memory management func-

tions. It contains two physically distinct, separately addressed memories: management memory and digit memory. Management memory has a fixed word-width (22 bits) while the word-width of digit memory is dictated by the width of the arithmetic unit being built. The structure of the Cascade arithmetic chip requires that digit memory be at least 16 digits wide. Each digit in memory is represented by 5 bits, making digit memory at least 80 bits wide. To avoid the use of division in the Cascade memory management scheme, Cascade requires that the arithmetic unit and its associated digit memory be expanded by powers of two.

Agents external to Cascade refer to variable precision integers through handles. This permits Cascade to relocate variable precision integers in a hardware controlled garbage collection scheme. The use of handles rather than a direct address also completely hides the implementation of variable precision integers from the outside world. Most of the arithmetic methods supported by Cascade require the allocation of at least one new variable precision integer (multiplication and square root extraction require two and division requires three).

The size of a handle is an important system design consideration. Ideally, a handle would never be reused, but this is not practical. Consider the following equation which, given the useful lifetime of a system (L_s) and the steady-state frequency of object creation (F_c), indicates the number of bits (B_h) required in a handle if handles are never reused.

$$B_h = \log_2(L_s F_c)$$

If, for example, a system is to have a useful lifetime of 10 years and a new object is created every millisecond during that time then the size of a handle would be 39 bits. Even to run for one day without reusing a handle under an object creation frequency of only one thousand objects per second would require 27 bit handles. Consider also that we need to mark each handle with one bit, indicating that the object to which it refers has been destroyed. This one management bit alone would require nearly 17 megabytes of storage every day for the useful lifetime of the system! Clearly, reusing handles to objects is required in a practical and cost-effective system.

In the Cascade hardware, a handle is reusable as soon as the variable precision integer to which it refers is destroyed. There are three conceptually separate memory resident objects in Cascade: descriptor pointers (referred to directly by a handle), descriptors (referred to by descriptor pointers) and words of digits (referred to by descriptors). The descriptor pointers and descriptors are contained in management memory while digits are contained in digit memory.

Figure 5 shows the memory structure of Cascade. Cascade uses 20-bit handles; it can only manage about 1 million specific variable precision integers at any one time. The storage location in management memory that is referred to by a handle contains three data items: a bit indicating that the handle is currently free, a bit indicating that the variable precision integer referred to through the handle has been destroyed but has not yet been reclaimed by the garbage collector and a 20-bit reference to a variable precision integer descriptor. This reference is converted to a pointer by concatenating two extra bits on the end.

A variable precision integer descriptor contains four words (thus requiring a 22-bit pointer since the required size of descriptor memory is four times larger than that of the descriptor pointer memory). The first word contains a garbage bit and a pointer to the related handle. The handle pointer is used in the garbage collection algorithm which operates primarily on descriptors but which must update some information in descriptor pointers. The second word indicates the sign of and the number of significant digits in the number. The remaining two words indicate the bounds of the variable precision integer in digit memory. The third word is the address of the most significant word in digit memory and the fourth is the address of the least significant word in digit memory. This structure limits the size of digit memory to 4 mega-words. It also "limits" the size of any single variable precision integer to 2 mega-digits.

Digit memory is allocated from the top down. The Cascade hardware maintains a register containing a pointer to the top-most available word in digit memory. The garbage collection algorithm compresses all non-garbage descriptors to the top of management memory and all non-garbage digit-words to the top of digit memory.

