A Mixed Decimal/Binary Redundant Floating-Point Adder

By

Karim Yehia Fathy Mahmoud ElGhamrawy

A Thesis Submitted to the

Faculty of Engineering at Cairo University

In Partial Fulfillment of the

Requirement for the Degree of

MASTER OF SCIENCE

In

ELECTRONICS AND COMMUNICATIONS ENGINEERING

FACULTY OF ENGINEERING, CAIRO UNIVERSITY

GIZA, EGYPT

2011

A Mixed Decimal/Binary Redundant Floating-Point Adder

By

Karim Yehia Fathy Mahmoud ElGhamrawy

A Thesis Submitted to the
Faculty of Engineering at Cairo University
In Partial Fulfillment of the
Requirement for the Degree of
MASTER OF SCIENCE
In
ELECTRONICS AND COMMUNICATIONS ENGINEERING

Under the Supervision of

Hossam. A. H. Fahmy
Associate Professor
Elec. And Com. Dept.

Electronics and Communications Engineering Department
Faculty of Engineering, Cairo University

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
2011

A Mixed Decimal/Binary Redundant Floating-Point Adder

By

Eng. Karim Yehia Fathy Mahmoud ElGhamrawy

A Thesis Submitted to the

Faculty of Engineering at Cairo University

In Partial Fulfillment of the

Requirement for the Degree of

MASTER OF SCIENCE

In

ELECTRONICS AND COMMUNICATIONS ENGINEERING

Approved by the

Examining Committee

Prof. Dr. Mohamed Zaki Abd El Mageed

____

Prof. Dr. Amin M. Nassar

____

Assoc. Prof. Dr. Hossam A. H. Fahmy, Thesis Main Advisor

____

FACULTY OF ENGINEERING, CAIRO UNIVERSITY

GIZA, EGYPT

2011

Table of Contents

List of Figures

List of Tables

Abstract

Binary floating-point adders are used in processors to perform scientific computations. However, binary floating-point operations are not suitable for financial and commercial computations. Decimal floating-point operations were carried out in software until recently when IBM introduced decimal floating-point adders in their z9 and z10 processors.

In this research, initially a decimal floating-point adder is proposed. The floating-point numbers are encoded in a redundant format. The digit set is [-6, 6]. Redundancy allows for a carry-free addition so that the addition operation does not depend on the width of the operands. The adder was synthesized with the TSMC 65 nm LP technology and shows a latency of 2.02 ns (48 FO-4) and a total area of 24683 um^2.

The design is then extended to incorporate IEEE-compliant binary addition as well. This preserves the speed gains produced through redundancy while using the same hardware resources for both binary and decimal operations. Synthesis Results show a 68% increase in delay over the decimal one, and a 26.8% increase in area. This makes our mixed floating-point adder more area-efficient to rather than using a separate binary floating-point adder and another separate decimal one.

The Mixed floating-point Adder was further pipelined into five stages to increase the maximum frequency of operation, and to allow for overlapping the execution of binary operations and decimal ones. Synthesis results show that our pipelined mixed floating-point adder can operate on a clock with a frequency up to 1.04 GHz.

Chapter 1

1 Introduction

1.1 Floating-Point System

Scientific and Engineering applications work on real numbers. Real numbers can be represented in computers in fixed-point representations where the fractional point has a fixed position in the number. This allows for using the same integer units to perform real number computations. However the range of real numbers in fixed-point representation is very small.

Another alternative is to represent real numbers in floating-point representation. In this representation, a real number ‘a’ is represented as (S_a, M_a, E_a) where S_a is the sign of the real number ‘a’, M_a is the significand or Mantissa, and E_a is the base-r exponent of the real number ‘a’.

$$a = (-1)^{S_a}.M_a.r^{E_a} \tag{1-1}$$

1.1.1 Dynamic Range

Dynamic range is defined as the ratio between the largest real number to the smallest positive real number. The dynamic range of a fixed-point representation is

$$DR_{fixed} = r^n - 1 \tag{1-2}$$

where r is the base and n is the number of digits.

For floating-point representation, if we divide the n digits into two parts: an m-digit mantissa and an (n-m)-digit exponent. We get $DR_{\text{floating-point}}$ such that

$$DR_{floating-point} = (r^m - 1).r^{(\ n-m} {}^{-1)} \tag{1-3}$$

For instance, if n = 64, m = 55, and r = 2

$DR_{fixed} \approx 2 \times 10^{19}$

$DR_{\text{floating-point}} \approx 10^{168}$

It is obvious that floating-point representation has a high dynamic range which makes it suitable for applications requiring a wide range of real numbers.

1.1.2 Precision

Precision is defined as the total number of digits in the significand. For the same number of digits n, the precision of the fixed-point representation is higher than the precision of the floating-point representation because in the later, the total number of digits is divided between the significand and the exponent. So the higher dynamic range of the floating-point representation comes at the expense of a less precision.

1.1.3 Rounding

Real-numbers in computers are represented in a finite number of bits. Hence, not all real numbers can be represented exactly. That is why rounding is an essential step in floating-point systems.

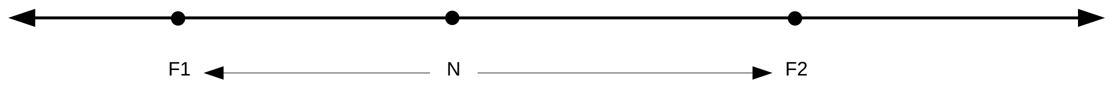


Figure 1-1 The real number N between two floating-point numbers F1 and F2

In a floating-point system, a real number that is exactly represented in this system is called a floating-point number. If another real number, N that lies between the two floating-point number F1 and F2, is to be represented using this floating-point system, then N should be rounded to either F1 or F2 according to the rounding mode required.

1.2 IEEE 754-1985

The IEEE 754-1985 is the first IEEE standard for binary floating-point computations. The standard was later revised in 2008 (IEEE 754-2008) to incorporate decimal floating-point computations as well.

The 754-1985 standard defines formats for representing floating-point numbers and special values (infinities, and NaNs) together with a set of floating-point operations that operate on these values. It also specifies four rounding modes.

The IEEE 754-1985 defines four binary floating-point formats with different precision:

1- Single precision (32-bit)

2- Double precision (64-bit)

3- Single-extended precision (≥ 43-bit)

4- Double-extended precision ((≥ 79-bit)

1.2.1 Representation of the Significand and Exponent

The IEEE 754-1985 uses a sign-magnitude representation for the significand. A sign-magnitude representation facilitates comparing significands. It also facilitates the multiplication operation.

A floating-point number can be represented in more than one way. For example, the number 2.0 can be represented as 2 x 10^0 or 0.2 x 10^1. This makes comparing two floating-point numbers difficult. This redundancy in representation can be avoided through normalized representation. Normalization means that the most significand digit of the floating-point number must be a non-zero digit (except for the zero number). The IEEE 745-1985 standard uses normalized representation for significands.

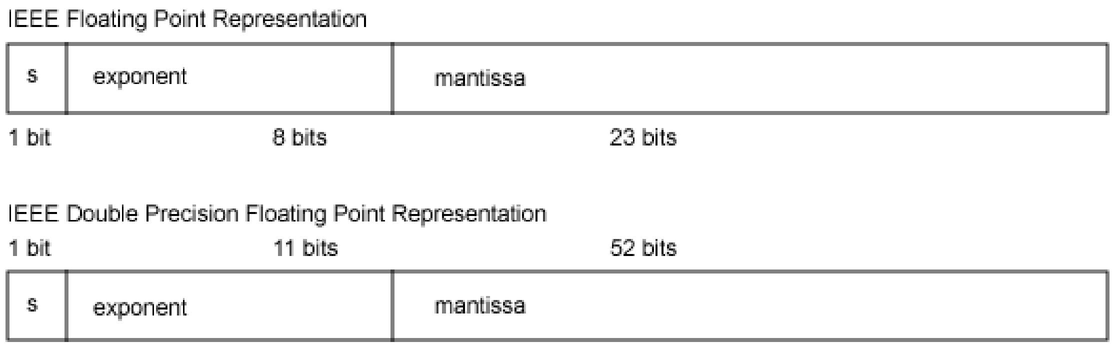

IEEE Floating Point Representation
s
exponent
mantissa
1 bit
8 bits
23 bits
IEEE Double Precision Floating Point Representation
1 bit
11 bits
52 bits
s
exponent
mantissa

Figure 1-2 Significand and Exponent Representation in Single and Double Precision

The exponent is represented in a biased format in order to be able to represent positive and negative significands. The bias is equal to 2^{n-1}-1 where n is the number of bits of the exponent.

A floating-point number consists of three fields as shown in Figure 1-2: The sign bit, the fraction, and the exponent. Since the significand is normalized, the most significant bit (MSB) must be ‘1’, hence it is not explicitly stored and it is called a ‘hidden 1’. Only the fraction is explicitly represented.

1.2.2 Special Values

In addition to the normal floating-point numbers, the IEEE 754-1985 defines some special cases like infinities and NaNs.

Table 1-1 Special cases

Case	Sign	Biased Exponent	Fraction
+0	+	0	0
-0	-	0	0
+infinity	+	Maximum	0
-infinity	-	Maximum	0
NaN		Maximum	≠ 0

The IEEE 754-1985 allows also denormalized numbers to be represented. Denormalized numbers (Denormals) are floating-point numbers that are less than the value of the minimum normalized floating-point number. Denormals are represented by a minimum exponent (biased exponent = 0) and non-zero fraction.

1.2.3 Rounding Modes

The IEEE 754-1985 standard defines four rounding modes:

1- **Round toward +∞ or round up (RP)**, directed rounding toward positive infinity.

2- **Round toward -∞ or round down (RN)**, directed rounding toward negative infinity.

3- **Round toward 0 (RZ)**, directed rounding toward zero.

4- **Round to nearest tie to even (RNE)**, rounds to the nearest value; if the number falls midway it is rounded to the nearest value with an even (zero) least significant bit.

1.3 IEEE 754-2008

The IEEE 754-1985 standard was revised in 2008 when the IEEE 754-2008 replaced it [1]. It includes the entire original IEEE 754-1985 standard in addition to decimal floating-point computations.

The standard defines arithmetic formats for binary and decimal floating-point data as shown in Table 1-2. It also defines interchange formats (encodings) for the floating-point data.

Table 1-2 The IEEE 754-2008 arithmetic formats

Name	**Common Name**	**Base**	**Digits (including the hidden one for binary representations)**	**Max. exponent**	**Min. exponent**
binary16	Half precision	2	11	15	-14
binary32	Single precision	2	24	127	-126
binary64	Double precision	2	53	1023	-1022
binary128	Quadruple precision	2	113	16383	-16382
decimal32		10	7	96	-95
decimal64		10	16	384	-383
decimal128		10	34	6144	-6143

One main difference between binary floating-point data and decimal floating-point data is that decimal-floating point numbers are not normalized. Hence, the same floating-point number can be represented in more than one form.

Unlike in a binary floating-point format, a number in a decimal floating-point format can have more than one representation. The set of representations of a floating-point number is the floating-point number's cohort. For example, a decimal floating-point number that is represented as (sign, significand, exponent) and another decimal floating-point number (sign, significand/10, exponent+1) belongs to the same cohort, assuming the significand is a multiple of 10.

For decimal arithmetic, the specified decimal operation selects a member of the result's cohort. Since we are interested in decimal floating-point addition here, it is important to mention that the preferred exponent for an exact result is the smaller exponent of the two operands. The preferred exponent for an inexact result is the least possible exponent.

1.4 Binary Floating-Point Addition

Floating-point addition is the most frequent floating-point operation, and addition is the most basic floating-point operation because other floating-point operations depend on addition. If addition is fast, the performance of the whole floating-point unit improves. A lot of research has been done to enhance the algorithm and the implementation of floating-point adders [2] [3] [4] [5] [6].

1.4.1 Binary Floating-Point Addition Algorithm

We will now investigate the algorithm and the basic implementations of binary floating-point adders.

Let A, B be two binary floating-point numbers whose addition results in a new floating number Z where CA, CB, and CB are the significands of A, B, and Z respectively. EA, EB, and EZ are their exponents. SA, SB, and SZ are the signs of their significands. CA*, CB*, and CZ* are their signed significands (The significand including the sign information).

$$A = (-1)^{SA}.CA.2^{EA} \tag{1-4}$$
$$B = (-1)^{SB}.CB.2^{EB} \tag{1-5}$$
$$Z = (-1)^{SZ}.CZ.2^{EZ} \tag{1-6}$$

$$CZ^* = \begin{cases} \left(CA^* + \left(CB^* * 2^{EB-EA}\right)\right) * 2^{EA} \; if \; EA \geq EB \\ \left(CB^* + \left(CA^* * 2^{EA-EB}\right)\right) * 2^{EB} \; if \; EB < EA \end{cases} \tag{1-7}$$

First, the difference between the exponents EA, and EB must be computed. According to this difference, the number with the smaller exponent is multiplied by 2 raised to the power of the exponent difference according to eq. (1-7) (This process is called *alignment*). Then addition is performed.

In order to be compliant with the IEEE 754 standard, the resulting number Z must be normalized and rounded according to the required rounding mode.

The detailed algorithm of binary floating-point addition is:

1- Compute the exponent difference d and set the result exponent to be the larger exponent.

2- Significand alignment: This is performed by shifting the number with the smaller exponent d positions to the right.

3- Add or subtract the aligned significands according to the effective operation. The effective operation depends on the floating-point operation and the sign of the operands according to the Table 1-3

Table 1-3 Effective Operation Computation

Floating-point operation	**Sign of the operands**	**Effective operation**
Addition	Equal	Addition
Addition	Different	Subtraction
Subtraction	Equal	Subtraction
Subtraction	Different	Addition

4- Normalization: The resulting significand is not necessarily normalized. If the effective operation is addition, there is a chance a final carry might have occurred. This is resolved by shifting the result one position to the right and incrementing the result exponent. If the effective operation is subtraction, there is a chance some leading zeros might occur in the significand of the result. This is resolved by first detecting the amount of leading zeros, then shifting the result to the left a number of positions equal to the number of leading zeros. The exponent must be adjusted as well.

5- Rounding: The result should be rounded according to the specified rounding mode.

6- Final Adjustment: Rounding itself can generate a final carry or a special value (infinities for instance). If a final carry has occurred, normalization has to be performed again by shifting the number one position to the right and incrementing the exponent. If rounding yields a special value, the resulting number must reflect this change.

1.4.2 Binary Floating-Point Implementation

The basic algorithm for binary floating-point addition can be implemented by the block diagram of Figure 1-4.

First EB is subtracted from EA to determine the absolute difference between the two exponents in order to perform the significand alignment operation. Moreover, the number who has the larger exponent is determined.

Aligning the significands requires that the number with the smaller exponent to be shifted m-positions to the right according to the difference between the exponents. Instead of having two alignment shifters, a swapping unit is used and only one alignment shifter. When shifting CY to the right, we should keep record of the most recent two shifted-out bits (Guard bit, Round bit), and a sticky bit as shown in Figure 1-3. A sticky bit contains the information of whether the remaining shifted-out bits were all zeros or not. In case all the other shifted-out bits were zeros, the sticky bit is zero. If not, the sticky bit is set. The round, guard, and sticky bits are imperative for correct rounding.

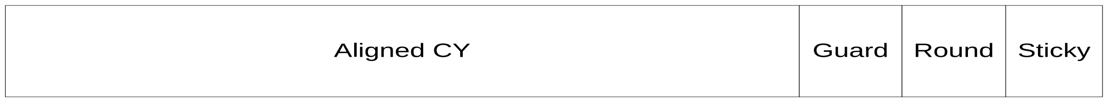


Figure 1-3 the Round, Guard, and Sticky bits

If EA is greater than or equal EB, the swap signal is not raised. But if EB is less than EA, the swap signal is raised. According to the block diagram in Figure 1-4, after the swapping unit, we have two numbers: X, the number with the higher exponent, and Y, the number with the smaller exponent.

