IEEE-COMPLIANT BINARY/DECIMAL UNIT BASED ON A BINARY/DECIMAL FMA

by
Ahmed Adel Abdelghany Wahba

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

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
2014

IEEE-COMPLIANT BINARY/DECIMAL UNIT BASED ON A BINARY/DECIMAL FMA

by
Ahmed Adel Abdelghany Wahba

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

Under the Supervision of

Associate Prof. Hossam A. H. Fahmy
Principal Adviser

*

Adviser

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
2014

IEEE-COMPLIANT BINARY/DECIMAL UNIT BASED ON A BINARY/DECIMAL FMA

by
Ahmed Adel Abdelghany Wahba

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

Approved by the
Examining Committee

_____

Associate Prof. Hossam A. H. Fahmy, Thesis Main Advisor

_____

———————————, Member

_____

———————————, Member

_____

———————————, Member

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
2014

Acknowledgment

In the name of Allah the most merciful the most gracious; all thanks to Allah the Lord of the Heavens and Earth and peace be upon Mohamed and his companions.

First of all I want to thank my family, especially my parents, for their invaluable support during my whole my life; After Allah, without their help and support I wouldn't have accomplished anything in my life.

Also many thanks to my students and friends, especially 2011 TAs, who were more than supportive during my journey.

Finally, I would like to express my sincere gratitude to my advisor and mentor Dr. Hossam Fahmy for his enriching advice, help, and support during the last four years of my life.

Ahmed Adel Wahba,
May, 2014.

Abstract

In this work a combined binary/decimal floating point unit based on a combined binary/decimal floating point Fused Multiply-Add unit is proposed.

A binary/decimal multiplier that uses SD-radix5 encoding for decimal, and SD-radix4 for binary is used to multiply the first two operands.

A redundant octal/decimal adder that uses the digit set [-6,6] is used to add the addend to the multiplication result. Leading zeros are anticipated in parallel with the redundant addition to remove the delay of the Leading Zeros Anticipator (LZA) out of the critical path. In order to eliminate the carry propagation caused by rounding, the result is rounded while still in the redundant format.

A new redundant to decimal/octal converter based on a look-ahead carry tree is proposed. Also a new binary LZA that generates its output in base 3 to simplify the final binary normalization shifting is proposed.

Decimal verification was done using more than 1.1 million test vectors that were generated specifically to test every part of the FMA. For binary a number of random test vectors were used to test the design in different cases. All of the binary and decimal test vectors passed correctly, and the design showed complete functionality as an adder/subtractor, multiplier, or an FMA.

The design was synthesized using TSMC65LP kit, on typical temperature and process, and 1.2V supply. The results showed a delay of 118 FO4 gate delays, an Area of 122,000 NAND2 gates, and power of 112mW.

Synthesis results also showed that the proposed combined binary/decimal FMA has 8% less delay and 30% more area than the fastest previously published decimal FMA.

Contents

List of Tables

List of Figures

List of Symbols and Abbreviations

Abbreviations

BCD Binary Coded Decimal.

BID Binary Integer Decimal.

CPA Carry Propagate Adder.

CSA Carry Save Adder.

DCSA Decimal Carry Save Adder.

DFP Decimal Floating Point.

DPD Densily Packed Decimal.

EOP Effective Operation.

Exp Exponent.

IEEE Institute of Electronics and Electrical Engineering.

FMA Fused Multiply Add.

FO4 Fan Out of Four.

FPU Floating Point Unit.

GD Guard Digit.

ITD Input Transfer Digit.

LSB Least Significant Bit.

LSD Least Significant Digit.

LZA Leading Zeros Anticipator.

LZC Leading Zeros Count.

LZD Leading Zeros Detector.

MSB Most Significant Bit.

MSD Most Significant Digit.

MUX Multiplexer.

NaN Not a Number.

OTD Output Transfer Digit.

RD Round Digit.

RndMode Rounding Mode (Direction).

This thesis is dedicated to my family and friends.

Chapter 1

Introduction

Decimal floating point arithmetic is getting more and more urgent to be implemented in computer systems especially in financial, military, and space applications where the small truncation error of the binary-based units can lead to massive losses in big companies [1].

For example, in some financial applications such as phone billing using decimal is a must. Some fractions that we -humans- use a lot in our daily transactions and expect the computer to use it as accurate as we do, may be inexactly represented in binary. For example the fraction $\frac{1}{10}$ when converted to binary single precision (binary 32), it will be $(0.0001100110011\cdots)$, the value of that fraction is not exactly 0.1, it's 0.099999964. This conversion error can cause millions of dollars loss per year in large transactions such as banks transferring credit, or huge companies paying their phone bills [1].

Other urgent need for decimal representations in computer systems, is that the user of some human oriented applications expect some trailing zeros, not a normalized number which is the case in binary. For example a resistor is said to have value of 4.700K Ohm, we expect it to be 4700 Ohm accurate to three decimal places, or in other words the precision of the value of this resistor is 1 Ohm. This is totally different than saying it's 4.7 K Ohm, in that last case the precision is 100 Ohm not 1 Ohm as in the first case. The IEEE754-2008 decimal standard [2] solved this problem by allowing for different cohorts of the same number.

Due to its importance, decimal floating point arithmetic was included in the IEEE754-2008 standard [2]. The decimal operations can be performed either in a software layer that uses a binary-based Floating Point Unit (FPU), or in a separate decimal FPU. Performing decimal operations on a hardware that is just made for

decimal is much faster and more power efficient than using software libraries to perform them on a binary hardware.

Fused Multiply Add (FMA) operation is one of the operations defined in the IEEE754-2008 standard. The FMA operation has three operands: A, B, and C, The result of the FMA operation is $A \times B + C$, it's equivalent to multiplying A and B then adding the result, without rounding, to the third operand (C) and performing one final rounding. The main advantage of the FMA operation is its accuracy, instead of performing two rounding steps, after the multiplication and after the addition, only one rounding step is performed at the end of the FMA operation. Also FMA is a separate instruction with only one fetch and decode stages. Performing the multiplication then the addition requires a fetch and a decode cycle for each instruction.

Multiplication and addition can also be performed using the FMA unit. Multiplication can be performed as $A \times B \pm zero$, and addition/subtraction can be performed as $A \times one \pm C$. Hence in some applications where the area is critical, the FMA unit can replace the adder and the multiplier. FMA is very vital in many applications, especially DSP applications where the accumulation equation $sum = sum + a_i \times b_j$ appears a lot, which is basically an FMA operation [3].

The first decimal FMA was presented in [4], and it was a combined binary/decimal architecture. The first verified architecture was proposed by SilMinds in [5], which will be explained in details later.