Handles refer to descriptor pointers that are allocated from zero up. Cascade maintains a register containing the handle last allocated. The circular handle reuse algorithm tries to find a free handle from that point up to the top of descriptor pointer space and back around to that point. The use of the circular handle reuse algorithm dictates that the initial allocation pattern will proceed from the bottom of the handle namespace to the top. It is based on the assumption that older variable precision integers are more likely to be destroyed than ones that were just created. The circular handle reuse algorithm permits the allocation of space for a new variable precision integer to be made as rapidly as possible. In addition to using free handles that have either never been used or have been garbage collected, the allocation algorithm will also reuse free handles that have been used but which have not been subjected to the garbage collector if there is enough digit memory already allocated. The destruction of a variable precision integer consists simply of marking it as free and as

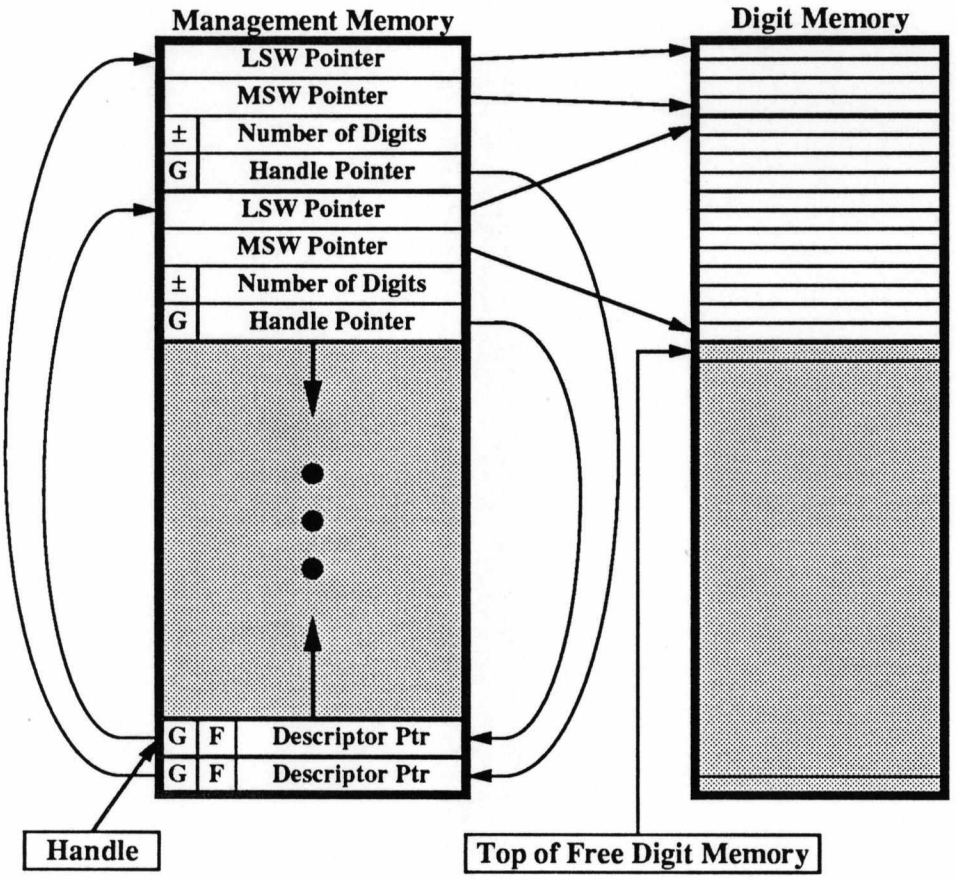

Figure 5: Cascade's Memory Organization

garbage (in both the descriptor pointer and the descriptor). The storage for it will automatically be directly reused or garbage collected in Cascade using the pseudo-coded allocation and garbage collection algorithms that follow below. The statements in {} are not valid Common Lisp.

```
(defun vpimgr::allocate_vpi (op vpi1 vpi2)
  (let ((w  0) (h  0) (free-handle 0))
    (setf w (maximum-result-size-in-words op vpi1 vpi2))

    % search for a usable handle and enough digit memory

    (loop with h from (1+ last-handle-allocated)
                to   last-handle-allocated
                by 1
          do (when (and {handle h is free}
                        {handle h does not refer to garbage}
                        {there are at least w words of digit
                            memory available})
               {assign the next free descriptor to handle h}
               {assign w words of digit memory to that descriptor}
               {make handle h not free}
               (return h))
             (when (and {handle h is free}
                        {handle h does     refer to garbage}
                        {that garbage has at least w words of
                            digit memory})
               {make handle h not free}
               {make handle h not contain garbage}
               (return h))
             (when ({handle h is free}) % keep searching, avoid gc
               (setf free-handle h))
             (when ({there are no free handles})
               (return {an error indicating no free handles}))
             (when (= h top-handle) (setf h bottom-handle)))

    % there is a free handle but not enough digit memory

    (collect_garbage)
    (when ({there is enough digit memory available after gc})
      {assign the next free descriptor to handle free-handle}
      {assign w words of digit memory to that descriptor}
      {make handle free-handle not free}
      (return free-handle)))
    (return {an error indicating not enough digit memory}))
```

```
(defun vpimgr::collect_garbage) ()
  (let ((d  nil)                              % temp descriptor ptr
        (distance-descriptors-rise 0)
        (distance-digits-rise      0)
        (u 0)
        (w 0))
    (loop with d from top-descriptor
                 downto top-available-descriptor
          do (setf w {\# of words of digit memory referenced by d})
             (if ({d contains garbage})
               (progn
                 (incf distance-descriptors-rise)
