AN INTEGRATED RATIONAL ARITHMETIC UNIT

Peter Kornerup
David W. Matula

Computer Science Department, Aarhus, Denmark
Computer Science Department, Southern Methodist University, Dallas, Texas

Abstract
Based on the classical Euclidian Algorithm, we develop the foundations of an arithmetic unit performing Add, Subtract, Multiply and Divide on rational operands. The unit uses one unified algorithm for all operations, including rounding. A binary implementation, based on techniques known from the SRT division, is described. Finally, a hardware implementation using ripple-free, carry-save addition is analyzed, and adapted to a floating-slash representation of the rational operands.

Keywords: Rational numbers, Floating-slash, Euclidian Algorithm, SRT-techniques, Carry-save addition.

1. Introduction and background

This paper develops the foundations and possible implementations of a unified arithmetic unit capable of performing the standard arithmetic operations on rational operands. The fundamental idea used in the implementation is that the classical Euclidian algorithm, when applied to the numerator and denominator of some rational number p/q, can be used to define some transformations on a given 2 x 2 matrix. With initialization of the matrix depending on the operand r/s and the operation to be performed, the exact same algorithm computes successive approximations to the result wanted.

This work is a continuation of previous work on the fixed- and floating-slash rational number representations [1], the underlying number system [2, 3] and rational arithmetic [4]. However the previous work on arithmetic has been concentrating on the properties of the "natural" rounding procedure used, which is called the mediant rounding. This rounding is based on choosing the "last representable" continued fraction convergent, which in the number theory is called the "best rational approximation". The "last representable" convergent is chosen as to fit suitable size constraints on numerators and denominators, depending on the packing of these in a computer word. In fixed-slash it is assumed that two fixed and equal size fields are used to represent numerator and denominator respectively. In floating-slash the boundary between the two fields can be moved (an extra field is used to position the "slash"), thus providing a more flexible representation, and greater range of representable numbers.

In section 2 the foundations of the arithmetic unit are developed and a number of observations concerning its operation is made. Section 3 deals with a binary implementation of the Euclidian algorithm and its adaption to the arithmetic unit. In section 4 it is demonstrated that a redundant "carry-save" representation may be used in a hardware implementation of the algorithm, to speed up the additions/ subtractions needed. Finally, section 5 outlines a hardware implementation of a unit based on floating-slash represented operands.

The paper assumes only a very limited knowledge of number theory; some good references are [5] and [6]. The previously mentioned references by these authors provide a background on the underlying number system. The terminology on continued fractions used in this paper is fairly standard, however we will extend the standard notion of a canonical continued fraction expansion to include expansions of negative numbers (by negation of all partial quotients).

2. The foundations for an arithmetic unit

An implementation of the mediant rounding may be based on the fact that the well-known Euclidian Algorithm can be extended to compute the continued fraction convergents p_i/q_i of a rational number $p/q = [a_0, a_1, \ldots, a_m]$, utilizing implicitly the quotients a_i.

Algorithm EC (Euclidian Convergent Algorithm)
For any $p \geq 0$, $q \geq 1$, let

$$b_{-2} = p; \quad p_{-2} = 0; \quad q_{-2} = 1;$$
$$b_{-1} = q; \quad p_{-1} = 1; \quad q_{-1} = 0.$$

For $i = 0, 1, \ldots,$ while $b_{i-1} \neq 0$, determine a_i as the quotient and b_i as the non-negative remainder of the division of b_{i-2} by b_{i-1}, so

(*) $$b_i = -b_{i-1} \cdot a_i + b_{i-2},$$

and compute

$$p_i = p_{i-1} \cdot a_i + p_{i-2},$$
$$q_i = q_{i-1} \cdot a_i + q_{i-2}. \quad \square$$

Notice that the algorithm works as well if p and/or q are negative, $q \neq 0$, by choosing the remainder b_i of the same sign as the dividend b_{i-2} in (*). If $q = 0$, $p_{-1}/q_{-1} = 1/0$ may be considered the (only) convergent.

We may now observe that the initial matrix (the seed) of Algorithm EC

$$\begin{Bmatrix} p_{-2} & q_{-2} \\ p_{-1} & q_{-1} \end{Bmatrix} = \begin{Bmatrix} 0 & 1 \\ 1 & 0 \end{Bmatrix}$$

may be substituted by an arbitrary 2 x 2 matrix:

$$\begin{Bmatrix} a & c \\ b & d \end{Bmatrix}$$

to yield a sequence of pairs (u_i, v_i) instead of the (p_i, q_i) pairs, where u_i and v_i are linear combinations of p_i and q_i.

