CADAC: AN ARITHMETIC UNIT FOR CLEAN DECIMAL ARITHMETIC AND CONTROLLED PRECISION *

M. Cohen, V.C. Hamacher and T.E. Hull

Departments of Electrical Engineering and Computer Science
University of Toronto
Toronto, Ontario, Canada, M5S 1A4

ABSTRACT

This paper describes the design of an arithmetic unit called CADAC (Clean Arithmetic with Decimal base And Controlled Precision). A brief indication of programming language specifications for carrying out "ideal" floating-point arithmetic is given. These specifications include detailed requirements for precision control and exception handling at the level of a programming language such as Fortran. CADAC is an arithmetic unit which performs the four floating-point operations add/subtract/multiply/divide on decimal base numbers in accordance with the language requirements. A three-level pipeline is used to overlap 2-digit-at-a-time ("double digit") 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.

1. INTRODUCTION

There has recently been increased interest in the detailed specification of floating-point arithmetic [1]. The goal of such work is, or at least should be, to make scientific computing convenient from the point of view of the ultimate user. It should be powerful and flexible enough, efficient, and at the same time simple and easy to use. It is therefore appropriate to begin by specifying the needs of the user, and then determining the extent to which these needs can be met, in an efficient way, through the development of suitable software and hardware. In this paper, we are especially interested in the hardware implications of such an approach.

One view of the user's needs in terms of programming language specifications has been given in [2]. Much of the motivation behind those proposals was to provide the user with two techniques which are not currently available in scientific programming languages, but which would be very convenient to have:

(1) The first requirement is 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. Here is an example of what might appear in the program:

```
float(8) x(1:100), y(1:100), dotprod
   :
begin precision(10)
    float(10) 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 all in precision 10. The begin-end block is known as a precision block.

In some situations, it is desirable to increase the range of the exponent as well as the precision and the syntax can be extended in a simple way to include this feature.

(2) The other requirement is 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 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 p
end while
```

* This research was partially supported by NSERC (Canada) Grants A-5192 and A-0055.

Here, the precision block is executed repeatedly, in higher and higher precision, until the error requirement is satisfied.

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) be kept separate from the precision of the operations carried out on the data (which is concerned only with the arithmetic unit). By controlled precision we mean to have control over both the length of the significand and the range of the exponent.

Besides the key ideas of having control over the precision, and of separating the precision of the data from the precision of the operations, it is also desirable to satisfy several other requirements:

(1) The floating-point arithmetic should be perfect in some sense, and easy to remember; normalized and properly rounded arithmetic would be best (ties being broken by rounding to the nearest even).

(2) Interval arithmetic should also be supported. (This requires provision of round up and round down (algebraic) rounding modes.) Interval arithmetic is needed, for example, in the calculation of error bounds.

(3) Comparisons should be done properly, so that, for example, "IF (A > B)" cannot cause overflow/underflow.

(4) The most appropriate base is 10. This avoids I/O conversion errors; but, most important, it makes matters much simpler for the user.

(5) Provision must be made for handling exceptions such as overflow, underflow, and zero-divide.

One of the principle requirements of any design is that the user knows exactly what to expect under all circumstances. For example, the precision (of, say, 8 decimal digits) should be exact, not "at least" (8 decimal digits). The result after overflow or underflow should be the wraparound result (which is probably the most useful result under such circumstances - see, e.g., Sterbenz [3]). Special values would also be needed for unassigned, exact 0, infinity (i.e., A/0, A≠0), and indeterminate (e.g., 0/0). Hardware representations of these special values would therefore be needed.

2. OVERVIEW OF CADAC OPERATION

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 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

$$F = (B + C)/A$$

is to be evaluated. A machine language code sequence corresponding to this expression is:

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

Suppose the precisions of the significands of A, B, and C are declared to be 8 decimal digits, and the significand precision of F is 6. If the current significand precision of CADAC is 10, then the above code executes as follows:

- the value in B is padded out with two zeroes when it is loaded into the accumulator (ACC).
- the value in C is padded out with two zeroes as it is moved from main memory into CADAC and added to the contents of ACC, generating a properly rounded 10-digit sum in ACC.
- the value from A is padded out when loaded into CADAC where it is divided into the contents of ACC, generating a rounded 10-digit quotient.
- the contents of ACC are rounded to 6 digits and stored at location F in the main memory.