1.1 Decimal Redundant representations

Beside the usual well known BCD-8421 encoding, there are some redundant encodings to represent any decimal digit. Some properties of these representations can simplify the decimal operations such as carry save addition, getting the 9's complement, and multiplication by some constants. One of these representations is the (4221) encoding, where each decimal digit is represented in four bits with weights of 4,2,2, and 1, i.e. the most significant bit has a weight of four, and the least significant bit has a weight of 1. For example, 8 is represented as (1110). An example of the redundancy in this representation is that some decimal values have more than one representation. For example, 6 can be represented as (1010) or (1100). In this work we used, as well as the 8421 and 4221 encodings, the 5211 and 5421 encodings which represent each digit in four bits with the weights (5,2,1,

Digit	8421	4221	5211	5421
0	0000	0000	0000	0000
1	0001	0001	0001 | 0010	0001
2	0010	0010 | 0100	0100 | 0011	0010
3	0011	0011 | 0101	0101 | 0110	0011
4	0100	1000 | 0110	0111	0100
5	0101	1001 | 0111	1000	1000
6	0110	1010 | 1100	1001 | 1010	1001
7	0111	1011 | 1101	1100 | 1011	1010
8	1000	1110	1101 | 1110	1011
9	1001	1111	1111	1100

Table 1.1: Decimal Representations

and 1), and (5,4,2, and 1). The different decimal representations used in this work are shown in Table 1.1.

The following are some useful properties that were used in this work and their proof:

1) The decimal encodings that have the sum of their weights = 9, such as (4221) and (5211) encodings, have an important property, that is all the sixteen 4-bit combinations represent a decimal digit ($X_i \in [0,9]$). Therefore, any Boolean function (AND, OR, XOR,. . .) operating over two or more input digits in these encodings produces a 4-bit vector that represents a valid decimal digit (input and output digits represented in the same code). Hence we can use simple full adders to perform a carry save addition to add three numbers represented in one of those codes, and the resulting sum and carry vectors will be in the same representation as the inputs.

2) In the encodings that have the sum of their weights = 9, it's very easy to get the 9's complement of any decimal number represented in any of these representations, simply by inverting each bit. Hence we can get negative numbers by inverting each bit of the original positive number, and add a (+1) increment at the least digit position.

It's important to be stated that it's very easy to convert between any of the four decimal representations shown in Table 1.1 using a simple few gate delays logic.

Proof:

Assume X is a decimal digit represented in a decimal format that has the bit weights of $w_3w_2w_1w_0$, where $w_3+w_2+w_1+w_0=9$, as $x_3x_2x_1x_0$. Hence X = $w_3*x_3+w_2*x_2+w_1*x_1+w_0*x_0$.

By inverting each bit we get $\overline{X} = w_3*(1-x_3)+w_2*(1-x_2)+w_1*(1-x_1)+w_0*(1-x_0) = w_3+w_2+w_1+w_0-(w_3*x_3+w_2*x_2+w_1*x_1+w_0*x_0) = 9-X$.

Hence by inverting each bit of a digit represented in a decimal format that has the sum of its weights = 9, we get the nine's complement of this digit. So we can easily get the nine's complement of a decimal number simply by inverting the bits representing each of its digits.

3) Multiplying by 2:

If a decimal number X is encoded such that the 4-bits of each digit are weighted as (5421) then shifted left by '1' bit and and the result is read again as if weighted as (8421) for each digit, it is the same as multiplying the decimal number X by 2.

$$(L_{1shift}\{X_{5421}\})_{8421} = (2 \times X)_{8421} \tag{1.1}$$

This can be simply proved as follows: (v_i^j bit number (i) in digit (j) in 5421 encoding)

$X_{5421} = (\cdots v_0^{i+1} v_3^i v_2^i v_1^i v_0^i \; v_3^{i-1} \cdots)_{5421}$

$= \cdots 10^{i+1} \times (\cdots + v_0^{i+1}) + 10^i \times (5v_3^i + 4v_2^i + 2v_1^i + v_0^i) + 10^{i-1} \times (5v_3^{i-1} + \cdots)$

with one bit left shift:

$(L_{1shift}\{X_{5421}\})_{8421} = (\cdots v_0^{i+1} v_3^i v_2^i v_1^i v_0^i v_3^{i-1} \; v_2^{i-1} \cdots)_{8421}$

$= \cdots 10^{i+1} \times (\cdots + v_3^i) + 10^i \times (8v_2^i + 4v_1^i + 2v_0^i + v_3^{i-1}) + 10^{i-1} \times (5v_2^{i-1} + \cdots)$

$= 2 \times (\cdots 10^{i+1} \times (\cdots + v_0^{i+1}) + 10^i \times (5v_3^i + 4v_2^i + 2v_1^i + v_0^i) + 10^{i-1} \times (5v_3^{i-1} + \cdots))$

$= 2 \times (\cdots v_0^{i+1} v_3^i v_2^i v_1^i v_0^i v_3^{i-1} \cdots)_{5421} = (2X)_{8421}$

This proves the initial claim.

4) Another way to multiply by 2: If a decimal number X is encoded such that the 4-bits of each digit are weighted as (5211) then shifted left by '1' bit and the result is read again as if weighted as (4221) for each digit, it is the same as multiplying the decimal number X by 2.

$$(L_{1shift}\{X_{5211}\}p)_{4221} = (2 \times X)_{4221} \tag{1.2}$$

This can be simply proved as follows: (v_i^j bit number (i) in digit (j) in 5221 encoding)

$X_{5211} = (\cdots v_0^{i+1} v_3^i v_2^i v_1^i v_0^i \; v_3^{i-1} \cdots)_{5211}$

$= \cdots 10^{i+1} \times (\cdots + v_0^{i+1}) + 10^i \times (5v_3^i + 2v_2^i + 2v_1^i + v_0^i) + 10^{i-1} \times (5v_3^{i-1} + \cdots)$

with one bit left shift:

$(L_{1shift}\{X_{5211}\})_{4221} = (\cdots v_0^{i+1} v_3^i v_2^i v_1^i v_0^i v_3^{i-1} \; v_2^{i-1} \cdots)_{8421}$

$= \cdots 10^{i+1} \times (\cdots + v_3^i) + 10^i \times (4v_2^i + 2v_1^i + 2v_0^i + v_3^{i-1}) + 10^{i-1} \times (4v_2^{i-1} + \cdots)$

$= 2 \times (\cdots 10^{i+1} \times (\cdots + v_0^{i+1}) + 10^i \times (5v_3^i + 2v_2^i + 2v_1^i + v_0^i) + 10^{i-1} \times (5v_3^{i-1} + \cdots))$

