MS Project Final Report

Signed Digit Representation of Numbers

Submitted by: Savithri Venkatachalapathy

(Computer Science Department, Oregon State University)

Major Professor: Dr. Bella Bose

Minor Professor: Dr. Toshimi Minoura

Committee Member: Dr. Tim Budd

Table of Contents

1 Introduction

In conventional digital computers, integers are represented as binary numbers of fixed length. There are several other number systems that are useful for certain applications. These include the Redundant Signed Digit Number System. In a redundant signed digit representation with radix r each digit is allowed to take more than r-values. Redundancy in representation makes faster addition and subtraction in which each sum or difference digit is a function of only the digits in the adjacent portions of the operands. In this project work we emphasize on the floating point representation of XLU numbers, which is based on signed digits. And various rounding schemes for them, namely truncate, nearest to zero, nearest to even, nearest to positive and negative infinity are discussed.

1.1 Conventional Number Representation

A non-redundant radix –r number has digits from the set {0, 1… r-1} and all the numbers can be represented in a unique way. The conventional number system like binary system or the commonly used decimal systems are non-redundant weighted and positional number systems. There is a sequence of weights associated, which determines the values of the number.

$$(X_{n-1}, X_{n-2} \ldots X_0) \text{ gives the value } X = \sum_{i=0}^{n-1} X_i W_i$$

where $W_{n-1}, W_{n-2}, \ldots W_0$ are the sequence of weights. The weights depends only on the position i of the digit X_i.

The sequence of n digits may represent a mixed number that has a fractional part as well as an integral part. This is done by partitioning the n digits into 2 sets: k digits in the integral part and m digits in the fractional part, satisfying: k + m = n.

$$\underbrace{(X_{k-1}, X_{k-2} \ldots X_1, X_0}_{\text{Integral Part}} . \underbrace{X_{-1}, X_{-2}, \ldots\ldots\ldots X_{-m})}_{\text{Fractional Part}}\ r$$

The value is: $X = X_{k-1}\, r^{k-1} + X_{k-2}\, r^{k-2} + \ldots X_1\, r + X_0 + X_{-1}\, r^{-1} + \ldots + X_{-m}\, r^{-m}$

$$= \sum\nolimits^{k-1}_{i=-m} X_i\ r^{i}$$

The radix point is understood to be in a fixed position between the k most significant digits and the m least significant digits. These representations are called Fixed-Point Representations as described in Koren [1].

In the number system seen above the digit set is restricted to {0, 1 … r-1}.

1.2 Redundant Number Representation

Adding some redundancy to a number system is very beneficial as will be seen later. This defines a new type of number system: Redundant number system or the Signed Digit Representation.

Each digit is either positive or negative and therefore there is no need for a separate sign digit.

A radix – r redundant signed digit number system is based on digit set:

$S = \{-\beta, -(\beta-1), \ldots -1, 0, 1, \ldots \alpha\}$ where $1 \le \beta, \alpha \le r - 1$.

The digit set S contains more than r values, implies multiple representations for any number in signed digit format. Hence, it is called the Redundant Number System.

Here is an example:

Consider radix r = 10. The allowed digits are $\{\bar{9}, \bar{8}, \ldots\ldots, \bar{1}, 0, 1 \ldots 8, 9\}$ and if n=2, the range is from $\bar{9}\bar{9}$ to 99 which includes 199 numbers. However with 2 digits each having 19 possibilities there are $19^2 = 361$ representations and hence some numbers have more than one representation. Therefore the number system is called Redundant. For example: (01) = ($1\bar{9}$) = 1, ($0\bar{2}$) = ($\bar{1}8$) = -2. The representation of Zero is unique and so is the representation of 10.

In a Signed Digit Representation with radix- r each digit can assume q values.

$$r+2 \leq q \leq 2r - 1$$

that is more than r values allowed in the conventional representation. Both positive and negative digit values are allowed for this purpose.

A Symmetric Signed Digit has $\alpha = \beta$. It uses the digit set D = {$-\alpha$, … -1, 0, 1, ….. , α } where r is the radix and α is the largest digit in the set. A number in this representation is written as:

$$X = X_{w-1}, X_{w-2} \ldots X_0 = \sum_{i=0}^{w-i} X_{w-i} * r^i$$

The sign of a number is given by the sign of the most significant non-zero digit. The following table shows the digit set and its redundancy factor, which determines the extent of redundancy that exists in the digit set.

Digit Set D	α	Redundancy Factor P
Incomplete	< (r-1) /2	< _
Complete, non-redundant	= (r-1) /2	= _
Redundant	$\geq \lceil r/2 \rceil$	> _
Minimally Redundant	$= \lfloor r/2 \rfloor$	> _ and < 1
Maximally Redundant	= r – 1	= 1
Over-Redundant	> r – 1	> 1

On the other hand, a high level of redundancy might be too costly since a larger digit requires a larger number of bits to represent each digit. The amount of redundancy might be reduced by restricting the digit set to:

$$X_i \in \{ \bar{a}\text{-}1, \bar{a}\text{-}2, \ldots \bar{1}, 0, 1, \ldots, a-1 \} \quad \text{with}$$

$$\lceil (r-1)/2 \rceil \leq a \leq r-1 \qquad (1)$$

where the ceiling of $\lceil X \rceil$ of a number is the smallest integer that is greater than or equal to X. At least r different digits are needed to represent a number in a radix r system and with $\bar{a} \leq X_i \leq a$ we have 2a + 1 digits. Therefore the inequality $2a + 1 \geq r$ must be satisfied and the lower bound from equation (1) follows as given by Koren [1].