We are currently implementing a CADAC unit prototype to be interfaced with a PDP-11/34 minicomputer. A number of format details in the CADAC design to follow are thus influenced by the 16-bit wordlength of the PDP-11 architecture. It should be clear, however, that the design concepts generalize easily to other machine evironments.

3. NUMBER REPRESENTATION

A main memory representation (suitable for a 16-bit wordlength machine) for controlled-precision, decimal, floating-point numbers is shown in Figure 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.

The exponent field, E, is a binary integer ranging from 1 to 1023; and represents the true exponent, E', in excess-512 notation. Therefore, $E' = E - 512$. We use the symmetric range $-511 \le E' \le 511$, 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 of the significand as $2L + 2$. Therefore, the significand length ranges from 2 to 32 decimal digits, in 2-digit increments.

A significand length of 2 to 32 digits, along with a ±511 exponent range is sufficient for practically all computational requirements. However, a longer significand and/or 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 is used to contain the length descriptor and additional exponent bits. The significant digits follow this second attribute word.

Let us now consider the representation of the special values: exact 0, unassigned, infinity, 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 unassigned, infinity, or indeterminate. These three conditions are distinguished by the first digit of the significand following the attribute word, labelled d_{2L+2} in Figure 1. If this digit is hexadecimal E, then the operand is unassigned; if it is hexadecimal F the operand is indeterminate (0/0); and if it is hexadecimal D the operand is infinity (A/0). When 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 leading digit of the significand is set to hexadecimal E. A later assignment will change the special value to a normal value.

4. 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.

4.1 Precision

The floating-point number precision, which is a part of the main memory number representation, is independent of the current precision of the CADAC unit. Both significand precision and exponent range of calculations can be specified for the CADAC unit. In our main memory representation we have allowed programmer-determined significand precision. However, programmer-determined exponent range has not been included in the number representation. We feel that for most calculations exponent range control in CADAC is sufficient. Independent exponent range checking of main memory numbers can be done by host software.

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. 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.

4.2 Rounding

CADAC performs proper rounding by implementing "round to nearest or 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 occurs frequently:

1) During alignment of operands.
2) After an operation (rounding of results).
3) When loading operands.
4) When storing results.

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

5. INSTRUCTION SET AND EXCEPTION HANDLING

5.1 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 flag. The default precision is 16 digits and the default exponent range is ±255. (This can be doubled when calculating Euclidean Norm, etc.).

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.

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. This operation can generate an underflow exception.

CEIL: Load the accumulator with the smallest integer greater than or equal to the current contents of the accumulator. This operation can generate an overflow exception.

COMPARE: Sets the condition code from the result of subtracting from the accumulator the number whose address follows (if the signs are the same). This operation is similar to subtraction except that underflow/overflow exceptions will not occur. The accumulator is unaltered. If the operand signs differ, the subtraction is not required.

5.2 Floating-Point Integers

Our floating-point (FP) number representation can be used to represent large integers. These can be generated by the host or by the FLOOR or CEIL operations. The condition for a FP number to be an integer is

$$\text{LENGTH} > \text{EXPONENT} \ \underline{\text{AND}} \ \sum_{i=\text{EXP}+1}^{i=\text{LENGTH}} \text{DIGIT}_i = 0$$

$$\underline{\text{OR}}$$

$$\text{LENGTH} = \text{EXPONENT}$$

where DIGIT_1 is the most significant digit of the significand. If it is known that a number is a FP integer, the programmer can test for a rounding "error". This can occur during any rounding operation.

5.3 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 occurred. Let us consider the results of exponent overflow. If the exponent range is < ±511, the result of an addition may still be correctly represented in the machine and can be correctly stored in the destination. If the exponent range was ±511 during the operation, the wraparound result will be stored in the accumulator. Now consider multiplication. If the exponent range is < ±255, the result of an operation will be correctly represented. Otherwise the wraparound result will be stored in ACC. An overflow can also occur when CADAC is loaded with a number whose exponent exceeds the current exponent range of the unit. This situation should be avoided.

If the overflow exception was caused by a zero-divide error, the infinity code will be stored in the accumulator if the numerator was non-zero, and the indeterminate code will be stored if the numerator was zero.