$= 2 \times (\cdots v_0^{i+1} v_3^i v_2^i v_1^i v_0^i v_3^{i-1} \cdots)_{5221} = (2X)_{5211}$

5) Multiplying by 5: If a decimal number X is encoded such that the 4-bits of each digit are weighted as (4221) then shifted left by '3' bits and the result is read again as if weighted by (5211) for each digit, it is the same as multiplying the decimal number X by 5.

$$(L_{3shift}\{X_{4221}\}p)_{5211} = (5 \times X)_{4221} \tag{1.3}$$

This can be simply proved as follows: (v_i^j bit number (i) in digit (j) in 4221 encoding)

$X_{4221} = (\cdots v_0^{i+1} v_3^i v_2^i v_1^i v_0^i \; v_3^{i-1} \cdots)_{4221}$

$= \cdots 10^{i+1} \times (\cdots + v_0^{i+1}) + 10^i \times (4v_3^i + 2v_2^i + 2v_1^i + v_0^i) + 10^{i-1} \times (4v_0^{i-1} + \cdots)$

with three bits left shift:

$$(L_{3shift}\{X_{4221}\})_{5211} = (\cdots v_1^i v_0^i v_3^{i-1} v_2^{i-1} v_1^{i-1}\ v_0^{i-1}\cdots)_{5211}$$

$$= \cdots 10^{i+1} \times (\cdots + v_1^i) + 10^i \times (5v_0^i + 2v_3^{i-1} + v_2^{i-1} + v_1^{i-1}) + 10^{i-1} \times (5v_0^{i-1} + \cdots)$$

$$= 5 \times (\cdots 10^{i+1} \times (\cdots + v_0^{i+1}) + 10^i \times (4v_3^i + 2v_2^i + 2v_1^i + v_0^i) + 10^{i-1} \times (4v_0^{i-1} + \cdots))$$

$$= 5 \times (\cdots v_0^{i+1} v_3^i v_2^i v_1^i v_0^i v_3^{i-1} \cdots)_{4221} = (5X)_{5211}$$

This proves the initial claim.

Using properties 3,4,5 we can easily get the multiples 2X, 4X, by two cascaded 2X operations, and 5X. These multiples are easy decimal multiples, that we can get without carry propagation. Note that we can also get 2X, 4X, and 5X multiples without carry propagation also in the 8421 format.

1.2 IEEE Decimal Floating-Point Standard

As previously indicated, there was an increasing need to Decimal Floating Point (DFP) arithmetic. Hence, there were many efforts to find out the most appropriate DFP formats, operations and rounding modes that completely define the DFP arithmetic. These efforts ended up with the IEEE 754-2008 floating-point arithmetic standard. This section gives a brief overview to this standard [2].

1.2.1 Decimal Formats

The IEEE 754-2008 defines DFP number as : $(-1)^s \times (10)^q \times c$, where: S is the sign bit, q is the exponent, $c = (d_{p-1} d_{p-2} \cdots d_0)$ is the significand where $d_i \in \{0,1,2,3,4,5,6,7,8,9\}$, and p is the precision.

Figure1.1 shows the basic decimal interchange format specified in the IEEE 754-2008 standard. S is the sign bit which indicates either the DFP number is positive ($S = 0$) or negative ($S = 1$) and G is a combination field that contains the exponent, the most significant digit of the significand, and the encoding classification. The rest of the significand is stored in the trailing significand field (T), using either the Densely Packed Decimal (DPD) encoding or the Binary Integer

Table 1.2: Parameters for different decimal interchange formats

Parameter	decimal32	decimal64	decimal128
Total storage width	32	64	128
Combination Field (w+5)	11	13	17
Trailing significand Field (t)	20	50	110
Total Significand Digits (p)	7	16	34
Exponent Bias	101	398	6176
Exponent Width	8	10	14

Decimal (BID) encoding, where the total number of significand digits corresponds to the precision, p. The DPD encoding represents every three consecutive decimal digits in the decimal significand using 10 bits, and the BID encoding represents the entire decimal significand in binary.

Before encoded in the combination field, the exponent is first encoded as binary excess code and its bias value depends on the precision used. There are also minimum and maximum representable exponents for each precision. The different parameters for different precision values are presented in Table 1.2.

In decimal floating-point format a number might have multiple representations. This set of representations is called the floating-point number's cohort. For example, if c is a multiple of 10 and q is less than its maximum allowed value, then (s, q, c) and $(s, q+1, c/10)$ are two representations for the same floating-point number and are members of the same cohort. In other words, a one-digit floating-point number might have up to p different representations while a p-digit floating-point number with no trailing zeros has only one representation (An n-digit floating-point number might have fewer than $p-n+1$ members in its cohort if it is near the extremes of the format's exponent range). A zero has a much larger cohort: the cohort of $+0$ contains a representation for each exponent, as does the cohort of -0. This property is added to decimal floating-point to provide results that are matched to the human sense by preserving trailing zeros. In brief, for decimal arithmetic, besides specifying a numerical result, the arithmetic operations also select a member of the result's cohort. And thus, decimal applications can make use of the additional information cohorts convey.

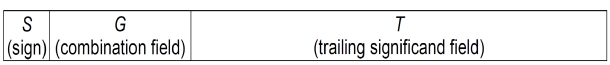

Figure 1.1: Decimal interchange floating-point format

1.2.2 Rounding

There are five rounding modes defined in the standard, Round ties to even, Round ties to away, Round toward zero, Round toward positive infinity, and Round toward negative infinity. Also, there are two well-known rounding modes supported in the Java BigDecimal class [6]. Table 1.3 summarizes the different rounding modes with their required action.

Rounding Mode	Rounding Behavior
Round Ties To Away RA	Round to nearest number and round ties to nearest away from zero, the result is the one with larger magnitude.
Round Ties to Even RNE	Round to nearest number and round ties to even, the result is the one with the even least significand digit.
Round Toward Zero RZ	Round always towards zero (truncate) , the result is the closest DFP number with smaller magnitude.
Round Toward Positive RPI	Round always towards positive infinity , the result is the closest DFP number greater than the exact result.
Round Toward Negative RNI	Round always towards negative infinity, the result is the closest DFP number smaller than the exact result.
Round Ties to Zero RZ	Round to the nearest number and round ties to zero, i.e. truncate in case of ties.
Round To Away RA	Round always to nearest away from zero, the result is the one with larger magnitude.

Table 1.3: Different Rounding Modes

1.2.3 Special numbers and Exceptions

Special numbers:

Operations on DFP numbers may result in either exact or rounded results. However, the standard also specifies two special DFP numbers, infinity and NaN.