Algorithm EAA (Euclidian Arithmetic Algorithm)
For any (p, q), $q \neq 0$ and a, b, c, d let

$$b_{-2} = p; \quad u_{-2} = a; \quad v_{-2} = c;$$
$$b_{-1} = q; \quad u_{-1} = b; \quad v_{-1} = d.$$

For $i = 0, 1, \ldots$, while $b_{i-1} \neq 0$ determine a_i as the quotient and b_i as the remainder (of the same sign as b_{i-2}) of the division of b_{i-2} by b_{i-1}, so

$$b_i = -b_{i-1} \cdot a_i + b_{i-2}$$

and compute

$$u_i = u_{i-1} \cdot a_i + u_{i-2},$$
$$v_i = v_{i-1} \cdot a_i + v_{i-2}. \quad \square$$

Then the proof of the following is obvious:

Lemma 1: The (u_i, v_i) $i = 0, 1, \ldots, m$ determined by algorithm EAA with seed

$$\begin{Bmatrix} a & c \\ b & d \end{Bmatrix}$$

satisfies the following relations

$$u_i = aq_i + bp_i,$$
$$v_i = cq_i + dp_i$$

where p_i/q_i, $i = 0, 1, \ldots, m$ are the convergents of p/q. $\square$

Lemma 2: Given any bilinear form

$$f(x) = \frac{a + bx}{c + dx}$$

and any rational number p/q, then Algorithm EAA computes (u_i, v_i) such that

$$f(p_i/q_i) = u_i/v_i,$$

where p_i/q_i, $i = 0, 1, \ldots, m$ are the convergents of p/q. $\square$

Observe that although p_i/q_i is in reduced form, u_i/v_i as computed by EAA need not be so. Also note that u_i/v_i does not in general form a sequence of convergents of f(p/q).

If f has no pole in the closed interval $[p_i/q_i, p_{i+1}/q_{i+1}]$ (or equivalently v_i and v_{i+1} are non-zero and of same sign) then

$$\left| \frac{u_i}{v_i} - f(\tfrac{p}{q}) \right| \leq \left| \frac{u_i}{v_i} - \frac{u_{i+1}}{v_{i+1}} \right|$$
$$\leq \left| \frac{ad - bc}{v_i v_{i+1}} \right| \quad \text{for } i < m,$$

where the numerator (ad - bc) is the determinant of the "seed-matrix".

We may now pick particular seed matrices to realize basic arithmetic operations. These will correspond to "curried" operations of LISP, e.g. they combine an operation with an operand, the operand being a rational number.

Theorem 1 Algorithm EAA, when seeded with the following matrices:

$$\begin{Bmatrix} r & s \\ s & 0 \end{Bmatrix}, \begin{Bmatrix} -r & s \\ s & 0 \end{Bmatrix}, \begin{Bmatrix} 0 & s \\ r & 0 \end{Bmatrix} \text{ and } \begin{Bmatrix} 0 & r \\ s & 0 \end{Bmatrix}$$

implement the curried operations:

Add r/s, Sub r/s, Mult r/s and Div r/s

respectively, applied to the operand p/q.

Proof With:

$$\begin{Bmatrix} a & c \\ b & d \end{Bmatrix} = \begin{Bmatrix} r & s \\ s & 0 \end{Bmatrix}$$

it follows that

$$u_i = rq_i + sp_i \text{ and } v_i = sq_i,$$

hence

$$\frac{u_i}{v_i} = \frac{rq_i + sp_i}{sq_i} = \frac{r}{s} + \frac{p_i}{q_i}.$$

The other three cases follow equivalently. $\square$

Observation 1 Applying a unit operator (e.g. Add 0/1 or Mult 1/1) produces the sequence $u_i = p_i$, $v_i = q_i$, i.e. performs the EC algorithm. $\square$

Observation 2 The computed u_i and v_i are the very same numerators and denominators that would have been obtained using standard arithmetic rules on p_i/q_i and r/s. $\square$

Observation 3 p/q need not be in reduced form, as only its convergents will affect the growth of u_i and v_i. p/q may thus be the (possibly unreduced) result of a previous arithmetic operation. □

Theorem 1 and these three observations form the key ideas for an arithmetic unit working on rational operands.

Observation 4 Consider p/q represented as an integer tuple placed in a pair of registers, which together may be identified as one "accumulator" (out of possibly many). Take an operand r/s (say fetched from a memory in fixed or floating slash packed format), together with an operator, forming the seed matrix. The arithmetic unit then performs the EAA algorithm, running it to completion, or until u_i and/or v_i exceeds predetermined bounds (in which case the previous u_{i-1} and v_{i-1} will be chosen). Then write the resulting u and v into the registers originally containing p and q, thus forming the new accumulator contents.

