CADAC: A Controlled-Precision Decimal Arithmetic Unit

MARTY S. COHEN, T. E. HULL, AND V. CARL HAMACHER, SENIOR MEMBER, IEEE

***Abstract*—This paper describes the design of an arithmetic unit called CADAC (clean arithmetic with decimal base and controlled precision). Programming language specifications for carrying out "ideal" floating-point arithmetic are described first. These specifications include detailed requirements for dynamic precision control and exception handling, along with both complex and interval arithmetic at the level of a programming language such as Fortran or PL/I.**

CADAC is an arithmetic unit which performs the four floating-point operations add/subtract/multiply/divide on decimal numbers in such a way as to support all the language requirements efficiently. A three-level pipeline is used to overlap two-digit-at-a-time serial processing of the partial products/remainders. Although the logic design is relatively complex, the performance is efficient, and the advantages gained by implementing programmer-controlled precision directly in the hardware are significant.

***Index Terms*—Arithmetic unit, language requirements, numerical computation, pipelined multiplier, variable precision.**

I. INTRODUCTION

THERE has been a great deal of recent interest in the detailed specification of floating-point arithmetic, as evidenced, for example, by the work of the IEEE subcommittee on floating-point standards [1]. The goal of such work is to make scientific computing convenient from the point of view of the ultimate user. It should be powerful and flexible, and at the same time simple and easy to use. It is therefore appropriate to begin, in this section, by specifying a possible set of user needs. This is followed by a description of how these needs can be met, in an efficient way, through the development of suitable software and hardware. This paper is an updated version of recent work by the authors [2]-[5].

Existing programming languages tend to restrict our ability to describe the numerical processes we would like to implement. We believe that the situation would improve if a number of specific capabilities could be provided in the language in a straightforward and convenient way. These capabilities will be outlined in some detail in this section. An arithmetic unit, called CADAC, has been designed to support these capabilities in an efficient way. It has been implemented as an attached processor, and is currently undergoing extensive testing and evaluation. CADAC will be described in later sections.

We assume that all floating-point numbers have a normalized sign-and-magnitude decimal representation, and that all arithmetic operations on them are properly rounded (ties being broken by rounding to the nearest even). The main reason for this assumption is simply that it is easy and natural from the user's point of view. It also avoids I/O conversion errors.

Precision is to be under the control of the programmer and will be denoted by p, the number of decimal digits in the significand. The exponent range will be under the control of the programmer as well. Somewhat arbitrarily we assume that the exponent range is $[-10p, 10p]$ when the precision is p. Floating-point numbers can therefore be viewed by the user as being of the form

$$\pm 0 \cdot dd \cdots d \times 10^{\pm ee \cdots e}$$

p decimal digits	exponent range is $[-10p, 10p]$

We also assume that real variables can have two "values" in addition to the floating-point numbers, namely "null" and "indeterminate." Null is the uninitialized value, and wrap-around results (see Sterbenz [6]) are left after overflow/underflow, except that indeterminate is the result of zero-divide.

Next, we assume that both complex and interval arithmetic are to be available. They can be provided directly as a part of the language, or, alternatively, through the specifications of user-supplied modules. In either event, their efficient implementation requires an arithmetic unit that can round up or round down (algebraically), as well as perform proper rounding and control of both precision and exponent range.

Precision control is now described in more detail. We identify two requirements which are not currently met by scientific programming languages.

1) The First Requirement: A means to enable the user to carry out intermediate stages of a calculation in precisions which differ from that currently being used in the program.

To consider one simple illustration, suppose that x and y are two arrays, and that we wish to calculate their dot product in higher precision in order to reduce the cumulative effect of roundoff error.

Manuscript received April 23, 1982; revised October 11, 1982. This work was supported in part by NSERC (Canada) Grants A-0055 and A-5192.

M. S. Cohen was with the Department of Electrical Engineering, University of Toronto, Toronto, Ont., Canada M5S 1A4. He is now with MSC Consulting, Toronto, Ont., Canada.

T. E. Hull is with the Departments of Computer Science and Mathematics, University of Toronto, Toronto, Ont., Canada M5S 1A4.

V. C. Hamacher is with the Departments of Electrical Engineering and Computer Science, University of Toronto, Toronto, Ont., Canada M5S 1A4.

0018-9340/83/0400-0370$01.00

Here is an example of what might appear in the program:

```
real(8) x(1:100), y(1:100), dotprod
   ⋮
begin precision(16)
    real(16) temp
    integer i
    temp = 0
    for i = 1, 2, ···, 100
        temp = temp + x(i)*y(i)
    end for
    dotprod = temp
end begin
   ⋮
```

The arrays x and y, and the result "dotprod," are in precision 8, but the intermediate calculations and the intermediate result "temp" are in precision 16. The value of "temp" is rounded to precision 8 on being assigned to "dotprod." The begin-end block is known as a *precision block*.

There are many other situations in which one might want to perform intermediate stages of a calculation in higher precision. These include calculating "residual" vectors in linear algebra, calculating Euclidean norms, and both determining and solving the normal equations associated with least-squares problems. See [3] and [4] for other examples. In some situations, the precision is increased in order to take advantage of the corresponding increase in exponent range.

