Applications of Redundant Number Representations to Decimal Arithmetic

R. Sacks-Davis
Department of Computer Science, Monash University, Clayton, Victoria, 3168, Australia

A decimal arithmetic unit is proposed for both integer and floating-point computations. To achieve comparable speed to a binary arithmetic unit, the decimal unit is based on a redundant number representation. With this representation no loss of compactness is made relative to binary coded decimal (BCD) form. In this paper the hardware required for the implementation of the basic operations of addition, subtraction, multiplication and division are described and the properties of floating-point arithmetic based on a redundant number representation are investigated.

1. INTRODUCTION

When representing numbers, positional notation using radix r is defined by the rule

$$(\ldots a_{-2}a_{-1}a_0 \cdot a_1a_2a_3 \ldots)_r = \cdots + a_{-2}r^2 + a_{-1}r + a_0 + a_1r^{-1} + a_2r^{-2} + \cdots$$

and conventional number systems are obtained when r is a positive integer and the a's are restricted to be integers in the range $0 \leq a_j < r$. Thus the a's are allowed to assume r different values.

In a series of papers, Avizienis described a class of number systems called signed-digit representations.[1-3] With these representations the a's are allowed to assume more than r values and both positive and negative digit values are allowed for this purpose. (A signed-digit number representation is sometimes referred to as a redundant representation.) For example, if negative values are identified by a bar over the digit, then the decimal ($r = 10$) signed-digit number $3\bar{6}4.\bar{5}$ has the algebraic value $3 \times 10^2 - 6 \times 10^1 + 4 \times 10^0 - 5 \times 10^{-1} = 243.5$.

Let A be a signed-digit number represented by the $n + m + 1$ digits a_i, $i = -n, \ldots, -1, 0, 1, \ldots, m$ with the algebraic value

$$A = \sum_{i=-n}^{m} a_i r^{-i} \qquad (1)$$

Then some of the properties of signed-digit number systems are the following.

(i) The sign of the algebraic value A is indicated by the sign of the most significant non-zero digit.
(ii) The algebraic value of A is zero if, and only if, each of the digits of its signed-digit representation has the value 0.
(iii) Given a signed-digit representation of the algebraic value A, the signed-digit representation of $-A$ is formed by changing the sign of each of the non-zero digits a_i.

However, the property of a signed-digit number system that makes it so attractive to designers of arithmetic units is the following:

(iv) Each of the digits of two signed-digit numbers may be added or subtracted in a parallel fashion so that the time required for addition or subtraction is independent of the length of the arguments.

Redundant number representations are used implicitly in many algorithms for which the final result appears in a conventional non-redundant form. Examples of such algorithms are carry-save schemes for multiplication and SRT division algorithms. A survey of the role of redundancy in computer arithmetic is given in Ref. 4.

In this paper we consider using a signed-digit representation to implement decimal arithmetic. Because base 10 is the preferred radix for machine/human interface, the use of decimal arithmetic has been advocated by a number of people.[5-7] We will consider a representation of decimal numbers for which the allowed values of the digits a_i are $-6, -5, \ldots, -1, 0, 1, \ldots, 5, 6$. In the notation of Avizienis, this range of 13 permissible values is called the minimally redundant range. Since with this choice, only 4 bits are required to store 13 values, no loss of compactness of representation is made relative to BCD representation. However, this choice does preclude Chen's 10 bit encoding of 3 BCD digits which gives almost the same density of decimal and binary numbers.[8] It will be seen that conversion from BCD notation to the redundant representation is easily achieved and, of course, no errors are introduced when converting between representations.

Based on signed-digit representation, the basic operations of addition, subtraction, multiplication and division can be performed very efficiently. Implementations of these basic operations are described in the following section. Multiplication is performed in a similar fashion to carry-save multiplication so that the time required is $O(n)$ for n digit arguments. An iterative process is proposed for division and with this technique, the time required for division is approximately $2\frac{1}{2}$ times the multiplication time. The hardware used to implement these algorithms is very simple being based on bipolar read-only-memories (ROMs) and conventional MSI 4-bit adders.

One possibility for implementing decimal arithmetic is to use a conventional representation together with a fast carry-look-ahead adder such as the one proposed in Ref. 7. Signed-digit representation can then be used for the implementation of multiplication and division algorithms. Alternatively, the internal representation of numbers and all the basic operations can be based on a

CCC-0010-4620/82/0025-0471 $03.50

redundant number system. However, the properties of floating-point arithmetic based on a signed-digit representation are different to those resulting from a conventional representation. Some of these properties are described in Section 3. In particular, for floating-point arithmetic based on a signed-digit representation, a round-off process that is based on truncation yields errors that are unbiased and small in magnitude. Finally, in Section 4 we show that signed-digit arithmetic can also facilitate algorithms for multiprecision arithmetic.