Example: For r = 10, the range of a is $5 \leq a \leq 9$. If we select a = 6, the digit set = { $\bar{6}$, $\bar{5}$, $\bar{4}$, …., $\bar{1}$, 0, 1, …. 6 } then for n = 2, there are 133 numbers that are in the range $\bar{6}\bar{6} \leq x \leq 66$. With each digit taking any of 13 possible values, there are 169 representations and the redundancy is 27 %.

2 Redundant Signed Digit Numbers

Redundancy in the number system allows a method for fast addition and subtraction. Carry free addition is an attractive property of redundant signed digit number. The requirement for totally parallel addition and subtraction determines the minimum redundancy (r + 2 values) which is necessary in the representation of one digit. The upper limit for redundancy of digit values results from the requirement for a unique representation of the Zero algebraic value of a number. To satisfy this requirement the magnitude of allowed digit values may not exceed r – 1 as given by Avizeni [2].

Requirement of totally parallel addition and subtraction and of a unique representation for the Zero is satisfied by a class of Redundant Representation for radices r > 2, called the Signed Digit Representation. The digits of a Signed Digit Representation assume both positive and negative integer values and contain the sign information for the number. So no special sign digit is necessary. The number of digit values in a radix r > 2 representation ranges from required minimum of r+2 to an allowable maximum of 2r –1.

2.1 Properties of Redundant Signed Digit Numbers

Redundant Signed Digit representation is useful when developing algorithms for fast addition and subtraction of two numbers by eliminating carry propagation chains.
Consider the operation:
$(X_{n-1}, X_{n-2}, \ldots, X_0) \pm (Y_{n-1}, Y_{n-2}, \ldots, Y_0) = (S_{n-1}, S_{n-2}, \ldots, S_0)$
Here the carry chain is eliminated by having the sum digit S_i depends only on the four operand digits X_i, Yi, X_{i-1}, Y_{i-1}. This can be achieved using the signed digit; the time duration of the operation is independent of the length of the operands and is equal to the time required for the addition or subtraction of two digits.

The desired principal properties of Signed Digit Representations are specified by the following description of a Signed Digit Number given by Avizeni [2]:
A signed digit number is represented by n+m+1 digits Zi (i = -n, …, -1, 0, 1, …, m) and has the algebraic value:

$$Z = \sum_{i=-n}^{m} Z_i * r^i$$

where the values of r and Z_i are such that the following requirements are satisfied.

1) The radix r is a positive integer > 2.

2) The algebraic value Z = 0 has a unique representation.

3) There exists transformation between conventional representation and Signed Digit Representation.

4) Totally parallel addition and subtraction is possible for all digits in corresponding positions of two representations.

The arithmetic operation of totally parallel addition of two digits x_i and y_i from the corresponding i^{th} positions of the representation of numbers X and Y are defined as follows:

(i) The sum digit s_i (The digit of the sum $S = X + Y$) is a function only of the digits x_i and y_i and the transfer digit t_i from the $(i+1)^{th}$ position on the right.

$s_i = f(x_i, y_i, t_i)$

(ii) The transfer digit t_{i-1} to the $i-1^{th}$ position on the left is a function only of the digits x_i and y_i.

$$t_{i-1} = f(x_i, y_i).$$

The general algorithm for addition is quoted here from Koren [1] and described in two steps:

Step 1: Compute an interim sum w_i and transfer digit t_{i-1} to the next position on the left.

$$w_i = x_i + y_i - r\,t_{i-1}$$

Step 2: The sum digit s_i is calculated $s_i = w_i + t_i$.

The requirement for conversion from non-redundant to redundant representation is satisfied if the procedure of totally parallel addition applied to non-sign digit x_i:

$x_i = r\,t_{i-1} + w_i$

$s_i = w_i + t_i$

will yield an allowed value of digit s_i for every positive and negative value for the digit x_i.

The required and allowed values for t_i and w_i are established as:

$$t_i = -1, 0, 1 \qquad (2)$$

are the least sufficient set of values for the transfer digit t_i and the condition

$$|\,w_i\,| \le r-2 \qquad (3)$$

as the upper limit for the magnitude of the interim sum. The immediate result of (3) is the restriction of allowed values of radix r to the integer values $r > 2$ because for $r = 2$, the only allowed value of $w_i = 0$.