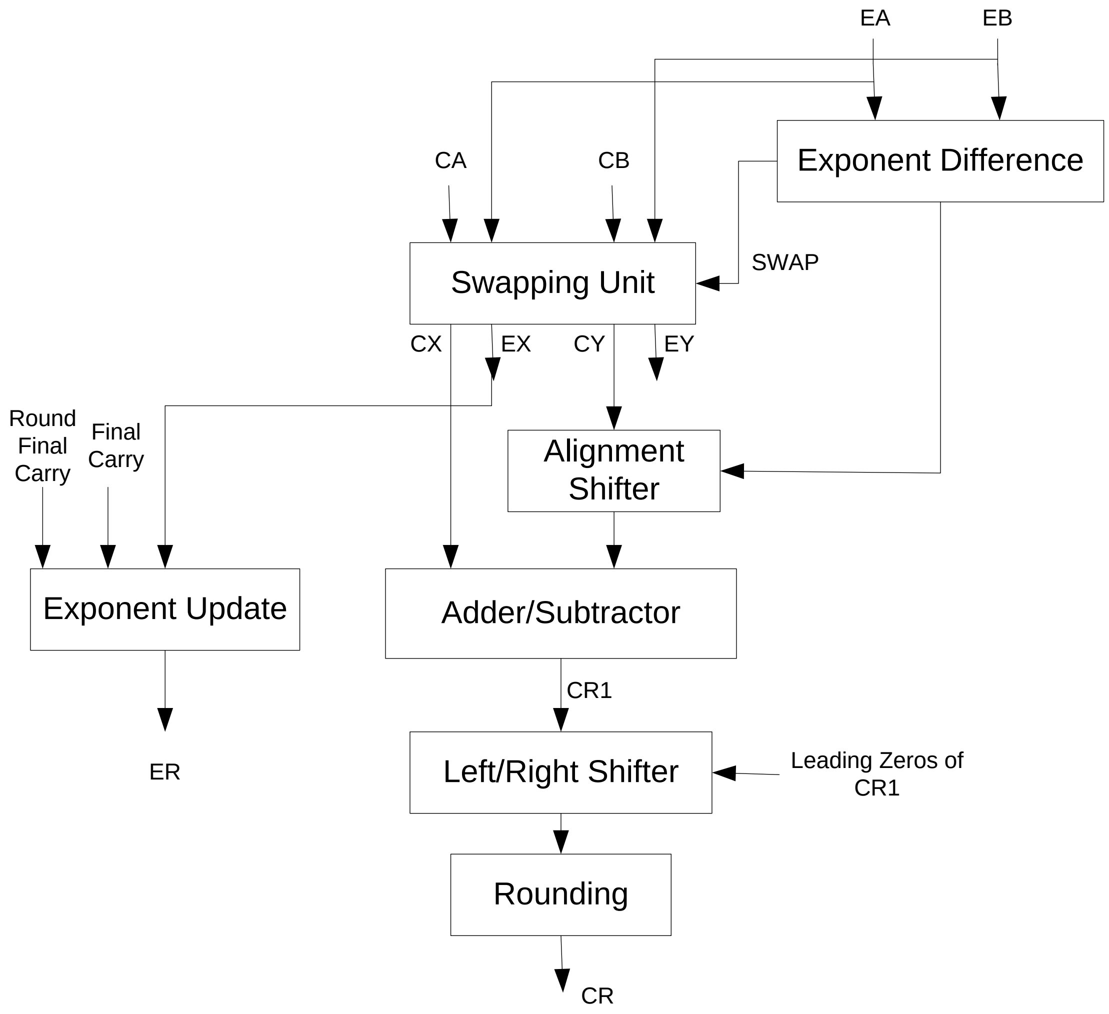


Figure 1-4 Block Diagram of a Conventional IEEE-Compliant Binary Floating-Point Adder

Now, the two operands are aligned and the exponent of the result is set primarily to be the higher power EX. The two operands are added or subtracting depending on the effective operation to generate CR1.

After addition/subtraction, the resulted CR1 is not necessarily normalized, so it should be normalized. We have two cases:

1- The effective operation was addition and a final carry was generated. In this case, CR1 should be shifted one position to the right.

2- The effective operation was subtraction. In this case, CR1 may contain some leading zeros. It should be shifted to the left with the same amount of the leading zeros. This process requires a leading-zero detector (LZD) to detect the number of zeros in CR1.

After normalization, rounding should be performed based on the specified rounding mode. Because of the alignment process, CY may be shifted m-positions to the left. Moreover, during normalization of CR1, it is shifted to the left depending on the number of leading zeros in CR1. This means that CY should be represented using a number of bits that is greater than the f-fractional bits. These extra bits are necessary for correct rounding. However, after rounding, the excess bits of CR1 are disposed of.

The question which arises is, how many extra bits should we keep track of? To answer this question, we discuss the following different rounding modes:

- For rounding toward zero, only the f fractional bits are required.
- For rounding to nearest, an additional bit is required. So (f+1) fractional bits are required. For the tie case, we should have information about whether the shifted-out bits are all zeros or not. This information is stored in the sticky bit.
- For rounding toward positive or negative infinity, we still need the information contained in the sticky bit discussed above.

So in general, to be able to perform any of the rounding modes, (f+1) fractional bits of the normalized significand are required in addition to a sticky bit which gives an information about the shifted-out bits, whether they were all zeros or not.

In order to answer the question posed above about the number of extra bits that should be stored after addition/subtraction so that we can guarantee an (f+1) fractional bits in the normalized CR1 and a sticky bit, we discuss the case of an effective addition, and the case of an effective subtraction:

In the case of effective addition, CR1 is either normalized or it contains a final carry. If it contains a final carry, it will be shifted one position to the right. In the worst case, CR1 is normalized and we need an extra bit to have an (f+1) fractional bits of the normalized significand, and we also need a sticky bit.

In the case of effective subtraction, there are two sub-cases:

1- The exponent difference is greater than 1

In this case, CY has more than one leading zero, but CR1 is either normalized or has at most one leading zero. Normalizing CR1 in this case requires a shift-left by one bit position. Hence, for normalization, we need to store this extra bit. For rounding, we need to store one additional bit in addition to the sticky bit. Therefore CR1 should have (f+3) fractional bits including the sticky bit. These bits are the guard bit, the round bit, and the sticky bit as shown in Figure 1-3.

2- The exponent difference is zero or 1

In this case, the result CR1 might contain more than one leading zero. But since the exponent difference is at most one, then CY will at most be shifted one position to the right. Only one additional bit is required after subtraction.

Table 1-4 Round to nearest tie to even mode

Least significant bit (L)	Guard	Round	Sticky	Action
X	0	x	x	truncate
0	1	0	0	truncate
1	1	0	0	Add one to L
X	1	1	x	Add one to L
X	1	0	1	Add one to L

Table 1-4 shows the required actions if the specified rounding mode is ‘Round to nearest tie to even’ (RNE). If rounding to positive infinity is required, if any of the extra bits is set, and the sign of the result is positive, one should be added to the least significant bit. If rounding to negative infinity is required, if any of the extra bits is set, and the sign of the result is negative, one should be added to the least significant bit.

Rounding toward zero is just a mere truncation of the extra bits.

1.4.3 Double-Path Implementation

There are several modifications that have been developed for the implementation of binary floating-point addition. The main objective of these modifications was to enhance the speed of the floating-point adder. For instance, a leading zero anticipator (LZA) might be used to anticipate the number of leading zeros of CR1 in parallel with the addition/subtraction process instead of a leading zero detection (LZD) [7] [8].

We should notice that the critical path of the single-path implementation includes two variable shifters: one for alignment and the other for normalization. However, as mentioned earlier, normalization requires a variable shifter only in the case of an effective subtraction and an exponent difference that is less than or equal to one. In this particular case, the alignment shifter is at most a one bit shifter.

In order to decrease the overall latency of the floating-point adder, one solution is to define two separate paths as shown in Figure 1-5. The *close path* if the effective operation is subtraction and the exponent difference is less than or equal to one, and the *far path* if the effective operation is addition or the exponent difference is greater than one.

In the close path, there is only a one-bit right shifter for alignment, the adder, the variable left shifter. On the other hand, the far path has a variable right shifter, the adder, and a one-bit left shifter for normalization.

To reduce latency, rounding can be performed in parallel with adding before normalization. This can be achieved through implementing an adder that generates the (sum) and the (sum+1). The rounding mode selects one of these sums to be the rounded sum.

Leading zero detection is a complex operation. It is as time consuming as addition itself. As a result, a leading zero anticipator can be used in the close path in parallel with addition.

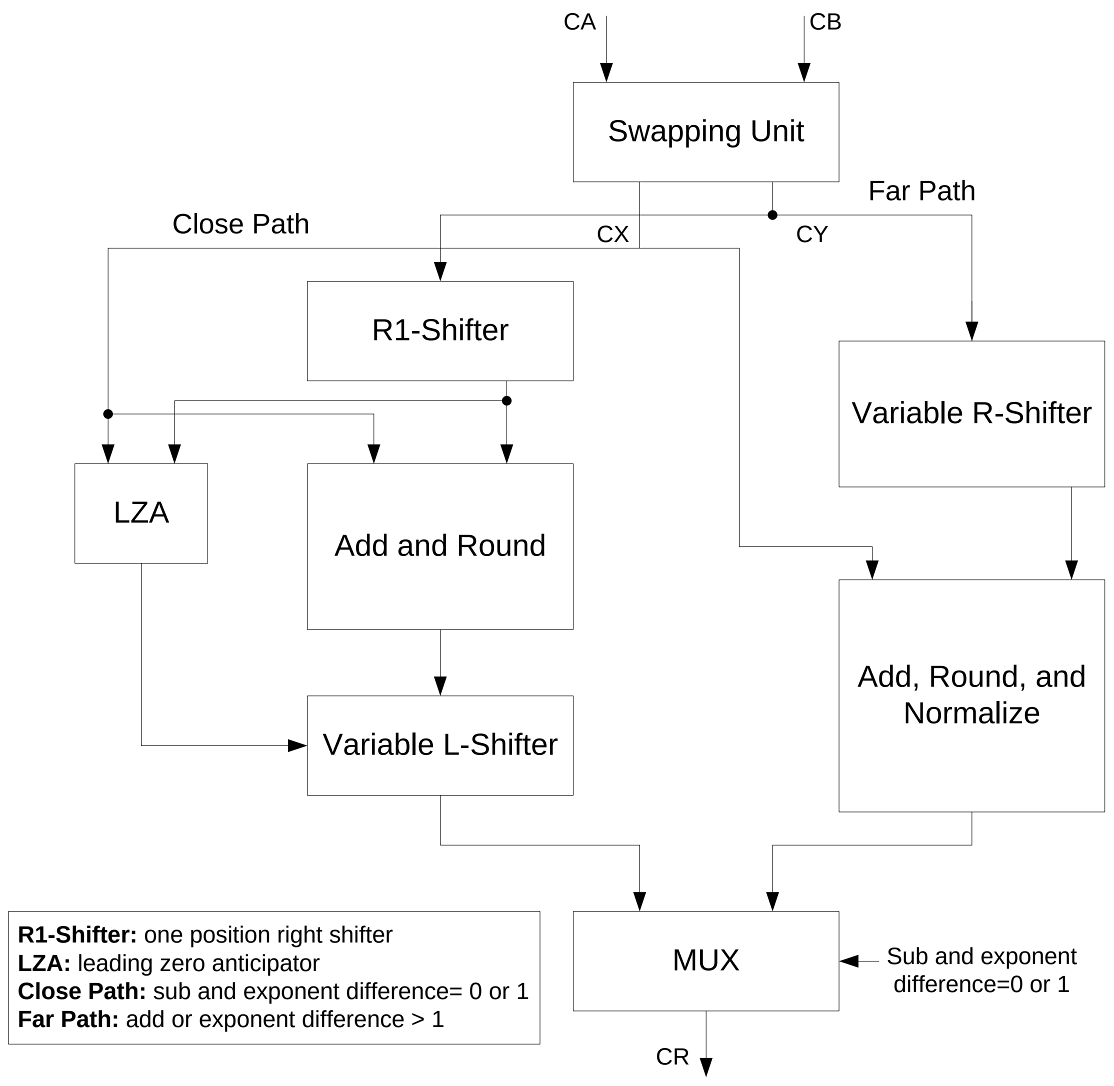

CA
CB
Swapping Unit
Far Path
Close Path
CX
CY
R1-Shifter
Variable R-Shifter
LZA
Add and Round
Add, Round, and Normalize
Variable L-Shifter
MUX
Sub and exponent difference=0 or 1
CR
R1-Shifter: one position right shifter
LZA: leading zero anticipator
Close Path: sub and exponent difference= 0 or 1
Far Path: add or exponent difference > 1

Figure 1-5 double-path implementation of the binary floating-point adder

1.5 Decimal Floating-Point Addition

1.5.1 The Rationale behind Decimal Arithmetic

Financial and business applications use decimal based arithmetic to perform arithmetic operations. These applications require accuracy.

We have mentioned that floating-point arithmetic introduces a roundoff error. This is because of the finiteness of the floating-point numbers representable in a given floating-point system. The accumulation of these roundoff errors can result in a totally different numbers than the number expected.

Consider the following fragment of code:

```
double a,b,x,y;
a=0.3;
b=0.1+0.1+0.1;
println(a==b);
x=0.5;
y=0.1+0.1+0.1+0.1+0.1;
println(x==y);
```

Surprisingly, the above code will print "false" for the first statement and "true" for the second one. This is because the simple decimal floating-point number '0.1' needs an infinite precision in a binary floating-point system to be completely represented.

This unpredictability in the program behavior can cause big problems for applications that depend greatly on the accuracy of the numbers. Moreover, it puts the burden on the programmer to handle these problems.

Early solution to the above problem was to implement decimal floating-point arithmetic in software [9] [10]. However, to increase the performance of decimal floating-point arithmetic, decimal floating-point units were recently implemented in hardware [11] [12] [13].

The revised IEEE 754-2008 defines the standard for decimal floating-point operations.

1.5.2 IEEE-compliant Decimal Adders

To the best of my knowledge, the first hardware implementation of an IEEE compliant decimal floating-point adder was proposed by Thompson et al. [14]. He presented the design and implementation of a 64-bit decimal floating-point adder that is compliant with the IEEE 754-2008 standard. The design performs addition and subtraction on 64-bit operands and can be pipelined to achieve substantial improvements in its critical path delay. The adder can also be extended to perform 32-bit and 128-bit decimal floating-point addition as well.

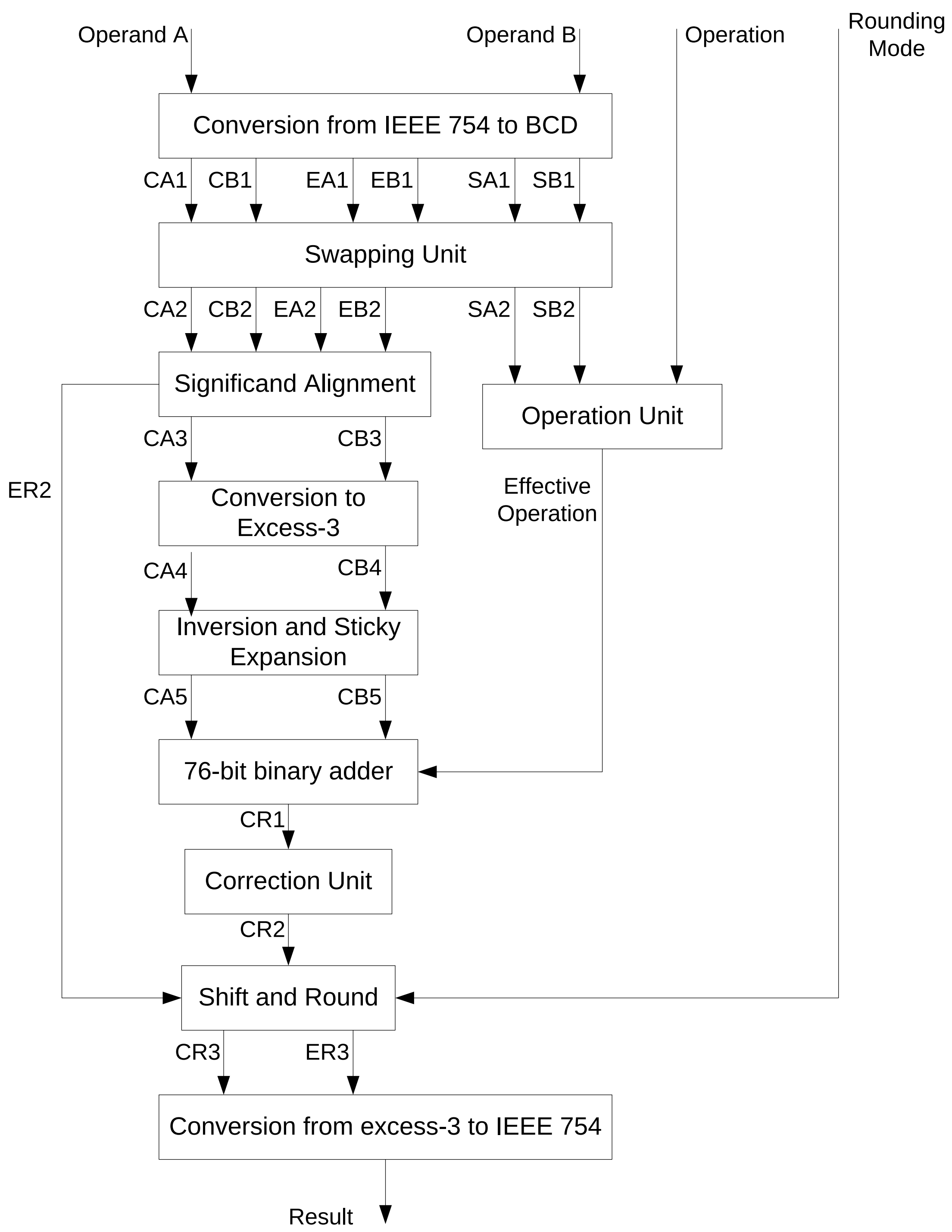


Figure 1-6 a 64-bit decimal FP adder/subtractor by Thomas et al.