When the rational number in an accumulator is to be stored away in packed format, a unit operator seed matrix is used in the EAA algorithm, and the algorithm is stopped with the last values of u and v which will fit the packed representation, thus realizing the mediant rounding. □

We may picture the unit as consisting of 6 registers organized and initialized the following way:

$$\begin{array}{c|cc} p & a & c \\ q & b & d \end{array}$$

In the next section we will discuss a possible implementation of such an arithmetic unit, but before that we will make some final observations on the functioning of the unit.

Observation 5 Assume the arithmetic unit is capable of holding u's and v's large enough to hold the numerator and denominator of the result of any dyadic arithmetic operator (+, −, *, /), applied to any pair of operands from F, e.g. for the add operator to hold u = ps + rq and v = qs.

Assume r/s ∈ F, where F is a rational number system (e.g. a particular fixed or floating slash system). Now given any p/q, p/q can be rounded into F by the mediant rounding Φ_F, say $\Phi_F(p/q) = p_k/q_k$ (the k'th convergent of p/q).

When the arithmetic unit is initialized with p/q and seeded with r/s and any operator, the resulting u/v will be the exact result of the operator applied to r/s and some convergent p_j/q_j of p/q, where $j \geq k$. That is, the unit will compute a result u/v such that

$$\left| \frac{u}{v} - \frac{p}{q} \otimes \frac{r}{s} \right| \leq \left| \Phi_F(p/q) \otimes \frac{r}{s} - \frac{p}{q} \otimes \frac{r}{s} \right|$$

where ⊗ is the operator applied. □

Having noticed that the unit needs registers large enough to hold results which correspond to "double precision" one might ask whether there is any way that say two "normal precision" words could be used used for storing a "double precision" result. The following observation provides a partial answer to this question.

Observation 6 When the result of a mediant rounding of some "accumulator contents" has been built up as described in observation 4, the (u_i, v_i) pair which has been packed away represents the "most significant part" of the original p/q. However the partial remainders b_i and b_{i+1} represent all the information needed to compute the "remaining" quotients of the expansion of p/q. Hence, if after storing (u_i, v_i) away, the unit is seeded again with the identity matrix, the unit can go on computing a rounded value of the b_i/b_{i+1}, and so on until all information from the original p/q has been extracted in the form of "packed" rational numbers. The original p/q can later be restored (in reduced form) from its components, by seeding the unit with the identity matrix, and successively loading the components into the "p" and "q" registers and breaking them down into quotients, while building up (u, v) pairs. □

The previous observation only provides a sketch of handling "multiple precision" operands. There are some problems concerning very large quotient values, whose solution requires floating-slash representations allowing the slash to move "outside" the word, thus providing scaled representations. We will not discuss the details here, but raise another obvious question. Since the integers form a subset of the rationals, the arithmetic unit as described handles integers in a natural way, but what about efficiency? Say if the unit is to divide two integers i and j, represented as i/1 and j/1. Thus the unit will be initialized as follows:

$$\begin{array}{c|cc} i & 0 & j \\ 1 & 1 & 0 \end{array}$$

and the unit will divide 1 into i, while multiplying 1 by the quotient found (i). By shifting out the top row and adding the new, the result is:

$$\begin{array}{c|cc} 1 & 1 & 0 \\ 0 & i & j \end{array}$$

which is fine, but it was a cumbersome way to get i/j. Also the add and subtract operation on integers looks inefficient, but fortunately there is a trick:

Observation 7 When the arithmetic unit is seeded and loaded with values

$$\begin{array}{c|cc} p & a & c \\ q & b & 0 \end{array}$$

such that |q| = |b|, then the following initialization

$$\begin{array}{c|cc} \hline b & a & c \\ q & p & 0 \end{array}$$

will yield the same result of the EAA algorithm (in one cycle), except possibly for common factors of u_i and v_i in the (u_i, v_i) pairs. □

Observation 8 When the arithmetic unit is loaded and seeded with values

$$\begin{array}{c|cc} \hline p & 0 & c \\ q & b & 0 \end{array}$$

such that $|p| = |c|$, then the following initialization

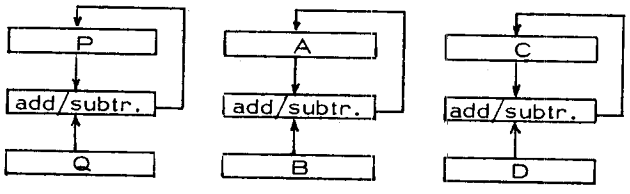
$$\begin{array}{c|cc} \hline p & 0 & q \\ c & b & 0 \end{array}$$

will yield the same result of the EAA algorithm (in one cycle), except possibly for common factors of u_i and v_i in the (u_i, v_i) pairs. □