2) *The Second Requirement:* A means to enable the user to carry out one part of a calculation two or more times in different precisions; for example, the calculation could be carried out in higher and higher precision until some error estimate becomes small enough.

To illustrate this second capability, we consider a program for solving a nonlinear equation within some prescribed accuracy, say "tol." We suppose that we have a procedure (such as Newton's method) for finding an approximation to the solution. And we also suppose that we have a way of determining a guaranteed bound on the error in an approximate solution, once that approximation has been found. The following informal program outline shows the essential features of what we have in mind:

```
initialize precision
flag = true
while (flag = true)
    begin precision p
        find approximate solution
        find error bound
        if bound ≤ tol
            root = approximation
            flag = false
        end if
    end begin
    increase precision
end while.
```

Here, the precision block is executed repeatedly, in higher and higher precision, until the error requirement is satisfied. Specific examples are given in [3] and [4].

The main implication of these two requirements, as far as the hardware is concerned, is that the precision of the data (which is concerned with storage) is chosen independently of the precision of the operations carried out on the data (which is concerned only with the arithmetic unit).

One of the principle requirements of any design is that the user know exactly what to expect under all circumstances. For example, the precision (of, say, eight decimal digits) should be exact, not "at least" (eight decimal digits).

The final topic to be discussed in connection with language facilities is that of exception handling.

3) *The Third Requirement:* A means to allow the user to have control over what is done in case of exceptions such as underflow and overflow.

The following example shows how a user might be allowed to specify what action is to be taken in case such exceptions arise:

```
begin
    on(underflow)
        result = 0
      (overflow)
        action to be taken
       ⋮
    end on
       ⋮} scope
end.
```

In this example, underflow anywhere in the scope of the block would lead to a value of 0 being assigned to "result" (a reserved word) in place of the wraparound value provided by the arithmetic unit before trapping, and then the calculation would resume from the point at which the trap took place.

Such a facility enables the user to choose alternative actions, such as terminating the calculation after output of an appropriate message, or perhaps repeating the calculation in a higher precision.

The difficulties with this third requirement are mostly in the software, and in connection with verification techniques. The essential hardware features are relatively easy to provide, since they only involve the provision of appropriate trapping on exceptions.

II. Overview of CADAC

The machine program view of CADAC is that it is a single-accumulator arithmetic unit with a specifiable precision. A decimal floating-point number with declared precision is stored in the main memory as a packed binary coded decimal (BCD) string with a short header that specifies sign, exponent, and length (precision of the significand).

Let us consider a typical sequence of operations involving CADAC. Suppose the Fortran expression

$$(B + C)/A$$

is to be evaluated and assigned to F. An assembly language code sequence corresponding to this task is

```
LOAD  B
ADD   C
DIV   A
STORE F.
```

Suppose the precisions of the significands of A, B, and C are declared to be eight decimal digits, and that the significand precision of F is six. If the current significand precision of CADAC is ten, then the above code executes as follows:

- the value in B is padded out with two zeros when it is loaded from main memory into the accumulator (ACC) in CADAC;
- the value in C is padded out with two zeros as it is moved from the main memory into CADAC and added to the contents of ACC, generating a properly rounded ten-digit sum in ACC;
- the value from A is padded out with two zeros when loaded into CADAC where it is divided into the contents of ACC, generating a rounded ten-digit quotient;
- the contents of ACC are rounded to six digits and stored at location F in the main memory.

A CADAC unit prototype has been built using high-speed SSI and MSI logic in the main data flow paths that process the significands. The unit is microprogram controlled by an AMD2910 microsequence generator, and exponent arithmetic is performed in three AMD2903 slices. The complete unit consists of two 7 × 16 in boards, each containing about 100 IC's, including all registers and buffers needed to interface the unit to a DMA controller. There are about 1K 88-bit microinstructions in the controller ROM. CADAC is interfaced to a PDP-11/34 minicomputer as an attached processor I/O device, and is currently undergoing extensive testing and evaluation. More details of this prototype will be provided in later sections. A number of data format details in the CADAC design, as reported below, are influenced by the 16-bit wordlength of the PDP-11 architecture. However, it should be clear that the design generalizes easily to other machine environments.

III. Number Representation

A main memory representation (suitable for a 16-bit word-length machine) for controlled-precision decimal floating-point numbers is shown in Fig. 1. The first word, termed the attribute, contains the sign, extend flag, exponent, and significand length descriptor. Succeeding words contain the fractional digit-normalized significand, low-order digits first. We have determined, from a consideration of typical user computations, that it is convenient to set the exponent range as $[-10p, 10p]$ when significand precision is p decimal digits.

The exponent E' is a binary integer ranging from 1 to 1023 and represents the true exponent E in excess-$(10p + 1)$ notation. Therefore, $E = E' - (10p + 1)$. A maximum symmetric range of $-511 \leq E \leq 511$ is available, reserving $E' = 0$ to indicate exact 0 and other special values, as described later.