Infinities:

Infinity represent numbers of arbitrarily large magnitudes, larger than the maximum represented number by the used precision. That is: $-\infty < \{each\ representable\ finite\ number\} < +\infty$. In Table 1.4, a list of some arithmetic operations that involve infinities as either operands or results are presented. In this table, the operand x represents any finite non-zero number.

Operation	Exception	Operation	Exception
$\infty + x = \infty$	None	$\infty / x = \pm\infty$	None
$\infty + \infty = \infty$	None	$x / \infty = \pm 0$	None
$\infty - x = \infty$	None	$\infty / \infty = NaN$	
$\infty - \infty = NaN$	Invalid	$\sqrt{\infty} = \infty$	None
$\infty \times x = \infty$	None	$\sqrt{-\infty} = NaN$	Invalid
$\infty \times \infty = \infty$	None	$x/0 = \pm\infty$	Division by Zero
$\infty \times 0 = NaN$	Invalid	$subnormal \div \infty$	Underflow

Table 1.4: Examples of some DFP operations that involve infinities

NaNs (Not a Number):

Two different kinds of NaN, signaling and quiet, are supported in the standard. Signaling NaNs (sNaNs) represent values for uninitialized variables or missing data samples. Quiet NaNs (qNaNs) result from any invalid operations or operations that involve qNaNs as operands. When encoded, all NaNs have a sign bit and a pattern of bits necessary to identify the encoding as a NaN and which determines its kind (sNaN vs. qNaN). The remaining bits, which are in the trailing significand field, encode the payload, which might contain diagnostic information that either indicates the reason of the NaN or how to handle it. However, the standard specifies a preferred (canonical) representation of the payload of a NaN.

Exceptions:

There are five different exceptions which occur when the result of an operation is not the expected floating-point number. The default nonstop exception handling uses a status flag to signal each exception and continues execution, delivering a default result. The IEEE 754-2008 standard defines these five types of exceptions as shown in Table 1.5.

Parameter	binary32	binary64	binary128
Total storage width	32	64	128
Exponent Field	8	11	15
Mantissa	23	52	112
Exponent Bias	127	1023	16383

Table 1.6: Parameters for different binary formats

Exceptions	Description	Output
Invalid Operation (Description shows only common examples)	-Computations with sNaN operands -Multiplication of $0 \times \infty$ -Effective subtraction of infinities -Square-root of negative operands -Division of $0/0$ or ∞/∞ -Quantize in an insufficient format -Remainder of $x/0$ or ∞/x (x: finite non zero number)	Quite NaN
Division by Zero	The divisor of a divide operation is zero and the dividend is a finite non-zero number.	Correctly signed ∞
Overflow	The result of an operation exceeds in magnitude the largest finite number representable.	The largest finite number representable or a signed ∞ according to the rounding direction.
Underflow	The result of a DFP operation in magnitude is below $10^{e_{min}}$ and not zero	zero, a subnormal number or$\pm 10^{e_{min}}$ according to rounding mode.
Inexact	The final rounded result is not numerically the same as the exact result (assuming infinite precision)	The rounded or the overflowed result.

Table 1.5: Different Types of Exceptions

1.3 Binary IEEE Standard

The value of the binary number consists of three parts: sign, exponent, and significand. The sign is the most significant bit of the bits representing the binary number. Exponent is calculated as biased exponent, which is the field next to the sign and its width depends on the precision as shown in Table 1.6, minus the

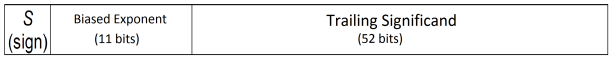

Figure 1.2: Binary64 floating-point format

Exponent	Trailing Significand	Value of the number
111111...11111	$= 0$	NAN
111111...11111	$\neq 0$	(-1)$^{Sign}* \infty$
000000...00000	$= 0$	0
000000...00000	$\neq 0$	$(-1)^{Sign} * 0.Matissa * 2^{-exponent\ bias+1}$

Table 1.7: Special Cases in Binary

exponent bias, which is also shown in Table1.6. Finally significand is the remaining part, and is concatenated from the left by one (the hidden one), unless the number was subnormal or zero. The value of any normalized binary number is $(-1)^{Sign} * 1.Trailing\ Significand * 2^{Exp}$. Special cases of the binary representation are shown in Table 1.7.

The binary standard is the same as the decimal standard concerning exceptions, special cases handling, and rounding directions.

The rest of the thesis is organized as follows: In Chapter 2, we present an overview of the previous binary and decimal FMAs. We present our design in details in Chapters 3 and 4. Chapter 5 shows how the exceptional data paths were handled. Chapter 6 shows the verification and synthesis results and comparisons with the previously published FMAs. Finally Chapter 7 concludes the thesis.

Chapter 2

Previous Work

2.1 General FMA Architecture

Most of the FMAs consist of the same major blocks. The difference is mainly in the implementation of each of these blocks. In this section we will present the common parts, without going through the details of implementing each block. And in the next sections, we will present how each of the previously published FMAs implemented each block.

2.1.1 Decoding the Operands

First The Three operands (OpA, OpB, and OpC) are read in the IEEE 754-2008 format, then each operand is decoded into sign bit, exponent, significand, and flags for special values (NaN or infinity). For decimal, the significand is then decoded to BCD-8421 format.

2.1.2 Multiplier Tree

After decoding the operands, the first two operands are sent to the multiplier tree to get the multiplication result. The multiplication process is performed in two steps:

1) Generating the partial products: In this step, the partial products are generated, the multiplier might be recoded from the usual binary or decimal format to a new format to simplify the multiplicand multiples that needs to be generated.

2) Reducing the partial products: In this step, the partial products are reduced to only two vectors in order to add them with the addend to produce the final result.

2.1.3 Addend preparation

In order to align the addend with the multiplication result, the addend is shifted by the shift amount, which is calculated from the exponent difference. Shifting is performed to the left or right depending on the sign of the exponent difference, and is done in parallel with the multiplication tree.

2.1.4 Final Adder

The two resulting vectors of the multiplier tree, as well as the prepared addend has to be added. First they are reduced to only two vectors using a Carry Save Adder (CSA), then the resulting two vectors are added using the final Carry Propagate Adder (CPA).

2.1.5 Leading Zeros Anticipator (LZA)

In order to produce a normalized result, the leading zeros has to be counted and shifted out of the result. Unless the preferred exponent (preferred exponent is defined ad Min(ExponentA + ExponentB, Exponent C) or the minimum allowable exponent is reached in decimal, or the minimum exponent is reached in binary and in that case the binary result is subnormal.

The LZA operates on the two operands of the final adder and produces the leading zeros count in their sum.

2.2 SilMinds' Architecture

The first hardware implementation of a fully parallel decimal floating-point FMA unit is presented in [5]. This design supports decimal-64, and decimal-128 specified by the IEEE754-2008 standard [2]. The top level of the design is shown in Figure 2.1.