We recall at this time that overflow can also occur when a result is returned to main memory. However, the exponent range of the number is not stored with the number and so overflow on a store operation cannot be detected by CADAC. Overflow can be checked easily by the software. The only overflow on a store that is detected by the unit is the case of storing an extended exponent range number in a non-extended destination. Overflow is set if the exponent of the result is > +511.

2) UNDERFLOW: Underflow is handled similar to overflow. Note that the wraparound result is always stored; it is never replaced with a true zero.

3) ILLEGAL OPERATION: This flag is set when the unit attempts to operate on an unassigned, indeterminate, or infinity operand. The indeterminate code is stored.

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

6. THE PIPELINE

CADAC uses a hardware multiplier to achieve high performance. The multiplier includes a 3-stage pipeline running at 10 Mhz. This pipeline will multiply two 32-digit significands in about 30 μs. The design chosen for the multiplier unit uses BCD ALUs in a 2-digit x 2-digit structure. This structure multiplies 2 multiplicand digits by 2 multiplier digits and adds overlapped digits to produce 2 digits of partial product and 2 overlapped digits that are internally fed back. The first cycle produces 2 final product digits and 2 overlapped digits. All the multiplication is done in digit pairs and does not require shifting during product formation.

The pipe requires 17 x 16 + 4 iterations to multiply two 32-digit significands. At 10 Mhz this takes 28 μs. It does not appear to be cost-effective to increase the level of parallelism employed in the above design.

6.1 Operation of the Pipeline

To understand the operation of the pipeline, consider the multiplication of 2-digit numbers, $a_1a_0 \times b_1b_0$, as illustrated in Figure 2. Four 2-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 lookup tables. Now note the significance of the 2-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 ALUs. 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 4-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 $a_1\ a_0 \times b_1\ b_0 = \alpha_{3_0}\ \alpha_{2_0}\ \alpha_{1_0}\ \alpha_{0_0}$. In the second cycle the pipeline produces $a_3\ a_2 \times b_1\ b_0 = \alpha_{3_1}\ \alpha_{2_1}\ \alpha_{1_1}\ \alpha_{0_1}$. 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 $p_1\ p_0$. The complete set of additions performed in the pipe is illustrated in Figure 3. The 8 required additions are indicated with brackets.

The 3-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 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 Figure 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 $p_1\ p_0$ (to which

$a'_{1_{i-1}} \times b'_{1_{i-1}}$ have 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 Figure 4 and its interconnection with the other components of CADAC is shown in Figure 5. The inputs to the pipeline are:

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

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

Clearly all 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).

7. MEMORY REQUIREMENTS

7.1 Partial Product Storage (PROD)

The partial product digits emerging from the pipe must be bufferred 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 Figure 5.

7.2. 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 is loaded into the machine and written back in blocks of 32 digits as required by a calculation.

8. STANDARD (NON-EXTENDED) ARITHMETIC

8.1 Multiplication

We will consider the multiplication process in detail since the pipe has been optimized for this operation. See Figure 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. Zeroes 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 non-zero, 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.

Typical execution times for all four arithmetic operations are shown in Table 1.

8.2 Division

Division is more complex than multiplication. We provide only an overview here. A straightforward long-division implementation is used. A PROM lookup 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.

8.3 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. The minuend and subtrahend are chosen so that the difference will most likely be positive. If the difference is negative (10's complement), another pass through the ALUs is required to complement the result.

9. 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 RAMs 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.

10. CONTROL

So far we have not discussed the control of CADAC. The system uses an AMD 2910 µsequence controller and three 2903 4-bit ALU slices. The 2903s are used for:

1) exponent arithmetic
2) length calculation and handling
3) provision of 12 bit registers to hold
   i) exponent limit
   ii) precision
   iii) operand lengths
   iv) trailing zero counts
   v) other
4) control logic

The 2910 µsequence controller addresses up to 4K words of µcode. It requires a 12-bit branch address field in the µcode word. This field can also be used to provide constants for the 2903s or for the pipeline, if required.

REFERENCES

[1] "The Proposed IEEE Floating-Point Standard", four articles in COMPUTER, Vol. 14, No. 3, March 1981.