The 4-bit length descriptor L is a binary integer that ranges from 0 to 15 and specifies the length p of the significand as $2L + 2$. Therefore, the significand length ranges from 2 to 32 decimal digits, in two-digit increments. For $2 \leq p \leq 32$, the exponent ranges are $[-20, 20]$-$[-320, 320]$, which can be accommodated in the 10-bit E'-field specified above.

A significand length p in the range [2, 32] along with an exponent in the range $[-10p, 10p]$ is sufficient for practically

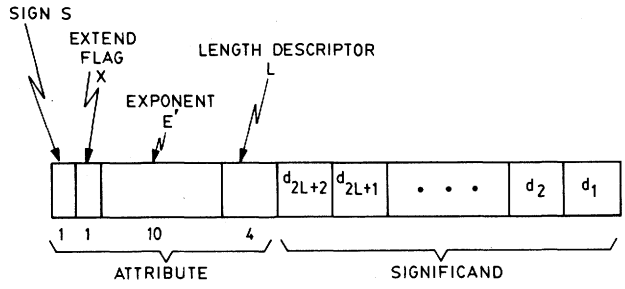


Fig. 1. Standard floating-point number representation.

all computational requirements. However, a longer significand and exponent range may be needed. This can be accommodated in an extended representation, indicated by setting the X bit to 1. (It is 0 in the above standard representation.) In extended representation, a second attribute word can be used to contain the length descriptor and additional exponent bits. The significand digits follow this second attribute word. We will not discuss the details of such a representation here because it will not be needed in this paper.

Let us now consider the representation of the special values: exact 0, null (or unassigned), "zero-divide," and indeterminate. As mentioned earlier, this class of values is indicated by an $E' = 0$ field. Exact 0 is represented by also setting $S = 0$. When $S = 1$ (with $E' = 0$) the operand is either null, zero-divide, or indeterminate. These three conditions are distinguished by the first digit of the significand following the attribute word, labeled d_{2L+2} in Fig. 1. If this digit is hexadecimal E, then the operand is null; if it is hexadecimal F, then the operand is indeterminate; and if it is hexadecimal D, then the operand is zero-divide. When space for a floating-point number is allocated to main memory, its attribute word is loaded with the appropriate L-field and X-field values. At this time, $S = 1$, $E' = 0$, and the first digit of the significand is set to hexadecimal E. A later assignment will change this special value. It should be noted that CADAC provides more than required by the language specifications of the preceding section in terms of special values. CADAC distinguishes between zero-divide ($A/0$, with $A \neq 0$) and indeterminate, while the language specification assigns indeterminate to either case.

IV. The Environment

CADAC is a single-accumulator arithmetic unit. It performs operations based on opcodes and operand addresses sent to its device input registers from the host. CADAC moves numbers between main memory and itself in DMA mode. A status register in CADAC indicates exceptions that occur as a result of instruction execution. This register can be read by the host. A control register is available in association with a maskable interrupt facility.

A. Precision

The floating-point number precision that is a part of the main memory number representation is independent of the current precision of the CADAC unit. Both the significand precision and the exponent range to be used in calculations can be independently set by appropriate commands to the CADAC unit. The particular case of $[-10p, 10p]$ for the exponent range, which should be sufficient for most applications, is thus

easily supported by CADAC. Exponent range checking of main memory numbers can be done either by CADAC or by host software or firmware. For example, suppose CADAC precision p' is greater than the precision p of a result variable V. In association with rounding and storing from ACC to V, a check could be made in CADAC or the host for overflow/underflow with respect to the $[-10p, 10p]$ exponent range.

Suppose that the CADAC unit is set to operate with a given significand precision and exponent range. A number loaded into the unit is automatically rounded or padded to the current precision. If the exponent of the number exceeds the current CADAC range, an overflow/underflow will occur. When a result is returned to the main memory, the number is rounded or padded to the precision specified by the destination, and exponent range checking can also be performed. Since CADAC is a single accumulator machine, the high level language statement $X \leftarrow X$ is translated to a LOAD operation with appropriate rounding or padding, followed by a STORE operation. By setting the precision of the unit to some value less than that of a number, we have a convenient method for performing rounding. Thus we do not need a separate ROUND instruction. In normal operation we expect the precision of the unit to be greater than or equal to the precision of the operands.

B. Rounding

CADAC performs proper rounding by implementing "round to nearest, or, in case of a tie, to nearest even." The unit supports interval arithmetic with algebraic round up and round down. In addition to these rounding modes, CADAC also supports round up magnitude and round down magnitude (truncation). Rounding can occur:

- as part of an operation (rounding of results)
- when loading operands
- when storing results.

To properly support interval arithmetic, the rounding mode must be a property of the operation, not of the environment.

V. Instruction Set and Exception Handling

A. Instructions

This section introduces a basic instruction set consisting of instructions for hardware control and for basic arithmetic operations. Several additional instructions are included to support floating-point integer arithmetic. The instruction set consists of the following.

INIT—Initialize the unit and set the default precision and exponent range. Reset the exception flags. The default precision is 16 digits and the default exponent range is ±160.