                 (incf distance-digits-rise w)
                 {re-init descriptor d}
                 {re-init the descriptor-pointer that refers to d})
               (progn
                 {move d upwards by distance-descriptors-rise}
                 (setf u
                       {\# of UNUSED digit words referred to by d})
                 (incf distance-that-digits-rise u)
                 {move each of the USED digit memory words referred
                     to by d upwards by distance-digits-rise}
                 {adjust the msw and lsw pointers of d to reference
                     the relocated block of digit memory}))))
    (incf top-of-available-digit-memory distance-digits-rise)
    (incf top-available-descriptor distance-descriptors-rise)))
```

These memory management algorithms are quite simple to implement in hardware. When we restrict the width in digits of an a Cascade arithmetic unit to be a power of two, a simple 22-bit two's complement adder and a shift register (or barrel shifter) suffice to implement the arithmetic required for memory management. A few registers for holding handles, descriptor pointers, descriptors and a few other useful quantities are required. The sequencing is done via a state machine that controls the complete Cascade system including memory management, arithmetic operations and interfacing with external agents. The state machine can be implemented to perform certain portions of the memory management algorithms in parallel such as storing the previous and fetching the next descriptor for garbage collection while moving words in digit memory for the current descriptor.

Physically, management memory is bifurcated and contains both descriptors and descriptor pointers (which never move and are referred to by handles). Descriptors are maintained such that there are no crossing pointers

into digit memory, thereby facilitating garbage collection.

The memory management strategy attempts to avoid garbage collection if at all possible. If a number has been destroyed but has not yet been reclaimed by the garbage collector and if it has adequate storage for the next result it will be reused immediately without garbage collection. A setup register in the control chip permits less than 4 megawords of real memory to be installed in the system.

5.3 Model Division

The SRT division algorithm used in Cascade is described in [8]. The model division used a three digit estimate of the divisor and a two digit estimate of the partial remainder. The three digit estimate of the divisor is stored in a special register since multiples of it must be constantly computed as part of the model division. The model division produces radix-16 quotient digits that are broadcast to all arithmetic chips where they are multiplied by the divisor and subtracted from the current partial remainder in a recursion step. The computation of the quotient digits takes place with very limited precision while the computation of the next partial remainder is a full-precision multiply/subtract operation.

6 Performance Estimates

A software model of this architecture has been simulated to verify correctness of the control algorithms for the arithmetic operations. Detailed SPICE [16] simulation of the individual arithmetic circuit modules (called operators) that make up the arithmetic unit have been done. Cascade's estimated operation execution times are summarized in Table 1. Message reception is not accounted for in this table since procedure call overhead was factored out in the performance benchmarks. These timing estimates are based on a slightly pessimistic 25 MHz clock rate.