3. A binary implementation

It is obvious that in an implementation of the EAA algorithm it is only necessary to store two consecutive values of b_i's, u_i's and v_i's, and it is not necessary either to record the quotient values a_i - not even to have these exist explicitly. In [4] a shift-subtract binary implementation of the EC algorithm was described, based on the idea that bits of the binary representation of a_i (the quotient of b_{i-2} and b_{i-1}) can be determined and immediately used to accumulate multiples of p_{i-1} and q_{i-1}, thus building up p_i and q_i respectively. Utilizing hardware parallelism, the subtractions of the division and the additions involved in the multiplications can be executed in true parallel.

For the implementation of the EAA algorithm we will use a binary, 2's complement representation of the integers, and gain some speed by using techniques known from the SRT division algorithm. Let us assume the existence of 6 registers, named P, Q, A, B, C and D corresponding to their initializations in EAA. The registers are pairwise connected to an add/subtract circuitry, controlled by the signs of the contents of the P and Q registers.



Furthermore a shift-register K is used to keep track of any un-alignment of the contents of registers. The capacity (size) of the registers is for the moment left unspecified.

Assuming proper initialization with $(p, q) \neq (0, 0)$ in the P and Q registers, and seeding of the A, B, C and D registers, the extended Euclidian algorithm may be implemented as follows:

Algorithm BNE (binary normalized/nonrestoring Euclidian algorithm)

```
{K is an auxiliary register initialized to 1}
while {P or Q register not normalized}
  do {leftshift P and Q}
while Q ≠ 0 do
begin
L: while {Q register not normalized}
     do {leftshift Q, B, D and K} ;
   loop
     while {P register not normalized} do
       begin
         exitloop if K = 1;
         {leftshift P and rightshift B, D and K}
       end;
     if sign (P) = sign (Q)
     then P:=P-Q and A:=A+B and C:=C+D
     else P:=P+Q and A:=A-B and C:=C-D;
   end;
   swap (P,Q) and swap (A,B) and swap (C,D);
end;
```

Possible parallelism has been expressed using and between statements, as opposed to ";" for sequential execution. □

Upon exit the registers B and D contain values u and v such that $(u, v) = (u_n, v_n)$ or $(u, v) = (-u_n, -v_n)$ with

$$u_n = aq_n + bp_n$$
$$v_n = cq_n + dp_n,$$

where p_n/q_n is the reduced version of p/q, and a, b, c, d are the original values of the A, B, C, D registers.

We are interested in possibly stopping the algorithms prematurely based on certain size limitations of the u_i, v_i values computed by the algorithm. Unfortunately, the BNE algorithm does not compute the pairs (u_i, v_i) corresponding to "convergents" (defined in the usual way) based on a signed continued fraction expansion

$$\frac{p}{q} = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \ddots + \cfrac{1}{a_m}}}$$

where the partial quotients a_i are arbitrary integers, except that $a_i \neq 0$ and $[a_i, a_{i+1}, \ldots, a_m] \neq 0$ for $1 \leq i \leq m$.

The sequence of a_i's used in the BNE algorithm in computing (u_i, v_i), i.e. determining the convergents p_i/q_i, are the partial quotients of such a signed continued fraction. To see this, and to analyze the sequence of convergents computed by the BNE algorithm, we need the following lemma.

Lemma 3: The sequence of partial quotients a_i, implicitly computed by the BNE algorithm, satisfies the following

i) $|a_i| \geq 1$ for $1 \leq i \leq m$

ii) $|a_i| = 1 \Rightarrow \text{sign}(a_i) = \text{sign}(a_{i+1})$ for $1 \leq i < m$

iii) $|a_i - [a_i, a_{i+1}, \ldots, a_m]| < 1$ for $1 \leq i \leq m-1$ and for $i = 0$ when $m \geq 2$.

Proof For $1 \leq i \leq m$ let b_i be the i'th partial remainder, as found in the register Q at label L of the algorithm BNE, after the i'th execution of the outer loop. Also b_{i-1} is in P at label L, b_{i-1} is normalized in P, b_{i-1} and b_i are scaled to the same position of unity. b_i is not normalized in Q, as the only way to escape the inner loop is with P not normalized (which is later swapped with Q). Hence $|a_i| \geq 1$, as the inner loop will execute at least once, which proves (i).