RESET—Reset the exception flags.

SET—Set the exponent range and significand precision.

LOADADDR—The unit uses temporary storage areas in the host main memory. These are required when extended precision numbers are processed. The starting addresses of these areas must be provided to the unit before extended precision processing can be requested.

LOAD—Load the accumulator from main memory.

STORE—Store the accumulator in main memory.

ADD—Add from main memory to the accumulator.

SUB—Subtract the main memory operand from the accumulator.

MUL—Multiply the accumulator by the main memory operand.

DIV—Divide the accumulator by the main memory operand.

NEG—Change the sign of the number in the accumulator.

FLOOR—Load the accumulator with the largest floating point integer less than or equal to the current contents of the accumulator.

CEIL—Load the accumulator with the smallest integer greater than or equal to the current contents of the accumulator.

COMPARE—Condition codes are set based on a comparison of the contents of the accumulator with the main memory operand. If the signs of these two values are the same, an actual subtraction ([ACC]-memory operand) is performed. Underflow/overflow exceptions are not set and the accumulator remains unaltered. If the signs differ, the condition codes can be set directly.

B. Floating-Point Integers

Our floating-point number representation can be used to represent large integers. These can be generated by the host or by the FLOOR or CEIL operators. Let NZ be the number of digit positions from the decimal point to the least-significant nonzero digit in a floating-point number. The condition for this number to be an integer is that $NZ \leq E$ where E is the exponent value. If it is known that a number is an integer, the programmer can test for a rounding "error." This can occur during any rounding operation.

C. Exception Handling

A number of exception conditions are detected. They are indicated by setting appropriate bits in the status register. An exception condition can generate an interrupt if its corresponding bit in the control register is set, and interrupts are enabled. The status register can always be read after an operation to determine if an exception occurred. The following exceptions are implemented.

1) OVERFLOW: This flag is set if exponent overflow or a zero-divide error occurs. On zero-divide (or indeterminate) the appropriate special value is loaded into the accumulator; otherwise, the wraparound exponent result [6] will be stored in the accumulator.

2) UNDERFLOW: Underflow is handled similar to overflow, with the wraparound result always being stored in the accumulator.

3) ZERO DIVIDE: A zero-divide error occurs when performing $A/0$ where A is a nonzero floating-point number. Both the overflow and zero-divide flags are set and the zero-divide code is stored in the accumulator.

4) INDETERMINATE: If an attempt is made to perform $0/0$, both zero-divide and indeterminate flags are set, and the indeterminate code is stored in the accumulator.

5) ILLEGAL OPERATION: This flag is set when the unit attempts to operate on a null, indeterminate, or zero-divide

number. The indeterminate code is stored in the accumulator.

6) *INVALID FLOATING-POINT:* This flag is set if an invalid floating-point number is fetched. An invalid number may be an unnormalized number or a number containing a non-BCD digit. The indeterminate code is stored in the accumulator.

7) *NOT EXACT:* This flag is set whenever a rounding error occurs. If floating-point integer numbers are involved in an operation, any rounding operation produces an error. The rounded result is returned.

VI. The Pipeline

CADAC uses a hardware multiplier to achieve high performance. The multiplier includes a three-stage pipeline capable of running at 10 MHz. At this rate, the pipeline can multiply two 32-digit significands in approximately 30 μs. The design chosen for the multiplier unit uses BCD ALU's in a two-digit × two-digit structure. This structure multiplies two multiplicand digits by two multiplier digits and adds overlapped digits to produce two digits of partial product and two overlapped digits that are internally fed back. The first cycle produces two final product digits and two overlapped digits. All the multiplication is done in digit pairs and does not require shifting during product formation. The pipeline requires 17 × 16 + 4 iterations to multiply two 32-digit significands.

A. Operation of the pipeline

To understand the operation of the pipeline, consider the multiplication of two-digit numbers $a_1a_0 \times b_1b_0$ as illustrated in Fig. 2. Four two-digit product terms are required, $a_0 \times b_0$, $a_1 \times b_0$, $a_0 \times b_1$, and $a_1 \times b_1$. In the pipeline, these are obtained in parallel from look-up tables. Now note the significance of the two-digit product pairs. Pairs $a_1 \times b_0$ and $a_0 \times b_1$ have the same significance. The least significant (even) and most significant (odd) digits of these pairs are added together in parallel with two BCD ALU's. The even sum digit is added to the most significant digit of $a_0 \times b_0$. Similarly, the odd sum digit is added to the least significant digit of $a_1 \times b_1$. These two additions are done in parallel. A four-digit product is obtained.

Now consider how the pipeline would multiply $a_3a_2a_1a_0 \times b_1b_0$. In the first cycle the pipeline produces $a_1a_0 \times b_1b_0 = \alpha_{3_0}\alpha_{2_0}\alpha_{1_0}\alpha_{0_0}$. In the second cycle the pipeline produces $a_3a_2 \times b_1b_0 = \alpha_{3_1}\alpha_{2_1}\alpha_{1_1}\alpha_{0_1}$. Now the digit pairs $\alpha_{3_0}\alpha_{2_0}$ and $\alpha_{1_1}\alpha_{0_1}$ have the same significance and must be added together in the pipeline. In general, the digit pairs $\alpha_{1_i}\alpha_{0_i}$ emerge from the pipe. The pairs $\alpha_{3_i}\alpha_{2_i}$ are fed back to be added to $\alpha_{1_{i+1}}\alpha_{0_{i+1}}$.