2. IMPLEMENTATION OF BASIC OPERATIONS

When adding signed-digit numbers, the carries are propagated just one position. The propagation of carries over the whole number associated with the addition of numbers using standard arithmetic is therefore eliminated by using a redundant representation. Given two signed-digit numbers, X and Y,

$$X = \sum_{i=-n}^{m} x_i 10^{-i}, \qquad Y = \sum_{i=-n}^{m} y_i 10^{-i}$$

addition is performed in two stages. Firstly, from each of the terms $x_i + y_i$, $i = -n, \ldots, m$, a transfer (or carry) digit t_{i-1} and an interim sum digit w_i are formed satisfying

$$x_i + y_i = 10t_{i-1} + w_i$$

The values for t_{i-1} and w_i are given in Table 1.

In the second stage the digits, s_i, of the sum are formed:

$$s_i = w_i + t_i$$

It may be observed from the addition table that $|w_i| \leq 5$ and $|t_i| \leq 1$ so that s_i will be formed without any further carry propagation.

The formation of each of the pairs of digits t_{i-1} and w_i in the first stage and the formation of each of the sum digits s_i in the second stage may be performed in a totally parallel fashion. Thus the addition time in signed-digit arithmetic is a constant (independent of the number of digits in the arguments) number of gate delays. An example of addition is given in Fig. 1 and a block diagram of a totally parallel adder is given in Fig. 2.

In order to implement Fig. 2, some 4-bit encoding for the signed-digits $-6, -5, \ldots, 6$ must be chosen. In the following we will assume that a 1's complement representation is used. As discussed earlier, in order to complement a signed-digit number, A, it is necessary to complement each of the digits of A in parallel. This will simply involve inverting each of the bits of A if a 1's complement representation for the signed-digits is chosen.

With this representation for the signed-digits, a detailed section of a parallel add/subtract unit is given in Fig. 3. The unit consists of identical 256 × 7 ROMs for generating the interim sum and transfer digits and 4-bit binary adders for generating the sum digits. Note that an extra output from the ROMs is used to predict the end-around carry required in 1's complement addition.

x_i: 2 4 $\bar{3}$ 5 $\bar{1}$ (Algebraic value 23749)
y_i: 1 1 $\bar{6}$ 1 6 (Algebraic value 10416)
w_i: 3 5 1 $\bar{4}$ 5
t_i: 0 $\bar{1}$ 1 0 0
s_i: 3 4 2 $\bar{4}$ 5 (Algebraic value 34165)

Figure 1. An example of addition using signed-digit arithmetic.

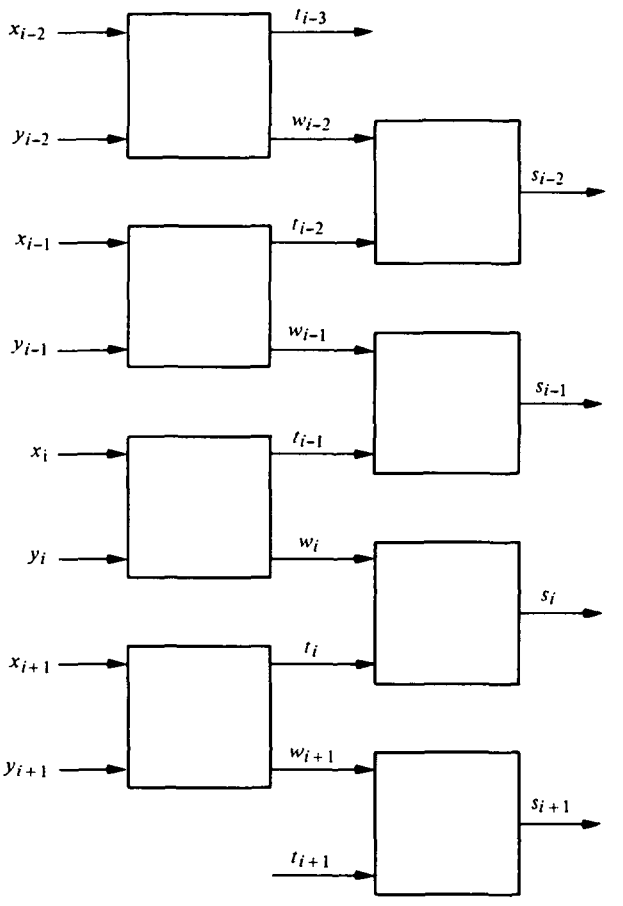

Figure 2. Section of a totally parallel adder.