Figure 1-6 shows the block diagram of the decimal floating-point adder.

First, the two IEEE 754 operands are unpacked into their corresponding sign bits (SA1 and SB1), 10-bit biased binary exponent (EA1 and EB1), and 16-

digit binary coded decimal (BCD) significands (CA1 and CB1). Note that the IEEE 754 standard uses densely packed decimal (DPD) for significand encoding for decimal floating-point data. This is good for storage purposes as DPD represents three decimal digits in 10 bits instead of 12 bits.

Second, the two operands are ordered by the swapping unit according to their exponent values. The swapping unit requires the two operands to be exchanged if EB1 is greater than EA1. The effective operation is also determined based on the signs of the two operands and the specified operation.

Next, the operands are aligned so that they have the same exponent ER2. Since the decimal floating-point numbers are not necessarily normalized, operand alignment should be performed so that ER2 is the preferred exponent. This usually requires shifting the number with the larger exponent to the left and/or shifting the operand with the smaller exponent to the right.

The two operands are then converted to an excess-3 decimal encoding. Inversion is also performed in case of an effective subtraction.

The two significands (CA5 and CB5) are passed into the binary adder, which performs the addition. The adder produces CR1 which is then rounded according to the specified rounding mode. ER2 is updated as well.

Last, the result is converted back to the DPD format.

Later, Wang and Schulte [15] presented another decimal floating-point (DFP) adder. The DFP adder uses a parallel method for decimal operand alignment, and a modified Kogge-Stone (K-S) parallel prefix network for significand addition and subtraction. A novel decimal variation of the injection-based rounding method is also applied. The DFP has a 21 percent less delay and 1.6 percent less area than the DFP proposed by Thompson et al. Figure 1-7 shows the block diagram of the proposed adder.

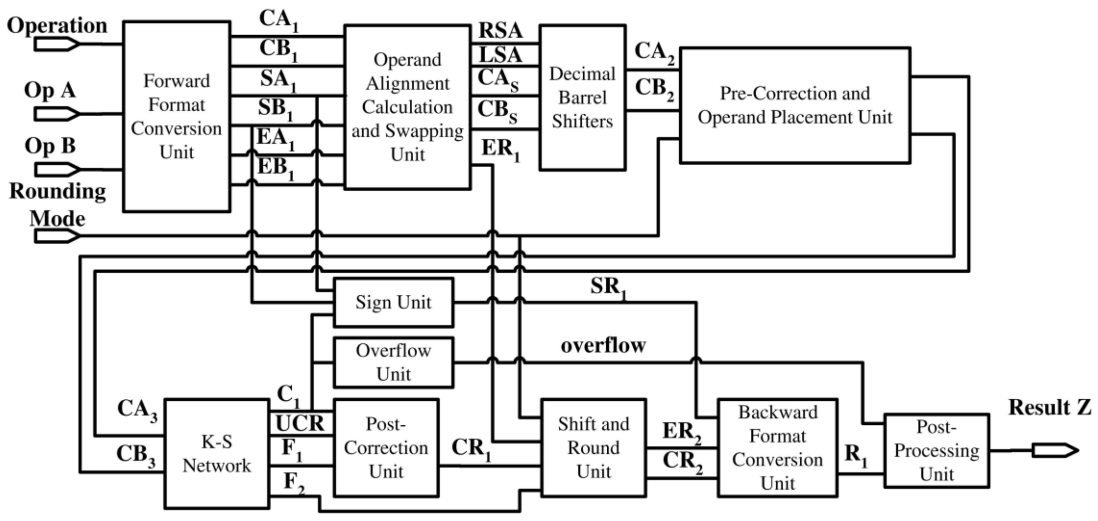


Figure 1-7 the DFP adder by Wang and Schulte **[15]**

Generally the algorithm used for binary floating-point addition is the same one used for decimal floating-point addition. The only difference originates from the fact that decimal floating-point data are not normalized. Hence, a member of the result's cohort must be selected so that the result has a preferred exponent as stated by the standard. This usually requires an operand to be shifted left while the other is shifted right to perform alignment. Later, Wang and Schulte enhanced their decimal FP-adder even more with a decoded operand and a decimal leading zero anticipator [16].

1.6 Redundancy

Addition is a basic building block in computer arithmetic. If addition is slow, all other operations are slow. Generally, the critical path of an adder is the carry propagation path. One method to enhance the carry propagation delay is through complex carry look-ahead adders. Another method is to try to limit the carry propagation to within a small number of bits, or to try to eliminate carry propagation altogether. This can be achieved through redundancy.

From a linguistic point of view, “redundancy” means something that is excessive, superfluous, and unneeded. In the computer arithmetic realm, “redundancy” means that a given number can be represented in more than one form. This allows for carry-free addition.

For example, assume the addition of two decimal operands is required given that the digit-set of the operands is conventional non-redundant [0, 9] digit-set.

If we assume that the sum can be represented in a redundant format such that the digit-set of the result is [0, 18], then we can perform a carry-free parallel addition as shown in Figure 1-8.

$$\begin{array}{ccccc} & 8 & 9 & 5 & 2 \\ + & 1 & 6 & 5 & 4 \\ \hline & 9 & 15 & 10 & 6 \end{array}$$

Figure 1-8 Redundant addition example.

In the previous example, we are adding the decimal number 8952 to 1654. The result, represented in conventional non-redundant decimal representation should be 10606. However, if we allow redundant representation of the result, the sum can be written as shown in Figure 1-8 which is a redundant decimal representation of the same number 10606.

$$\mathbf{6} * 10^0 + \mathbf{10} * 10^1 + \mathbf{15} * 10^2 + \mathbf{9} * 10^3 = 10606 \qquad (1\text{-}8)$$

Generally, redundant representation means more storage for the same range of numbers. The gain of redundant representation, however, is that it allows for a carry-free parallel addition. In the last example of Figure 1-8, each two digits belonging to the same arithmetic position can be added in parallel with digits from other position. Hence, the latency of addition becomes the latency of adding two one-digit numbers only. It is independent on the width of the operands, i.e. adding two 4-digit number takes the same time as adding two 40-digit numbers.

The problem with the last example is that it does not allow for consecutive additions, because of our assumption that the digit-set of the operands is [0, 9]. To allow for consecutive additions, and efficiently make use of the speed gains of redundancy, the digit-set of the operands must be the same as the digit-set of the result.

To achieve this, assume we use an operand digit-set [0, 18]. For consecutive additions, we need a result whose digits belong to the digit-set [0, 18] as well.

The question which arises is: what if the sum of the two digits is greater than 18? For example, adding the digit (18) to itself gives (36) which is not a digit in our digit-set. However, the number (36) can be represented by a digit (16) and a (2) added to one position left to the digit (16). This next example illustrates how to handle results that are greater than the maximum digit in the digit-set.

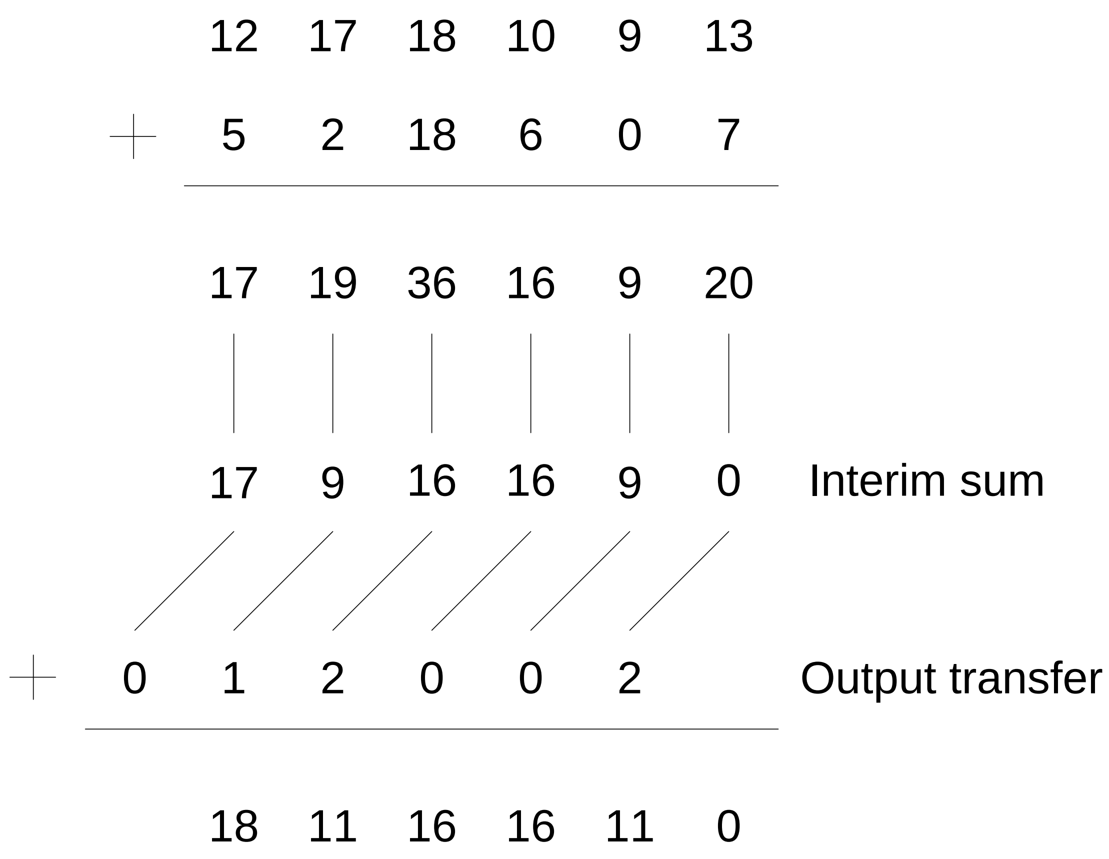


Figure 1-9 Adding two decimal numbers with the digit-set [0, 18]

As illustrated in Figure 1-9, adding two digits gives an interim sum and an output transfer such that

$$sum_i = 10 * t_{i+1} + w_i \qquad (1\text{-}9)$$

where

sum_i is the sum of adding the two digits at position i,

t_{i+1} is the transfer digit that is transferred to the position i+1,

and w_i is the interim sum produced at position i.

Each two digits of a certain position *i* are added to produce an interim sum at position *i*, and a transfer digit to the position *i+1*. All the digits of the operands are added in parallel leaving us with two vectors: an interim sum vector, and a transfer vector. Adding these two vectors together gives the final result.

As long as the digits of the interim sum belongs to the set [0, 16], and the transfer digit belong to the set [0, 2], then adding the two vectors gives the results whose digits belong to the digit-set [0, 18] as required. This allows for successive additions at the expense of an additional adder that adds the interim sum vector and the transfer vector.

1.6.1 Signed-Digit Redundant Representation

In the early 1960's, Avizienis [17] defined a class of signed-digit number systems with symmetric digit set $[-\alpha, \alpha]$ and radix $r > 2$, where α is any integer in the range $\lfloor r/2 \rfloor + 1 \leq \alpha \leq r-1$. These number systems allow at least $2\lfloor r/2 \rfloor+3$ digit values, which is larger than the conventional r digit values. Hence, these number systems are redundant.

Avizienis states that for a parallel carry-free addition/subtraction using the signed-digit representation, the following conditions have to be met:

1. For a radix-r system, each digit can assume q values such that $\boldsymbol{r + 2 \leq q \leq 2r - 1}$
2. Each digit can assume positive and negative integer values, and contains the sign information of the number.
3. Zero has a unique representation.
4. There exist transformations from the redundant representation to the conventional representation.

1.6.2 Redundant Decimal Integer Adders

There are many integer adders /subtractors in literature that used the signed-digit representation introduced by Avizienis to implement parallel carry-free addition.

Shirazi et al. in 1989 [18] proposed a redundant binary coded decimal (BCD) adder. He introduces a VLSI design of a Redundant BCD (RBCD) adder. The design consists of two small PLA's and two 4-bit binary adders for one digit of the RBCD adder. The time delay of the RBCD adder is of course independent of the width of the BCD operands.

The RBCD adder uses a digit-set [-7, 7]. This digit-set satisfies the conditions stated by Avizienis for carry-free addition. The digits are encoded in their two’s complement format as shown in Table 1-5. The interim sum digit-set is [-6, 6] and the transfer digit-set is [-1, 1].

Table 1-5 RBCD digits

Digit	**RBCD**	**Digit**	**RBCD**
0	0000		
1	0001	-1	1111
2	0010	-2	1110
3	0011	-3	1101
4	0100	-4	1100
5	0101	-5	1011
6	0110	-6	1010
7	0111	-7	1001

The algorithm used is:

1. Add the two RBCD digits using conventional 4-bit binary adder.
2. Compare the result from above with ±7. If the result is larger or equal to +7, then set the transfer digit to be 1. If the result is less than or equal -7, then set the transfer digit to be -1. If the result lies between -7 and +7, set the transfer digit to be 0.
3. Generate the interim sum *w* from the result produced from step 1 and the transfer digit *t* of step 2 such that: $\boldsymbol{w = r - 10t}$
4. Add the interim sum vector and the transfer vector to get the final RBCD result using another 4-bit conventional adder.

Table 1-6 and Table 1-7 show the RBCD addition table for the RBCD adder.

Table 1-6 RBCD addition table part 1

	-7	**-6**	**-5**	**-4**	**-3**	**-2**	**-1**
-7	(-1)(-4)	(-1)(-3)	(-1)(-2)	(-1)(-1)	(-1)0	(-1)1	(-1)2
-6	(-1)(-3)	(-1)(-2)	(-1)(-1)	(-1)0	(-1)1	(-1)2	(-1)3
-5	(-1)(-2)	(-1)(-1)	(-1)0	(-1)1	(-1)2	(-1)3	0(-6)
-4	(-1)(-1)	(-1)0	(-1)1	(-1)2	(-1)3	0(-6)	0(-5)
-3	(-1)0	(-1)1	(-1)2	(-1)3	0(-6)	0(-5)	0(-4)
-2	(-1)1	(-1)2	(-1)3	0(-6)	0(-5)	0(-4)	0(-3)
-1	(-1)2	(-1)3	0(-6)	0(-5)	0(-4)	0(-3)	0(-2)
0	(-1)3	0(-6)	0(-5)	0(-4)	0(-3)	0(-2)	0(-1)
1	0(-6)	0(-5)	0(-4)	0(-3)	0(-2)	0(-1)	00
2	0(-5)	0(-4)	0(-3)	0(-2)	0(-1)	00	01
3	0(-4)	0(-3)	0(-2)	0(-1)	00	01	02
4	0(-3)	0(-2)	0(-1)	00	01	02	03
5	0(-2)	0(-1)	00	01	02	03	04
6	0(-1)	00	01	02	03	04	05
7	00	01	02	03	04	05	06

Table 1-7 RBCD addition table part 2

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
-7	(-1)3	0(-6)	0(-5)	0(-4)	0(-3)	0(-2)	0(-1)	00
-6	0(-6)	0(-5)	0(-4)	0(-3)	0(-2)	0(-1)	00	01
-5	0(-5)	0(-4)	0(-3)	0(-2)	0(-1)	00	01	02
-4	0(-4)	0(-3)	0(-2)	0(-1)	00	01	02	03
-3	0(-3)	0(-2)	0(-1)	00	01	02	03	04
-2	0(-2)	0(-1)	00	01	02	03	04	05
-1	0(-1)	00	01	02	03	04	05	06
0	00	01	02	03	04	05	06	1(-3)
1	01	02	03	04	05	06	1(-3)	1(-2)
2	02	03	04	05	06	1(-3)	1(-2)	1(-1)
3	03	04	05	06	1(-3)	1(-2)	1(-1)	10
4	04	05	06	1(-3)	1(-2)	1(-1)	10	11
5	05	06	1(-3)	1(-2)	1(-1)	10	11	12
6	06	1(-3)	1(-2)	1(-1)	10	11	12	13
7	1(-3)	1(-2)	1(-1)	10	11	12	13	14

Figure 1-10 shows the block diagram of the RBCD adder.