The quotient/root digit selection hardware has not yet been fully designed so the speed of division and square-root extraction cannot be determined exactly. However, we estimate that quotient digit selection can be done in approximately 240 ns. With the exception of the first four digits of the root, root digit selection is as fast as quotient digit selection.

A critical issue in addition and subtraction is memory access time since three or four memory cycles are required (two to fetch the operands and one or two to store the result). If fast static RAM is not used as suggested in table 1 then the times for addition and subtraction slow down dramatically. In general, memory cycle time has less effect on the execution times of single-precision multiplication and division than on addition

Table 1: Operation Delays

Descriptor Fetch	t_d	160 ns
Descriptor Store	t_D	160 ns
Operand Fetch*	t_f	40 ns
Result Allocation*	t_a	80 ns
Result Store*	t_s	40 ns
Raw Add (N digit)*	t_+	40 ns
Raw Mul (1 digit)	t_*	120 ns
Raw Div (1 Q digit)	$t_/$	320 ns
Raw Sqrt(1 R digit)	$t_\surd$	360 ns
Compute Sign/N_d	$t_{\pm N_d}$	$\lceil (N/16) \rceil * 200$ ns
Add (N digits)	$2t_d + t_D + t_{\pm N_d}$	$480 + \lceil (N/16) \rceil * 200$ ns
Mul (N digits)	$2t_d + t_D + Nt_*$	$480 + N * 120$ ns
Div (N Quot. digits)	$2t_d + 2t_D + Nt_/$	$640 + N * 320$ ns
Sqrt (N Root digits)	$t_d + 2t_D + Nt_\surd$	$480 + N * 360$ ns

* Can be easily overlapped with t_d and/or t_D.

and subtraction.

Memory management functions are dependent on the speed of memory as well as on dynamically varying distributions of allocated memory blocks. Depending on memory speed, it will take approximately 200 nanoseconds to retrieve a descriptor given a handle. The allocation of a new descriptor when a search for an unused descriptor is not required would take approximately 240 ns. A complete scan of a maximally sized management memory to find a free handle would require approximately 60 milliseconds; in the average case, this operation would take around 1 microsecond. Garbage collection takes about 600 nanoseconds for garbage numbers and about 1.2 microseconds for numbers still in use (exclusive of the time to move digits which depends on the average size of a number). Considering an average case where half of the numbers are garbage with 2 words in each number, about 960 nanoseconds will be required for each descriptor. In the infrequently occurring worst case, allocation involving garbage collection would take approximately one second.

Table 2 shows measured performances of HP Common Lisp (Rev. 2.01), Lucid Common Lisp (HP Rev. A.2.01), Utah's Portable Standard Lisp and the estimated performance of the Cascade hardware for operand sizes varying between 1 and 1023 radix-16 digits. The operations studied in this table are addition, multiplication, division and extraction of the square root. The functions computed were $n + 1$ for addition, $n * (n - 1)$ for

multiplication, $(n/3)$ for division and isqrt(n) for square root extraction. These were all taken so that measurement error was less than 0.5%. In HP Common Lisp, care was taken to avoid garbage collections during the measurements since the `time` function does not factor it out. The overhead for loop control and function application was factored out so that only the time involved in the arithmetic function computation is included. The code used in these measurements is shown schematically as follows:

```
(defvar nMMMM (- (expt 16 (1- MMMM) 1))

(defun null-fn (x y) nil)
(defun add-fn  (x y) (+ x y))
(defun mul-fn  (x y) (* x y))
(defun div-fn  (x y) (/ x y))
(defun sqrt-fn (x y) (isqrt x))

(defun timing (n f x y)
  (gc)
  (time-subtract
    (time (dotimes (i n) (apply f (list x y))))
    (time (dotimes (i n) (apply 'null-fn (list x y))))))
```