The conversion from traditional number system representation to signed Digit representation can be satisfied only when $w_i = -1, 0, 1$ are included for all $r > 2$. The relationship between the greatest value w_{max} and the smallest value w_{min} of w_i described by Avizeni [2] is:

$$w_{max} - w_{min} \geq r-1$$

where the equality sign applies only if r values of w_i are chosen. The required set of values of w_i therefore must consist of r integers which include –1, 0, and 1. More than one such set may exist but a symmetric sequence around zero is preferable in further development. The set of all allowed values of w_i is unique and consists of 2r – 3 integers from –(r – 2) to r – 2. The digit x_i assumes the minimum number of values when the required set of w_i is chosen as a sequence of r integers $\{w_{min}, ..., -1., 0, 1, \dots ,w_{max}\}$ in which additive inverses exist for all (when r is odd) and all but one (when r is even) non-zero values of w_i. In both cases the sum digit s_i assumes a value from the set:

$$\{w_{min} -1, w_{min}, \dots, -1, 0, 1, \dots, w_{max}, w_{max+1}\}$$

For odd radix ($r \geq 3$) $w_{max} = -w_{min}$ = _ (r -1), here an additive inverse exists for every value of w_i and s_i. The required set(minimum) of values for digits x_i or y_i consists of sequence of r + 2 integers: {-_ (r + 1), … -1, 0, 1, …, _(r + 1)}

Example: Consider r =5. The digit set is: {-3, -2, -1, 0, 1, 2, 3}. Here all the digits of s_i have additive inverse.

For an even radix $r \geq 4$ either $w_{min} -1$ or $w_{max} + 1$ in the set for the required values of s_i has no additive inverse. The required (minimum) set of values for the subtrahend digit y_i consists of the sequence of r + 2 integers {-(_ r + 1), .. –1, 0, 1, ..., (_ r + 1)}.

Example: Consider r =4. The digit set is: {-2, -1, 0, 1, 2, 3}. Here the digit 3 does not have additive inverse.

The minimum set of values for the sum digit s_i and the digit z_i requires only r + 2 integers and either _ (r + 1) or - _ (r + 1) is omitted to give this set.

The digit-level parallelism for addition operation provided by signed-digit representation is demonstrated by an example of addition of two 2-digit numbers.

Regular addition

Consider: X = 555, Y = 555 where X and Y are decimal numbers with 3 digits.
In normal addition the carry from the right most digit is propagated to its immediate left. And for every digit, carry from the previous position is added to generate the sum.

$$\begin{array}{lcccc} & & 5 & 5 & 5 \\ + & & 5 & 5 & 5 \\ \cline{2-5} \text{Carry:} & 1 & 1 & 1 & 0 \end{array}$$

Signed digit addition

With signed digit numbers the sum depends only on the digits at the current position and on the carry generated from the immediate position on the right. Fast addition is done in just two steps using the algorithm stated above. And an example is given below to illustrate this concept.

Consider: X = 154, Y = 103 where X and Y are signed digit numbers with 3 digits..
Radix r = 10, and the digit set is {-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6}

$$\begin{array}{cccll} 1 & 5 & 4 & & \\ 1 & 0 & 3 & & \\ \cline{1-3} 2 & 5 & \bar{3} & \leftarrow & \text{Interim sum} \\ 0 & 1 & 0 & \leftarrow & \text{Carry bits} \\ \cline{1-3} 2 & 6 & \bar{3} & \leftarrow & \text{Sum} \\ \cline{1-3} \end{array}$$

Here the interim sum and carry is calculated in parallel for all the digits. This is followed by the addition of carry digits from the previous position to the interim sum digits. Hence in just two steps we have performed the addition.
Also in this case, the interim sum did not exceed the provided range and we could perform fast addition in just 2 steps.

But there are situations where the digits in the interim sum exceed the range and it has to be converted into a different set of digits from the specified range.

As an example consider two numbers X = 5554 and Y = 1114. Let radix r = 10 and digit set is {-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5}

$$\begin{array}{ll} 5\ 5\ 5\ 4 & \\ 1\ 1\ 1\ 4 & \\ \hline 6\ 6\ 6\ \bar{2} & \leftarrow \text{Interim sum} \\ 0\ 0\ 1\ 0 & \leftarrow \text{Carry bits} \\ \hline \end{array}$$

Here the interim sum has digits 6 which are out of range. In this case it has to be converted into an alternative form to fit in the range.

Therefore $6\ 6\ 6\ \bar{2}$ is again represented as $1\ \bar{3}\ \bar{3}\ \bar{4}\ \bar{2}$. And in the subsequent step, sum is calculated using the new representation of interim sum.

$$\begin{array}{ll} 1\ \bar{3}\ \bar{3}\ \bar{4}\ \bar{2} & \leftarrow \text{Interim sum (converted)} \\ 0\ 0\ 0\ 1\ 0 & \leftarrow \text{Carry bits} \\ \hline 1\ \bar{3}\ \bar{3}\ \bar{3}\ \bar{2} & \leftarrow \text{Final Sum} \\ \hline \end{array}$$

Similarly subtraction can be performed on the same lines as described above.

3 XLU Representation

An alternative design to the Computer Arithmetic Architecture is based on the Redundant Number Representation as described by Avizeni [2]. This architecture is called the "XLU architecture" which is an interesting alternative to the traditional approach. The key point to the sign is a single unifying number representation.

The XLU architecture provides only one number representation and others necessary can be derived from it. The two representations fixed point and floating point are discussed. Then the rounding schemes, suitable for a XLU floating Point number is explained.

3.1 XLU number

1) It must carry enough information to allow it to serve as a basis upon which other number representations (integer, floating point, fixed point etc) may be efficiently realized by software or firmware using the XLU primitive operations.

2) Achieving Instruction-Level-Parallelism is one of the goals of this architecture. Therefore the base number Representation should be viable to manipulation promoting as much instruction –level- parallelism as possible.