Let A = $a_3a_2a_1a_0$ and B = $b_3b_2b_1b_0$ be the two RBCD digits to be added. Let the binary sum of A and B is S such that S = $s_3s_2s_1s_0$. We have three cases for the sum:

- **Case 1:** The sum lies in the range [-6, 6]. This is the final result and the sum does not need correction. The transfer digit in this case = 0.
- **Case 2:** The sum lies in the range [7, 14]. In this case, the sum should be corrected by adding 6 (denoted by $\boldsymbol{f_6}$) and sending a carry to the next digit (denoted by $\boldsymbol{f_{c=1}}$)

$$\boldsymbol{f_6 = f_{c=1} = (s_3 + s_2s_2s_0)\overline{a_3}\overline{b_3}} \qquad (1\text{-}10)$$

- **Case 3:** The sum lies in the range [-14, -7]. In this case, the sum should be corrected by subtracting 6 (denoted by $\boldsymbol{f_{\bar{6}}}$) and sending a borrow to the next digit (denoted by $\boldsymbol{f_{c=\bar{1}}}$)

$$\boldsymbol{f_{\bar{6}} = f_{c=\bar{1}} = a_3b_3(\overline{s_3} + \overline{s_2s_1s_0}) + (a_3 + b_3)(s_3\overline{s_2s_2s_0})} \qquad (1\text{-}11)$$

It is obvious that the sum must be corrected by adding -6, 0, or 6. Generally, this digit accepts a transfer digit also which is 0, 1, or -1. The correction digit thus belongs to the set {-7, -6, -5, -1, 0, 1, 5, 6, 7}. Another 4-bit binary adder is used to add the sum to the correction digit (W= $w_3w_2w_1w_0$).

$$w_3 = f_{c=\bar{1}}\overline{f_6} + f_{\bar{6}} \qquad (1\text{-}12)$$

$$w_2 = f_{c=\bar{1}}\overline{f_{\bar{6}}} + f_6 \qquad (1\text{-}13)$$

$$w_1 = \overline{f_{\bar{6}}}\,\overline{f_6}f_{c=\bar{1}} + f_6\overline{f_{c=\bar{1}}} + \overline{f_{c=\bar{1}}}f_6 \qquad (1\text{-}14)$$

$$w_0 = f_{c=1} + f_{c=\bar{1}} \qquad (1\text{-}15)$$

In Figure 1-10, PLA1 is responsible for generating $\boldsymbol{f_6}$, $\boldsymbol{f_{\bar{6}}}$, $\boldsymbol{f_{c=1}}$, and $\boldsymbol{f_{c=\bar{1}}}$. PLA2 is responsible for generating the correction digit W.

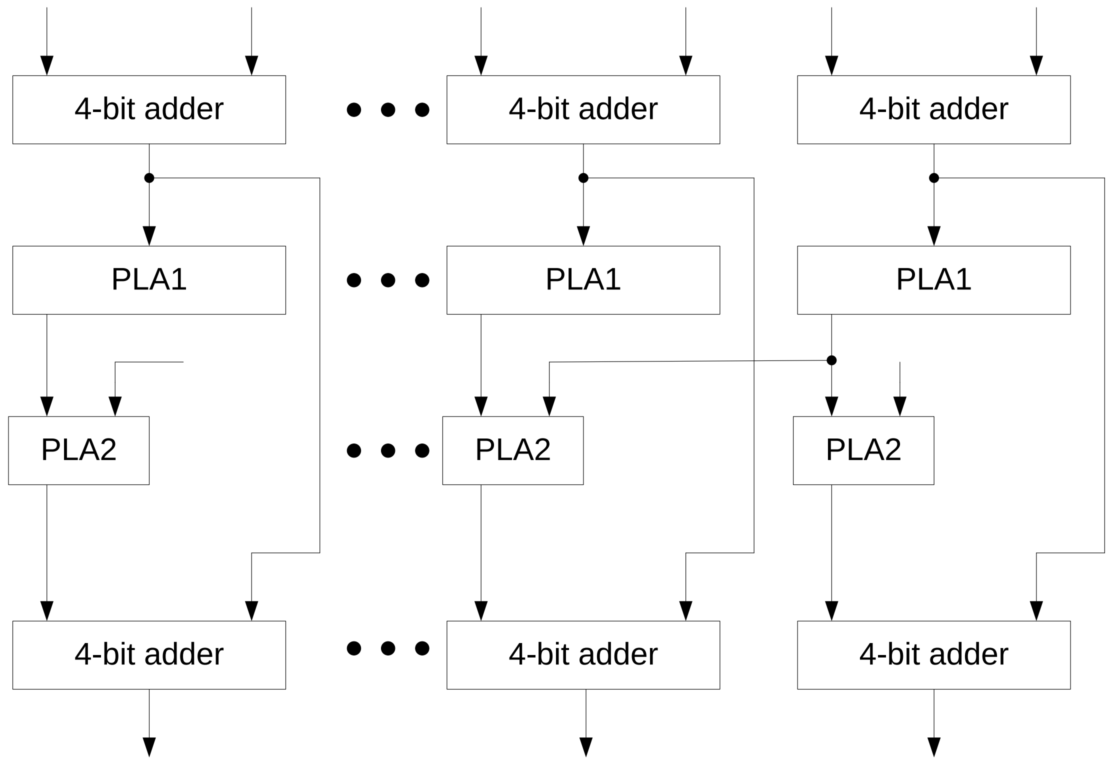


Figure 1-10 RBCD Adder

Another recent redundant decimal integer adder is proposed by Saeid Gorgin and Ghassem Jaberipur [19]. The decimal signed-digit set is [-7, 7]. This digit-set is called a decimal septa signed-digit (DSSD). Each DSSD is represented as a two's complement number. The method they used is that they tried to divide the two operands into two groups of numbers containing posibits and negabits then gradually adding these groups until they reach a final sum. Table 1-8 shows that, according to [18], the DSSD adder provides the least delay and the least area of all the signed-digit redundant adders (SD adders) encountered in literature. These adders were synthesized using Synopsis Design Compiler. The target library was based on TSMC 0.13 um standard CMOS technology.

Table 1-8 delay/area for SD decimal adders

Adder; reference	**Digit set**	**Delay (ns)**	**Ratio**	**Area (um^2)**	**Ratio**
Svoboda; [20]	[-6,6]	2.5	2.87	781	1.25
RBCD; [18]	[-7, 7]	1.22	1.4	668	1.07
Nikmehr; [21]	[-9, 9]	1.39	1.6	2333	3.75
Moskal; [22]	[-9, 9]	1.65	1.9	2112	3.39
DSSD; [19]	[-7, 7]	0.87	1	622	1

1.7 Outline of the thesis

The rest of the thesis is organized as follows: Chapter 2 discusses the proposed IEEE-compliant redundant decimal floating-point adder, the algorithm used and the block diagram of the adder. In chapter 3, we discuss how the decimal floating-point adder proposed in chapter 2 can be extended to incorporate binary addition as well. The decimal floating-point adder presented in Chapter 2 and the mixed floating-point adder presented in Chapter 3 were synthesized using Design Compiler and the TSMC 65 nm LP technology. Chapter 4 shows the synthesis results for each of them and discusses the future work.

Chapter 2

2. The Proposed Redundant Decimal FP Adder

As was the case with binary floating-point adders, starting with a simple hardware implementation of the most basic algorithm, then attempting to improve the latency of the FP adders either by subtly changing the algorithm as in the double-path implementation, or by changing the representation of the binary floating-point data as in redundant representations, decimal floating-point adders started also with the most simple implementation of the basic algorithm, followed by some attempts to improve the latency of the decimal floating-point adders. However, to the best of my knowledge, our proposed decimal floating-point adder is the first *redundant* decimal floating-point adder implementation.

The design is based on Avizienis signed-digit redundant representation. The digit-set [-6, 6] was chosen to represent the decimal floating-point data.

A decimal floating-point data is stored in memory in the densely packed decimal (DPD) format. A decimal floating-point number is converted into our proposed signed-digit redundant representation when brought from memory. A decimal floating-point number maintains this redundant representation inside the floating-point registers. It is converted back to the densely packed format when it is written back to memory.

2.1 Design and Implementation

Our representation uses a digit-set [-6, 6] encoded in the two's complement format instead of the conventional [0, 9] representation which does not allow for a carry-free addition/subtraction [23].

The design implements the IEEE decimal64 format; however, it can be easily extended to implement the decimal128 format or any other format. The precision for the decimal64 format is 16, i.e. the significand has 16 digits.

In the decimal64 format, the largest significand is 9999999999999999. To represent the decimal64 16-digit significand in our redundant format with digits in the range [-6, 6], we should be able to represent all the floating-point numbers that can be represented with the conventional [0, 9] digit-set. To achieve this, we will need to append another digit to the 16-digit significand. We will call this new digit, the “addendum”.

Table 2-1 2's complement encoding of the digit-set [-6, 6]

Digit	Encoding	Digit	Encoding
-6	1010	1	0001
-5	1011	2	0010
-4	1100	3	0011
-3	1101	4	0100
-2	1110	5	0101
-1	1111	6	0110
0	0000		

To eliminate the time consumed for leading zero detection, our representation includes the leading zero count (LZC) so that we do not have to count the lading zeros before determining the left shift amount and the right shift amount.

We also use explicit bits to indicate special values like infinities and NaNs (not a number). This facilitates handling the addition process of special cases.

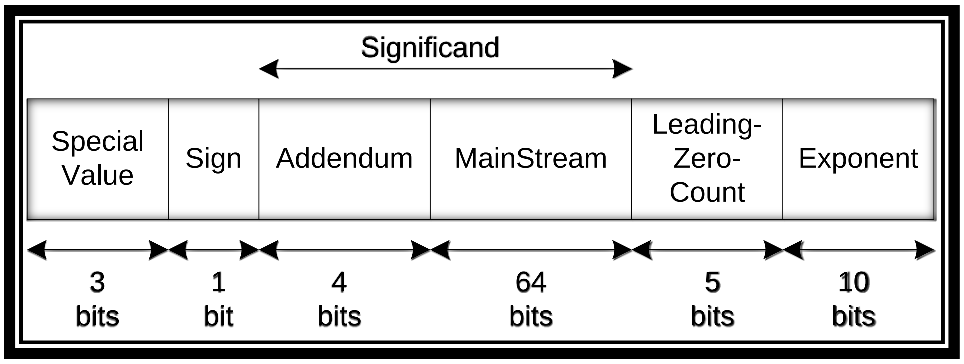


Figure 2-1 the proposed internal representation

Figure 2-1 presents our proposed internal representation which is divided to several fields:

- **Special Value:** It is used to indicate if the floating-point number is a special case. Special cases include infinities, signaling and quiet NaNs and Zeros. Special cases' operations are handled separately in our design. Special cases are encoded in three bits to differentiate among five different situations: No special case, Infinity, Signaling NaN, Quiet NaN, and Zero.

- **Sign:** This bit is used to store the sign of the number. The significand is always positive. A negative significand is not allowed; therefore the sign of the decimal floating-point number is explicitly represented in the *sign* bit.
- **Addendum:** This is the most significant digit (MSD) of the significand. It is used to allow for the representation of all the significands representable in the decimal64 format. This digit can only take two values: zero or one.
- **MainStream:** The MainStream is 64-bit wide (16 decimal digits). Together with the Addendum, they represent the significand of the floating-point number.
- **Leading-Zero Count:** It stores the leading-zero count of the operand. This eliminates the need for detecting the leading-zeros of the operands which improves the overall latency of the adder.
- **Exponent:** This field stores the exponent of the floating-point number. The minimum exponent is -398 and the maximum exponent is 369. The exponent is represented by a 10-bit biased representation.

2.2 Block Diagram and the Algorithm

Figure 2-2 shows the proposed block diagram of the decimal floating-point adder.

First, the exponent difference between the two operands is computed on the top right of the figure. If the first exponent EA is smaller than the second exponent EB, the swap flag is raised in order to swap the operands. If not, the operands remain unswapped.

The absolute difference of the exponents is used as well for the computation of the amounts by which the large operand (the one with the greater exponent) should be shifted left, and the small operand (the one with the smaller operand) should be shifted right for the purpose of aligning the two operands. The small operand should keep track of the conventional guard, round digits and sticky bit. However, we also need a *Sticky-Sign* bit that indicates whether the number to the right of the guard digit is positive or negative which is important during the rounding process. The aligned operands are then fed into the signed-digit redundant adder/subtractor.

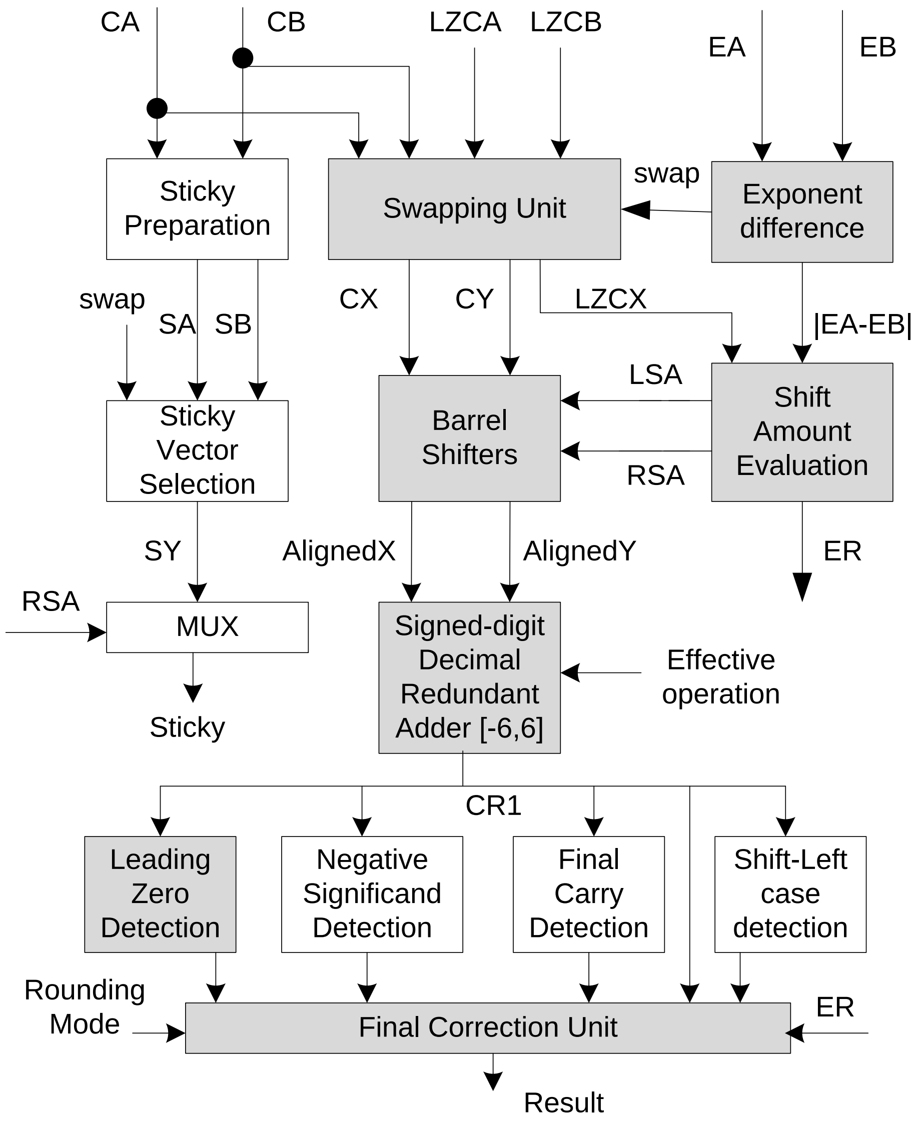


Figure 2-2 block diagram of the proposed decimal FP adder

The signed-digit cell is similar to that of [18] except for the fact that this adder works on a [-6, 6] digit-set. The signed-digit cell works as follows:

1- Add the two corresponding digits conventionally using a conventional 4-bit binary adder to produce an interim sum, concurrently add the input transfer digit to all the potential correction digits {+10, -10, 0} to get the final potential correction digits.

2- Compare the interim sum with ±6. If the interim sum is greater than or equals +6 the output transfer is set to 1. If it is less than or equals -6 the output transfer is set to -1. Otherwise, the output transfer is set to 0.

3- The output transfer digit selects one of the final potential correction digits and passes it to another conventional 4-bit binary adder which adds it to the interim sum to produce the final sum.

The adder handles the special cases as well (if any) and produces the intermediate result CR1. CR1 however is not suitable for rounding yet for the following reasons:

- CR1 might be a negative significand.
- CR1 might contain a final carry.
- In some cases, CR1 should be shifted left to select the appropriate cohort suggested by the IEEE standard. We will call this the *shift-left* case.

