4

A Multiprocessor Architecture Using Modular Arithmetic for Very High Precision Computation

Henry M. Wu
Artificial Intelligence Laboratory
and
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology

A.I. Memo No. 1119 April, 1989

Abstract

~~We~~ outlines a multiprocessor architecture that uses *modular arithmetic* to implement numerical computation with 900 bits of intermediate precision. A proposed prototype, to be implemented with off-the-shelf parts, will perform high-precision arithmetic as fast as some workstations and mini-computers can perform IEEE double-precision arithmetic. The author ~~We~~ discusses how the structure of modular arithmetic conveniently maps into a simple, pipelined multiprocessor architecture. ~~We~~ presents techniques ~~we~~ developed to overcome a few classical drawbacks of modular arithmetic. Our architecture is suitable to and essential for the study of *chaotic dynamical systems*.

Keywords: Modular Arithmetic, Residue Number System, Computer Architecture, Computer Arithmetic, Multiprocessors, Pipelining, Chaos.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the Laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-86-K-0180.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AI Memo 1119	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) A Multiprocessor Architecture Using Modular Arithmetic for Very High Precision Computation		5. TYPE OF REPORT & PERIOD COVERED memorandum
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Henry M. Wu		8. CONTRACT OR GRANT NUMBER(s) N00014-86-K-0180
9. PERFORMING ORGANIZATION NAME AND ADDRESS Artificial Intelligence Laboratory 545 Technology Square Cambridge, MA 02139		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Advanced Research Projects Agency 1400 Wilson Blvd. Arlington, VA 22209		12. REPORT DATE April 1989
		13. NUMBER OF PAGES 12
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Office of Naval Research Information Systems Arlington, VA 22217		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) Distribution is unlimited		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES None		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) modular arithmetic; computer architecture; multiprocessor; residue number system; computer arithmetic; pipelining; chaos		

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

We outline a multiprocessor architecture that uses modular arithmetic to implement numerical computation with 900 bits of intermediate precision. A proposed prototype, to be implemented with off-the-shelf parts, will perform high precision arithmetic as fast as some workstations and mini-computers can perform IEEE double-precision arithmetic. We discuss how the structure of modular arithmetic conveniently maps into a simple, pipelined multiprocessor architecture. We present techniques we developed to overcome a few classical drawbacks of modular arithmetic. Our architecture (OVER)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Block 20 cont.

is suitable to and essential for the study of chaotic dynamical systems.

1. Introduction

We have designed and functionally simulated a multiprocessor architecture which uses modular arithmetic to implement fast, extremely precise arithmetic operations. The structure of modular arithmetic exhibits immense parallelism, allowing an implementation of high-precision fixed-point arithmetic that is comparable in speed to the IEEE double-precision arithmetic (64 bits) provided by some of the floating-point units in workstations and mini-computers. By implementing multiple fixed-point number systems on top of a modular number system with 18 moduli ranging from $2^{29} - 1$ to 2^{64}, we obtain over 900 bits of intermediate precision.

High-precision computation is essential in numerical studies of chaotic systems. The behaviour of these systems are extremely sensitive to their initial conditions, rendering numerical simulation with low-precision arithmetic extremely difficult and frequently useless. Chaos theory, combined with precise numerical simulation, has been applied in orbital mechanics [Sussman 88], and work is in progress on using computation and chaotic phenomena in physical systems. These applications require tremendous amounts of high-precision computation, which our architecture will effectively provide. We also believe that this architecture can be adapted to perform efficient symbolic algebra and cryptography.

The idea of using the modular number system to speed up computer arithmetic is not new. Extensive work was done in the 60's to investigate its viability [Szabo 67]. Even more effort was spent on digital signal processing applications [Soderstrand 86]. However, owing to difficulties in performing division, sign detection, and magnitude comparison in this representation, modular arithmetic is seldom used in general-purpose computer arithmetic.

Our approach of using medium-sized (< 64 bit) binary numbers to support a 900 bit modular arithmetic system, which in turn implements high-precision fixed-point arithmetic, allowed us to overcome some of the problems associated with modular arithmetic. The approximate magnitudes of the 900 bit numbers are tracked by a conventional floating-point unit. This information can be used to reduce in the number of normalizations. The floating-point estimates produce initial guesses for a Newton-Raphson division routine, and are used in rough magnitude comparisons.