3) The number representation must scale well, that is the performance of multiples of the XLU machine word must perform well relative to the performance of operations involving only the XLU machine word size.

A Signed Digit number modified to carry additional information for signaling overflow or special conditions such as positive / negative infinity fulfills the requirements mentioned above. Therefore the XLU number representation is based upon the signed digit representation discussed earlier.
So when we say an XLU digit, it is an entity having both magnitude and sign. The sign of an XLU number is determined by the highest non-zero digit.
A particular XLU digit is defined as having the same number of bits as a particular XLU processor implementation’s word size. For example: if the XLU architecture is implemented on a 32-bit processor with 32 bits per word, the XLU digit for this machine will have 32 bits.

The magnitude of the signed digit could be represented using 2’s complement form as an integral power of two is the default encoding for integers on most modern machines. Even though the magnitude portion of the XLU digit is represented in 2’s complement form an XLU digit with ‘m’ bits of magnitude has a range that is one less than a standard m-bit, 2's complement integer. The extra negative pattern, characteristic of two's complement representation is reserved as a special pattern by the XLU definition.

Using the XLU architecture the basic arithmetic operations can be performed as follows. The interesting characteristic of signed digit representation is that they eliminate carry propagation chains in addition and subtraction and this property shapes the way the XLU performs addition as well.

The general algorithm stated by Koren:

To perform the operation X + Y = S, for a signed digit number system of radix r, where X, Y, S consists of signed digit x_i, y_i, s_i:

$$X = (x_{n-1}, \ldots, x_0) \quad Y = (y_{n-1}, \ldots, y_0) \quad S = (s_{n-1}, \ldots, s_0)$$

The addition operation consists of two steps:

1) For each x_i and y_i compute an interim sum u_i and a carry digit c_{i-1} to the next position on left.

$$u_i = x_i + y_i - r\,c_{i-1}$$

where c_{i-1} is:

$$c_{i-1} = \{1 \text{ if } (x_i + y_i) >= a\}$$
$$c_{i-1} = \{0 \text{ if } |\, x_i + y_i \,| < a\}$$
$$c_{i-1} = \{\bar{1} \text{ if } (x_i + y_i) \le \bar{a}\}$$

with (for a maximally redundant system) $\overline{r\text{-}1} = \bar{a}$ and $a = r-1$.

2) Sum is: $u_i + c_i$ where c_i is the carry from i+1th position on the right.

The XLU primitive operations for addition directly implement the various steps of the Signed Digit addition algorithm. For a single- digit-by- single –digit (i.e. word by word) addition with a single digit result, the XLU can deliver the result via a single operation.

Other Representations:

As mentioned earlier, XLU representation can be used as a base from which other number representations may be created.

3.2 XLU Fixed – Point Representation

The fixed- point representation stated earlier is mentioned again here. As per the definition given by Koren:

A fixed – point number X is a sequence of n radix r digits x_{n-1}, x_{n-2}, …, x_1, x_0 that is partitioned into fraction parts of m digits and integral parts of k digits and the value is:

$$X = X_{k-1} r^{k-1} + X_{k-2} r^{k-2} + \ldots\ldots X_1 r + X_0 + X_{-1} r^{-1} + \ldots\ldots + X_{-m} r^{-m}$$

$$= \sum^{k-1}{}_{i=-m} X_i\ r^i$$

The radix point is understood to be in a fixed position between the k most significant digits and the m least significant digits.

Substituting the “XLU digit” for digit in the discussion above shows how naturally Fixed- Point numbers can be represented by the XLU architecture. A fixed – point number can be represented by one or more XLU digits with the additional constraint that the radix point lies somewhere to the right of the leftmost bit of the leftmost digit and somewhere to the left of the rightmost bit of the rightmost digit. The information carried by the XLU digit and the particular bit-pattern it shows at a given point in time does not change, although the interpretation of what it represents at that point in time may differ depending upon where the higher- level abstraction chooses to place the radix- point.

The arithmetic operations described earlier applies without any modification to XLU digit sequence no matter whether it represents a pure integer, a pure fraction or a fixed – point number. But in fixed – point arithmetic, the possibility of scaling of operands should also be considered. When scaling involves moving the radix point by fixed digits, the change is transparent to the digits themselves. But if the radix point is to be shifted between the bits of a single digit then XLU primitives are provided. They accept an XLU digit x and a value s to scale it by. The scaling primitives work by shifting the magnitude bits of x by s positions in the appropriate direction.

4 Floating point number system

To obtain a dynamic range of represent-able real numbers without having to scale the operands, the floating point numbers are used instead of fixed- point-ones. The representation of floating point numbers is similar to the commonly used scientific notation and consists of two parts, the significant (or mantissa) M and then exponent (or characteristic) E.

The floating point number represented by pair (M, E) is given by

$$M.\ \beta^{E}$$

where β is the base of the exponent. This base is common to all floating point numbers in a given system, it is not included in the representation of a floating point number, but it is rather implied. The n digits that represent a floating point number are partitioned into two parts, one holding the significant M and the other the exponent E.
The designer of a floating point representation must find a compromise between the size of the significant and the size of the exponent because a fixed word size means you must take a digit from one to add a digit to the other. This trade – off is between accuracy and range. Increasing the size of the significant enhances the accuracy of the significant while increasing the size of the exponent increases the range of the numbers that can be represented.