2.2.1 Negative Significand Detection

A negative significand might occur only in case of an effective subtraction. The detection of a negative significand in our redundant case is more difficult than that of the conventional case. To detect a negative significand, the sign of the most significant non-zero digit should be examined, and this is equivalent to a carry-propagation delay as we do not know apriori which digit contains the sign information of the significand. However, the gain is that if the significand is found to be negative, converting it to its positive counterpart is done in a fixed amount of time regardless of the width of the significand as all we have to do is to get the two's complement of each digit independently contrary to the conventional conversion which requires a carry propagation delay.

Note that a negative significand indicates that there is no rounding needed, because for CR1 to be negative, it is impossible that the operand with the smaller exponent was shifted to the right. Hence, the guard, round, and sticky are all zeros. We can start rounding right away after the generation of CR1 assuming a positive significand while detecting the sign of the significand concurrently.

2.2.2 Final Carry Detection

A final carry means that the result CR1 is larger than the 16-digit significand of the IEEE format. In conventional adders, the final carry is explicitly represented by a carry originating from the addition of the most significant digits of the two significands. In our redundant case, a final carry occurs if:

- Effective operation is addition.

- Addendum Value = 2

For instance, If CR1 = $2\overline{6}\overline{6}\overline{6}\overline{6}\overline{6}\overline{6}\overline{6}\overline{6}\overline{6}\overline{6}\overline{6}\overline{6}\overline{6}\overline{6}\overline{6}\overline{6}$ which is the minimum number that has an addendum value of 2. CR1 in this case is equal to $(13333333333333334)_{10}$ which is greater than the maximum representable significand in the IEEE format. Hence, an addendum value of 2 indicates a final carry case.

- Addendum Value = 1 and the most significant non-zero digit is not-negative

For instance, if CR1 = 10000000000000001 which is the minimum number that has an addendum value of 1 and a positive most-significant non-zero digit. CR1 in this case is equivalent to $(10000000000000001)_{10}$ which indicates a final carry as well. The same applies if CR1 = 10000000000000000.

In case of a final carry, the significand should be shifted right, the exponent should be incremented by 1, and then rounding should be performed according to the new *Guard*, *Round*, and *Sticky*. If the exponent is already the maximum allowable exponent, the number remains as is and the Final Correction block either sets the final result to infinity or the maximum representable floating-point number depending on the rounding mode.

2.2.3 Shift-Left Case Detection

Figure 2-3 shows an example that illustrates the shift-left case. For simplicity, we assume that the significand is composed of an addendum and a 3-digit MainStream

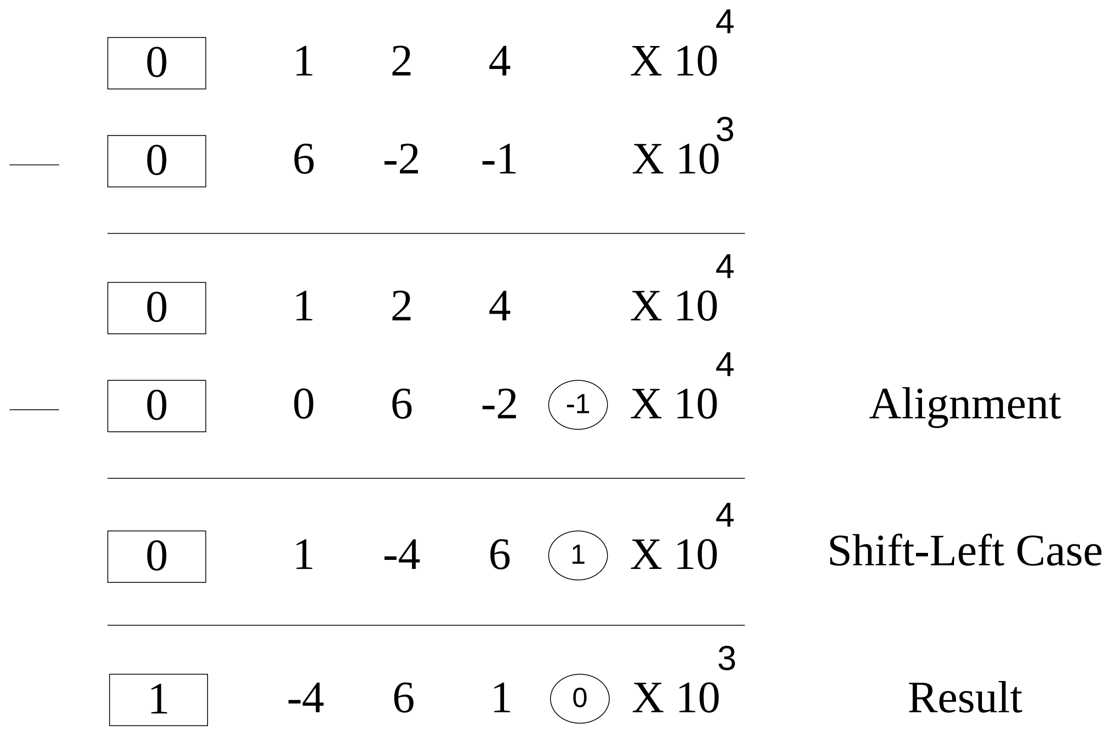


Figure 2-3 shift-left case example

A left-shift to CR1 might be necessary to make the result exact or to approach the minimum possible exponent (The preferred exponent of the result is the smaller exponent according to the IEEE 754-2008 standard).

If the following conditions are satisfied, then a *Shift-Left* is the case:

- Effective operation is subtraction.
- Current exponent of CR1 is greater than the preferred exponent (the smaller exponent).
- Addendum of CR1 = 0.
- MainStream has at least one leading zero.[1]

If a shift-left case is detected, CR1 should be shifted one position to the left before performing rounding.

[1] Leading zeros might be explicit zeros or implicit ones. Note that a most significant digit of ‘1’ followed by a negative digit as in Figure 2-3 indicates that one leading zero exists. Since our digit-set is [-6, 6], it is not possible that more than one implicit zeros exist. It is either one implicit zero, or no implicit zeros at all.

2.2.4 Handling Special Cases

The adder also handles special cases in accordance with the IEEE standard.

If any of the operands or both of them are NaNs, either a signaling NaN or a quiet one, CR1 is a quiet NaN.

If any of the operands is infinity and the other operand is a normal number, then CR1 is infinity, and the sign of CR1 is the same as the sign of infinity if the infinity is the first operand (before swapping). The sign of CR1 is the xor[2] between the infinity sign and the operation if infinity is the second operand (before swapping).

If both operands are infinities, then CR1 is infinity if the effective operation is addition. If the effective operation is subtraction, then CR1 is a quiet NaN.

2.2.5 Sticky Generation

The purpose of sticky generation is to produce a *sticky bit* and a *sticky sign*. A sticky sign is important as to tell whether the shifted-out digits adds or subtracts from the current number CR1. This is important for correct rounding.

In a conventional decimal floating-point adder, the sticky bit generation is not in the critical path as it is performed concurrently with the conventional addition of the aligned significands. However, in our redundant floating-point adder, the conventional sticky bit generation would be in the critical path thus eliminating the speed improvement gained by redundancy. For this reason, the sticky generation starts as soon as the operands arrive. The Sticky-sign and Sticky-bit are generated for every possible value of the right-shift-amount which is fed into the selection lines of a multiplexer to choose the appropriate *sticky-sign* and *sticky-bit*.

[2] We assume that a negative sign and subtraction operation are encoded as a logic 1, whereas a positive sign and addition operation are encoded as a logic 0.

2.3 Rounding

From the analysis discussed in the previous sections, it is clear now that the rounding of CR1 should not be performed right away, CR1 should be checked first to discover if it has a *final carry*, or if there is a *Shift-Left* case.

Generally, we have one of three possibilities:

- Rounding CR1 as is.
- Shifting CR1 to the left before performing the rounding.
- Shifting CR1 to the right before performing the rounding.

Normally, Rounding can be performed after examining CR1. However, in our design, rounding is performed immediately after the generation of CR1 for the three possibilities discussed above. Fortunately, we do not have to replicate CR1 entirely three times for this purpose as CR1 has a feature that guarantees that the rounding process will, at most affect, the least significant two digits of the MainStream.

Lemma:

The result of signed-digit decimal addition of the digit set [-6, 6] never has two consecutive `6's or `-6's.

Proof:

Assume A and B are the numbers fed into the signed-digit redundant adder where $A = a_n a_{n-1} a_{n-2} \dots \dots a_0$, and $B = b_n b_{n-1} b_{n-2} \dots \dots b_0$.

Let S_{j+1} and S_j be the interim sums at positions j+1 and j respectively. Also, let t_{j+1} and t_j be the input transfers at positions j+1 and j respectively.

Remember that the interim sum belongs to {-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5}, and that a transfer digit belongs to {-1, 0, 1}.

Assuming that the final sum m_j at position j is equal to 6, this can only occur if $S_j = 5$ and $t_j = 1$. However, if $S_j = 5$, there is no way $t_{j+1} = 1$, therefore two consecutive ‘6’s can never occur. Similarly, we can prove that two consecutive ‘-6’s can never occur.

According to the previous lemma, only the two least significant digits of the MainStream are rounded for each of the three cases discussed above which reduces the interconnect area drastically. Figure 2-4 shows the three blocks responsible for the rounding process for each case discussed above. Each block accepts the appropriate *Rounding Digits* (the digits that are examined to determine the *Rounding Decision*), and the *Rounding mode*. It outputs the *Rounding Decision* and the expected two least significant digits of the result (those digits affected by the rounding process). The outputs of these three rounding blocks are passed to the *Final Correction Block* to produce the final result. Rounding decisions can be made according to Table 2-2 and Table 2-3.

In Table 2-2, the "Pivot" is the *Guard* digit if it is a non-zero digit. It is the *Round* digit if the *Guard* digit is zero.

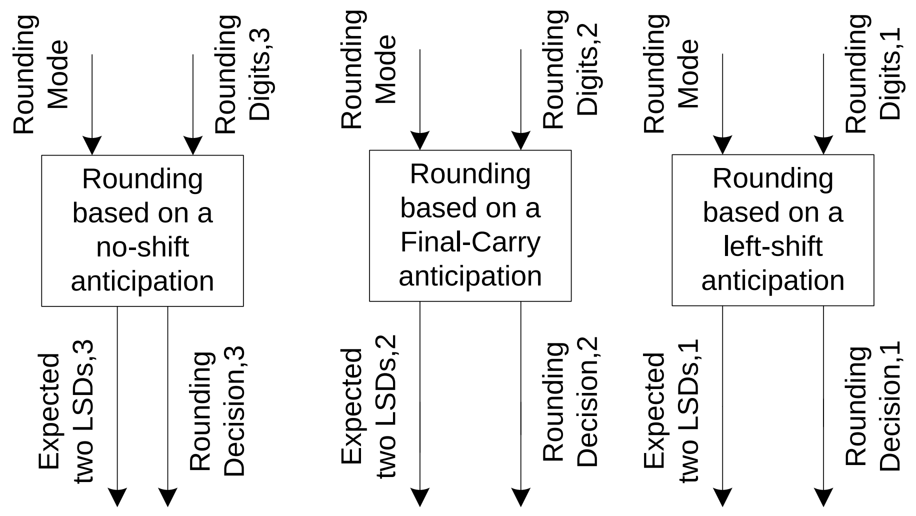


Figure 2-4 rounding block diagram

Table 2-2 rounding to positive infinity (RP) and rounding to negative infinity (RN)

Sign	**Pivot**	**Sticky**	**RP Decision**	**RN Decision**
X	0	0	0	0
+	0	+1	+1	0
-	0	+1	0	+1
+	0	-1	0	-1
-	0	-1	-1	0
+	>0	x	+1	0
+	<0	x	0	-1
-	>0	x	0	+1

-	<0	x	-1	0

In case of a Round-toward-zero (RZ) mode, the decision is the same as the decision of RN if the sign of the number is positive. If the sign is negative, the RZ decision is the same as RP decision. For a Round-away-from-zero mode, the decision is the same as the decision of RN if the sign is negative. It is the same as the decision of RP if the sign is positive.

Table 2-3 round to nearest tie to even

Guard	**Round**	**Sticky**	**LSD**	**Decision**
>5	x	x	x	Round away from zero
0<Guard<5	x	x	x	Round toward zero
5	<0	x	x	Round toward zero
5	>0	x	x	Round away from zero
5	0	+1	x	Round away from zero
5	0	-1	x	Round to zero
5	0	0	even	0 (tie case)
5	0	0	odd	+1 (tie case)
-5<Guard<0	x	x	x	Round away from zero
<-5	x	x	x	Round toward zero
-5	>0	x	x	Round away from zero
-5	<0	x	x	Round toward zero
-5	0	+1	x	Round away from zero
-5	0	-1	x	Round toward zero
-5	0	0	even	0 (tie case)
-5	0	0	odd	-1 (tie case)

The other Rounding-to-nearest modes are exactly like Table 2-3 except for the tie case, the exact rounding mode decides the appropriate action taken in case of a tie.

2.4 Simulation and Results

The proposed design was simulated using the test vectors suggested by IBM and another test vectors developed in Cairo University by Amr A. R. Sayed-Ahmed for the decimal64 IEEE format [24], a software tool was developed to convert the standard representation to our proposed internal representation, then the output of the decimal floating-point adder is converted back to the standard representation and then they are compared.

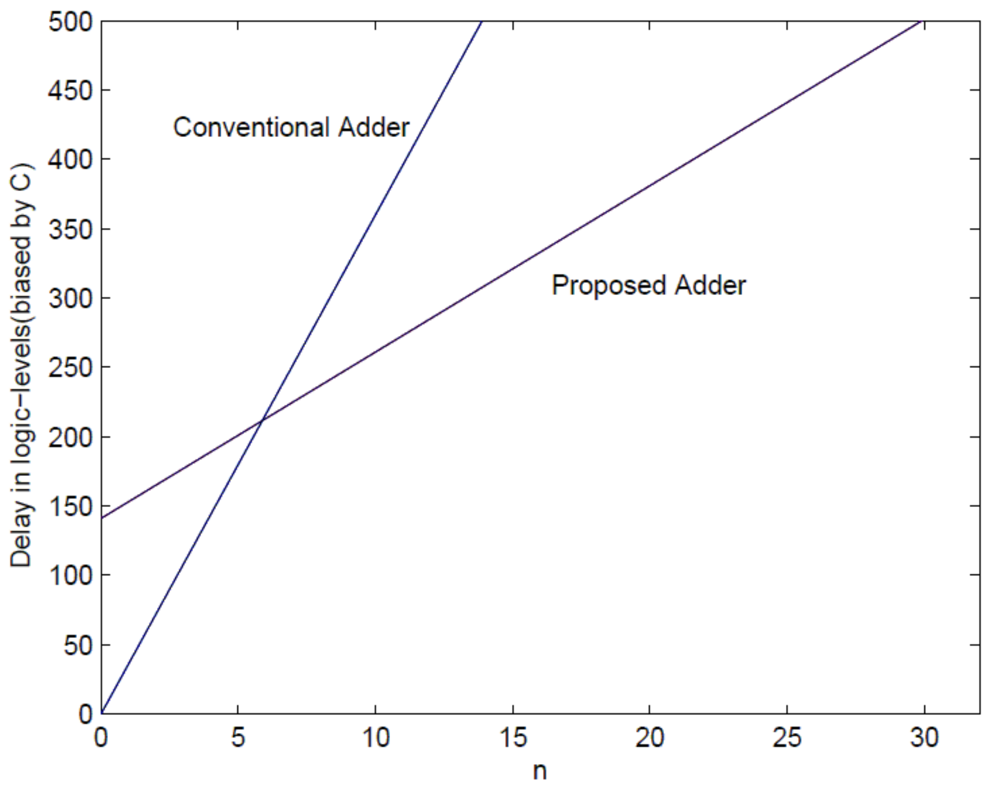
The critical path of our design is the path connecting the shaded boxes in Figure 2-2. An estimate of the delay of our proposed design and a comparison with the conventional one is performed based on the following assumptions:

- The delay estimation is performed at the logic-level. This will simplify the process of estimation. However, it does not give a very accurate estimation of the delay because many of the circuit-level effects, like the interconnect delays, for instance, are not taken into consideration.
- For a fair comparison, we assume that only two-input gates are available. We also assume that the delay of a Leading-Zero Detector and Sticky-Generation of an n-bit number is equivalent to the delay of n-bit carry propagation although there are novel faster techniques for implementing such circuits.
- The BCD adder of the conventional decimal floating-point adder is assumed to be a ripple-carry adder that would also take a delay of an n-bit carry propagation ignoring the delay required for digit correction. If another faster adder is used, a carry look-ahead adder for instance, this can be compensated for in our design by using a faster Leading-Zero Detector.
- A one bit carry-propagation delay is 3 logic-levels assuming two-input gates.