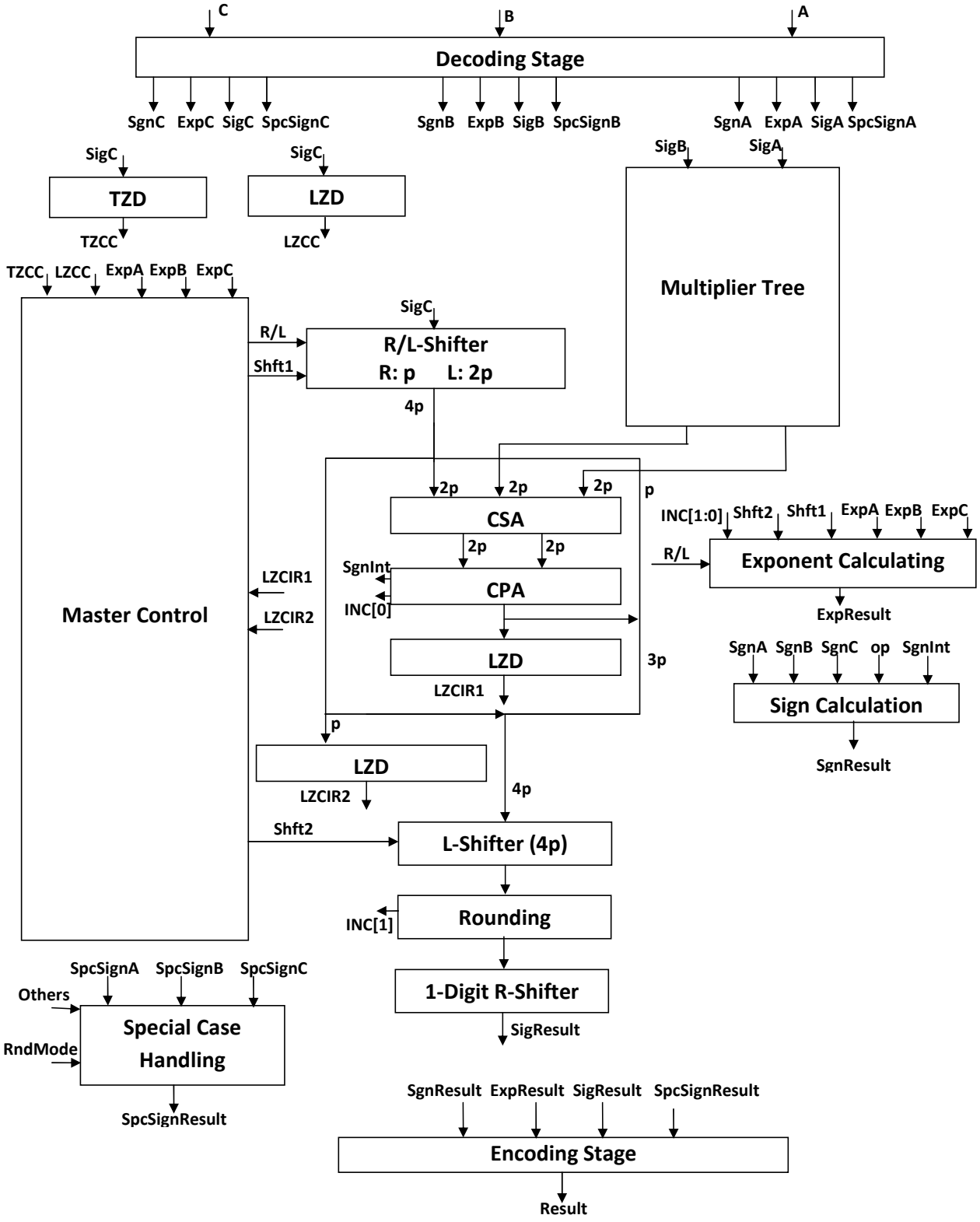

Figure 2.1: SilMind's Architecture

2.2.1 Multiplier Tree

2.2.1.1 Multiplier Recoding

In the multiplier tree used in this work, the multiplier (B) is recoded into the Radix-10 format presented in [7], where the multiplier digits are recoded from the regular digit set $\{0,\cdots,9\}$ to the SD Radix-10 digit set $\{-5,-4\cdots,4,5\}$. The multiplier (B) is recoded into (p+1) digits, where p is the number of digits in the significand. Each digit generates a corresponding partial product from ($\pm$A, $\pm$2A, $\pm$3A, $\pm$4A, $\pm$5A), where A is the multiplicand.

2.2.1.2 Partial Products Generation

The required multiples are generated in a few levels of logic gates, using recoders and wired left shifts. Negative multiples are obtained by getting the 10's complement of the positive corresponding multiples.

2.2.1.3 Reduction Tree

After selecting the correct multiples, the (p+1) partial products are aligned according to their decimal weights. A decimal Carry Save Adder (CSA) tree is used to reduce the middle (2p) digits of the partial products, as well as the middle (2p) digits of the addend, into two vectors: the sum vector and the carry vector. The least significant p digits and the most significant p digits are treated separately outside the CSA. finally a (2p) width Carry Propagate Adder (CPA) is used to add the sum and carry vectors to get the intermediate result. the carry out of the CSA is used to get the sign of the result, and complement it if needed.

2.2.2 Leading Zeros Counter

The result of the CPA might need some left shift to be in the IEEE754-2008 standard. A leading zeros counter (LZC) is used to get the amount of shifting needed. After shifting, the most significant (p+1) digits (p digits + the rounding digit) are sent to the rounding module.

2.2.3 Rounding

The inputs to this stage are the p+1 digits resulting from the left shifting after the LZC, which contain the round digit, and the sticky bit, which is calculated in parallel with the LZC.

The rounding stage selects either the most significant p digits of the result and truncate the less significant digit, or their increment. This decision is done according the rounding direction, the sign of the result, as well as the round digit, and sticky bit. For example, if the rounding direction is rounding towards zero, then we always select the most significant p digits of the result without any increment. On the other hand, of the rounding direction was towards positive infinity, then the most significant p digits of the result are selected if the result is negative and their increment is selected if the result is positive. In case of rounding to nearest, ties away from zero. If the round digit less than five, the most significant p digits of the result are chosen, otherwise their increment is chosen. The sticky is needed in rounding to nearest ties to even, to determine if it was a Tie case or not.

The rounding unit supports the five rounding directions specified in the IEEE754-2008 standard, as well as two more rounding directions.