Compared to the fixed-point representation the range of represent-able floating point number is larger but the precision is smaller. The total number of different values (represent-able in n bits) is still 2^n and since the range between any two consecutive values must increase as well. Floating point numbers are thus sparser than fixed point numbers, resulting in a lower precision. Any real number whose value lies between two consecutive floating point numbers is mapped onto one of these two numbers.

Floating point numbers are usually a multiple of the size of a word. The representation of a floating point number is shown below where 'S' is the sign of the floating point number (1 meaning negative), exponent is the value of the 8- bit exponent field (including the sign of the exponent) and significant is the 23 bit number in the fraction. This representation is called Sign and Magnitude, since the sign has a separate bit from the rest of the number.

S	Exponent E	Unsigned Significant M

In general, floating point numbers are of the form:

$$(-1)\ S * F * 2^{E}$$

F involves the value in a significant field and E involves the value in the exponent field. The exact relationship to these fields will be spelled out soon.

The chosen sizes of exponent and significant gives an extra- ordinary range, but not infinite. Thus there are chances that overflow occurs in floating point arithmetic as well as integer arithmetic. Overflow means the exponent is too large to be represented in the exponent field.

Floating point representation offers another kind of exception as well, to know if the non-zero fraction is so small that it cannot be represented. This situation occurs when the exponent is too large to fit in the exponent field. This is called underflow.

One way to reduce the chances of underflow or overflow is to use a notation that has a larger exponent, either single precision floating point or double precision floating point could be used.

Single Precision:

Sign (1 bit)	Exponent (8 bits)	Mantissa (23 bits)

Double Precision:

Sign (1 bit)	Exponent (11 bits)	Mantissa (52 bits)

To pack even more bits into the significant, IEEE 754 makes the leading 1 bit of normalized binary numbers implicit. Hence the significant is actually 24 bits long in single precision (implied 1 and a 23 bit fraction) and 53 bits long in double precision (1 + 52). Since zero has no leading 1 it is given the reserved exponent value 0 so that the hardware would not attach a leading 1 to it. The representation of the rest of the number uses the form from before with the hidden 1 added:

$(-1)^{s\,*}$ (1+ Significant) * 2^{e}

where the bits of the significant represent the fraction between 0 and 1 and E specifies the value in the exponent field.

Also by placing the exponent before the significant simplifies sorting of floating point numbers using integer comparison instructions since numbers with bigger exponents look larger than numbers with smaller exponents as long as both have the same sign.

4.1 Floating point Operations

The way floating point operations are executed depends on the specific format used for representing the operands.

4.1.1 Floating-point addition and Subtraction

Floating point addition and subtraction are complicated by the fact that exponents of the 2 input operands must be equal before the corresponding mantissas can be added or subtracted. This is achieved by aligning its exponent at the same time, until it equals the exponent. In other words the significant of the smaller number (i.e. the number with the smaller exponent) is shifted $| E1 – E2 |$ base β positions to the right. For example, if E1≥ E2, which are biased, then

$$F1 \pm F2 = ((-1)^{S1} * M_1 \pm (-1)^{S2} * M_2 * \beta^{-(e1-e2)}) * \beta^{e1-bias}.$$

The exponent of the larger number is not decreased to make it equal to the other exponent, since this will result in a significant larger than 1 and a larger significant adder will be required.

If, based on the two sign bits and the originally required operations an addition is performed, then the resultant significant denoted by M:
$1/\beta \le M < 2$
If the significant M is greater than 1, a post normalization step is required. This consists of shifting the significant to the right to yield M3 and increasing the exponent by one.
At this point an exponent overflow may occur.

In summary the following steps are required when adding or subtracting two floating point numbers as given by Koren:
Step 1: Calculate the difference of two exponents, $d = |E1 – E2|$.
Step 2: Shift the significant of the smaller number by d base- β positions to the right.
Step 3: Add the aligned significant and set the exponent of the result equal to the exponent of the larger operand.
Step 4: Normalize the resultant significant and adjust the resultant exponent if necessary.
If the final operation called for is subtraction, then the resultant significant satisfies;

$$0 \le | M | < 1$$

and a post normalization step is required if the resultant significant is smaller than _. This step consists of shifting the significant to the left and decreasing the exponent simultaneously, which may lead to an exponent underflow. In extreme cases the post normalization step may require a shift left operation over all bits in the significant yielding a zero result.

Multiplication and Division are relatively simple since the mantissas and exponents can be processed independently. Floating point multiplication requires a fixed point multiplication of the mantissas and a fixed point addition of the exponents. Similarly floating point division requires a fixed point division involving the mantissas and a fixed point subtraction involving the exponents. Thus multiplication and division are not significantly more difficult to implement than the corresponding fixed point operations.

5 XLU Floating point Number Representation

The heart of a floating point representation is two separate pieces of information exponent and mantissa that together represent one numerical value. So this can be implemented with one or more XLU digits, the exponent as a pure integer and the fraction as either a fixed point value or a pure fraction. The extra negative pattern characteristic of 2's complement number system in the representation of a single XLU digit is reserved to indicate overflow or underflow.