The critical path of a conventional decimal floating-point adder comprises the following: a Leading-Zero Detector for the operands, the swapping unit, the barrel shifter, the BCD adder and the rounding. The critical path of our design comprises the following: the exponent difference block, the swapping unit, the shift amount evaluator, the barrel shifter, the redundant adder, the leading-zero detector and the final correction unit. It is obvious that the swapping unit and the barrel shifter are present in both critical paths so there is no need to compute their delays. We will assume that their collective delay is C logic-levels. The exponent difference block latency is equivalent to a 10-bit carry propagation delay, hence 30 logic-levels. The shift amount evaluator is constrained by the evaluation of the Right Shift Amount (RSA). The evaluation of an RSA requires two 10-bit adders and two multiplexers, hence 64 logic-levels. The redundant adder latency is 41 logic-levels. It does not depend on the width of the operands. The leading-zero detector (LZD) is the only block whose latency depends on the width of the operands. Assuming that the significand is n decimal digits, the latency of the LZD is 4n*3=12n. The Final

Correction unit latency is the latency of one 4*1 multiplexer and one 2*1 multiplexer, hence 6 logic-levels. The total delay of the critical path is (C+141+12n) logic-levels. For the conventional decimal floating-point adder, the LZD, the adder and the rounding unit each has a latency of (4n*3) logic-levels, so ignoring the final-correction unit of the conventional floating-point adder, and the digit-correction of the BCD adder. The delay would be (C+36n) logic-levels.

Table 2-4 delay comparison between the proposed design and the conventional one



Obviously, the proposed design gives a better performance for both decimal64 and decimal128 formats. However, as n gets larger, the critical path of the adder proposed changes to include the sticky-generation block which is definitely dependent on the width of the operands, so the slope of the delay is supposed to increase when n becomes large enough.

2.5 Conclusion

In this chapter, a novel faster implementation of a decimal floating-point adder was proposed. The adder uses a special internal representation for the decimal floating-point numbers to improve the speed of addition. The significand of a decimal floating-point number is represented in a redundant format to allow for a parallel carry-free addition. Sticky generation starts as soon as the operands arrive and it is performed concurrently with other tasks. Rounding is also performed in parallel with leading-zero detection. Results show that as the precision of the floating-point number increases, our design outperforms the conventional decimal floating-point adder.

Chapter 3

3. Binary Extension of the Decimal FP adder

In this chapter, we are trying to extend the above redundant decimal floating-point adder to include binary64 floating-point addition as well.

3.1 A Mixed Octal/Decimal Adder Cell

An integer mixed redundant octal/decimal adder/subtractor is proposed. This adder is the core unit of the mixed floating-point adder. The adder accepts the two digits to be added, namely ‘X’ and ‘Y’. It also accepts the input transfer digit, the radix of the required operation ‘B’, and ‘sub’ to specify whether the required operation is addition or subtraction. The adder performs signed-digit addition/subtractions based on the input radix ‘B’. It outputs an output transfer digit that takes a value from the set {-1, 0, 1}, and a sum that takes a value from the set {-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5}. First, we will discuss the implementation of the mixed adder. Then we will discuss how the adder was extended to accommodate subtraction as well.

3.1.1 Design and Implementation

The algorithm used is the same as that used in previous integer redundant adders. However, independent parts of the algorithm will run concurrently to reduce the delay of the addition process. We know beforehand that a correction digit of value 0, ±10, or ±8 has to be added to the interim sum, and then the result has to be added to the input transfer digit to get the final output sum. Instead, the input transfer digit can be added to all the possible correction digits concurrently with the conventional addition of the input digits to produce what we will call ‘modified correction digits’. The interim sum is then compared to the maximum and minimum digits in the digit-set so as to generate the output transfer digit and choose one of the modified correction digits. The modified correction digit is then added to the interim sum to produce the output sum digit.

The two inputs X and Y are two 4-bit digits encoded in the 16’s complement representation. A digit-set [-6, 6] is chosen for our redundant representation. A conventional 4-bit binary adder adds X and Y to produce the interim sum concurrently with the generation of the modified correction digits. Some flags are also evaluated as follows:

‘z1’ and ‘z2’ are flags that indicate whether the input digits are zeros or not. If X is of value zero, then ‘z1’ is raised. If Y is of value zero then ‘z2’ is raised.

‘eff_op’ is the effective operation. If both input digits have the same sign then the ‘eff_op’ is raised.

$$eff\_op = X(3) \oplus Y(3) \tag{3-1}$$

‘no_correction’ is a flag that is set if there is no correction digit required to be added to the interim sum. A potential correction might occur if the effective operation is addition (‘eff_op’ = 0) or if one of the two input digits are zeros regardless of the effective operation.

$$no\_correction = eff\_op\,.\overline{z1}.\overline{z2} \tag{3-2}$$

The ‘above_threshold’ flag is set if the interim sum is anything other than a positive digit ranging from [0, 5] while the ‘below_threshold’ flag is set if the interim sum is anything other than a negative digit ranging from [-5, -1].

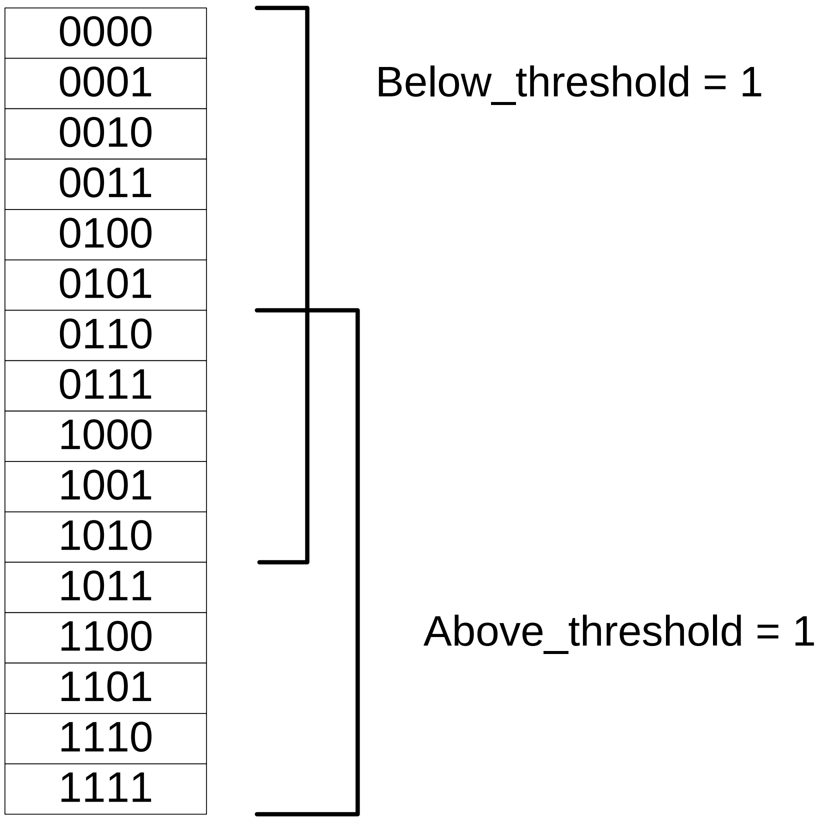


Figure 3-1 Interim Sum ranges for ‘Above_threshold’ and ‘Below_threshold’

A ‘need_correction’ flag is set if a correction digit is needed according to the following Boolean equation

$$need\_correction = \overline{no\_correction}\,[below\_threshold.X(3).Y(3) + above\_threshold(below\_threshold + \overline{X(3)})] \tag{3-3}$$

The ‘need_correction’ flag together with the ‘z1’, ‘z2’ flags and the most significant bits of X and Y are used to generate the output transfer digit according to the following Boolean equations:

$$tout\_posi = need\_correction[\overline{z1}.\overline{X(3)} + \overline{z2}.\overline{Y(3)}] \tag{3-4}$$
$$tout\_nega = need\_correction[\overline{z1}.X(3) + \overline{z2}.Y(3)] \tag{3-5}$$

The mathematical value of the output transfer digit is

$$output\ transfer\ digit = tout\_posi - tout\_nega \tag{3-6}$$

The output transfer digit and the radix of operation ‘B’ are used to pass one of the modified correction digits to the other 4-bit binary adder that adds this modified correction digit to the interim sum to output the final sum digit.

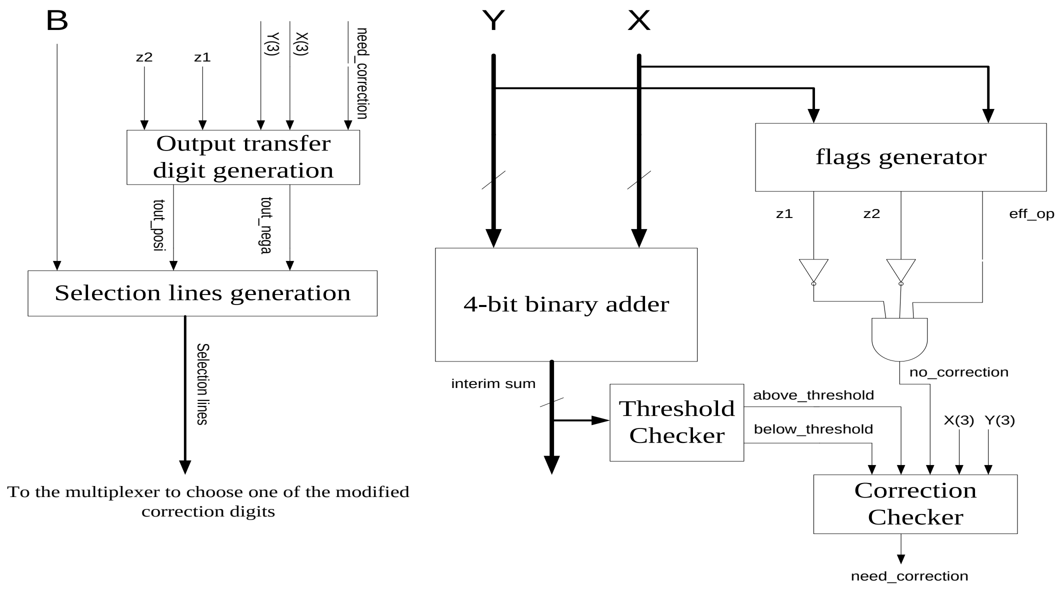


Figure 3-2 block diagram of the mixed adder

As stated above, the generation of the modified correction digits is performed concurrently with the above process. The block diagram of the modified correction digit formation is as follows.

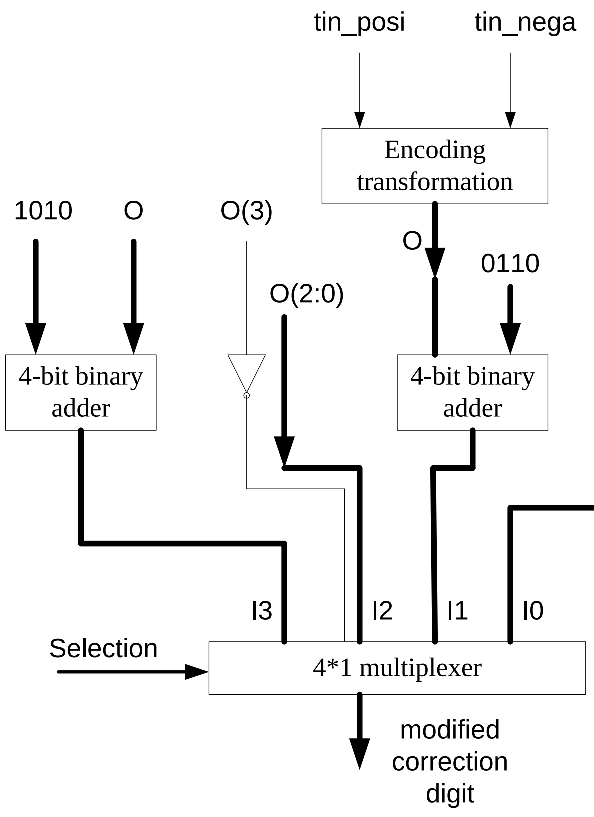


Figure 3-3 generation of the modified correction digit

O is the input transfer digit encoded in the 16's complement format

$$O(3) = O(2) = O(1) = tin\_nega.\overline{tin\_posi} \tag{3-7}$$
$$O(0) = tin\_posi \oplus tin\_nega \tag{3-8}$$

The modified correction digit is then added to the interim sum to generate the output sum.

3.1.2 Extending the Hardware to incorporate Subtraction

A fast and easy solution to incorporate subtraction into the previous redundant adder is to pass the 16's complement of the input Y if subtraction is the required operation. However, the 16's complement generation requires a 4-bit carry propagation delay. For the purpose of minimizing the delay required for extending the HW to include subtraction, the 15's complement only of the input Y is generated and then passed to the 4-bit binary adder. The remaining 'addition of one' is taken care of while generating the modified correction digit. This technique requires some modifications to the previous redundant adder. The thresholds will change as a result of using the 15's complement instead of the 16's complement.

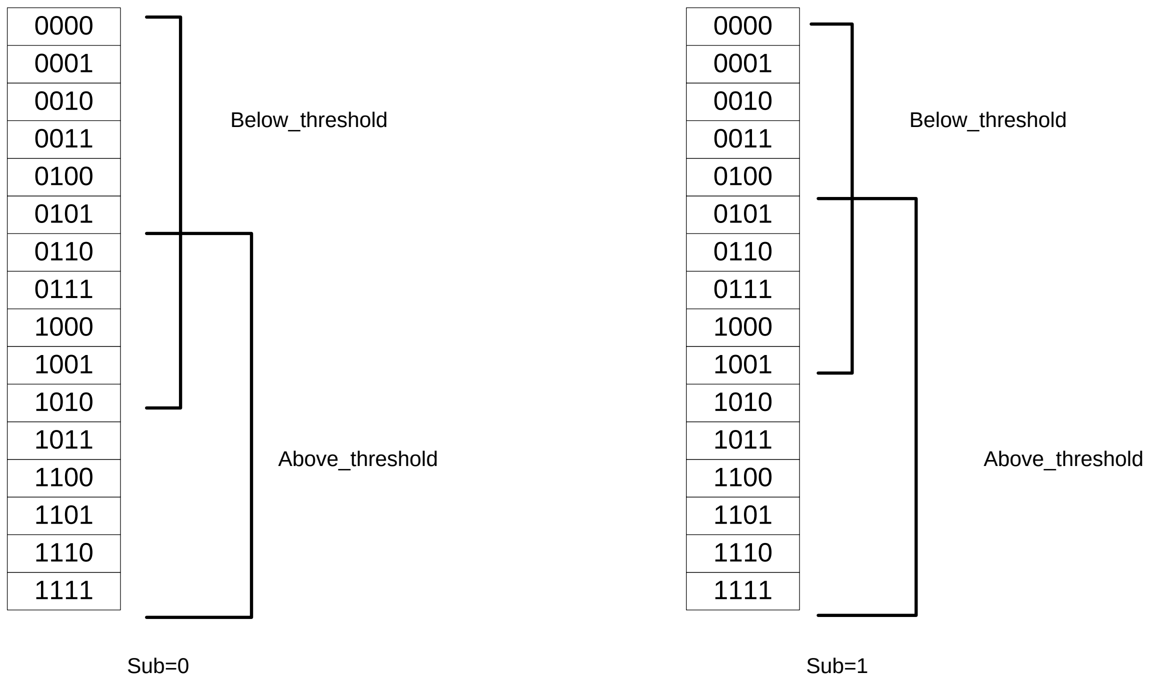


Figure 3-4 threshold ranges for addition and subtraction

'Sub' is an input that is set if the required operation is subtraction.

'Ym' is the 15's complement of the input Y.

These are the modification needed to include subtraction

$$\begin{aligned} above\_threshold &= (interim\_sum(1)\,.\,interim\_sum(2)\ ) \\ &+\ interim\_sum(3) \\ &+\ (sub\,.\,interim\_sum(0)\,.\,interim\_sum(2)) \end{aligned} \qquad (3\text{-}9)$$

$$\begin{aligned} above\_threshold &= (interim\_sum(1)\,.\,interim\_sum(2)\ ) \\ &+\ interim\_sum(3) \\ &+\ (sub\,.\,interim\_sum(0)\,.\,interim\_sum(2)) \end{aligned} \tag{3-10}$$

$$\begin{aligned} below\_threshold &= \overline{interim\_sum(3)} \\ &+ \overline{interim\_sum(2)}\,.\,\overline{interim\_sum(1)} \\ &+ \overline{interim\_sum(2)}\,.\,\overline{sub}.\,(interim\_sum(0) \\ &\oplus interim\_sum(1)) \end{aligned} \tag{3-11}$$

$$O(3) = (\overline{tin\_posi}\ ).\,(tin\_nega).\,\overline{sub} \tag{3-12}$$

$$O(2) = O(3) \tag{3-13}$$

$$O(1) = (tin\_nega \oplus sub)(tin\_posi \odot sub) \tag{3-14}$$

$$\begin{aligned} O(0) &= \overline{tin\_posi}.\,\overline{tin\_nega}.\,sub + \overline{tin\_posi}.\,tin\_nega.\,\overline{sub} \\ &+ tin\_posi.\,\overline{tin\_nega}.\,\overline{sub} + tin\_posi.\,tin\_nega.\,sub \end{aligned} \tag{3-15}$$

The output transfer digit and the ‘need_correction’ flag are generated using the same Boolean equations of the previous redundant adder except for changing every ‘Y’ with its 15’s complement ‘Ym’.