2.3 Decimal FMA using Combined Add/Round module

This design was presented by Ahmed El-Tantawi in [8]. The main disadvantage of this design is the LZA and the normalization shifting has to be done in the critical path before the addition, because the addition is combined with the rounding. And the rounding position will not be known before the normalization shifting. Which leads to higher than minimal delay.

2.3.1 Multiplier Tree

After decoding the significands, the multiplier and multiplicand are sent to the multiplier tree to get the multiplication result. The Radix-5 implementation proposed in [9] is used in this design as it provides the speed required with reasonable area. Moreover, it has a property that can simplify the rest of the design; that is one of the two resulting vectors of the multiplication (the sum vector) is always negative. This property is used to simplify the LZA design as will be discussed

later. The multiplier tree used in this design is essentially the same as the one proposed in [9].

The multiplier is divided into two main stages: Generation of the decimal partial products and reduction of the partial products into only two vectors (sum and carry vectors). Each digit in the multiplier is recoded from the regular BCD-8421 format where$Y_i \in \{0,\cdots,9\}$ into the radix-5 encoding $Y_i = 5 \times Y_i^U + Y_i^L$, where $Y_i^U \in \{0,1,2\}$ and$Y_i^L \in \{-2,\cdots,2\}$. This results in a 32 digit multiplier (16 digits Y^U and 16 digits Y^L). Each Y^L digit selects a positive multiplicand multiple of $\{0,X,2X\}$ in (4221) format, while each Y^U digit selects a positive multiple of $\{0,5X,10X\}$ in (5221) format. To generate the negative multiples $\{-X$ or $-2X\}$; the corresponding positive multiple (coded in 4221) is inverted to get the 9's complement. Then a (+1) has to be added to each negated partial product to get the 10's complement. this (+1) is inserted without extra delay as will be shown. The 32 partial products are then aligned according to their decimal weights in order to be used as an input to the 32:2 decimal CSA tree.

Multiplier Recoding As explained in the previous section, each digit of the multiplier $Y_i \in \{0,\cdots,9\}$ is recoded into two parts: Y_i^U and Y_i^L such that $Y_i = 5 \times Y_i^U + Y_i^L$, where $Y_i^U \in \{0,1,2\}$ and $Y_i^L \in \{-2,\cdots,2\}$. Each digit Y_i^U is represented as two signals $\{y1_i^U, y2_i^U\}$, and each digit Y_i^L is represented as four signals $\{y(+2)_i^L, y(+1)_i^L, y(-1)_i^L, y(-2)_i^L\}$ and a sign bit ys_i, .

Multiplicand Multiples Generation The following Multiples need to be generated in (4221) format for each multiplicand ($\pm$X, $\pm$2X, 5X, and 10X):

The X BCD multiplicand: is easily done by recoding the Multiplicand into (4221).

Multiple 2X: each BCD digit is recoded into (5421) format, Then the multiplicand is shifted one bit to the left, to get the 2X multiple in (BCD-8421). Then, result is recoded into (4221).

Multiple 5X: It is obtained by a simple 3-bits left shift of the (4221) recoded multiplicand, with resultant digits coded in (5211).

Multiple 10X: It is obtained by a simple 3-bit left shift of the $2X$ (4221) recoded multiplicand multiples, with resultant digits coded in (5211).

Negative Multiples: For negative multiples (i.e. $ys_i = 1$), the positive multiple is inverted to get the 9's complement. For 10's complement, a (+1) has to be added at the least significant digit position. Since only the Y_i^L multiples may be negative, the (+1) is inserted in the least significant bit of the corresponding Y_i^U multiple.

Partial Product Array As detailed before, the SD radix-5 architecture produces 32 partial products: 16 of them are coded in (4221) format and the other 16 are coded in (5211) format. The next step is to reduce these 32 partial products to only 2 vectors.

Partial Product Reduction In order to reduce the partial products , The longest columns (the middle ones) are first reduced using decimal counters. The result is then reduced using a tree of carry save adders. This tree is designed to reduce the critical path delay using parallelism as much as possible and balancing the delay of different paths. To achieve such parallelism, the intermediate results that needs to be multiplied by a factor of 2 or 4 is multiplied in parallel with the reduction of the remaining intermediate results that doesn't need to be multiplied. The$\times 4$ operation is simply composed of two cascaded $\times 2$ stages.

To reduce the multiplier array to only two vectors, 32 of this carry save adders tree $(n:2)$ are needed where $n \in \{2,4\cdots,16,32\}$. These CSA trees can be found in [7]. At the end of this stage, the 32 partial products are reduced to only two vectors (the sum vector M1, and the carry vector M2) both of 33 digits and in the (4221) format. As stated before, the sum vector is always negative and the carry vector is always positive.

2.3.2 Decimal Carry Save Adder

In this stage, the addend, which is aligned, negated if needed, and ready to be added, is inserted with the two resulting vectors of the multiplier tree into a decimal 3:2 carry save adder (CSA). The resulting two vectors are converted from (4221) to BCD-8421 format in order to be used in the LZA block. Finally selection is done to select 3p+1 digits that contains the result starting from the MSD and discard any digits to the left of the MSD, and any digits to the far right that will contribute only to the sticky bit.

2.3.3 Leading Zeros Anticipation

The leading zeros in the sum of the two resulting vectors of the decimal CSA are anticipated in this stage. The LZA used in this design is also used in our work, and will be explained later in Section 3.5.1. It has to mentioned that the used LZA has an error in anticipation of one digit. This error is handled in the combined add/round stage.

2.3.4 Final alignment

The two vectors are shifted by the amount of the leading zeros, unless the preferred exponent is reached. Alignment is done before the combined add/round module in order to know the location where the rounding will take place.

2.3.5 Rounding Set Up

2.3.5.1 Top Level Architecture

Figure 2.2 shows the top level architecture of the rounding set up module. The main target of this stage is to calculate, in parallel to the combined add/round module, the guard and the round digits, the sticky bit, and the possible carry in to the most significant (p-digits) fed to the combined add/round module. To calculate these signals, the two vectors passed to the rounding set up have to be added.

In order to maintain only p-digit carry rippling delay at the critical path, the rounding setup is implemented as a conditional adder. The 2p+1 width is divided into p and p+1, the most significant p+1 digits are calculated twice using two carry networks. One of them assumes $C_{in} = 1$ to this part of the adder, while the other assumes $C_{in} = 0$. In order to use a Kogge-Stone binary carry network [10], a pre-correction stage that adds 6 to each digit is necessary to perform decimal addition using the binary adder. The correct carry out resulting from the least significant p digits selects the appropriate carry signals out of the two carry networks.

A small post-correction circuitry is used to generate the correct guard and round digits. However, the less significant digits contribute only to the sticky and there is no need for their exact values. Hence, no post correction is needed, and they are only processed to get their correct sticky share. The carry out of the most significant carry network is fed to the combined add/round module.