To give an indication of the time required to perform addition we list below typical speeds of the components used in the adder. Schottky TTL circuits will be assumed for the 4-bit adders and AND gates and Bipolar ROMs are assumed for the first stage. The following times are taken from a Signetics data manual:[9]

Component	Typical time	Maximum time
82S114(PROM)	35 ns	60 ns
74LS283(4-BIT ADDER)	16 ns	24 ns
74SO7(AND gate)	4.5 ns	7 ns

Table 1. Addition Table

$x_i + y_i$	−12	−11	−10	−9	−8	−7	−6	−5	−4	−3	−2	−1	0	1	2	3	4	5	6	7	8	9	10	11	12
t_{i-1}	−1	−1	−1	−1	−1	−1	−1	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1
w_i	−2	−1	0	1	2	3	4	−5	−5	−3	−2	−1	0	1	2	3	4	5	−4	−3	−2	−1	0	1	2

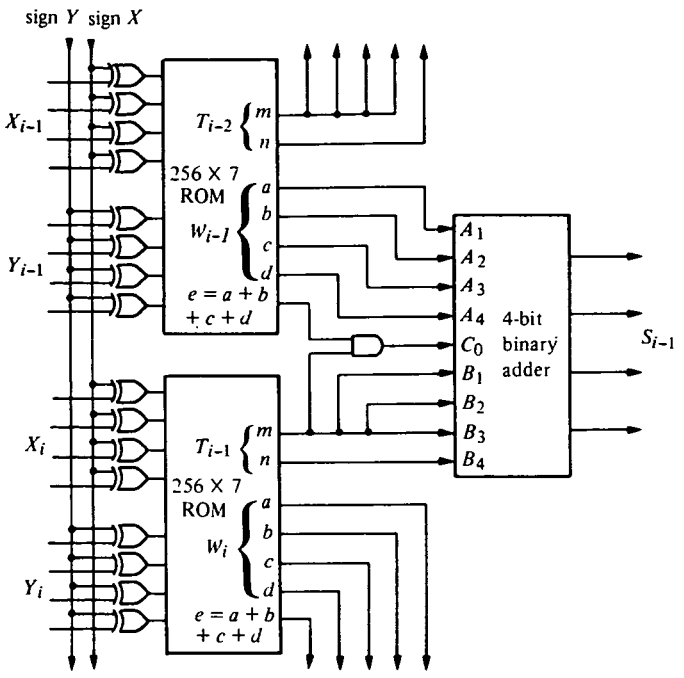

Figure 3. Detailed section of a parallel add/subtract unit.

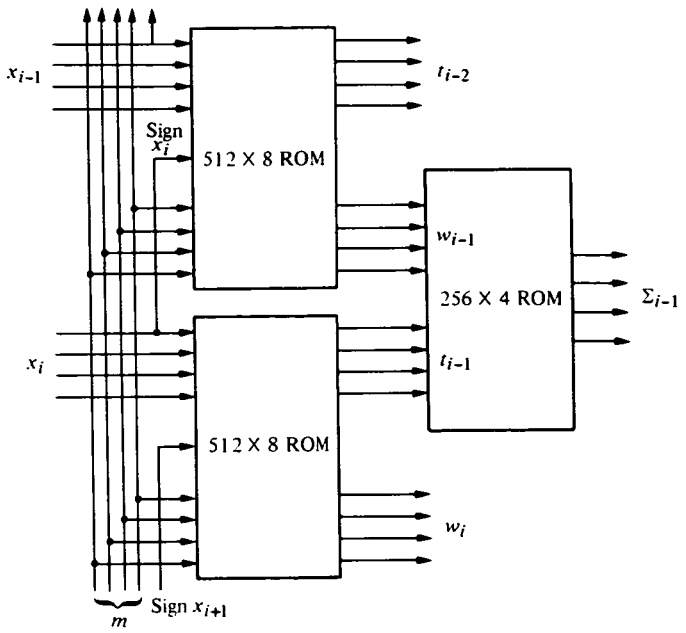

Figure 4. Forming mX, $-6 \leq m \leq 6$.