3.1.3 Simulation and Testing

The redundant mixed adder/subtractor was exhaustively tested using Mentor Graphic’s ModelSim, and it is giving correct results for all the possible inputs.

3.2 Mixed Binary/Decimal Floating Point Adder

A mixed binary/decimal floating-point adder is proposed. The floating-point adder accepts two operands, and then adds or subtracts these two operands according to the specified operation and radix. The floating-point adder can perform decimal addition compliant with the IEEE decimal64 format, or binary addition compliant with the IEEE binary64 format. We have discussed how decimal floating-point addition is performed in the previous chapter. In this chapter, we will discuss how the decimal floating-point adder was modified in order to accommodate binary floating-point addition as well. The binary64 floating-point numbers representing the input operands are first converted to a redundant octal format where each digit assumes a value from the digit-set [-6, 6]. Then, the operands propagate through the adder naturally until it reaches the rounding stage. Rounding algorithms were developed in order to generate a result that is equivalent to the IEEE binary64 result.

3.2.1 Representation of the Significand

The significand of the IEEE binary64 is a 52-bit fraction and a hidden 1, a total of 53-bit significand as shown in Fig. 3-5.

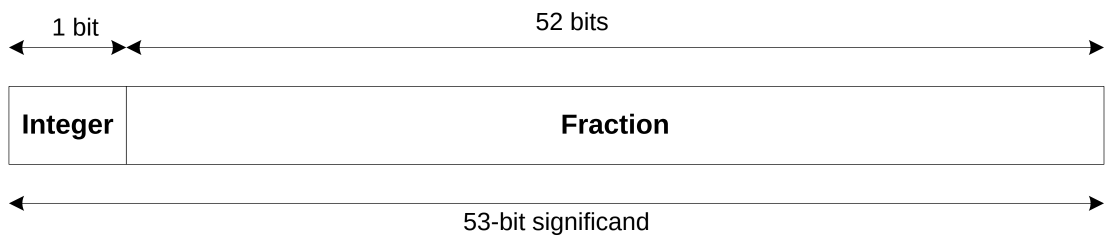


Figure 3-5 IEEE binary64 representation of the significand

To convert this conventional IEEE binary64 format into an equivalent conventional octal format. The integer part has to expand by 2 bits to accommodate for an octal digit, and two more bits are appended to the right of the LSB of the fractional part to accommodate for an 18 octal digits. The precision of the significand increases in the process of converting from the binary to the octal representation. Fig. 3-6 shows the structure of the non-redundant octal representation of the above FP number.

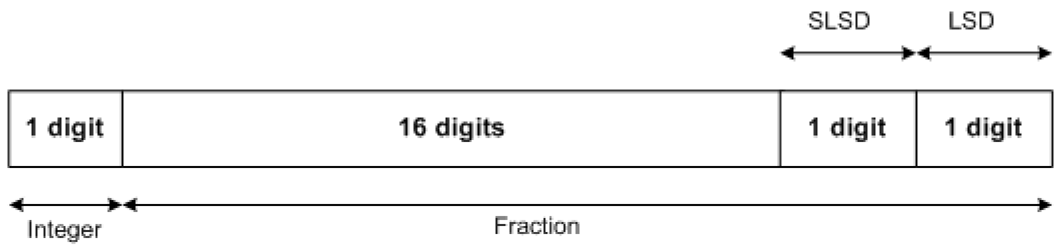


Figure 3-6 Non-redundant octal representation of the significand

The above non-redundant octal significand can be represented in a signed-digit redundant octal representation if an addendum digit is added to the left of the integer digit. Fig. 3-7 shows the structure of the proposed redundant representation.

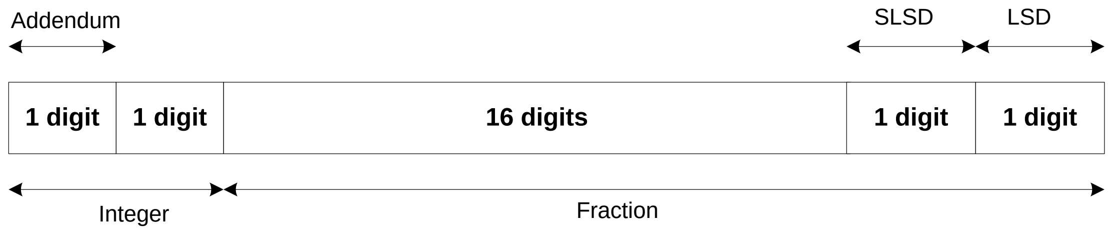


Figure 3-7 Redundant octal representation of the significand

For our mixed binary/decimal floating-point adder, we need an internal representation that accommodates for both the binary and decimal formats. The proposed internal representation of the mixed significand is shown in figure 3-8.

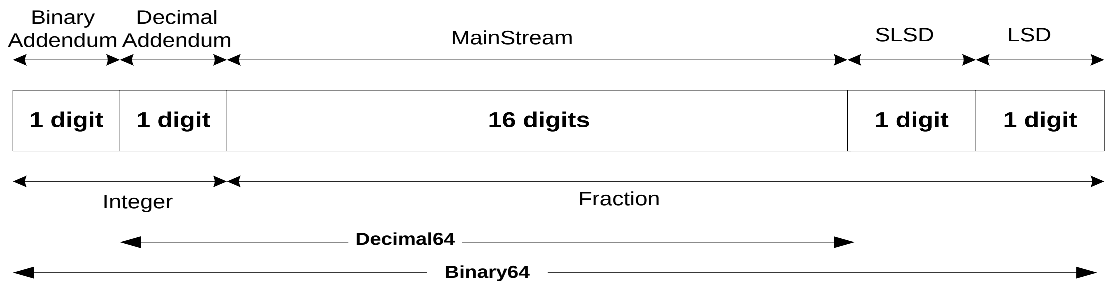


Figure 3-8 The internal representation of the mixed binary/decimal significand

The Decimal Addendum and the MainStream comprises the decimal64 representation while the binary64 is represented by the Binary Addendum, the Decimal Addendum, which together form the floating point integer, the MainStream, the SLSD, and LSD, which together form the binary floating-point fraction.

3.2.2 Block Diagram of the Mixed FP Adder

Figure 3-9 shows the proposed block diagram of the mixed floating-point adder. First, the exponent difference between the two operands is computed. If the first exponent EA is smaller than the second exponent EB, the swap flag is raised to swap the operands. The difference of the exponents is used in the computation of the shift amounts. Note that for the decimal format, the larger operand (the one with the larger exponent) may be shifted left and the smaller operand (the one with the smaller exponent) may be shifted right. However, for the binary format, the operands are normalized so operand alignment is performed by shifting the smaller operand to the right if necessary while the larger one remains as is. The small operand should keep track of the conventional round, and guard digits for the decimal operation. For binary operation, the round and guard bits are included in either the LSD or the SLSD. For both binary and decimal operations, the sticky generations starts immediately to generate the sticky bit and the sticky sign. This is important for rounding in both the binary and decimal operations.

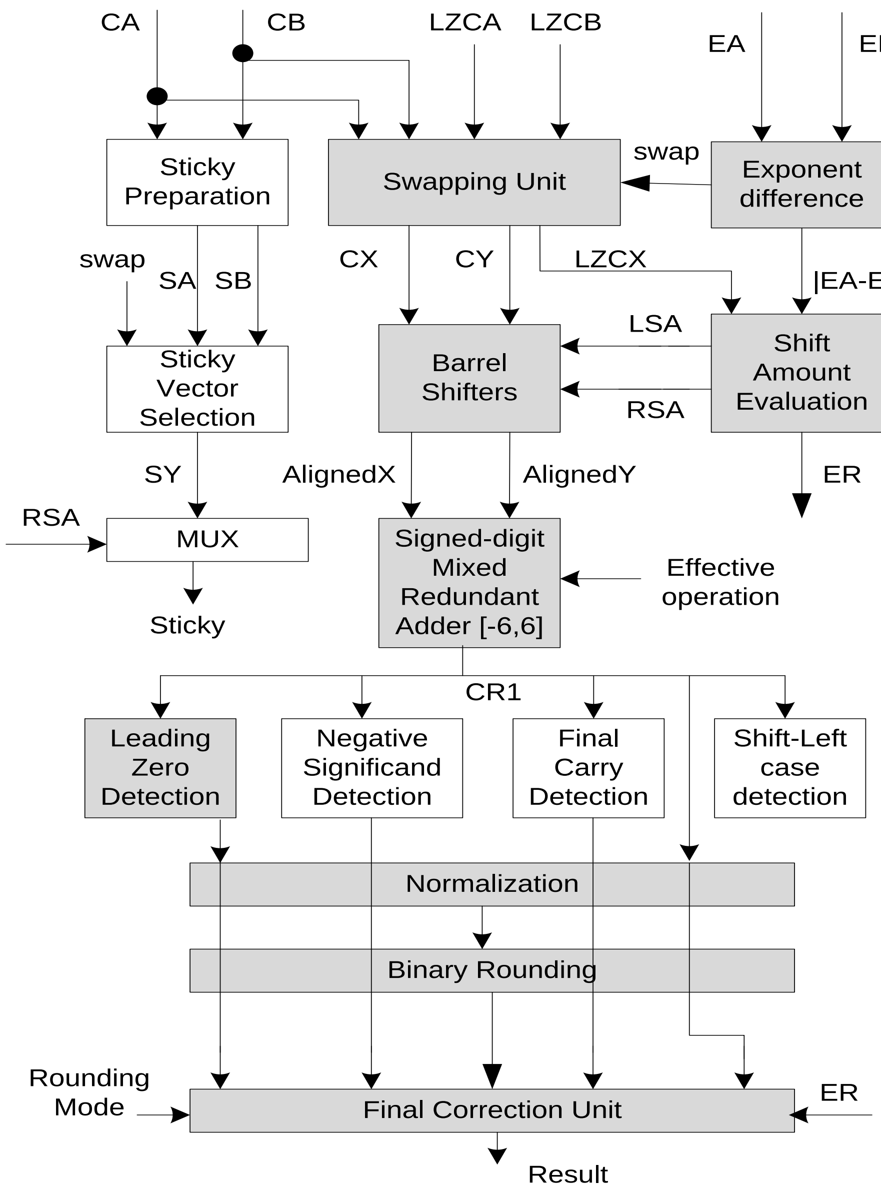


Figure 3-9 Block diagram of the mixed FP adder

During the CR1 leading-zero detection, the Group_ID of the significand CR1 is determined. CR1 can belong to one of three groups. The determination of the Group_ID is important during the binary rounding stage. The Group_ID is determined based on the value of the integer part of CR1 according to Table 3-1.

Table 3-1 Group_ID determination

	Group 1	***Group 2***	***Group 3***
Integer digit	1	2,3	4,5,6,7

3.3 Binary Rounding

In the next section, we treat the case of binary rounding for the redundant FP number above. This FP number is assumed to be generated from a signed-digit FP adder, and then normalized (i.e. the integer part is a non-zero digit). Note also that the round and guard bits from the binary format are included in the octal format (Recall that the octal representation is of more precision).

Problem Statement

Given a floating-point number N that is encoded in a signed-digit redundant octal format with a digit set [-6, 6]. Rounding of this number is needed so that a new floating-point number is generated, the rounded FP number, such that the rounded FP number is equivalent to that generated if binary rounding is performed on the IEEE binary64 floating point number equivalent to N. In this section, we treat the different cases of rounding modes for a redundant FP number. For each rounding mode, the rounding algorithm of the non-redundant octal format is developed first, then the corresponding rounding for the redundant format is deduced.

3.3.1 Rounding to Positive Infinity

The non-redundant octal format

Case 1: The significand belongs to Group 1

The rounding and guard bits are the least two significant bits in the LSD. If rounding in the binary format yields an addition of 1to the least significant bit of the binary64 format, then it should yield an addition of 1 to the third least significant bit in the octal format, in other words, an addition of 4 to the LSD.

The algorithm used is

```
If (sign_of_number is positive)
    If (sticky=0 and (LSD= (0 or 4)) No Change;
    Else Add 4 to the LSD;
Else No Change; //sign of the number is negative
```

Where ‘sign_of_number’ is the sign of the FP number.

Case 2: The significand belongs to Group 2

In this case the rounding bit is the third least significant bit, and the guard bit is the second least significant bit. Hence a rounding mode that yields a 1 to the binary64 format is equivalent to adding 1 to the SLSD.

The algorithm used is

```
If ((LSD=0 and Sticky=0) or sign_of_number is negative)
    No Change;
Else Add 1 to the SLSD;
```

Case 3: The significand belongs to Group 3

In this case the rounding bit is the fourth least significant bit, and the guard bit is the third least significant bit. Hence a rounding mode that yields a 1 to the binary64 format is equivalent to adding 2 to the SLSD.

The algorithm used is:

```
If (sign_of_number is positive)
    If (SLSD is even and LSD=0 and Sticky=0) No Change;
    Else Add 2 to SLSD;
Else No Change;
```

The redundant octal format

Case 1: The significand belongs to Group 1

The algorithm is

```
If (sticky=0 or sticky=1)
    If (sign_of_number is positive)
        If (sticky=0 and (LSD= (0 or 4 or -4)) No Change;
        Else Add 4 to the LSD;
    Else No Change;
Else //Sticky=-1
    If (sign_of_number is positive) Add 3 to the LSD;
    Else Add (-1) to the LSD;
```

Case 2: The significand belongs to Group 2

The algorithm is:

```
If (LSD is positive or (LSD=0 and (Sticky is either 0 or 1)))
    If ((LSD=0 and Sticky=0) or sign_of_number is negative)
        No Change;
    Else Add 1 to the SLSD;
Else
    If (sign_of_number is negative) Subtract 1 from SLSD;
    Else No Change;
```

Case 3: The significand belongs to Group 3

The algorithm is:

```
If (LSD is positive or (LSD=0 and (Sticky=0 or positive)))
    If (sign_of_number is positive)
        If (SLSD is even and LSD=0 and Sticky=0) No Change;
        Else Add 2 to SLSD;
    Else No Change;
Else
    If (sign_of_number is positive) Add 1 to SLSD;
    Else Subtract 1 from SLSD;
```

3.3.2 Round to Negative Infinity

The non-redundant octal format

Case 1: The significand belongs to group 1

The algorithm is:

```
If (sign_of_number is negative)
    If (sticky=0 and LSD $\varepsilon$ {0,4}) Add 0 to LSD;
    Else add 4 to LSD;
Else add 0 to LSD;
```

Case 2: The significand belongs to group 2

The algorithm is:

```
If ((LSD=0 and sticky=0) or sign_of_number is positive)
    Add 0 to SLSD;
Else add 1 to SLSD;
```

Case 3: The significand belongs to group 3

The algorithm is:

```
If (sign_of_number is positive) add 0 to SLSD;
Else
    If (SLSD is even and LSD=0 and sticky=0) add 0 to SLSD;
    Else add 2 to SLSD;
```

The redundant octal format

Case 1: The significand belongs to group 1

The algorithm is:

```
If (sticky $\geq$ 0)
    If (sign_of_number is negative)
        If (LSD=0 or 4 or −4) add 0 to LSD;
        Else add 4 to LSD;
    Else add 0 to LSD;
Else
    If (sign_of_number is negative)
```

```
        If (LSD=0 or 4 or -4) subtract 1 from LSD;
        Else add 3 to LSD;
    Else subtract 1 from LSD;
```

Case 2: The significand belongs to group 2

The algorithm is:

```
If (LSD is positive or (LSD=0 and sticky $\geq$ 0))
    If ((LSD=0 and sticky=0) or sign_of_number is positive)
        Add 0 to SLSD;
    Else add 1 to SLSD;
Else
    If (sign_of_number is positive) subtract 1 from SLSD;
    Else add 0 to SLSD;
```

Case 3: The significand belongs to group 3

The algorithm is:

```
If (LSD is positive or (LSD=0 and sticky $\geq$ 0))
    If (sign_of_number is positive) add 0 to SLSD;
    Else
        If (SLSD is even and LSD=0 and sticky=0)
            Add 0 to SLSD;
        Else Add 2 to SLSD;
```

```
Else
    If (sign_of_number is positive) subtract 1 from SLSD;
    Else
        If (SLSD is even and LSD=0 and sticky=0)
            Subtract 1 from SLSD;
        Else add 1 to SLSD;
```

3.3.3 Rounding towards zero

Rounding towards zero is just a round down if the sign of the number is positive and it is round up if the sign of number is negative.