A-1

We start with a brief overview of modular arithmetic and how it is used to implement efficient fixed-point arithmetic. We discuss how we avoid some intrinsic pitfalls of modular arithmetic, and how modular arithmetic can be implemented on pipelined, parallel hardware.

2. Modular Arithmetic

The modular arithmetic system is also known as the residue number system (RNS). A number is represented by the remainders (*digits*) formed when it is divided by a set of pairwise relatively prime numbers (*moduli*). For example, the integer 17_{10} is represented in an RNS with moduli $\{2,3,5,7\}$ by the digits $\{1,2,2,3\}$. We refer to an RNS number as a *modnum*.

2.1 Modular Arithmetic Operations

An RNS consisting of relatively prime moduli with product M can be used to represent signed integers $[(-M/2),(M/2)-1]$.[1] Three basic arithmetic operations - add, subtract, and multiply - on modnums can be implemented as digit-wise modular operations. So, in an RNS with moduli $\{m_{(n-1)},\dots,m_1,m_0\}$,

$$|\{x_{(n-1)},\dots,x_1,x_0\}\ op\ \{y_{(n-1)},\dots,y_1,y_0\}|_M$$

where $|x|_m$ denotes $x\ mod\ m$, and op is $+$, $-$, and $\times$ yields

$$\{|x_{(n-1)}\ op\ y_{(n-1)}|_{m_{(n-1)}},\dots,|x_1\ op\ y_1|_{m_1},|x_0\ op\ y_0|_{m_0}\}.$$

Digit-wise addition or subtraction modulo an RNS modulus (written as $\oplus$ and $\ominus$) is easy because the "carry" is guaranteed to be less than the modulus. The remainder of result can be computed in at most one more subtraction (called *carry-adjust*). So, if x and y are *mod m*,

$$x \oplus_m y = \begin{cases} x+y & \text{if } x+y<m; \\ x+y-m & \text{if } x+y \geq m. \end{cases}$$

Digit-wise modular multiplication ($\otimes$) requires a full remaindering operation. Efficient hardware implementation is possible if the moduli are restricted to numbers of the forms 2^p+1, 2^p, and 2^p-1. Using the *casting out nines* algorithm [Knuth 69],

[1] As in two's-complement a negative number X is represented as $M+X$.

$$|X|_{2^p} = X \ mod \ 2^p,$$

$$|X|_{2^p-1} = (X \ mod \ 2^p) \ \oplus_{2^p-1} \ (X \ div \ 2^p), \ and$$

$$|X|_{2^p+1} = (X \ mod \ 2^p) \ \ominus_{2^p+1} \ (X \ div \ 2^p),$$

involving only bit extraction and simple digit-wise modular operations. Choosing moduli of these forms also facilitates carry detection in modular addition and subtraction.

Therefore, simple modnum operations can be reduced to digit-wise operations modulo the respective moduli with no information carried between the digits. This lack of a carry chain eliminates the inherent sequentiality when operating on successive digits in a weighted number system, allowing full parallelism in digit-wise operations. Since digit-wise operations may occur concurrently, it is possible to implement, on parallel hardware, modulo M arithmetic in the time required to perform modulo m arithmetic. This makes modular arithmetic an attractive platform for implementing high-precision, long word-length arithmetic on a multiprocessor.

2.2 Modnum Division

Although generalized division in the residue number system is complicated and ill-defined, the particular case of division by a product of any of the moduli is possible. When a division has remainder zero, it is equivalent to multiplying the dividend by the divisor's multiplicative inverse. Since each modulus is relatively prime to all the other moduli, its multiplicative inverses modulo each of the other moduli is defined.[2] Hence division by a product of moduli may be decomposed into a series of multiplication of the dividend by the inverses of the divisor's factors, *provided* we guarantee, at each step, that the remainder is zero. For numbers in modular form, the modnum digit at each modulus predicts the remainder when the modnum is divided by that modulus. Therefore, to truncate the original dividend or each intermediate quotient to a multiple of the next divisor modulus, we simply subtract[3] the entire number by the value of the modnum digit at the divisor modulus.

Since each of the divisor's factors has no multiplicative inverse modulo itself, the quotient we form is in an RNS of only the non-divisor moduli. Unique repre-

[2] They can be computed by the Euclid GCD algorithm.