2.3.5.2 The addend 10's complement

In case of effective subtraction, the addend is 9's complemented without adding the (+1) required for 10's complement. Hence, we will use the carry in (C_{in}) of the least significant carry network to perform the required (+1) increment.

However, if the intermediate result is negative, it will need another 10's complementing which requires adding (+1). In order to avoid this, we use the following property:

$$10's\,comp.(X) = 9's\ comp.(X) + 1 = 9's\ comp.(X-1) \tag{2.1}$$

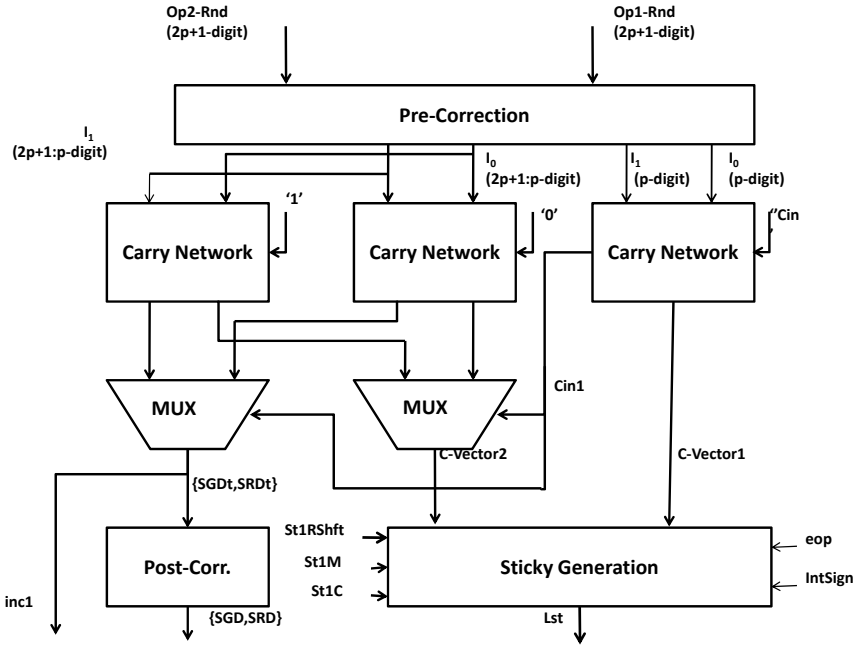

Figure 2.2: Rounding Setup

Hence, if the effective operation is subtraction and the intermediate result is negative no need to add $C_{in} = 1$. It will be sufficient to get the 9's complement of the intermediate result. However, if the intermediate result is positive, the $C_{in} = 1$ must be added, to account for the (+1) required for the 10's complement of the addend.

2.3.6 Combined Add/Round

This stage performs both BCD addition and rounding. The Combined Add Round Technique is proposed in [11]. However, this design is implemented for floating point adders and needs some modifications to fit this FMA.

These differences require different modifications in the design to handle them without delay or area overheads.

2.3.6.1 Rounding Position

As discussed previously, the rounding position is not exactly determined. It has an uncertainty of one digit due to the uncertainty of the preliminary anticipation of leading zeros. Figure 2.3 shows the unrounded result supposed to be produced from the two operands fed to the combined add/round module with three different

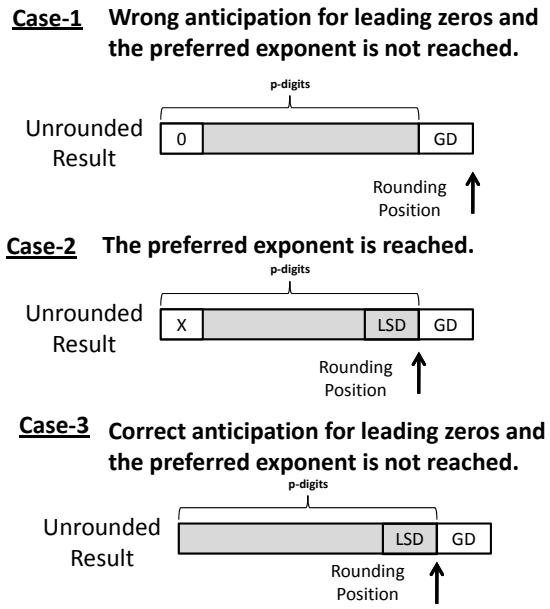

Figure 2.3: Rounding Position

cases in which the rounding position has two possibilities: either the least significant digit, or the guard digit produced from the rounding set-up stage.

Hence, the rounding increment $inc2$, fed to the most significant highlighted p-digits, must be calculated correctly according to the current case. This requires detecting the MSD of the unrounded result to determine whether it is zero or not, besides using the signal that indicates if preferred exponent is reached. This signal is generated from the final alignment module.

Figure 2.4 shows a top-level block diagram for the combined add/round module.

2.3.6.2 Pre-Correction

To allow the use of a fast binary adder that uses a Kogge-Stone carry network, p-digit BCD operands Op1-Add and Op2-Add are processed in the pre-correction stage. It performs the digit additions $(Op1_{Add})_i + (Op2_{Add})_i + 6$ in a 4p-bit binary 3:2 CSA, obtaining the 4p-bit sum and carry operands S and C_{int}.

Each +6 bias, coded in BCD as (0, 1, 1, 0), is connected to an input of a 4-bit binary 3:2 CSA. The p-digit BCD operands Op1-Add and Op2-Add are introduced into the other two inputs.

However, this stage is also used to prepare for the computation of the $SUM - 1$ required in case of $\overline{S_{lsb}}.cmp = 1$ where $SUM = (Op1_{Add}) + (Op2_{Add})$. Therefore,

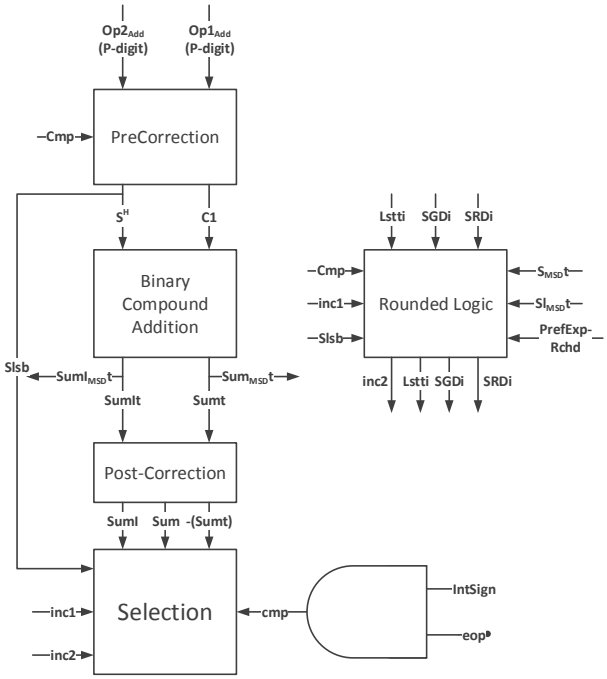

Figure 2.4: Combined/Add Round Module

the least significant digit of the correction vector is replaced by '5' instead of '6' in this case. This implements the (-1) decrement without needing any extra logic.