3.3.4 Rounding to nearest tie to even (RNE)

The non-redundant octal format

<u>*Case 1: The significand belongs to group 1*</u>

The algorithm is:

```
If (LSD ε {0, 1, 4, 5} or (sticky = 0 and LSD = 2))
    Add 0 to LSD;
Else Add 4 to LSD;
```

<u>*Case 2: The significand belongs to group 2*</u>

The algorithm is:

```
If (LSD ε {0, 1, 2, 3}) Add 0 to SLSD;
Else if (LSD = 4 and Sticky = 0)
```

```
    If (SLSD is even) Add 0 to SLSD;
    Else add 1 to SLSD;
Else add 1 to SLSD;
```

<u>Case 3: The significand belongs to group 3</u>

The algorithm is:

```
If (SLSD is even) Add 0 to SLSD;
Else
    If (LSD != 0) add 2 to SLSD;
    Else
        If (sticky=1) add 2 to SLSD;
        Else
            If (SLSD=1 or SLSD=5) add 0 to SLSD;
            Else add 2 to SLSD;
```

The redundant octal format

<u>Case 1: The significand belongs to group 1</u>

The algorithm is:

```
If (sticky $\geq$ 0)
    If (LSD $\varepsilon$ {0, 1, 4, -4, 5, -3} or (sticky=0 and LSD $\varepsilon$ {2,-6}))
        Add 0 to LSD;
    Else add 4 to LSD;
Else
```

```
    If (LSD ε {1, 2, 5, -3, 6, -2}) subtract 1 from LSD;
    Else add 3 to LSD;
```

Case 2: The significand belongs to group 2

The algorithm is:

```
If (LSD > 0 or (LSD = 0 and sticky ≥ 0))
    If (LSD ε {0, 1, 2, 3} or (LSD=4 and sticky is negative))
        Add 0 to SLSD;
    Else if (LSD = 4 and sticky = 0)
        If (SLSD is even) add 0 to SLSD;
        Else add 1 to SLSD;
    Else add 1 to SLSD;
Else
    If ((LSD ε {-6, -5} and sticky ≥ 0) or (LSD ε {-5,-4} and
    sticky is -ve))
        Subtract 1 from SLSD;
    Else if (LSD=-4 and sticky=0)
        If (SLSD is even) add 0 to SLSD;
        Else subtract 1 from SLSD;
    Else add 0 to SLSD;
```

Case 3: The significand belongs to group 3

The algorithm is:

```
If (LSD > 0 or (LSD = 0 and sticky $\geq$ 0))
    If (SLSD is even) add 0 to SLSD;
    Else
        If (LSD!=0) add 2 to SLSD;
        Else
            If (sticky = 1) add 2 to SLSD;
            Else
                If (SLSD $\in$ {1, 5}) add 0 to SLSD;
                Else add 2 to SLSD;
Else
    If (SLSD is odd) subtract 1 from SLSD;
    Else add 1 to SLSD;
```

3.4 Illustrative Binary Example

Assume that X and Y are two binary floating-point numbers such that:
X = +1.0101101 * 2^7, and Y = +1.1100100 * 2^4
For the sake of simplicity, we will assume a precision of 8 bits.
To add these two numbers according to the IEEE standard, the significand of Y is shifted to the right by three bits to align the operands,
Y_aligned = 0.0011100*2^7, Guard = 1, Round = 0, Sticky = 0
Then we add the two aligned operands to get Z,
Z = 1.1001001 *2^7, Guard = 1, Round = 0, Sticky = 0
If the rounding mode is 'Round to positive infinity' or 'Round to nearest even', the significand of Z becomes 1.1001010. The significand remains unchanged if the required rounding mode is 'Round to negative infinity' or 'Round towards Zero'.

Now we will discuss how our proposed adder performs binary addition. First, in the process of converting X and Y into an octal representation, two zero-bits are appended to the right of the significands of X and Y such that:

X = 001. 010 110 100 * 2^7 and
Y = 001. 110 010 000 * 2^4

In the process of converting the binary base to an octal base, X and Y becomes:

X = 2. 6 -3 0 * 8^2, Y = 3.5 -4 0 *8^1

Y is shifted right by one digit to align the operands, then the two operands are added to produce Z,

Z = 3.1 2 -4 * 8^2, Sticky=0

Then we round this number Z according to the above algorithms, note that Z belongs to group 2 since the integer digit is 3

Round to positive infinity:

According to the algorithm, we should not change Z.
Z = 3.1 2 x * 8^2, note that the LSD is irrelevant because the precision of the binary number is only 8 bits. Converting Z into its binary equivalent yields 011.001 010 * 8^2, which is the same number resulted from above. It just needs another step of transforming the number to base 2, which is performed in our case by shifting the number one bit to the right and modifying the exponent. This step is only performed when storing the number back into the memory.

Round to negative infinity:

According to the algorithm, we should subtract 1 from the SLSD
Z becomes 3.1 1 x * 8^2. Following the same steps as above, this gives the same number resulted from conventional binary rounding.

Round to nearest tie to even:

According to the algorithm, we should add 0 to SLSD (i.e. no change) which gives the same number as ‘rounding up’ and this is correct.

3.5 Conclusion

In this chapter, a mixed floating-point adder was proposed. The decimal floating-point adder proposed in Chapter 2 was extended to incorporate IEEE binary addition. The internal representation of floating-point numbers changed slightly. Rounding algorithms for the basic four rounding modes were suggested. The adder was simulated using random test vectors. The next chapter shows the synthesis results of the decimal, and the mixed floating-point adders. They were synthesized using Design Compiler and the TSMC 65 nm LP technology.

Chapter 4

4. Results and Future Work

4.1 Results

The decimal floating-point adder was simulated using the test vectors generated by IBM and Cairo University [24]. It was synthesized using Synopsys Design Compiler and the TSMC 65 nm LP technology. The following table shows delay comparison with all the decimal floating-point adders encountered in literature.

Table 4-1 Decimal FP Adder Delay Comparison

Adder	**delay (ns)**	**delay (FO-4)**	**Area (um^2)**
[14]	3.5 (1.1 um)	63.6	145100
[15]	2.76 (1.1 um)	56	142800
[16]	NA	48	NA
Proposed Adder	2.02	48	24683

The following chart shows an area profiling of the decimal FP adder

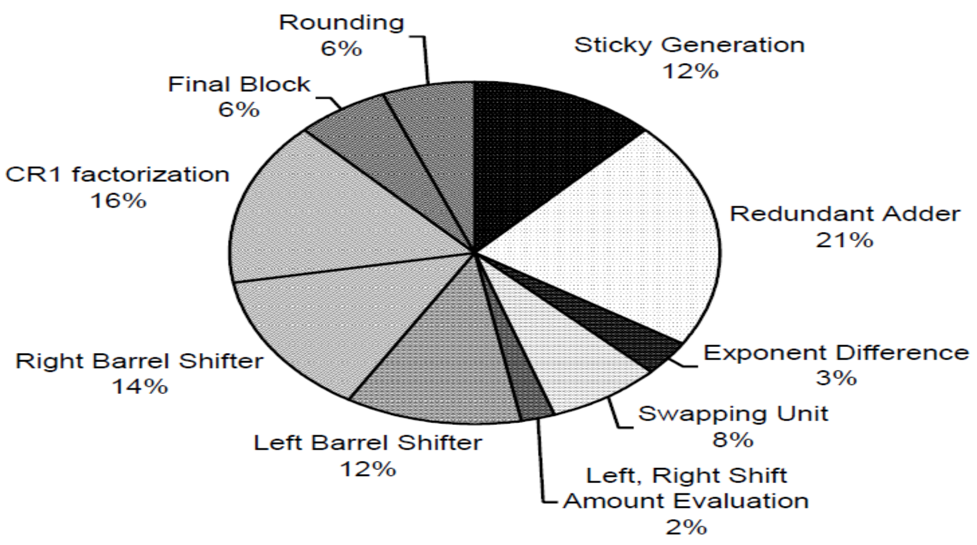


Figure 4-1 Area Profiling of the Decimal FP Adder

The mixed floating-point adder was simulated using random vectors. It was then synthesized with the same TSMC 65 nm LP technology. The mixed floating-point adder shows a latency of 3.4 ns and an area of 31333 um^2. In other words, it shows a latency increase of 68% over the decimal floating-point adder, and an area increase of 26.8% over the decimal floating-point adder. These results deem our mixed-adder favorable, from an area-efficiency point of view, to use rather than using a separate decimal adder and another separate binary one.

4.2 Pipelining the Design

The mixed floating-point adder was further pipelined into five stages. Pipelining the mixed adder has some advantages.

First, pipelining the mixed floating-point adder allows for a smaller clock period (hence a higher frequency of operation). Our implementation allows for a maximum frequency up to 1.04 GHz.

Second, pipelining the design allows for a higher decimal/binary floating-point throughput. For instance, consider the following instructions in a processor that has only one mixed floating-point adder.

DFPADD A, B, C

BFPADD D, E, F

DFPADD G, H, I

For the non-pipelined design, the second instruction will not start until the first one finishes. Similarly, the third one will not start until the second one finishes. If we assume that the non-pipelined design takes 5 cycles to complete, then the three instructions will finish after 15 cycles.

For the 5-stage pipelined design, the executions of those instructions are overlapped. A stage in the adder could be performing a decimal operation while another stage performing a binary one at the same time. The three instructions will finish after 7 cycles instead of 15. If we ignore the cycles required to fill the mixed adder pipes, we will get a result every cycle.

Figure 4-2 shows the stages of the pipelined mixed floating-point adder.

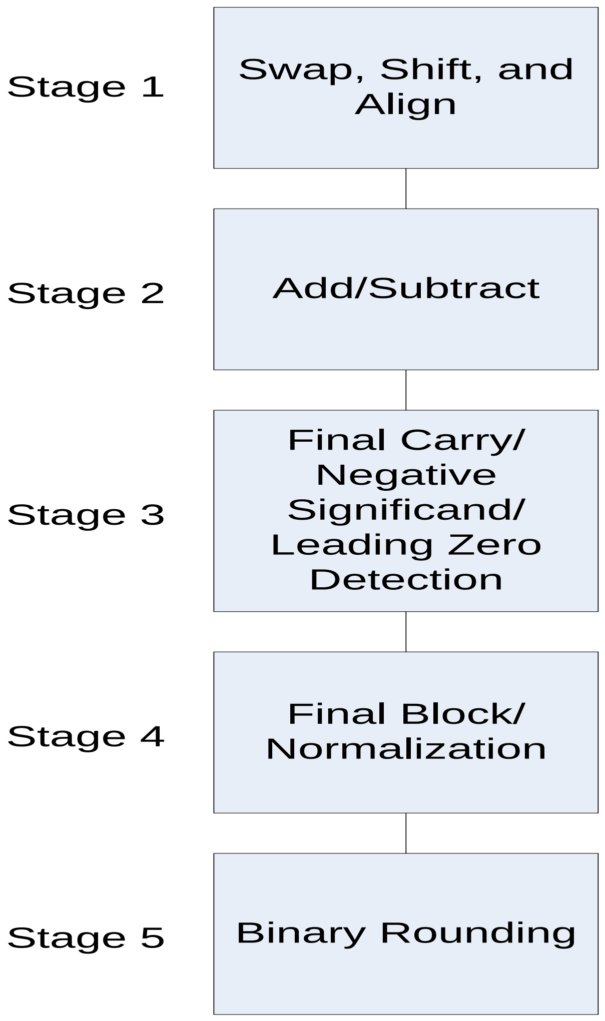


Figure 4-2 The 5 stages of the pipelined mixed adder

4.3 Future Work

CUSPARC is a processor developed in Cairo University based on the SPARC architecture. Currently, the processor supports only integer operations. The functionality of the CUSPARC can be extended by introducing our mixed floating-point adder. This will allow for both binary and decimal floating-point addition.

The same internal format can be used to implement a totally mixed redundant floating-point unit.

References

[1] IEEE, IEEE Standard for Floating-Point Arithmetic, 2008.

[2] E. M. Schwarz and C. A Krygowski, "The S/390 G5 floating-point unit," IBM Journal of Research and Development , pp. 707-721, 1999.

[3] A.M. Nielsen, D.W. Matula, C.N. Lyu, and G. Even, "An IEEE compliant floating-point adder that conforms with the pipeline packet-forwarding paradigm," IEEE Transactions on Computers, vol. 49, pp. 33-47, 2000.

[4] H. A. H. Fahmy, "A Redundant Digit Floating Point System," PhD thesis, Electrical Engineering Department, Stanford University, USA, 2003.

[5] Ercegovac M. D. and Lang T., Digital Arithmetic.: Morgan Kaufmann, 2004.

[6] Parhami B., Computer Arithmetic: Algorithms and Hardware Designs.: Oxford University Press, 2001.

[7] H. Suzuki, Y. Nakase, H. Makino, H. Morinaka, and K. Mashiko, "Leading-zero anticipatory logic for high-speed floating point addition," in Proceedings of the IEEE Custom Integrated Circuits Conference, 1995.

[8] M.S. Schmookler and K.J. Nowka, "Leading zero anticipation and detection-a comparison of methods," in Proceedings of the 15th Symposium on Computer Arithmetic, 2001.

[9] Intel Corporation. Intel Decimal Floating-Point Math Library. [Online]. http://software.intel.com/en-us/articles/intel-decimal-floating-point-math-library/

[10] Sun Microsystems. BigDecimal class, Java 2 platform standard edition. [Online]. http://download.oracle.com/javase/1.4.2/docs/api/java/math/BigDecimal.html

[11] Duale A. Y., Decker M. H., Zipperer H.-G, Aharoni M., and Bohizic T.

J., "Decimal floating-point in z9: An implementation and testing perspective," IBM Journal of Research and Development, vol. 51, pp. 217-227, 2007.

[12] L. Eisen et al., "IBM POWER6 accelerators: VMX and DFU," IBM Journal of Research and Development, vol. 51, no. 6, 2007.

[13] E. M. Schwarz, J. S. Kapernick, and M. F. Cowlishaw, "Decimal floating-point support on the IBM z10 processor," IBM Journal of Research and Development, vol. 53, 2009.

[14] J. Thompson, N. Karra, and M.J. Schulte, "A 64-bit decimal floating-point adder," in Proceedings of the IEEE Computer Society Annual Symposium on VLSI, 2004, pp. 297-298.

[15] Liang-Kai Wang, M.J. Schulte, J.D. Thompson, and N. Jairam, "Hardware Designs for Decimal Floating-Point Addition and Related Operation," IEEE Transactions on Computers, vol. 58, pp. 322-335, 2009.

[16] Liang-Kai Wang and M.J. Schulte, "A Decimal Floating-Point Adder with Decoded Operands and a Decimal Leading-Zero Anticipator," in 19th IEEE Symposium on Computer Arithmetic, 2009.

[17] A. Avizienis, "Signed-digit number representation for fast parallel arithmetic," IRE Trans. El. Comp., 1961.

[18] B. Shirazi, D.Y.Y. Yun, and C.N. Zhang, "RBCD: redundant binary coded decimal adder," in IEE Proc. Computer and Digital Techniques, vol. 136, 1989, pp. 156-160.

[19] S Gorgin and G. Jaberipur, "Fully Redundant Decimal Arithmetic," in 19th IEEE Symposium on Computer Arithmetic, 2009.

[20] A. Svoboda, "Decimal Adder with Signed Digit Arithmetic," IEEE Transactions on Computers, vol. C-18, pp. 212-215, 1969.

[21] H. Nikmehr, B. J. Philips, and C. C. Lim, "A Decimal Carry-Free Adder," in Proceedings of the SPIE Conference, 2004, pp. 786-797.

[22] J. Moskal, E. Oruklu, and J. Saniie, "Design and Synthesis of a Carry-Free Signed-Digit Decimal Adder," in Proceedings of the IEEE Symposium on Circuits and Systems, 2007, pp. 1089-1092.

[23] K. Yehia, H. A. H. Fahmy, and M. Hassan, "A Redundant Decimal Floating-Point Adder," 2010 Conference Record of the Forty Fourth Asilomar Conference on Signals, Systems, and Computers, Pacific Grove, CA, USA, 2011.

[24] A. A. R. Sayed-Ahmed, H. A. H. Fahmy, and M.Y. Hassan, "Three engines to solve verification constraints of decimal Floating-Point operation," 2010 Conference Record of the Forty Fourth Asilomar Conference on Signals, Systems, and Computers., Pacific Grove, CA, USA, 2011.