Frem the same considerations it also follows that $|b_{i-1}/b_i| \geq 1$, where the inequality is sharp for $i < m$. Now notice that b_{i-1}/b_i represents the remainder term from which the rest of the quotients $a_i, a_{i+1}, \ldots, a_m$ are to be determined, i.e. $b_{i-1}/b_i = [a_i, a_{i+1}, \ldots, a_m]$. Also note from the algorithm that the sign of a_i is the same as that of b_{i-1}/b_i.

For $1 \leq i < m$, assume that $|a_i| = 1$. Since $b_{i+1} = b_{i-1} - a_i b_i$, it follows that $b_{i+1}/b_i = b_{i-1}/b_i - a_i$, so b_i/b_{i+1} will have the same sign as b_{i-1}/b_i, from which (ii) follows.

(iii) follows for $1 \leq i \leq m-1$ from $[a_i, a_{i+1}, \ldots, a_m] - a_i = b_{i+1}/b_i$, and in case $i = 0$, $m \geq 2$ from the fact that the value of such a continued fraction cannot be integral when it has 3 or more nonzero quotients. □

Theorem 2 Let $[\hat{a}_0, \hat{a}_1, \ldots, \hat{a}_n]$ be the canonical continued fraction expansion of p/q, and

$$\hat{p}_0/\hat{q}_0, \hat{p}_1/\hat{q}_1, \ldots, \hat{p}_n/\hat{q}_n$$

be the canoncial convergents of p/q. The BNE algorithm, seeded with the unit matrix, and applied to p/q, computes a sequence

$$p_0/q_0, p_1/q_1, \ldots, p_m/q_m,$$

where $(p_i, q_i) = (u_i, v_i)$, $i = 0, 1, \ldots, m$, satisfy the following

i) $\gcd(p_i, q_i) = 1$ $\quad i = 0, 1, \ldots, m$

ii) $p_m/q_m = \hat{p}_n/\hat{q}_n$

iii) If for some i, $1 \leq i \leq m-1$, $p_i/q_i \neq \hat{p}_j/\hat{q}_j$ for all $j = 0, 1, \ldots, n$ then there exists a k, $0 \leq k \leq n-1$ such that

$$\frac{p_{i-1}}{q_{i-1}} = \frac{\hat{p}_k}{\hat{q}_k} \quad \text{and} \quad \frac{p_{i+1}}{q_{i+1}} = \frac{\hat{p}_{k+1}}{\hat{q}_{k+1}}$$

iv) If for some i, $1 \leq i \leq n$ $\hat{p}_i/\hat{q}_i \neq p_j/q_j$ for all $j = 0, 1, \ldots, m$ then there exists a k, $0 \leq k \leq m-1$ such that

$$\frac{\hat{p}_{i-1}}{\hat{q}_{i-1}} = \frac{p_k}{q_k} \quad \text{and} \quad \frac{\hat{p}_{i+1}}{\hat{q}_{i+1}} = \frac{p_{k+1}}{q_{k+1}} .$$

v) If $\text{sign}(a_i) = \text{sign}(a_{i+1})$ then there exists a k such that

$$p_i/q_i = \hat{p}_k/\hat{q}_k.$$

vi) If $\text{sign}(a_i) \neq \text{sign}(a_{i+1})$ then there exists a k such that

$$\frac{p_i - p_{i-1}}{q_i - q_{i-1}} = \frac{\hat{p}_k}{\hat{q}_k} .$$ □

The proof of the theorem is quite lengthy and will be left out. It may be based on Lemma 3, the fact that the canonical expansion is unique, and the following transformation rule for continued fractions:

If $|u| = 1$ and $0 \leq i \leq n-1$ then

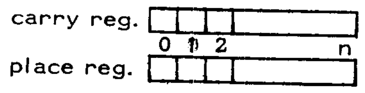
$$[a_0, a_1, \ldots, a_i, u, a_{i+2}, a_{i+3}, \ldots, a_n]$$
$$= [a_0, a_1, \ldots, a_i+u, -)a_{i+2}+u), -a_{i+3}, \ldots, -a_n].$$

The theorem can be used to modify the BNE algorithm such that, when stopped prematurely, it is possible to recover a (u, v) pair corresponding to a canonical convergent.

4. Implementation using Carry-Save representation

Since the BNE algorithm uses repeated additions and subtractions, the use of redundant number representations may be beneficial, in avoiding the time delay of carry propagation through registers. One possible redundant representation is the binary Carry-Save form, which is convenient when performing addition/subtraction. A number R is represented as two bit strings $R = R^c + R^p$, where R^c denotes the set of carries, and R^p the set of "place-values". If $R = A+B$, R^c and R^p can be computed from A and B by parallel, bitwise logical operations, i.e. without carry propagation. If the operands (A and B) themselves are in carry-save form, R^c and R^p (hence R) can be computed in two levels of bitwise adders (3 to 2 adders), without carry propagation (see Fig. 1).