Figures 6, 7, 8 and 9 plot these raw results and give a more intuitive feel for the results. There are several anomalies in the charts that deserve explanation. First, from the timings of division in Lucid Common Lisp, it is evident that a convergence division algorithm is being used. Second, measurements were made starting within the fixnum number range, so the measured performances of Lisp arithmetic exhibit a discontinuity in the curve where the computation enters the domain of bignums. Third, multiplication in Lucid Common Lisp exhibits an abnormal peak when $\log_2(digits + 1)$ is 3. This is most likely due to multiplying two fixnums and having a bignum result for every partial product as well as the final result. The peak is probably due to excessive number conversion overhead. It could most likely be eliminated by preconverting the two fixnum arguments to bignums and then proceeding with the multiplication.

7 Conclusion

Cascade is a hardware architecture designed specifically for performing arithmetic operations on high/variable precision integers. Relative to Common Lisp bignums running on a professional workstation, it is possible to realize speed improvements of several orders of magnitude in arithmetic computations involving bignums. The use of the Cascade hardware should

accelerate bignum addition by two to three orders of magnitude over the number ranges considered in this paper. Multiplication will be accelerated by one and one-half to three orders of magnitude. Division will be accelerated by two orders of magnitude at relatively small number sizes and by one and one-half orders of magnitude at larger number sizes. Square root will be accelerated by between two and five orders of magnitude. Additional state-machine circuitry and possibly some additional arithmetic circuitry could render it possible to compute other functions (natural logarithm, exponential, tangent or cotangent, sine, cosine and arctangent) on bignums with the same kind of performance increase seen for square root using the algorithms proposed by DeLugish [14].

The Cascade hardware exhibits only a moderate performance increase over worst-case optimized PSL [1] bignums running on a CRAY XMP; 20-30 times for addition, 10-20 times for multiplication, and only very small increases for division. It must be noted that the Cray PSL implementation of bignums has been optimized to utilize the Cray vector registers while PSL on other machines is straight Lisp code. If a more average case analysis were done it is expected that a fourfold decrease in running times for PSL bignums running on a CRAY XMP would be seen, making the Cascade hardware at best only a few times faster. The Cascade hardware, however, would cost significantly less and could easily be attached to a professional workstation making high-performance bignum arithmetic available to a much larger community. Its linear extensibility in space permits free tradeoffs of time versus space. The inclusion of hardware memory management functions in the object-oriented Cascade processor also implies that garbage collection time and memory usage in Common Lisp running on a professional workstation would be significantly reduced for applications that rely heavily on bignum arithmetic. This directly and significantly impacts user response time, which for interactive systems is the main concern.