The special patterns to indicate overflow and underflow are executed along with the rounding schemes in the next section.

Although there are many implementations of floating point numbers using XLU digit as a base for exponent and fraction, here is a simple implementation.

This floating point representation is synthesized from XLU digits as follows:

- An XLU digit composed of a 16 bit binary word in 2's complement.
- Radix r = 10.
- An exponent e composed of single or double XLU digits.
- A fraction f composed of two XLU digits playing the role of a pure fraction fixed point number.

- An implied radix point, the left of the leftmost fraction digit. This radix point is completely implicit and as stated in the sections dealing with fixed point representations.

Pre-normalizing the fractional of a floating point number involves scaling it up or down. The various scaling primitive operations are used to scale the numbers.

<u>XLU Floating Point Number</u>:

Exponent
Exponent
Mantissa
Mantissa

Addition or subtraction of 2 floating point numbers is broken down into sub-operations. The sub-operations implement, through sequence of XLU primitives, the steps in the floating point addition/ subtraction algorithms discussed previously.
At the completion of either a floating point addition or subtraction or multiplication sequence, the computed result may not be normalized. This may be acceptable in some instances, but more often the semantics of a particular floating point type implementation require that the result must be post-normalized. The goal of post normalization is the same as pre-normalization, to adjust the exponent and fractional parts of a floating point value so that together they fit in the definition of a floating point number. Here is a summary of Knuth's normalization algorithm.

1) Test the results fraction f, $| f | > = 1$ indicates "fraction overflow" and a need to scale the result down (i.e. to the right). $| f| = 0$ indicates that the exponent should be set to its lowest possible value, after which no further work is necessary, the normalization process is complete.

2) Test for normalization: for any other value, f may or may not be normalized. Testing whether $| f| > - 1/b$ determines this. If the number is normalized, proceed to rounding, otherwise scale the result up (i.e. to the left) and try again.

3) Scaling Up (left): Shift f to the left by one position and decrease the exponent by 1, and then test for normalization again.

4) Scaling down (right): Shift to the right by one position and increase e by 1, then proceeds to rounding.

5) Rounding: For a fraction f of at most n places, Knuth writes “We take this to mean that f is changed to the nearest multiple of b-n”. There are various methods for rounding a value of f greater than n places back down to n places. Some of these may result in | f | = 1, which is not allowable in Knuth’s algorithm. If this happens, return to the scaling down step and proceed from there.

6) Check e: Either the original result or prior operations of the normalization process may have resulted in an exponent underflow or overflow condition (e is either smaller or larger than its allowed range). Such cases indicate that the result computed cannot be expressed within the system and appropriate actions must be taken once per implementation basis. If e is safely within tolerance, then the normalization process is successfully completed.

As Knuth’s normalization algorithm shows, implementing normalization for floating point numbers based upon XLU digits is largely a matter of providing the primitive operation sequences to scale the fraction up or down, increment or decrement the exponent and test the value of the fraction at the appropriate point.

6 XLU Normalization Scheme

In our study we have considered the following hypothetical representation of a floating point number. The base is radix 10. The exponent is a single XLU digit and the mantissa part is stored in two XLU digits. There could be more than two digits representing the mantissa. In this representation there is no hidden 1. The floating point number is said to be normalized if the most significant digit of the mantissa is non-zero.
Ex: 1.66 * 10 ^{5} as normalized number is: 0.166 * 10 ^{6}. This is represented as:

6	1	6	6

There are two extra digits: Guard digit and Sticky digit for normalization and rounding purposes.

6.1 Extra Digits

To preserve the accuracy during floating point calculations one or more extra digits called guard digits are temporarily attached to the right end of the mantissa x_0, x_1, …. x_n. When a mantissa is right shifted during alignment step of addition or subtraction the digits shifted from the right end may be retained as guard digits. If the result does not have the most significant digit of 1, then it is shifted left until it has the form of 1.xxxxxx. This process is called normalization.

It is seen that only one guard digit is not sufficient as it gives incorrect result.

Consider this example: F_A = 1.0000…….00000 0(G)

F_B = 1.0000…….01111 0(G)

F_A and F_B are two binary numbers and G is the guard bit which is set to zero.

Assuming e_A-e_B = 4, the second number is shifted right 4 times, before subtraction.

Therefore F_A - F_B is: 1.00000………0 0(G)

0.00010………0 1(G)

0.11101………1 1(G)

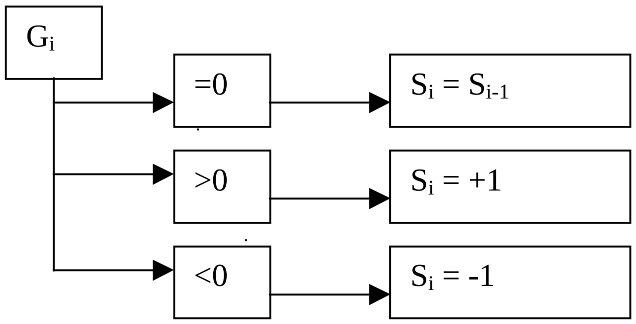
On normalization the result is: 1.11011….… 1

But if we had retained all the four shifted bits, the normalized result would have been 1.110111…..0

The result obtained with one guard digit is incorrect. This stresses the need for more than one guard digit. Hence two extra digits: guard and sticky digits are used. In order to check if there is a carry generated after shifting one of the operands, an extra digit known as sticky digit is added. The guard and round digits are used for normalization and rounding purposes. The sticky digit can take a value of 0, 1 and –1. Sticky digit indicates whether the shifted out so far has a value of 0, -1 or 1.