The pipe must also add the previous partial product, generated with the previous multiplier digit pair, to the pairs $\alpha_{1_i}\alpha_{0_i}$. This partial product is fed into the pipe in pairs p_1p_0. The complete set of additions performed in the pipe is illustrated in Fig. 3. The eight required additions are indicated with brackets.

The three-stage pipeline is designed with as much parallelism as possible. Note that α_{3_i} is the odd digit of $a_1 \times b_1$. Call this digit a'_{1_i}. It must be added to the α_1 column during cycle

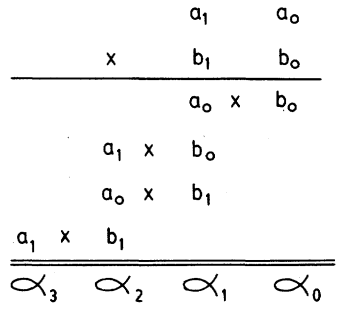


Fig. 2. A 2 × 2 multiply operation.

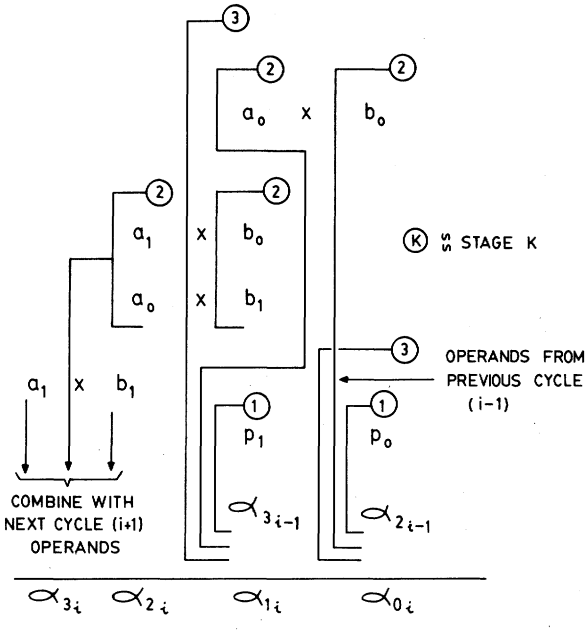


Fig. 3. Pipeline stage operations.

$i + 1$. It can be added to any of the digits in that column at any convenient time. It is most opportune to add it to p_1 in the first stage of the pipeline, as illustrated by the bracket labelled 1 in Fig. 3. Note that α_{2_i} consists of three terms, of which one is the even digit of $a_1 \times b_1$. Call this digit b'_{1_i}. It can be added to the α_0 column of cycle $i + 1$ in parallel with the addition of a'_{1_i}.

In the second stage of the pipe, the terms $a_1 \times b_0$ and $a_0 \times b_1$ can be added together. We can also add together the $a_0 \times b_0$ term and p_1p_0 (to which $a'_{1_{i-1}} \times b'_{1_{i-1}}$ has already been added).

We now consider the final stage of the pipe. The odd digit of $a_0 \times b_0$ (to which other terms have now been added) can be added to the even digit of $a_1 \times b_0 + a_0 \times b_1$. And lastly, the odd digit of $a_1 \times b_0 + a_0 \times b_1$ from the previous cycle, which is the remaining component of $\alpha_{2_{i-1}}$, can now be added to the least significant column.

The complete pipeline structure is illustrated in Fig. 4 and its interconnection with the other components of CADAC is shown in Fig. 5. The inputs to the pipeline are:

- the multiplier in R_{11}, R_{12}
- the multiplicand in R_{13}, R_{14}
- the previous partial product in R_{x1}, R_{x2}, denoted R_x.

The outputs from the pipeline are R_{41} and R_{42}.