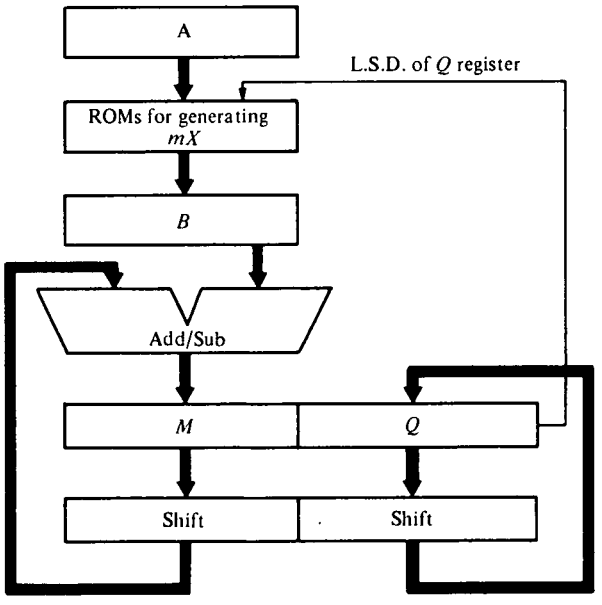

Figure 5. Multiply unit.

It can be seen that decimal addition can be achieved in under 60 ns and this addition time is independent of the length of the adders.

Multiplication is achieved by a sequence of additions or subtractions and right shifts. Given a multiplicand, X, and a multiplier Y, the formation of XY proceeds as follows. At each stage of the multiplication, an integer multiple, y_i, $-6 \leq y_i \leq 6$ of the multiplicand is added to the current partial sum and at the same time the next multiple, Xy_{i-1} is formed. The redundancy of representation permits the formation of an integer multiple of X by a parallel process similar to that used for addition.

Let successive digits of X be $\ldots x_{i-1}x_ix_{i+1}\ldots$ and suppose it is required to form mX, $-6 \leq m \leq 6$. For each digit, x_i, of X a transfer digit, t_{i-1}, and an interim sum digit, w_i, satisfying

$$mx_i = 10t_{i-1} + w_i \qquad (2)$$

are formed. However, in this case, the transfer digits may have modulus greater than one, so extra care must be taken to ensure that the condition $w_i + t_i \leq 6$ is not violated during the second stage. Now, the sign of t_i is determined by the signs of m and x_{i+1}. Hence if the sign of x_{i+1} is inspected when recoding x_i according to Eqn (2) then in those cases for which the redundancy of representation allows a choice of values for w_i and t_{i-1}, w_i may be chosen to have the opposite sign to t_i. The cases for which there is no choice of representation in w_i are those for which $|w_i| \leq 3$. Since $|t_i| \leq 3$, the condition $w_i + t_i \leq 6$ may be achieved in all cases. A configuration similar to Fig. 2 is then used to implement the formation of mX. One possible implementation is given in Fig. 4 where both stages are implemented using ROMs.

A diagram of a multiply unit is given in Fig. 5. Initially, A holds the multiplicand, M is set to zero and Q holds the multiplier. The double length product is formed in the MQ registers. To achieve a multiplication time of approximately n times the addition time, the formation of integer multiples of the multiplicand proceeds in parallel to the additions to the partial sums.

Signed-digit division may be performed as a sequence of additions or subtractions and left shifts based on an algorithm due to Robertson.[10] The redundancy of representation allows an inexact selection of quotient digits and at each stage only the leading digits of the jth partial remainder need be inspected to determine the correct quotient digit. However, the resulting circuitry is still relatively complex for the decimal arithmetic that we are considering and we describe a division algorithm based on a quadratically convergent iterative process.[11,12] The iterative process is based on repeated multiplication and requires no extra hardware beyond a ROM look-up which is used for starting the iteration.

Given a numerator, N, and a denominator, D, the quotient $Q = N/D$ is obtained by multiplying both N and D by the same factor R_k so the resultant denominator converges quadratically towards one and the resultant numerator converges towards Q. Thus if $N_0 = N$ and $D_0 = D$ we form

$$N_{k+1} = N_k R_k, \qquad D_{k+1} = D_k R_k, \qquad k = 0, 1, \ldots$$

so that $D_k \to 1$ and $N_k \to Q$. Writing

$$D_k = 1 - x_k$$

where $|x_k| < \varepsilon$, the multiplier R_k is given by

$$R_k = 1 + x_k$$

so that $D_{k+1} = 1 - x_k^2 = 1 - x_{k+1}$ where $|x_{k+1}| \leq \varepsilon^2$. Given D_k is of the form

$$D_k = 1.00 \ldots 0 d_m d_{m+1} \ldots d_M$$

the multiplier R_k is

$$R_k = 1.00 \ldots 0 \bar{d}_m \bar{d}_{m+1} \ldots \bar{d}_M$$

Unlike implementations based on conventional number representations, no explicit calculation of R_k from D_k is required. All that is needed is that D_k be copied into the Q register of the multiply unit prior to the next iteration and that the function control lines to the add/subtract unit be set during each stage of the multiplication. If $D_k = 1.00 \ldots 0 d_m d_{m+1} \ldots d_M$ then for $i = M, M-1 \ldots m$, the multiple of the multiplicand stored in the B-register will be subtracted from the partial sum.