Suppose Gi is the current guard digit. The values of the sticky digit depend on the guard digit being shifted. It captures the essence of all the digits shifted so far.
Initially both the guard and sticky digit are set to zero.

If: Gi = 0, then sticky digit retains the previous value.
Gi = Positive number, then sticky digit = 1.
Gi= Negative number, then sticky digit = -1.



Example: Consider 2 XLU numbers:
A: $0.1 * 10^4$
B: $0.8\bar{2}22 * 10^0$

B is shifted 4 times to the right to make the exponents equal.

Computing A-B:

```
           G  S
A: 0.1000
B: 0.0000  8  ¯1
```

C: 0.1000 $\bar{8}$ 1

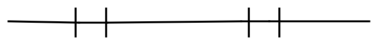
7 XLU Rounding Schemes

In this section different rounding schemes for XLU digits are described. Sticky digit can be 0, -1 or +1 and this is not used for rounding, otherwise it gives an error. The significant digits and the two extra digits, Guard digit and Round Digit are used for rounding. The different schemes are: Truncate, Towards Zero, Towards + ∞ , Towards -∞ , Towards nearest even, Towards nearest odd.

After Rounding, Normalization may be performed to make the most significant bit non zero. Also since the XLU digit itself carries the sign information, the guard and round digits can be positive or negative.

7.1 Truncate

The simplest scheme is called truncation. The extra digits are removed with no change to the remaining digits. The following example illustrates this concept.



-1.20 => -1.0 1.0 <= 1.25

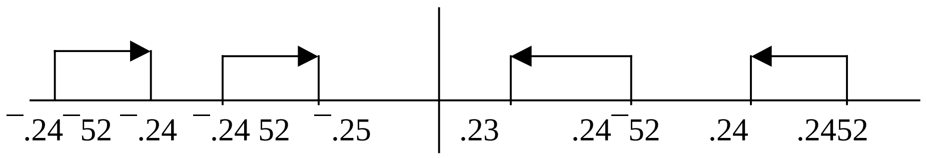
Significant	G and R	Rounding
.24	5 2	.24
.24	¯5 2	.24
¯24	5 2	¯.24
¯24	¯5 2	¯.24

7.2 Towards Zero

Under this scheme the extra digits are used to move the significant towards zero.

Example:

Significant	G and R	Rounding
.24	5 2	.24
.24	$\bar{5}$ 2	.23
$\overline{.24}$	5 2	$\overline{.25}$
$\overline{.24}$	$\bar{5}$ 2	$\overline{.24}$



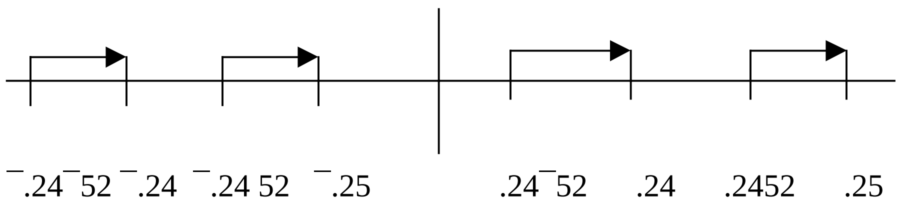
7.3 Towards +ve ∞

Here there are four cases to be considered. The significant is positive and extra digits are positive, significant is positive and extra digits are negative, significant is negative and extra digits are positive, and significant is negative and extra digits are negative.

Significant	Guard and Round digits	Rounding
+	+	Add +1
+	-	Truncate
-	+	Add +1
-	-	Truncate

Example:

Significant	G and R	Rounding
.24	52	.25
.24	¯52	.24
¯.24	52	¯.25
¯.24	¯.52	¯.24



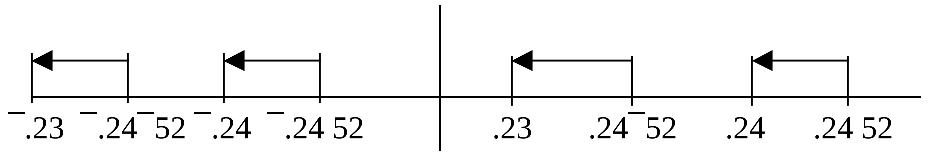
7.4 Towards -ve ∞:

Even here the four cases mentioned are to be considered. The significant is positive and extra digits are positive, significant is positive and extra digits are negative, significant is negative and extra digits are positive, and significant is negative and extra digits are negative.