Table 2: Performance Comparisons

Op.	Opnd Size (digits)	Per Operation Execution Times (in μS) HP CL (HP 9000/350)	Lucid CL (HP 9000/350)	PSL (Sun 4/260)	PSL (Sun 3/60)	PSL (Cray)	Cascade (est.)
Add	**1**	6.25	3.84	6.89	19.93	0.38	0.68
Add	**3**	5.92	3.01	7.23	20.06	0.39	0.68
Add	**7**	21.88	4.33	84.70	355.80	1.11	0.68
Add	**15**	136.70	105.20	65.62	262.80	19.54	0.68
Add	**31**	163.90	113.30	73.78	294.10	21.78	0.88
Add	**63**	219.50	126.10	82.96	319.60	40.84	1.28
Add	127	322.10	180.80	96.56	369.90	64.24	2.08
Add	255	540.80	257.00	130.60	459.70	143.80	3.68
Add	511	1038.00	427.10	185.00	696.30	80.00	6.88
Add	1023	1855.00	779.90	304.50	1109.00	132.40	13.28
Mul	1	12.80	13.47	9.96	24.85	0.63	0.60
Mul	3	12.59	12.93	10.84	25.94	0.56	0.84
Mul	**7**	324.10	227.70	75.02	335.70	19.93	1.32
Mul	**15**	672.90	103.30	81.76	307.10	13.70	2.28
Mul	31	2287.00	162.70	268.90	703.90	21.88	4.20
Mul	**63**	8669.00	385.60	710.60	1713.00	37.11	8.04
Mul	127	34200.00	1170.00	2523.00	5127.00	51.56	15.72
Mul	255	136200.00	4270.00	9456.00	16680.00	87.50	31.08
Mul	511	415750.00	16490.00	31030.00	58650.00	500.00	61.80
Mul	1023	2320000.00	64600.00	68000.00	137700.00	1500.00	123.24
Div	1	37.64	118.10	11.93	28.02	1.34	0.96
Div	3	37.84	253.30	15.05	28.53	1.34	1.60
Div	**7**	37.43	532.50	250.10	437.90	2.29	2.88
Div	15	13660.00	968.50	327.90	363.20	25.51	5.44
Div	31	40920.00	1114.00	747.10	390.10	29.30	10.56
Div	63	107900.00	1333.00	1162.00	498.10	46.88	20.80
Div	127	291900.00	1821.00	2890.00	510.00	54.69	41.28
Div	255	859000.00	2830.00	4420.00	850.00	93.75	82.24
Div	511	2852000.00	4881.00	9265.00	1190.00	125.00	164.16
Div	1023	10320000.00	9192.00	17680.00	2040.00	500.00	328.00
Sqrt	**1**	150.80	83.01				0.84
Sqrt	**3**	399.90	202.60				1.20
Sqrt	7	5507.00	405.30				1.92
Sqrt	**15**	44480.00	23280.00				3.36
Sqrt	31	188900.00	66720.00				6.24
Sqrt	**63**	1100000.00	172500.00				12.00
Sqrt	127	7682000.00	476000.00				23.52
Sqrt	255	58340000.00	1357000.00				46.56
Sqrt	511	458500000.00	4221000.00				92.64
Sqrt	1023	3635000000.00	14840000.00				184.80