Let us, for the present analysis, assume that we have numbers represented in pairs of registers, each of n+1 bits:



Negative numbers are represented in 2's complement notation, andthe positional value of a bit is the same, whether placed in position i of the carry reg., or in position i of the place reg. Hence the representation is symmetric in the two registers, and the value represented can be found by adding the contents of the two registers, interpreted as numbers having the same unit position.

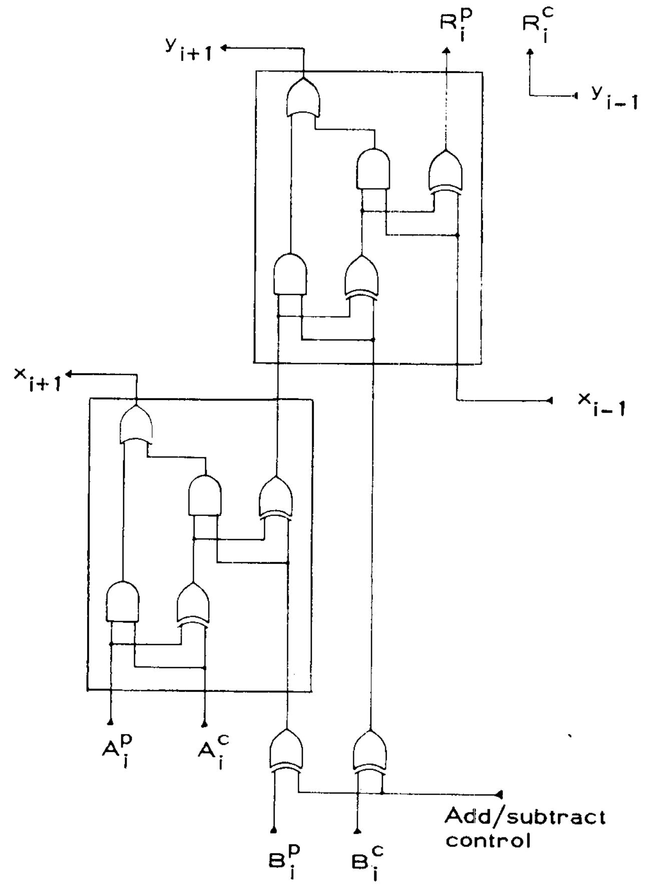


Figure 1. Bit slice of add/subtract circuitry computing R = A ± B in carry-save representation

When adding two numbers, each represented in two such registers, through the two levels of 3-2 adders, two carry values will be generated in the left end of the registers, and these will be discarded. At the right end, twice a carry may be brought in, and in the add operation these will be filled with zeroes.

In subtraction the contents of both registers representing the subtrahend must be inverted, and the two righthand carries brought in have to be ones.

As usual with redundant representations, there are some problems with normalization, sign determination, zero- and overflow-detection, which will be needed in the BNE algorithm.

For the analysis of normalization let us assume that the unit position of the carry and place registers is position 0, and furthermore that the number represented is in the half open interval $[-1, 1)$. Hence irrespective of which permissible number is chosen, it can always be represented in one of the registers, the other register containing all zeroes.

Without computing the sum of the contents of the carry and place registers (i.e. full carry propagation) it is not possible to normalize a number in carry-save form to be within two powers of two. However it turns out that using just addition with carry propagation of the leading 3 bits, a carry-save represented, nonzero number can be scaled to be in the interval $[-1, -1/4)$ or $[1/4, 1)$, and smaller intervals adding more positions.

To see this in the 3 bit case, notice that the contribution from positions 3 through n (which have not been added in) represents a variation h, $0 \leq h \leq 1/2 - 2^{-n}$. The following table shows the 8 possible combinations of the 3 bit result of the addition of pos. 0-2.

case	pos. 0-2	range
0	0 0 0	$[0, 1/2)$
1	0 0 1	$[1/4, 3/4)$
2	0 1 0	$[1/2, 1)$
3	0 1 1	$[3/4, 1)$ or $[-1, -3/4)$
4	1 0 0	$[-1, -1/2)$
5	1 0 1	$[-3/4, -1/4)$
6	1 1 0	$[-1/2, 0)$
7	1 1 1	$[-1/4, 1/4)$

Table 1

Clearly, cases 2, 3 and 4 represent properly normalized numbers, however the cases 1 and 5 represent numbers which may or may not be normalized in the ordinary sense.