[3] We can also round up by adding the additive inverse.

sentation is still guaranteed because the quotient has reduced range relative to the dividend. The *base extension* process [Szabo 67] recovers the missing digits. The algorithmic structure of this procedure resembles that of division.

Each step in modnum division starts with a modnum subtraction and a modnum multiplication, and then one of the digits is broadcast to the other digit positions for the next subtraction. The number of steps is equal to the number of moduli. Simultaneously, the algorithm yields digits in a weighted, *mixed-radix* representation, to allow sign detection and magnitude comparison.

The preceding section was meant merely as a reference for the following chapters. Complete and vigorous treatments of modular algorithms can be found in [Szabo 67] and [Knuth 69].

3. Implementing Fixed-point Arithmetic

Modnum additions, subtractions, multiplications, and scaling by products of moduli are used to implement fixed-point arithmetic. A fixed-point number f is represented by the modnum n, where $n = fR$. R is a fixed-point radix chosen to be a product of moduli. Each "tick" in this representation is $\frac{1}{R}$. Multiple radix points can be supported simultaneously to ensure precise representation over a wide range.

Fixed-point addition (and hence subtraction) is simply

$$f_1 + f_2 \rightarrow (f_1 + f_2)(R) = n_1 + n_2$$

and therefore equivalent to a modnum addition. Fixed-point multiplication is

$$f_1 \times f_2 \rightarrow (f_1 \times f_2)(R) = f_1 \times f_2(\frac{R^2}{R}) = \frac{(n_1 \times n_2)}{R}$$

Division by R "normalizes" the fixed-point number to its previous representation, at the expense of some precision. Since R is a product of moduli the division can be done as previously described. In order to hold the quantity $(m_1 \times m_2)$ before scaling, $M \geq f_1 f_2 R^2$, and so $f < \frac{\sqrt{M}}{R}$ to prevent overflow.

3.1 Minimizing Normalization Operations

Normalizing after each multiplication is expensive. The number of normalizations can be reduced by delaying them until necessary. For example, when summing a series of the form $f_T = f_{11}f_{12} + f_{21}f_{22} + \ldots + f_{n1}f_{n2}$ we can choose to sum the intermediate products with the radix temporarily raised to R^2. Only one normalization is needed for the entire multiply-accumulate operation.

A major advantage is that computation proceeds before precision is lost through normalization. We developed a scheme in which we track the approximate value of our fixed-point numbers with floating-point numbers (*flonums*). Whenever we perform a fixed-point operation, a corresponding flonum operation takes place, albeit with less precision. An accurate copy of the floating-point approximation can be constructed using the mixed-radix digits generated when normalizing products. The flonum's magnitude can be used to signal the need to normalize and avoid overflow.

Another trick is to pre-scale common factors. For example, if the expression $Y_i = X_i \times K$ appears within a loop, K may be scaled once outside the loop, eliminating the need to normalize within each iteration. If the approximate dynamic range and inherent accuracy of relevant numbers are known, either *a priori* or through the flonum approximation, pre-scaling can be handled with little or no loss in precision.

These optimizations can be statically managed by the programmer. However, we believe that automated optimization by a compiler is possible [Dally 89]. The problems of computing with fixed-point numbers were familiar to programmers before floating-point arithmetic was invented, and most of the solutions developed would apply here.

3.2 Division, Comparisons, and Sign-detection

With fixed-point addition, subtraction, and multiplication, we can implement fixed-point division (reciprocals) using the Newton-Raphson approximation method [AMD 88]. Because this algorithm has quadratic convergence, we only need about 10 iterations to generate a 900 bit reciprocal even if we start with an initial guess with one or two correct bits. Newton-Raphson approximation for other functions are equally applicable.

We can quickly compute an approximate answer to the function we are computing by performing floating-point arithmetic on tracking flonums. The answer is used to index into a precomputed table mapping flonums to modnums. A common table may be shared for all approximation methods.

The flonum approximation also allows gross magnitude comparisons to be done. Close calls are resolved by conversion of the modnum to the mixed-radix notation.

We can efficiently detect positive numbers close to zero. Since we ensure that our residue representation is non-redundant and that the moduli are pairwise relatively prime, the only case in which all the digits are equal is when modnum $n < min(m_{n-1}, ..., m_1, m_0)$.[4] Since this condition can be checked digit-wise, it can be done in parallel.[5]

4. The Modnum Parallel Architecture

A block diagram of the Modnum multiprocessor is shown in Figure 1.