Of course, if the number of digits in the multiplier could be reduced, the time for each multiply would be reduced. For the multiplication algorithm proposed in this paper, the time required is directly proportional to the number of digits in the multiplier and is independent of the number of digits in the multiplicand. Thus for the division process, it makes sense to use reduced length multipliers when this is possible. Now consider a typical stage of the division process. We have

$$D_k = 1 - x_k$$

so

$$R_k = 1 + x_k$$

Rather than use R_k as the multiplier let us consider using a reduced length multiplier, $\bar{R}_k$, for the next step. The quotient is not affected provided we multiply *both* the numerator and denominator by the same multiplier and the time for the multiplications will be reduced.

Let

$$\bar{R}_k = R_k + \eta_k$$

Then

$$D_k \bar{R}_k = 1 - x_k^2 + (1 - x_k)\eta_k$$

If $|(1 - x_k)\eta_k|$ is of the same order of magnitude as $|x_k^2|$ then using $\bar{R}_k$ rather than R_k will hardly affect the rate of convergence. For example, suppose that $x_k \sim 10^{-2}$, then $x_k^2 \sim 10^{-4}$. If $R_k = 1.00\, \bar{d}_3 \bar{d}_4 \ldots \bar{d}_m$ then we might use $\bar{R}_k = 1.00\, \bar{d}_3 \bar{d}_4$ as the multiplier for the next step without very much affecting the rate of convergence. For division based on this approach, the number of digits in the multipliers approximately doubles with each iteration. Further increase in the speed of each multiplication is achieved by skipping over the string of zeros in each multiplier.

As a final remark, it should be noted that since the division process consists of a sequence of multiplications, the intermediate terms $N_k \bar{R}_k$ and $D_k \bar{R}_k$ must be formed to a slightly greater precision than the final result in order that the accumulated 'roundoff' errors of these intermediate calculations do not affect the final result.

We will now briefly describe how numbers represented in BCD notation may be converted to signed-digit representation. The recoding of the BCD digits is done by a similar parallel process to those described previously. If $U = \sum_{i=-n}^{m} u_i 10^{-i}$, where $0 \leq u_i \leq 9$, then we may determine w_i and t_i satisfying

$$u_i = w_i + 10 t_{i-1}$$

where $|w_i| \leq 5$ and $|t_{i-1}| \leq 1$. The w_i and t_i are added in parallel in the second stage of the conversion process.

This parallel process may be implemented using the same hardware used for the addition process; no extra hardware is required. To see this, note that for addition the inputs x_i and y_i of Fig. 2 are restricted to the range, -6 to 6. Since we are using a 4-bit 1's complement representation of the digit values -6 to 6, two values, ± 7, are not used by the hardware for addition. Thus to implement conversion from BCD to signed-digit representation we might input the digits u_i as one of the arguments to the parallel adder (say x_i) and input $+7$ or -7 to the other input (y_i). With these inputs the ROMs would be configured for conversion rather than addition.

Conversion from signed-digit representation to BCD representation may be achieved by the serial process described in Fig. 6. Here a signed-digit number, X, is converted to BCD form using a simple carry-propagate converter. Another alternative would be to implement conversion to BCD form using a conventional BCD adder.

A characterization of an arithmetic unit for performing both single and double precision arithmetic is given in Fig. 7. It consists simply of two multiply units as

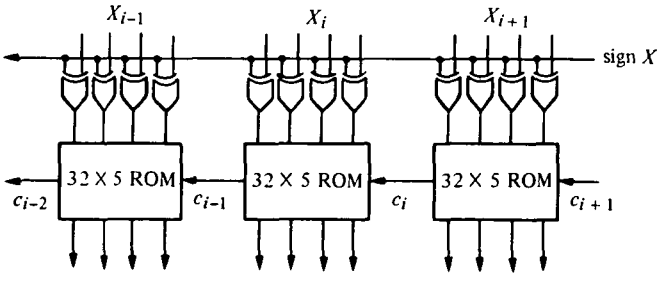

Figure 6. Serial signed-digit to BCD conversion.

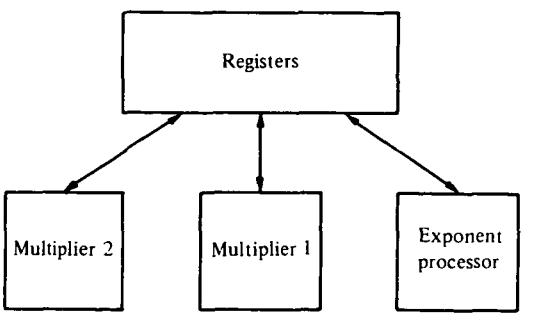

Figure 7. Arithmetic unit.

described in Fig. 5 together with a register set and exponent processor for floating-point arithmetic.