[2] Hull, T.E., "Desirable Floating-Point Arithmetic and Elementary Functions for Numerical Computation", Proceedings of 4th Symposium on Computer Arithmetic, pp. 63-69, IEEE Publ. No. 78CH1412-6C, Santa Monica, California, October, 1978.

[3] Sterbenz, P.H., Floating-Point Computation, Prentice-Hall, Englewood Cliffs, N.J., 1974.

Table 1. Typical execution times* (µsec.)

DIGITS:	8	14	24	32
ADD/SUB	6·6	7·8	10·3	12·3
MULT	4·1	7·7	18·2	29·6
DIV	20·3	39·9	46·2	131·0

*execution times do not include operand fetching.

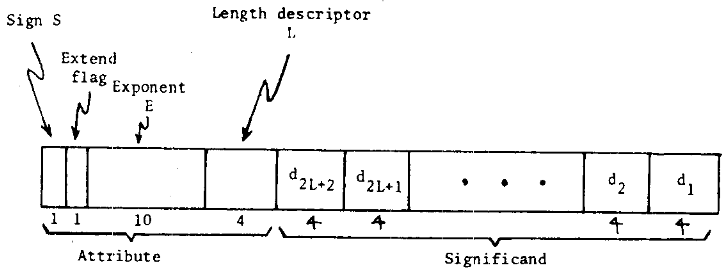

Figure 1. Standard floating-point number representation.

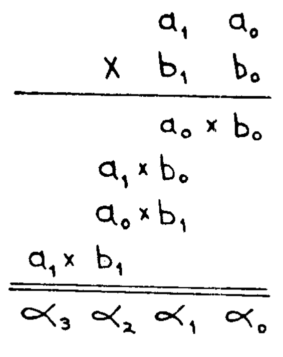

Figure 2. A 2x2 multiply operation.

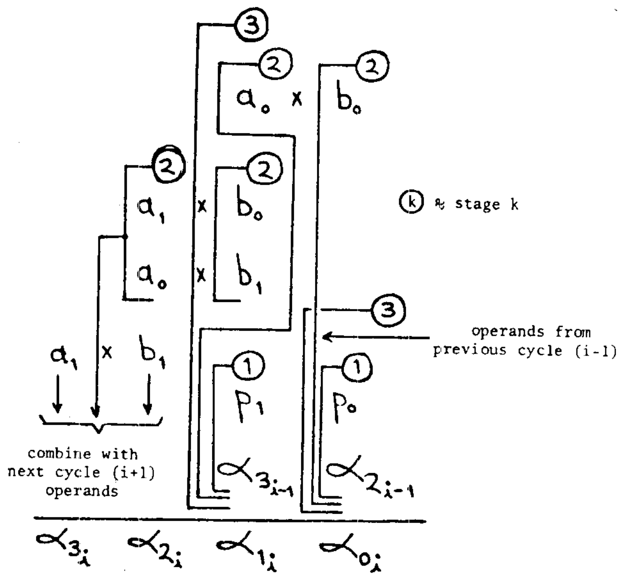
3
2
a_0 x
2
b_0
2
a_1
x
2
b_0
a_0
x
b_1
3
operands from previous cycle (i-1)
a_1
x
b_1
1
p_1
1
p_0
combine with next cycle (i+1) operands
$\alpha_{3_{i-1}}$
$\alpha_{2_{i-1}}$
α_{3_i} α_{2_i} α_{1_i} α_{0_i}
k ≈ stage k

Figure 3. Pipeline stage operations.