Notice that all additions are arranged in pairs. Since there

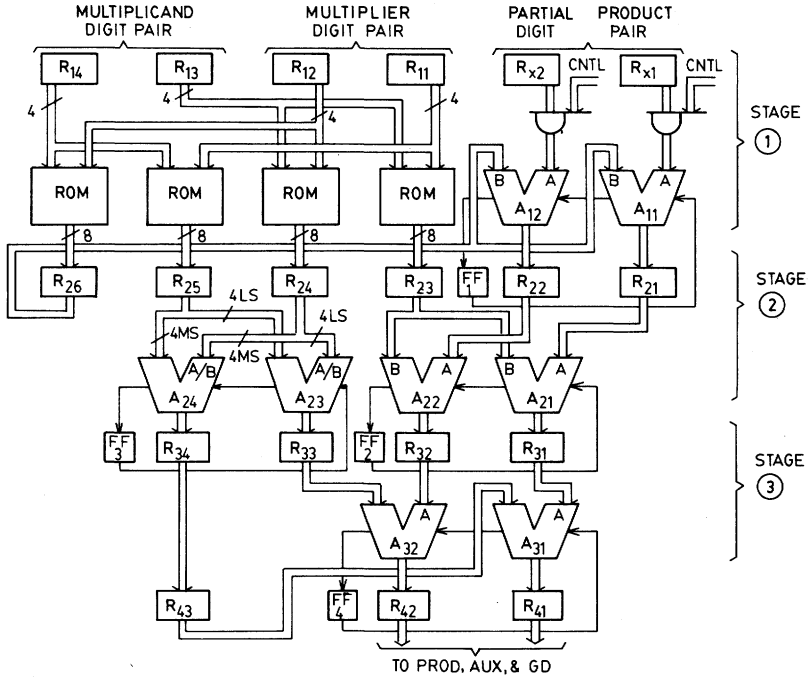


Fig. 4. Pipeline organization.

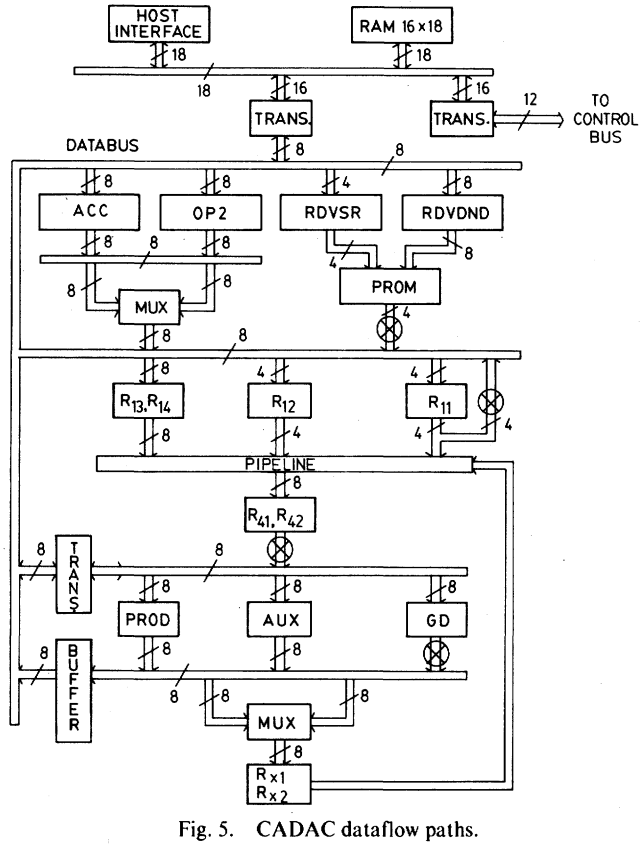


Fig. 5. CADAC dataflow paths.

is only one stage of carry propagation in each pair, there is no need for carry-lookahead. The carry output of the most significant adder is latched and fed back to the carry input of the least significant adder in the next iteration.

This is a convenient place to note that, with the exception of division, all arithmetic operations are performed with the least significant digit pair first. This is the reason that significands are stored least significant digit first. (Since we have decided to represent true zero by a special code, and not by a zero-most-significant digit, this storage arrangement poses no difficulty in zero detection.) This storage representation presents only a minor inconvenience in division, where slower execution times can be tolerated.

VII. Memory Requirements

A. Partial Product Storage (*PROD*)

The partial product digits emerging from the pipe must be buffered and fed back into the pipe at the appropriate time. A dual port memory, PROD, with 32-digit capacity, is used for this purpose. With this device a new partial product can be written into one port while the old partial product is read from the other. Separate counters are used to supply addresses to the two ports. See Fig. 5.

B. Operand Storage

To complement the 32-digit limit, we choose an accumulator (ACC) size of 32 digits. The second operand of an operation (*OP2*) is also stored. If a significand extends beyond 32 digits, it is not stored in CADAC but is buffered in main memory, in areas allocated to the unit. Regions are required for storing 1) the accumulator, 2) the second operand, and 3) the partial product. Data are loaded into the machine and written back in blocks of 32 digits as required by a calculation.

VIII. Standard (Nonextended) Arithmetic

A. Multiplication

We will consider the multiplication process in detail since the pipe has been optimized for this operation. See Fig. 5 while following this discussion. Consider the multiplication of two 32-digit numbers. The first multiplier pair is loaded into registers R_{11} and R_{12}. Now 16 pairs of multiplicand digits are loaded into R_{13} and R_{14} while partial product pairs appear at

registers R_{41} and R_{42}. A zero is now loaded into R_{13}, R_{14} to obtain the 17th product pair. Of these 17 pairs, the first is a final product pair and the others are partial product pairs (stored in the dual-port memory PROD).

If we required a 64-digit final product, the final product pairs would be written into an additional 32-digit memory. However, all we require is a properly rounded 32-digit result. This requires the storage of a guard digit and the provision of a sticky bit. We provide a register (GD) to store a pair of potential guard digits. During the multiplication process, the first product pair produced is written to GD and the others are written to PROD.