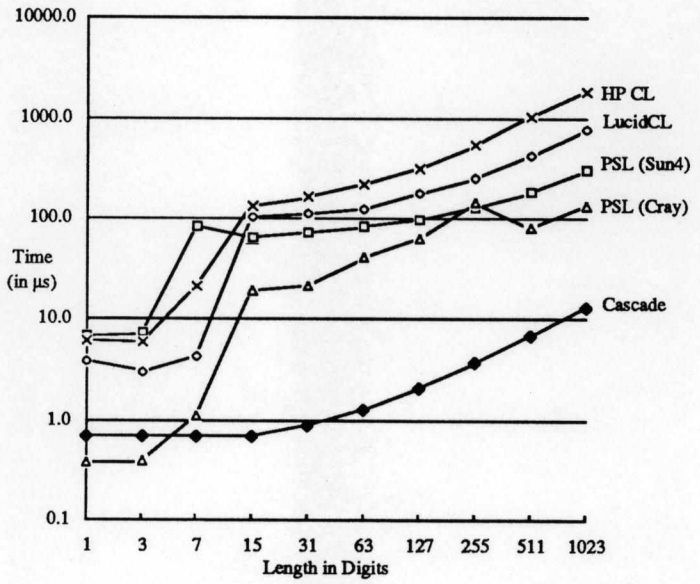

Figure 6: Addition/Subtraction Performance

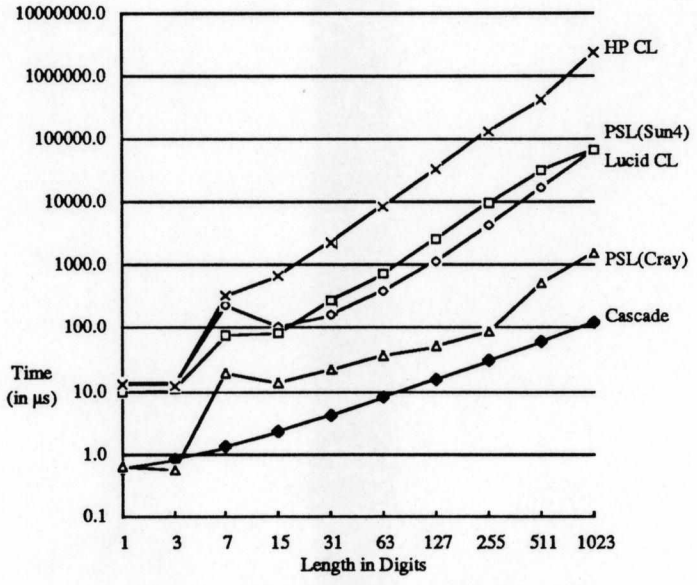

Figure 7: Multiplication Performance

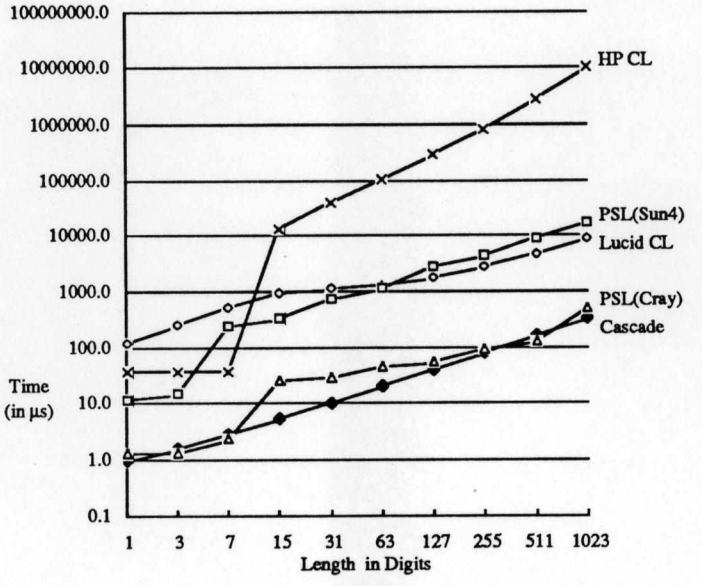

Figure 8: Division Performance

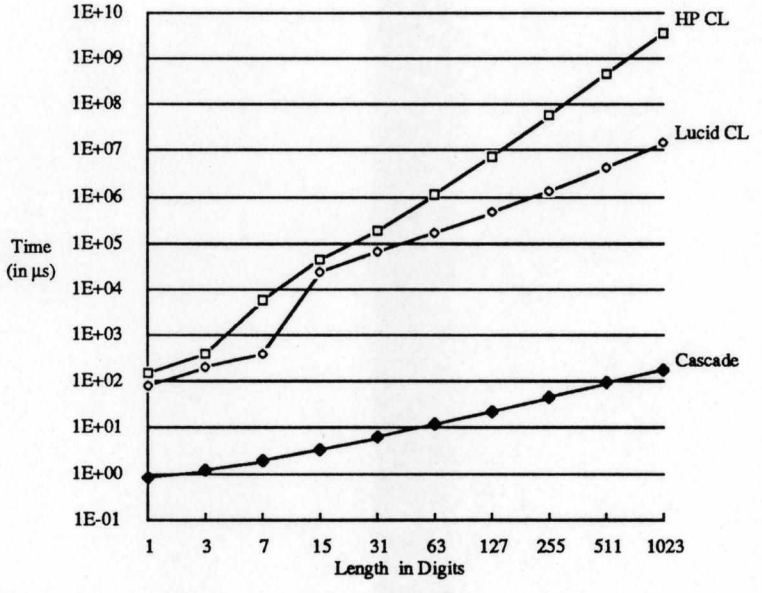

Figure 9: Square-Root Performance

References

1. Anderson, J. W., Galway, W. H., Kessler, R. R., Melenk, H., and Neun, W. Implementing and optimizing lisp for the cray. *IEEE Software* (July 1987) 74–83.