Single precision addition or subtraction (integer or floating-point) can be achieved using either of the adders in the two multipliers. In order to perform double-precision addition the overflow digit from the adder in Multiplier 1 is fed via a multiplexor to the least significant transfer digit position of Multiplier 2. The time required for double precision addition or subtraction will be the same as the single precision time.

Single precision multiplication (integer or floating-point) can be performed using either multiply unit. Alternatively, a fast scheme for multiplication consists of splitting the multiplier digits between the two multiply units and performing a double precision addition at the end. With this technique, sometimes referred to as split-multiplication,[13] the time required for multiplication is just over $n/2$ times the addition time. In order to perform double precision multiplication, the shift paths illustrated in Fig. 8 must be enabled. The multiplier is initially loaded into $Q_2 - Q_1$ and the quadruple precision product is formed in $M_2 - M_1 - Q_2 - Q_1$. In order to achieve the correct transfer-digit paths during the formation of the multiples of the multiplicand and during the addition to the partial sums, a few extra multiplexors are required to control the inputs to some of the ROM's in Figs 3 and 4.

The basic step in the division process is the multiplication of the numerator and denominator by the same factor. Since the arithmetic unit contains two multiply units these multiplications may proceed in parallel for single precision division. Double precision division is based on sequences of double precision multiplications. It should be noted that since division is based on a sequence of multiplications, intermediate results must be computed to a slightly higher accuracy so that accumulated round-off error does not affect the final result. If the length of the internal registers of the arithmetic unit correspond to the length of single-precision integers and therefore exceed the length of the mantissa or fraction parts of floating-point numbers, then the iterative division algorithm will be suitable for floating-point calculations but not for integer division. Single precision integer division may, however, be based on the double-precision floating-point algorithm.

3. FLOATING-POINT ARITHMETIC

In this section we consider floating-point arithmetic based on a redundant number representation. We will consider floating-point numbers of the form $f \times 10^e$ where $f = \sum_{j=1}^{t} a_j 10^{-j}$ is a t-digit mantissa normalized so that $a_1 \neq 0$. This normalization criterion is the simplest to implement in hardware. However, because the allowed digit values, a_i, are $-6 \ldots 6$ the permissible range of mantissas differs from the range $[\frac{1}{10}, 1)$ obtained from a conventional representation of decimal numbers. We have

$$0.1\bar{6}\bar{6}\ldots\bar{6} \leq |f| \leq 0.66\ldots6$$

or

$$m \leq |f| \leq M \tag{3}$$

where

$$M = \tfrac{2}{3} - \tfrac{2}{3}10^{-t}$$

and

$$m = \tfrac{1}{30} + \tfrac{2}{30}10^{1-t}$$

Avizienis[1] considers more complicated normalization criteria for which the resultant range of mantissas is close to $[\frac{1}{10}, 1]$.

It was noted that with a signed-digit representation, one algebraic number may be represented in more than one way. Thus a value, f, such as 0.5 may be represented as $1.\bar{5}$. Although the algebraic value of f lies within the permissible range (3), the latter representation of f will cause an overflow indication if f represents the mantissa of a floating-point number. The range of values for which this may occur is called the 'potential overflow' range by Avizienis[1] and based on the normalization criteria above is

$$M - \varepsilon \leq |f| \leq M$$

where

$$\varepsilon = \tfrac{1}{3} - \tfrac{1}{3}10^{-t}$$

The most attractive feature of floating-point arithmetic based on a signed-digit representation is that an accurate round-off process is obtained by truncation. Since the allowed digit values are symmetric around zero, the average error introduced by truncation is zero and the round-off is without bias. If x is a floating-point number with a $t + l$ digit mantissa, and if we denote by $fl(x)$ the representation of x based on truncating x to t digits then we have

$$\frac{|x - fl(x)|}{|x|} \leq \frac{\frac{2}{3}10^{-t}}{\frac{1}{30}} = 2 \times 10^{1-t}$$

We note that this bound on the relative error is over-pessimistic by a factor of 10 or more when the mantissa of x lies in the potential overflow range.