While the zero is loaded into R_{13}, R_{14}, the next multiplier pair is loaded into R_{11}, R_{12}. This process is repeated until all digit pairs have been processed. Zeros are now fed into R_{13}, R_{14} until the pipe is flushed. There will now be a 32-digit product and a pair of digits in GD. If the most significant product digit is nonzero, the odd digit in GD is the guard digit and is used for rounding. If the most significant product digit is zero, the product is normalized by shifting left, with the odd digit in GD becoming the least significant product digit. The even digit in GD becomes the guard digit and is used for rounding.

Achievable execution times for processing significands for all four arithmetic operations are shown in Table I. These figures assume a 10 MHz clocking of the pipeline, which can be achieved. However, the AMD 2910 microsequence generator and the ROM used in the current prototype do not support this rate.

B. Division

Division is more complex than multiplication. We provide only an overview here. A straightforward long-division implementation is used. A PROM look-up table is used to estimate the successive quotient digits, by considering the first divisor digit and the first two partial remainder digits. The division process can be described as follows. The first quotient estimate is loaded into R_{11}, with R_{12} set to zero. The dividend is written into PROD. Now divisor pairs are loaded into R_{13}, R_{14} where they are multiplied by the quotient, and subtracted from the dividend pairs (fed into R_x). The partial remainder (PR) is stored in PROD. If the PR is negative, the quotient estimate is decremented and the divisor is added to the PR. This is repeated until the PR is positive. The quotient digit is written to ACC and the entire process repeated. One extra quotient digit is generated for a guard digit and the final remainder determines whether or not the sticky bit is set.

C. Addition/Subtraction

We will not discuss these operations in any detail. They are sequenced through part of the hardware described above for multiplication in a reasonably straightforward fashion. For both add and subtract instructions on signed operands, a true subtract may be required. If the difference is negative (10's complement), another pass through the ALU's is required to complement the result.

IX. Extended Precision Arithmetic

Extended precision arithmetic presents some problems not previously encountered. Most important, we discover that the overlapped reading and writing of PROD during multiplication is not possible since the pairs involved may be in different blocks of 32 digits. Two separate RAM's are required. As before, we use PROD to contain the new partial product pairs. However, we add another RAM, which we call AUX, to hold the previous partial product. Arithmetic in extended precision is no different than before except that blocks of 32 digits are fetched from the host and written back as required.

The actual hardware cost of implementing extended precision arithmetic is therefore confined to extra buffers and ROM for the required additional microcode. This does not constitute a significant increase in cost of the basic CADAC unit.

TABLE I
ACHIEVABLE EXECUTION TIMES* (μs)

DIGITS:	8	14	24	32
ADD/SUB	7	8	11	13
MULT	4	8	18	30
DIV	20	40	85	131

*execution times are for significand processing and they do not include operand fetching.

X. Rounding

CADAC supports the five rounding modes discussed previously. These are:

- round to nearest
- round up/down algebraic
- round up/down magnitude.

Proper rounding requires a guard digit and a sticky bit. As was discussed previously, the unit retains two more digits than the precision of the calculation. This pair is stored in GD. This guarantees that the guard digit is retained even if a left shift is required (after multiplication or subtraction). The sticky bit is set to indicate that there are nonzero digits to the right of the guard digit. This bit is set automatically when overwriting a nonzero digit pair in GD. Rounding up is accomplished by applying a nonzero carry input to an ALU in the pipe.

XI. Zero Counting

It is important to know the number of leading and trailing zeros in a significand during various stages of processing in CADAC. The leading zero count is used:

- to skip over leading zeros in a partial remainder
- to normalize a product or sum/difference
- to determine if a partial remainder is zero.

The trailing zero count is used:

- to choose the multiplier from either ACC or *OP*2, picking the value whose significand has fewer nonzero significant digits
- to determine whether the sticky bit should be set when rounding a number
- to determine if a number is a floating-point integer
- to determine if a result is zero.

Leading zeros which are counted as digits emerge from the pipe and are transferred from R_{42}, R_{41} to PROD or GD. Trailing zeros which are counted as digits are written into ACC, *OP*2, or temporary storage.

XII. Control

So far we have not discussed the control of CADAC. The system uses an AMD2910 microsequence generator, ROM, and three AMD2903 4-bit ALU slices. The 2903's are used for:

- exponent arithmetic
- length calculation and handling
- provision of 12-bit registers to hold:
  - — exponent limit
  - — precision
  - — operand lengths
  - — trailing zero counts
  - — other
- control logic.

XIII. Concluding Remarks

We have presented a possible set of requirements for a decimal arithmetic processor that performs clean arithmetic with specifiable precision. Such a machine should have sufficient rounding modes to support interval arithmetic as well as standard arithmetic. Such a machine should have a very large precision size and exponent range.

We have described the logic design and implementation of such a machine, named CADAC. It handles significands up to 32 digits very efficiently. In extended precision mode, computation is much slower but significand length is greatly increased. The final stages of testing the CADAC hardware are currently in progress at the University of Toronto.