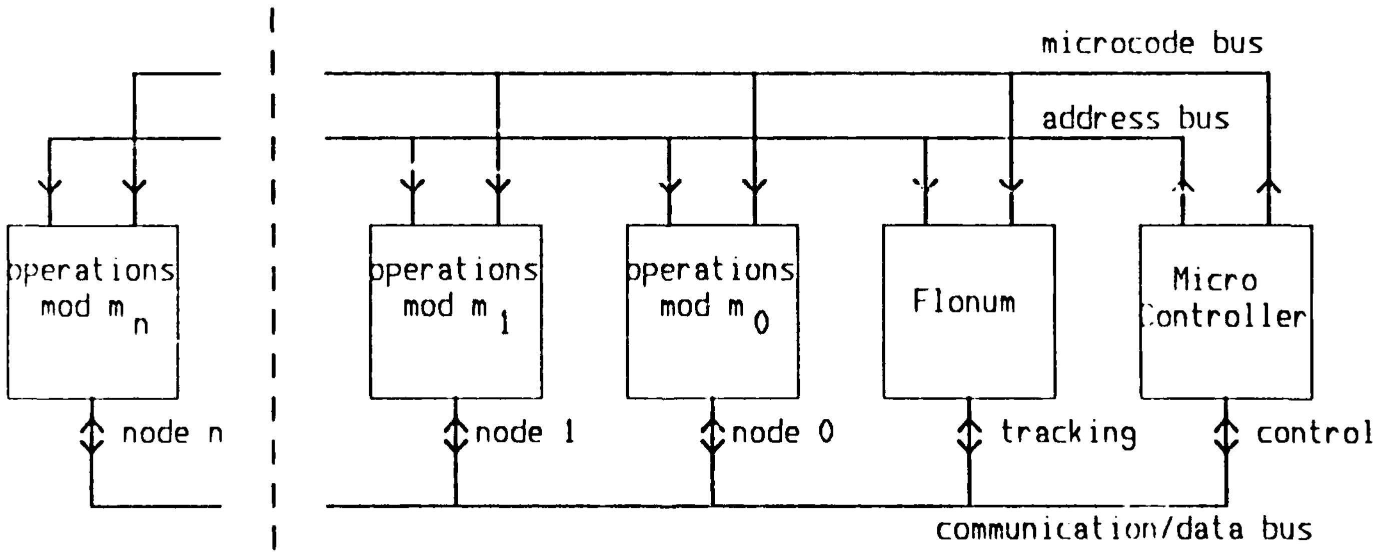

Figure 1: The Modnum Multiprocessor

The architecture specifies a number of *digit nodes*, each computing digit-wise

[4] Proof: It is "obvious" that the digits are in fact equal when m is smaller than the smallest moduli m_{min}, and since each number is uniquely represented and the representation is non-redundant, it follows that the digits are equal if and only if $n < m_{min}$.

[5] Assuming the accumulation of each digit's boolean result can be done at once, e.g. wire-ANDed in hardware.

operations of one modulus. They communicate through a synchronous, shared bus. Each node has modular arithmetic hardware, memory, and a sequencer, which executes *nanocode* that is potentially different on each node. The digit-node sequencer decodes *microcode* instructions fed from a central *controller*. In addition the controller performs address computations and feeds the computed addresses to the digit nodes. Also sitting on the shared bus is the *tracking node* a floating-point unit that snoops on the controller-supplied micro-instructions, memory addresses, and bus communication. It has its own nanocode to exercise proper control of its floating-point hardware.

For the prototype, we will use 18 digit nodes, each with 32-bit datapaths cycled twice to implement 64-bit arithmetic. Our chosen set of moduli ranges from $2^{29}-1$ to 2^{64}. These choices can be changed conveniently by redoing field-programmable logic devices.

4.1 Digit Node

Each node computes with one digit of the modular representation. Each digit of a modnum is stored on the corresponding node.