Significant	Guard and Round digits	Rounding
+	+	Truncate
+	-	Subtract 1
-	+	Truncate
-	-	Subtract 1

Example:

Significant	G and R	Rounding
.24	52	.24
.24	¯52	.23
¯.24	52	¯.24
¯.24	¯.52	¯.23



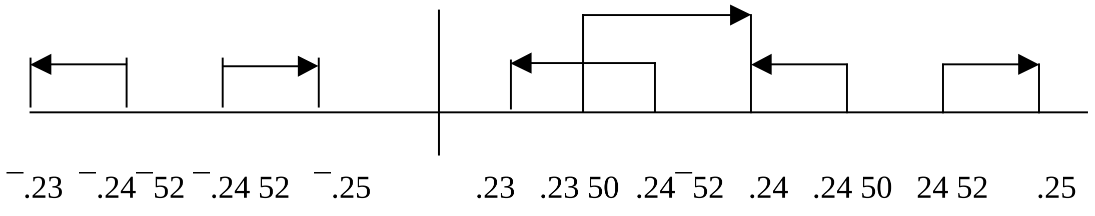
7.5 *Nearest even*

In this scheme in addition to the four cases stated earlier, we have to consider the value of the extra digits: whether they are equal to 0.5, greater than 0.5, lesser than 0.5, the rounding schemes are summarized in the table below.

Significant	G and R	Value of G and R	Rounding
+	+	< 0.5 = 0.5 > 0.5	Truncate. If last digit is odd: add 1, if last digit is even: truncate it. Add 1.
+	-	< 0.5 = 0.5 > 0.5	Truncate. If last digit is odd: subtract 1, if last digit is even: truncate it. Subtract 1.
-	+	< 0.5 = 0.5 > 0.5	Truncate. If last digit is odd: add1, if last digit is even: truncate it. Add 1.
-	-	< 0.5 = 0.5 > 0.5	Truncate. If last digit is odd: subtract 1, if last digit is even: truncate it. Subtract 1.

Example:

Significant	G and R	Rounding
.24	52	.25
.24	50	.24
.23	50	.24
.24	¯52	.23
¯.24	52	¯.25
¯.24	¯.52	¯.23



7.6 *Nearest odd*

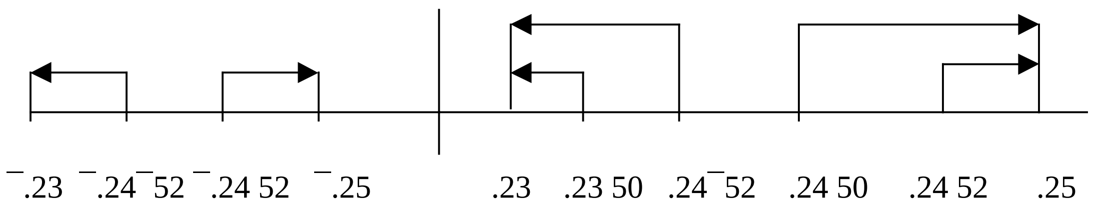
In this scheme in addition to the four cases stated earlier, we have to consider the value of the extra digits: whether they are equal to 0.5, greater than 0.5, lesser than 0.5, the rounding schemes are summarized in the table below.

``````````````````````````

| Significant | G and R | Value of G and R | Rounding |
|---|---|---|---|
| + | + | < 0.5<br>= 0.5<br>> 0.5 | Truncate.<br>If last digit is even: add 1, if last digit is odd: truncate it.<br>Add 1. |
| + | - | < 0.5<br>= 0.5<br>> 0.5 | Truncate.<br>If last digit is even: subtract 1, if last digit is odd: truncate it.<br>Subtract 1. |
| - | + | < 0.5<br>= 0.5<br>> 0.5 | Truncate.<br>If last digit is even: add1, if last digit is odd: truncate it.<br>Add 1. |
| - | - | < 0.5<br>= 0.5<br>> 0.5 | Truncate.<br>If last digit is even: subtract 1, if last digit is odd: truncate it.<br>Subtract 1. |

Example:

| Significant | G and R | Rounding |
|---|---|---|
| .24 | 52 | .25 |
| .24 | 50 | .25 |
| .23 | 50 | .23 |
| .24 | ¯52 | .23 |
| ¯.24 | 52 | ¯.25 |
| ¯.24 | ¯.52 | ¯.23 |
``````````````````````````



8 Conclusion

A non-redundant number system has the digit set limited ranging from 0 to radix -1. While the Signed digit numbers have digit set consisting of more than radix r values. Hence it is also called redundant number system. Furthermore, the advantage of redundant number system lies in its application for fast addition. In this project work we have considered the XLU architecture which is based on signed digit number system. We have developed methods for rounding, normalization of XLU floating point numbers. The various rounding schemes like truncation, towards zero, towards Zero, towards + ∞ , towards - ∞ , towards nearest even and towards nearest odd are discussed in detail with relevant examples.

9 Acknowledgements

Sincere acknowledgements and regards to Dr. Bella Bose for the timely advice and guidance provided during this project work. I also thank Dr. Toshimi Minoura and Dr. Timothy Budd for their co-operation.

10 References

1. Israel Koren: Computer Arithmetic Algorithms. Prentice Hall, Engelwood Cliffs, New Jersey, 1993.

2. Algirdas Avizenis: Signed-Digit Number Representations for Fast Parallel Arithmetic. IRE Transactions on Electronic Computers, EC-10:289-400, 1961.

3. David W. Matula: Radix Arithmetic-Digital Algorithms for Computer Architecture.

4. Kevin Djang: An Alternative Architecture for Performing Basic Computer Arithmetic Operations. Computer Science Department, Oregon State University.