In order to obtain a statistical measure of the accuracy of the round-off scheme based on truncation, it is necessary to know the distribution of the trailing digits of floating-point mantissas. Since the fundamental operation performed by the processor is the addition algorithm of Section 2, an indication of the distribution of the

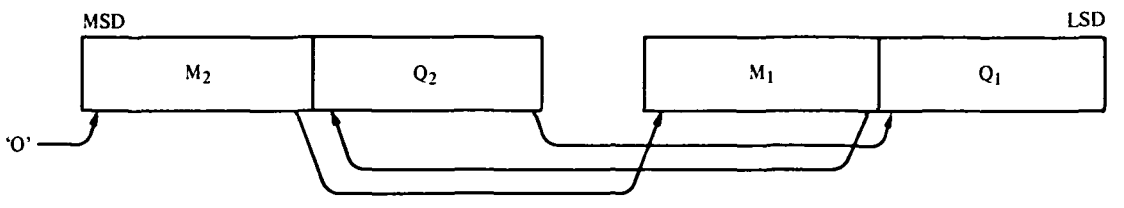

Figure 8. Double precision right-shift used in multiplication.

trailing digits may be obtained by determining the distribution of digits that is invariant under the addition algorithm. That is, if $p(i)$ is the probability that a digit is equal to i, $-6 \leq i \leq 6$, and the corresponding probability distribution after addition (assuming no correlation between the digits) is $F(p(i))$, then $p(i) = F(p(i))$. The values for $p(i)$ are given in Table 2.

Table 2. Probability distribution that is invariant under the add/subtract algorithm of Section 2

i	$p(i)$
±6	5.2296×10^{-3}
±5	5.0000×10^{-2}
±4	9.4770×10^{-2}
±3	1.0000×10^{-1}
±2	1.0000×10^{-1}
±1	1.0000×10^{-1}
0	1.0000×10^{-1}

The distribution of digit values given in Table 2 indicates that rounding by truncation is likely to be much more accurate than a worst-case analysis suggests. Let

$$f_{t+l} = \sum_{j=1}^{t+l} a_j 10^{-j}$$

denote a $t+l$ digit fraction and let e_{t+l} denote the absolute error incurred by truncating f_{t+l} to t digits. Then we may define

$$E_l = \{\text{expected value } (e_{t+l}^2)\}^{1/2}$$

Values for E_l based on the distribution of signed digits given in Table 2 appear in Table 3. For a comparison we also give the corresponding values obtained from exact rounding on a decimal machine based on a conventional (non-redundant) representation and for which all digit values (0–9) are equally likely.

Table 3. Comparison of RMS values

	E_l	
l	Rounding by truncation	Exact rounding
1	0.1588×10^{-t}	0.1581×10^{-t}
≥2	0.2966×10^{-t}	0.2887×10^{-t}

Other rounding schemes are more difficult to implement. We will illustrate some of the problems involved by considering the implementation of what we call S-rounding. The implementation of rounding schemes such as directed rounding, used for interval arithmetic, will pose similar problems.

Suppose it is required to reduce a $t+l$ digit mantissa, f_{t+l}, to t digits. We may express f_{t+l} as

$$f_{t+l} = g_t + r_l 10^{-t}$$

where $g_t = \sum_{j=1}^{t} a_j 10^{-j}$ and $r_l = \sum_{j=t+1}^{l} a_j 10^{t-j}$. Then $-M \leq g_t \leq M$ where $M = \frac{2}{3} - \frac{2}{3}10^{-t}$. The action to be taken on S-rounding will depend on the magnitude of r_l.

Define

$$\text{S-round}\,(f_{t+l}) = \begin{cases} g_t + 10^{-t} & \text{if } r_l > 0.5, \quad g_t < M \\ g_t & \text{if } r_l > 0.5, \quad g_t = M \\ g_t & \text{if } 0.5 \geq r_l \geq -0.5 \\ g_t & \text{if } -0.5 > r_l, \quad g_t = -M \\ g_t - 10^{-t} & \text{if } -0.5 > r_l, \quad g_t > -M \end{cases}$$

Thus conventional rounding occurs in all cases except for when

(i) $g_t = M$ and $r_l > 0.5$

or

(ii) $g_t = -M$ and $r_l < -0.5$.

Note that when

$$g_t = M = 0.\underbrace{66\ldots6}_{t \text{ digits}} \quad \text{and} \quad r_l > 0.5$$

conventional rounding would dictate that g_t be rounded up to

$$\underbrace{0.66\ldots67}_{t \text{ digits}}$$

But this is not representable as a t-digit signed-digit fraction and the most accurate rounding is achieved by truncation in these cases.