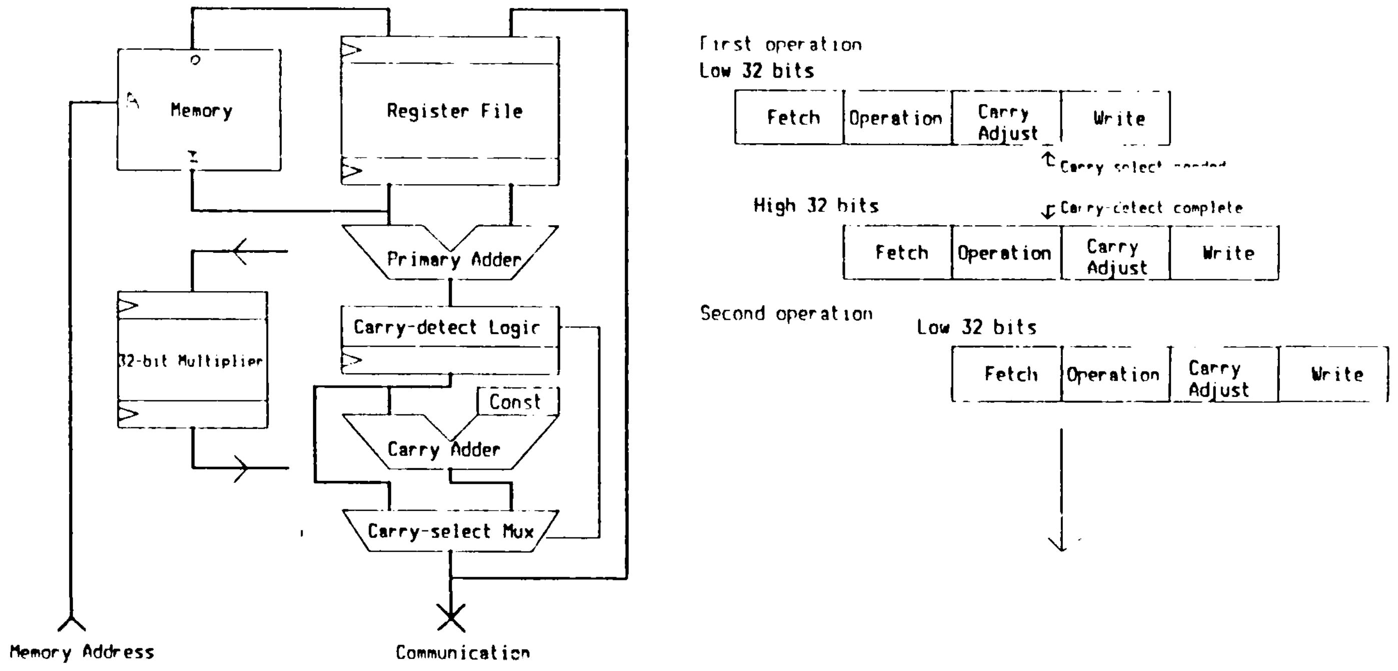

Figure 2: The Datapath Pipeline

The datapath pipeline, shown in Figure 2, is optimized to perform modular addition and subtraction. A new nano-instruction commences every pipestage.

The instruction and the least-significant 32-bit words (lsw) of the modnum operands are fetched in the first stage (`Fetch`). Binary addition on the lsw's happens during `Operation`. The result lsw is compared with the modulus lsw in `Carry-detect Logic`. Standard registered PAL's perform this function while simultaneously latching the result to maintain the pipeline. The next stage unconditionally generates a carry-adjusted lsw. Meanwhile the most-significant words of the operands (msw) have been fetched and have advanced to the `Operation` stage to yield the raw result msw. The full carry-detection may then complete, just in time to select whether the raw-result lsw or the carry-adjusted lsw gets written back to the register file. In the next nanocycle the correct msw is similarly selected. The pipeline allows a new modnum addition or subtraction to commence every 2 nanocycles.

A 32-bit CMOS multiplier performs 64-bit multiplication in 7 nanocycles. The multiplier is connected to the arithmetic datapath (not shown in diagram) so as to allow a new multiplication to commence every 4 nanocycles.

The entire datapath can be implemented with currently available, stock TTL and CMOS parts. Standard binary arithmetic units, coupled with a few programmable logic devices, efficiently perform modular arithmetic. Each pipestage can comfortably be completed in 80ns with the parts we have chosen.

4.2 Instruction Sequencing

The central controller supplies microcode to cause nanocode routines to be executed on each node. This approach combines the benefits of SIMD and MIMD architectures. Different nodes may have different nanocode. For example, the tracking node runs nanocode that is quite different from that on digit nodes, and we can program some digit nodes to perform two 32-bit operations with short moduli in the time it takes other nodes to complete one 64-bit operation. Fixed-point operations are sequenced as microcode instructions, while the modnum operations implementing them are programmed in nanocode. Synchronization is ensured either by carefully generated nanocode sequences of known length or by wired-ANDed status lines.