2. Avizienis, A. Signed-digit number representations for fast parallel arithmetic. *IRE Transactions on Electronic Computers*, EC-10, 9 (Sep. 1961) 389–400.

3. Brassard, G. *Modern Cryptography*. Springer-Verlag (1988).

4. Carter, T. M. *Structured Arithmetic Tiling of Integrated Circuits.* PhD thesis, Department of Computer Science (Dec. 1983).

5. Carter, T. M. Structured arithmetic tiling of integrated circuits. In *Proceedings of the* 8^{th} *Symposium on Computer Arithmetic*, Como, ITALY (May 1987) 41–48.

6. Carter, T. M. Cascade: hardware for high/variable precision arithmetic. In *Proceedings of the* 9^{th} *Symposium on Computer Arithmetic*, Santa Monica, CA (Sep. 1989).

7. Carter, T. M. and Hollaar, L. A. The implementation of a radix-16 digit-slice using a cellular vlsi technique. In *Proceedings IEEE Int'l Conference on Computer Design*, Port Chester, NY (Nov. 1983) 688–691.

8. Carter, T. M. and Robertson, J. E. *Radix-16 Signed-Digit Division.* Technical Report UUCS-88-004, University of Utah, Department of Computer Science (Apr. 1988).

9. Chow, C. Y. F. *A Variable Precision Processor Module.* PhD thesis, Department of Computer Science (1980).

10. Chow, C. Y. F. A variable precision processor module. In *Proceedings IEEE Int'l Conference on Computer Design*, Port Chester, NY (Nov. 1983) 692–695.

11. Group, CAGD Research. *Alpha_1 User's Manual.* University of Utah, Department of Computer Science (1983).

12. Hearn, A. C. The reduce program for computer algebra. In *Proceedings* 3^{rd} *Colloquium on Advanced Computing Methods in Theoretical Physics*, CNRS, Marseilles, France (June 1983) A–V–1–19.

13. Knuth, D. W. *The Art of Computer Programming.* Volume 2: Seminumerical Algorithms, Addison-Wesley (1981) chapter 4, 250–265.

14. Lugish, B. G. De. *A Class of Algorithms for Automatic Evaluation of Certain Elementary Functions in a Binary Computer.* PhD thesis, University of Illinois at Urbana-Champaign (June 1970).

15. Melenk, H., Möller, H. M., and Neun, W. *On Gröbner Bases Computation on a Supercomputer Using REDUCE.* Preprint SC 88-2, FB Mathematik und Informatik der Fernuniversität Hagen (Jan. 1988).

16. Nagel, L. W. *SPICE2: A Computer Program to Simulate Semiconductor Circuits.* Memo ERL-M520, University of California, Berkeley (May 1975).

17. Robertson, J. E. A systematic approach to the design of structures for arithmetic. In *Proceedings of the* 5^{th} *Symposium on Computer Arithmetic* (May 1981) 35–41.

18. Robertson, J. E. Design of the combinational logic for a radix-16 digit-slice for a variable precision processor module. In *Proceedings IEEE Int'l Conference on Computer Design*, Port Chester, NY (Nov. 1983) 696–699.

19. Robertson, J. E. *A Theory of Decomposition of Structures for Binary Addition and Subtraction.* Report UIUCDCS-R-81-1004, University of Illinois at Urbana-Champaign (Jan. 1983).

20. Thomas, S. W. *Modeling Volumes Bounded by B-Spline Surfaces.* PhD thesis, University of Utah (1984).

21. White, J. L. Reconfigurable, retargetable bignums: a case study in efficient, portable lisp system building. In *Proceedings ACM Conference on Lisp and Functional Programming* (1986) 174–191.

22. Zuras, D. and McCallister, W. H. Balanced delay trees and combinational division in vlsi. *IEEE Journal of Solid-State Circuits*, SC-21, 5 (Oct. 1986) 814–819.