Our primary goal in designing and implementing CADAC is to provide a thoroughly tested prototype that establishes the feasibility of supporting variable-precision decimal floating-point computations at a high execution speed. We do not claim that our programming environment requirements are the only reasonable set. However, they do provide powerful and flexible programming features, not currently supported in hardware, that are very useful in many practical applications.

The actual cost in integrated circuits, including ROM for microinstructions, of the CADAC prototype is not as interesting as the comparative cost of alternative implementations that either execute slower/faster or provide less/more functionality. For example, if only single (~8-digit) precision and double (~16-digit) precision is implemented in hardware, can a higher speed and a lower cost be achieved? Because of the significand alignment and normalization operations, our rough judgment is that both speed and cost cannot be significantly improved together by providing only single and double precision. Another interesting set of questions of the cost and speed type can be posed with respect to binary versus decimal implementations. It is clear that there are many more MSI and LSI components available for binary arithmetic, e.g., array multipliers. Therefore, it is tempting to conjecture that, with present technology, a binary implementation meeting similar, variable-precision functionality would be better. We have not carried out these or other comparative studies in quantitative detail. That type of study, interesting as it would be, requires significant effort; and it is outside of the scope of our present CADAC project.

To recap our project, we have demonstrated that it is feasible to build a decimal floating-point unit with very respectable performance, at reasonable cost, to meet very demanding requirements that focus on dynamically variable precision using small increments.

References

[1] "The proposed IEEE floating-point standard," four articles in *Comput.*, vol. 14, no. 3, Mar. 1981.

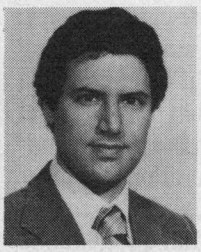
[2] M. Cohen, V. C. Hamacher, and T. E. Hull, "CADAC: An arithmetic unit for clean decimal arithmetic and controlled precision," in *Proc. 5th Symp. Comput. Arithmetic*, IEEE Publ. 81CH1630-3, May 1981, pp. 106–112.

[3] T. E. Hull, "The uses of controlled precision," in *Proceedings IFIP Working Conference on the Relationship Between Numerical Computation and Programming Languages*, J. K. Reid, Ed. North-Holland. Mar. 1982.

[4] ——, "Precision control, exception handling and a choice of numerical algorithm," in *Proc. Dundee Conf. on Numerical Analysis*, A. Watson, Ed. Springer-Verlag, to appear.

[5] ——, "Desirable floating-point arithmetic and elementary functions for numerical computation," in *Proc. 4th Symp. Comput. Arithmetic*, IEEE Publ. No. 78CH1412-6C, Oct. 1978, pp. 63–69.

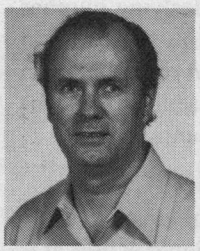
[6] P. H. Sterbenz, *Floating-Point Computation.* Englewood Cliffs, NJ: Prentice-Hall, 1974.



Marty S. Cohen was born in Toronto, Canada, in 1954. He received the B.A.Sc. degree in engineering science and the M.A.Sc. degree in electrical engineering, specializing in computer design, from the University of Toronto in 1977 and 1980, respectively.

From 1977 to 1979 he was an Engineer with Ontario Hydro where he was involved in several software projects. In 1980 he was invited to join the CADAC development project where he was responsible for all aspects of design and implementation. He is currently President of MSC Consulting, a Toronto-based firm of consulting engineers.

Mr. Cohen is a member of the Association of Professional Engineers of Ontario.



T. E. Hull was born in Winnipeg, Canada. He received the M.A. degree in physics and the Ph.D. in applied mathematics, both from the University of Toronto, Canada.

He continued to work at the university, first in the Department of Mathematics and then served as the Computing Centre Director at UBC. Currently he is a Professor of Computer Science and Mathematics at the University of Toronto. His research interests include numerical software and language facilities for scientific computation.

Mr. Hull is active in AMS, MAA, SIAM, the Canadian Math Society, the Canadian Applied Math Society, CIPS and IFIP.

V. Carl Hamacher (SM'79) was born in London, Canada, in 1939. He received the B.A.Sc. degree in engineering physics in 1963 from the University of Waterloo, Waterloo, Canada; the M.Sc. degree in electrical engineering in 1965 from Queen's University, Kingston, Canada; and the Ph.D. degree in electrical engineering in 1968 from Syracuse University, Syracuse, NY.

Since 1968 he has been at the University of Toronto, Toronto, Canada, where he is a Professor in the Departments of Electrical Engineering and Computer Science, and the Associate Chairman of the Division of Engineering Science. His current research interests include local area computer networks, arithmetic processors, and real-time computer systems. During 1978–1979, he was a Visiting Scientist at the IBM Research Laboratory in San Jose, CA. He is a coauthor of the textbook *Computer Organization* (McGraw-Hill, 1978).

Dr. Hamacher is a member of ACM, Sigma Xi, and the Association of Professional Engineers of Ontario.