4.3 Communication

To perform scaling, one selected digit of each step's result is broadcast to all other nodes. This is done sequentially on the shared bus. Since ownership of the bus is pre-determined by the scaling algorithm, it can be software controlled and requires no explicit arbitration. The controller can also access memory on a selected node and grant it ownership of the bus. The tracking node may also own the bus, e.g. when it has to broadcast a Newton-Raphson initial guess.

5. Performance Estimate

As discussed above, modnum (and hence fixed-point) additions and subtractions can start every two 80ns nanocycles, so the *peak* execution rate for these instructions is $6\frac{1}{4}$ million operations per second (MOPS). Primitive multiplies (without normalization) start every four nanocycles, to yield $3\frac{1}{8}$ MOPS peak. The *average* speed of the computer will depend on the number of normalizations required by the application program and whether operations may be effectively pipelined.

6. Conclusions

We designed a multiprocessor architecture to perform high-precision arithmetic very efficiently. We used the modular arithmetic representation, and developed the novel method of floating-point tracking to avoid some of its inherent pitfalls. Our architecture is suitable for implementation with currently available hardware, and the resulting system will provide high-precision arithmetic with performance comparable to common double-precision floating-point systems. Our system has immediate applications in the study of chaotic systems. We expect further applications to develop in symbolic algebra and cryptography.

7. Future and Relevant Work

We plan to have a prototype of this architecture implemented and tested by September, 1989. At that point we wish to conduct performance measurements on this architecture. We will continue to work on optimization techniques, and look into formal studies of roundoff errors in our fixed-point arithmetic.

Although this architecture is expected to deliver satisfactory performance for our initial applications, its implementation with off-the-shelf parts requires much hardware. By implementing each digit board (*sans* memory) in a single VLSI chip, it may be possible to implement an entire modnum machine on a single printed-circuit board. This board will be attractive as nodes in a multiprocessor, or as an accelerator board for a conventional computer.

The design of this architecture is just a means to an end. The primary goal of the project is to perform studies on chaotic systems. Not only will this computer be used to perform detailed numerical studies of chaotic systems, it will also be valuable for calibrating new simulation techniques using conventional arithmetic, and may have some impact on the field of numerical analysis.

Number representations using continued fractions (and continued logarithms) also promise to provide efficient arbitrary-precision arithmetic [Gosper PC]. Most recently [Vuillemin 88] reports some breakthroughs in that area. The Schönage-Strassen FFT multiplication algorithm is efficient for computing long word-length products. Its implementation is probably most suited for massively parallel machines (such as the Connection Machine) and word lengths a few orders of magnitude greater than those we plan to implement.

Acknowlegements

Gratitude is due to Mr. Andrew Berlin, who did initial studies on this subject, and has been a continual source of good ideas. Prof. Gerald J. Sussman suggested and supervises this work.

Bibliography

[AMD 88]
AM29C327 *User's Manual*, Advanced Micro Devices, Inc. Sunnyvale, Calif. 1988.

[Dally 89]
William J. Dally, "Micro-optimizations of Floating Point Operations," *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, 1989.

[Gosper PC]
W. Gosper, private communications, an unpublished memo on computing with continued fractions and continued logarithms, MIT Artificial Intelligence Laboratory.

[Knuth 69]
Donald E. Knuth, *The Art of Computer Programming, Vol. 2*, Addison-Wesley Publications, Reading, Mass. 1969.

[Soderstrand 86]
M. Soderstrand, W. Jenkins, G. Jullien, F. Taylor, editors, *Residue Number System Arithmetic: Modern Applications in Digital Signal Processing,* IEEE Press, 1986.

[Sussman 88]
Gerald J. Sussman, Jack Wisdom, "Numerical Evidence that the Motion of Pluto is Chaotic," *Science*, July, 1988.

[Szabo 67]
Nicholas S. Szabo, Richard I. Tanaka, *Residue Arithmetic and its Applications to Computer Technology,* McGraw-Hill, New York, 1967.

[Vuillemin 88]
Jean Vuillemin, "Exact Real Computer Arithmetic with Continued Fractions," *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, 1988