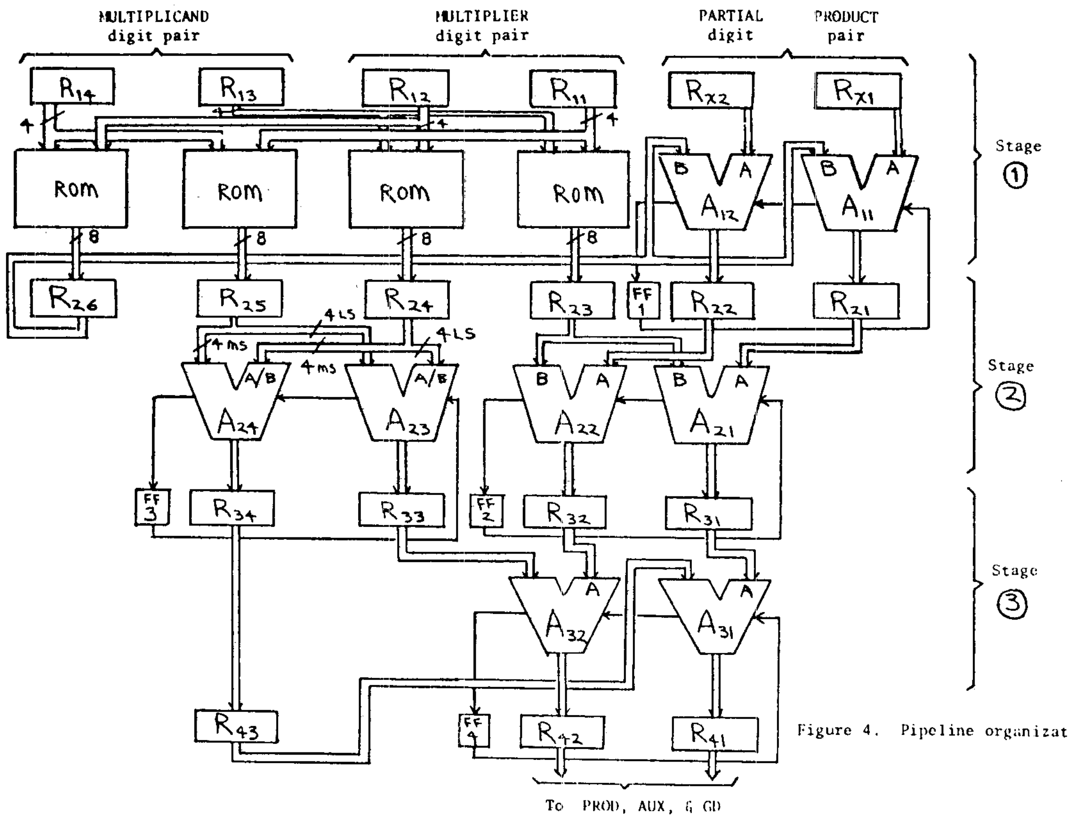

Figure 4. Pipeline organization.

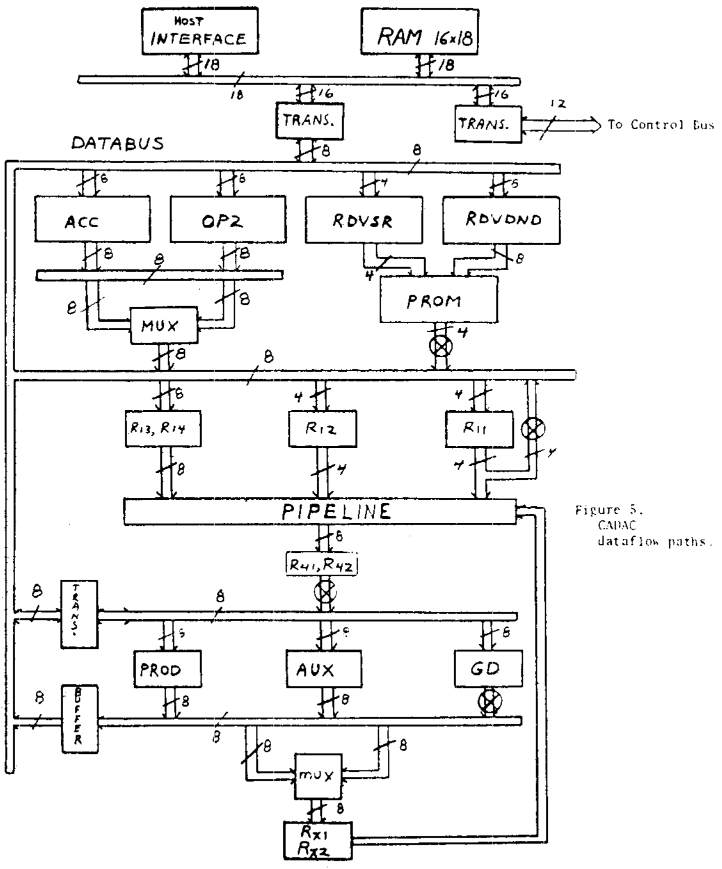

Figure 5. CADAC dataflow paths.