The BNE algorithm uses normalization of the P and Q registers to compute a "distance" $d(p, q) = p - q \cdot \text{sign}(pq)$ which is then substituted for p in the P-register. Actually normalization of p and q is used to get an approximation to the value $\min_i |d(p \cdot 2^i, q)|$ which may not be achieved just using ordinary normalization (e.g. $|d(3/8, 1/2)| < |d(3/4, 1/2)|$). Considering the cases 1 through 5 as representing properly scaled numbers implies that we accept values of p and q belonging to the intervals $[-1, -1/4)$ or $[1/4, 1)$.

Unfortunately, this choice may cause the BNE algorithm to get into an infinite loop. E.g. in the case $p \in [1/4, 1/2)$ and $q \in [3/4, 1)$, where $d(p, q) \in (-3/4, -1/2)$ will be substituted for p to be used in the next step. Since d(p, q) will be considered properly scaled (e.g. fall in one of the cases 1 through 5), the next step will compute $d(d(p, q), q) = (p-q)+q = p$, hence the algorithm will loop forever unless special precautions are taken.

In the "reversed" example $p \in [3/4, 1)$ and $q \in [1/4, 1/2)$ where $d(p, q) \in (1/4, 3/4)$, the algorithm may again go through two steps computing $d(d(p, q), q) = (p-q)-q = p-2q \in (-1/4, 1/2)$, or possibly (when $p-2q \in [1/4, 1/2)$) a third step computing $d(d(d(p, q), q), q) = p-3q \in (-1/4, 1/4)$.

Checking the various cases one finds that an attempt by the algorithm to do more than three add/subtracts, without intermediate scaling of the contents of the P-register, implies that the algorithm is in an infinite loop. However noticing that only in the infinite loop does the algorithm alternate between performing an add and a subtract operation without intermediate shifting, this can be used to force a shift after the second add/subtract step.

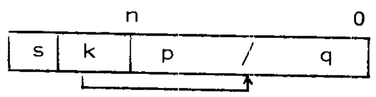
In the BNE algorithm it is necessary to determine the sign of the contents of the P and Q registers, to compute d(p, q). Considering the cases 1 to 5 of Table 1, only in case 3 is the sign not uniquely determined by bit zero. However the bit pattern 0 1 1 cannot be the result of a computation of d(p, q) as previously described, since an accepted value (substituted for p in the P-register) will always be in the interval $[-1/2, 1/2)$. Neither can this bit pattern appear by shifting (scaling) of a register contents previously falling in one of the cases 0, 6, 7. The only remaining possibility for a case 3 situation to occur is through a forced shift (to avoid infinite looping) of a number which was previously in one of the cases 1, 2, 4 or 5, but then the sign is already known.

Finally it is necessary to detect the presence of zero to stop the algorithm, and avoid an attempt to "normalize" the Q-register with zero contents. However, if the algorithm keeps track of the unit position of the actual integer contents of the Q-register, the "normalization" of the Q-register may be used to detect the presence of zero (when the unit position has been moved into position 2, and the value falls in case 0).

In this section we have only analyzed the use of carry-save representation of the contents of the P- and Q-registers of the BNE algorithm, where we have demonstrated that the flow of the algorithm can be controlled, apart from possible overflow situations in the A, B, C and D registers. The handling of these situations is closely related to the external representation of operands, and will be dealt with in the next section.

5. A floating-slash arithmetic unit

Let us assume that the representation of rationals external to the arithmetic unit (i.e. the storage representation), is a floating slash representation:



where s is the sign, p and q are the (unsigned) numerator and denominator, and k is a pointer to the "slash-position", to be interpreted as follows: When k has value i, p occupies positions i through n, and q occupies (bits reversed) positions 0 through i-1, plus a non-existing position -1 which has the value 1. In this way an extra position is gained, at the expense of not being able to represent zero denominators.

If q = 1 and k = 0 the word can be interpreted as "normal" sign-magnitude represented integer in the range $-2^{n+1} < p < 2^{n+1}$. If k > n there is no room for p, which then will be assumed to have the value 1, thus p and q have the same range.

The registers of the arithmetic unit have to be able to contain the numerator and denominator of the result of any operator $\odot \in \{+, -, *, /\}$ applied to any pair of such operands, p/q and r/s. This implies that the registers have to have at least 2n + 2 bits, plus room for sign and possibly overflow detection. We will choose N = 2n + 4 bits for all registers.

In section 4 it was described how the control of the BNE algorithm, using carry-save representation of operands, could be based on the contents of the P and Q registers, which we will consider the (only) "accumulator" of the arithmetic unit. The A, B, C and D registers and the associated add/subtract circuitry just operate as slaves, except for the problem of overflow which was postponed. We assume that these registers and associated logic are built much the same way as the P- and Q-registers, including shift capability, but with a four bit standard adder on the leading bits. Also associated with the A, B pair of registers is a shift register I, and with the C, D pair a shift register J. These will be used to keep track of the unit position of the (integer) contents of the associated register pairs, as the register contents are normalized in the same way as discussed for the P, Q pair.