To implement S-rounding we need to determine whether $r_l \in [-0.5, 0.5]$. Now $r_l = \sum_{j=t+1}^{l} a_j 10^{t-j}$. If $a_{t+1} = 6$ then $r_l > 0.5$. Similarly, if $a_{t+1} = -6$ then $r_l < -0.5$. If $-4 \leq a_{t+1} \leq 4$ then $-0.5 < r_l < 0.5$. The only cases which cause some difficulty are those for which $a_{t+1} = 5$. Then it is necessary to determine the sign of $\hat{r}_l = \sum_{j=t+2}^{l} a_j 10^{t-j}$.

When performing floating-point addition or subtraction this sign may be determined by detecting the sign of the last non-zero digit which is shifted past the $(t+1)$st digit-position during the pre-alignment shift. This is akin to the incorporation of a sticky-bit for the implementation of exact rounding on a conventional machine. A similar technique can be applied when performing multiplication. However, for floating-point division the situation is more difficult and a multiplication of the quotient by the denominator may be required to implement S-rounding.

4. SOME FURTHER EXAMPLES

There are a number of properties of redundant number representations which may be usefully exploited in applications such as multiprecision arithmetic. We will illustrate these properties with a couple of examples.

With 2's complement representation for binary arithmetic the algebraic value associated with the n bit integer, $a_{n-1}a_{n-2}\ldots a_0$, is $-a_{n-1}r^{n-1} + \sum_{j=0}^{n-2} a_i r^i$ where r is the radix. Thus only the leading digit, a_{n-1}, has negative weight. This leads to the problem of sign-extension when right-shifting numbers. A similar problem occurs with the choice of 9's or 10's complement arithmetic for decimal arithmetic. This problem does not arise with signed-digit arithmetic and this is useful when implementing multiplication algorithms where an array of numbers must be added to form the result. With signed-digit representation no account of the signs of the numbers forming the array need be taken. However for

2's or 10's complement representations, sign extension must be considered and the final array to be added might be represented as in Fig. 10 rather than in Fig. 9. (Gosling shows how sign-extension for 2's complement arithmetic may be implemented efficiently.[13])

As another example of how signed-digit representation can be utilized, consider the problem of adding two multiprecision floating-point numbers, u and v. With conventional number representations, the normalization of the result, w, must follow the addition of the mantissas.

```
        X  X  X  X
     X  X  X  X
  X  X  X  X
X  X  X  X
```

Figure 9. Array of numbers to be added in multiplication.

```
X  X  X  X  X  X  X
X  X  X  X  X  X
X  X  X  X  X
X  X  X  X
```

Figure 10. Array when the numbers are negative and sign extension is required.

However, with a signed-digit representation, these two processes can proceed in parallel. Because there are no carry propagation chains in addition, it is possible to add the mantissas of u and v from left to right (i.e. most significant part to least significant part). Thus the most significant digits of w are calculated first and the normalization of w can proceed in parallel with the addition of the mantissas. Early detection of overflow is possible and the same algorithm is directly applicable to subtraction.

REFERENCES

1. A. Avizienis, Signed-digit number representations for fast parallel arithmetic. *IRE Trans. Electronic Computers* **10**, 389–400 (1961).
2. A. Avizienis, On a flexible implementation of digital computer arithmetic, in *Information Processing*, ed by C. M. Popplewell, pp. 664–670. North-Holland, Amsterdam (1963).
3. A. Avizienis, Binary-compatible signed-digit arithmetic. *Proceedings of Fall Joint Computer Conference, 1964*, pp. 663–672 (1964).
4. D. E. Atkins, Introduction to the role of redundancy in computer arithmetic. *Computer* **8**, 74–77 (1975).
5. T. E. Hull, Desirable floating-point arithmetic and elementary functions for numerical computation. *ACM Signum Newsletter* **14** (No. 1), 96–99 (1979).
6. G. J. Myers, *Advances in Computer Architecture.* Interscience, New York (1978).
7. M. S. Schmookler and A. Weinberger, High speed decimal addition. *IEEE Trans. Computers* **20**, 862–866 (1971).
8. T. C. Chen and I. T. Ho, Storage-efficient representation of decimal data. *Communications of the ACM* **18**, 49–51 (1975).
9. Signetics Data Manual (1976).
10. J. E. Robertson, A New Class of Digital Division Methods. *IRE Trans. Electronic Computers* **7**, 218–222 (1958).
11. S. F. Anderson, J. G. Earle, R. E. Goldschmidt and D. M. Powers, The IBM system/360 Model 91: floating-point execution unit. *IBM Journal of Research and Development* **11**, 34–53 (1967).
12. C. S. Wallace, A suggestion for a fast multiplier. *IEEE Trans. Computers* **13**, 14–17 (1964).
13. J. B. Gosling, *Design of Arithmetic Units for Digital Computers*, Macmillan, London (1980).

Received January 1982