Since we want to operate in the leftmost part of the A, B, C and D registers, we will again slightly modify the BNE algorithm such that instead of left-shifting the contents of the B and D registers when normalizing Q, we will rightshift the A and C registers (with sign extension), and also rightshift the I and J registers to adjust for the change of the position of unity. Similarly when the P register is normalized, the B and D registers are rightshifted (but not I and J).

When the unit is seeded with an operand (and operator) the A and C registers are loaded left adjusted, the position of unity determined by the operand containing most bits, i.e. in general one of the operands will have to be shifted to be aligned. During the execution of the algorithm, when adding A, B pairs of same sign (and similarly with C, D pairs), the growth of the A (and C) register contents can at most require one extra bit position. The "normalization" thus has to provide one extra safeguard position, which means that the scaling just uses the three least significant bits of the four, but uses the same decision rules as described in section 4. Only once during the outer loop is it necessary to check the leading three positions (of the four), and perform a rightshift if they fall in one of the cases 1 through 5.

The I and J registers are checked whenever right-shifted to detect whether growth has moved the unit position beyond the rightmost end of either the A, B or the C, D pair. This can only occur outside the inner loop, and, based on the contents of all registers, it is then possible to find or possibly recover (cfr. Theorem 2) a (u, v) pair corresponding to a canonical convergent, which then can be transferred to the P, Q registers (the "accumulator").

If the contents of the P, Q registers are to be rounded by the mediant rounding for storing in the floating slash format, then the BNE algorithm has to be stopped by a different criteria. To estimate the total number of bits needed to represent the numerator (u) and denominator (v), the I and J registers are not convenient. However it is possible to keep track of an approximate value of the number of bits needed in an extra shift register (call it S), accumulating the total number of shifts in the I and J registers. The initialization of S is simple since in this case the seed is the identity matrix.

Unfortunately the S register cannot give the precise amount of bits needed, due to the slack in the normalization. But during packing of the external representation, an adder with carry-look-ahead is needed anyway, for the conversion from carry-save to 2's complement (and sign-magnitude), and can be used to get the exact amount of bits, in connection with the recovery of a canonical convergent.

6. Conclusions and future work

An arithmetic unit, which can perform all the standard arithmetic operations, has been described. The unit seems will suited as an "add-on" unit for a microprocessor, providing powerful arithmetic capabilities, which could easily be implemented in high-scale integration.

Of particular concern have been considerations concerning the speed of the unit. In particular it has been shown that the BNE algorithm can be implemented using carry-save representation, speeding up addition by avoiding the carry ripple of standard adders, at the expense of slowing down the normalization shifts. Whether this trade off is beneficial depends on the ratio of number of add/subtracts to shifts, in the BNE and other possible algorithms (e.g. the SS algorithm in [4]), and deserves further analysis.

Also the possibilities of supporting multiple precision operations needs further study. In [7] it has been proposed to use continued fractions, in the form of quotient sequences, as a number representation for "demand driven" expression evaluation. The scheme seems promising, but suffers the same problems with very large quotient values. Although rarely occurring, the large quotient values are crucial to the underlying number system. Using an extended floating-slash representation, where the "slash-position" is allowed to move beyond the field assigned for the numerator and denominator, it seems likely that large quotients can be handled, besides providing extra range for "single-precision" operands.

References

[1] D. W. Matula: "Fixed-Slash and floating-slash rational arithmetic". Proc. of the 3rd IEEE Symposium on Computer Arithmetic, Dallas, 1975.

[2] D. W. Matula & P. Kornerup: "A feasibility study of fixed-slash and floating-slash number systems". Proc. of the 4th IEEE Symposium on Computer Arithmetic, Santa Monica, 1978.

[3] D. W. Matula & P. Kornerup: "Foundations of finite precision arithmetic". Computing, Suppl. 2, 1980.

[4] P. Kornerup & D. W. Matula: "A feasibility study of fixed-slash arithmetic". Proc. of the 4th IEEE Symposium on Computer Arithmetic, Santa Monica, 1978.

[5] G.H. Hardy & E.M. Wright: "An introduction to the theory of numbers". 4th ed. Clarendon Press, Oxford, 1959.

[6] A.Y. Khintchin: "Continued fractions", translated from Russian by P. Wynn, P. Noordhoff Ltd., Grooningen, 1969.

[7] W. Gosper: "Continued fraction arithmetic". Unpublished manuscript.