2.3.6.3 Compound Adder

The compound adder proposed in [12] is used in this design. It is shown in Figure 2.5. This adder computes $Sumt = S + C$ and $Sumlt = S + C + 1$. Where S, and C are the inputs to the combined/add module after the pre-correction.

2.3.6.4 Rounding Stage

In parallel to the binary sum, the decimal rounding unit computes the increment signal inc2 and the guard digit of the result. Apart from the rounding mode, the rounding decision depends on the intermediate sign, the guard, the round and the sticky digits computed in the rounding set-up stage, the sum least significant bit (Slsb) and the unrounded result MSD. The MSD of the unrounded result depends on the MSD of Sumt, Sumlt, Slsb and inc1.

2.3.6.5 Rounding Conditions

In addition to the five IEEE 754-2008 decimal rounding modes [2], this design supports two additional rounding modes [6]: round to nearest down and away from zero.

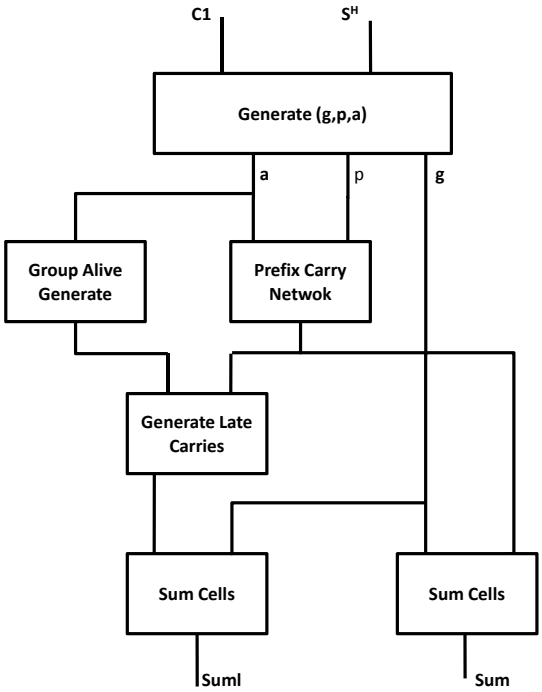

Figure 2.5: Compound Adder

2.3.6.6 Post-Correction and Selection

The post-correction stage is very simple. A '10' is added digit wisely to both Sumt and Sumlt to produce corrected sums: Sum and Suml. If the intermediate result is negative and the result will be complemented, it does not need post-correction. Hence, inverting of an excess-6 BCD number is equivalent to getting the 9's complement of an 8421 BCD number. The selection stage selects the correct output out of Sum, Suml, inverted version of Sumt or Sumlt.

2.3.7 A Decimal Floating-point Fused Multiply-Add Unit with a Novel Decimal Leading-zero Anticipator

In 2011 Akkas et al proposed w new Decimal-64 FMA in [13].

2.3.7.1 Multiplier Tree

The radix-10 multiplier presented in [7] is used to multiply the first two operands. The result is 32 digits that represent the multiplication result. This is the same multiplier explained in Section 2.2.1.

2.3.7.2 Operand Alignment

Instead of shifting the addend to align it with the multiplication result, to have the width of the result in 4P digits, where P is the number of digits in the significand, the multiplication result and the addend are both aligned at the same time. The result of the multiplication and the addend may be swapped according to which one has the larger exponent. The vector with the larger exponent is shifted to the left by the exponent difference, unless this shift amount is larger than the leading zeros count, in that case the other vector is shifted to the right by the difference between the required shift amount and the leading zeros count (the actual shift amount).

2.3.7.3 Addition

The addition is done using a Kogge-Stone parallel prefix adder. First the significands are pre-corrected, then the carries and flags are calculated using Kogge-Stone networks, and finally a post-correction is done using the generated carry and flag bits.

2.3.7.4 Leading Zeros Anticipation

In parallel to the Kogge-Stone adder, a leading zeros anticipator works on the pre-corrected operands, to calculate the leading zeros in the addition result. The LZA is divided into two stages:

The first stage computes two vectors (e1 and e2). Each bit of e1 has the information whether the corresponding digit in the sum of the two inputs are zero or not in case of effective addition. In case of effective subtraction e2 has this information.

The second stage is a binary Leading Zeros Detector (LZD) that detects the leading zeros in e1 and e2 in parallel.

Finally the effective operation signal (EOP) selects between the results of the two LZDs.

2.3.7.5 Final Shift And Rounding

The sum calculated by the Kogge-Stone adder is shifted by one digit location to the right, in case of an overflow in the addition, or by a massive amount, up to P digit locations. The result is finally rounded based on one of the five rounding directions specified in the IEEE-754 standard.

2.4 Binary Floating–Point Fused Multiply–Add with Reduced Latency

This architecture was proposed by T. Lang in [14].

The top level design of this binary FMA is shown in Figure 2.6.

2.4.1 Multiplier Tree

In this design a radix-4 binary multiplier tree is used. This is the same as the binary multiplier tree used in our work which is explained in Section 3.2.1, and 3.2.2.

2.4.2 Preparing the addend

The addend is prepared (shifted and complemented if needed) in parallel with the multiplier tree. The two resulting vectors from the multiplier tree as well as the prepared addend are then sent to a 3:2 CSA to get the final two vectors that need to be added in order to get the final sum.

2.4.3 Addition and Rounding

In this design, T. Lang tried to reduce the total latency by combining the addition with rounding. In the addition stage he prepared two vectors: sum, and sum+1, then the rounding increment selects the correct sum; if the rounding increment = 0, then the sum is chosen, and if it was 1 the sum+1 is chosen.

Since the addition is combined with rounding, the result has to be normalized before the final addition/rounding module. As if the result is not normalized before rounding, the rounding position is not known, and the rounding can not be done. Hence, normalization must be done before the addition/rounding module.

2.4.4 Leading Zeros Anticipation and Normalization Shifting

Instead of waiting for the LZA to finish Anticipation and get the final normalization shift amount, the LZA operates in parallel with the normalization shifter. The LZA generates the MSB of the shifting amount first, then the shifter uses this bit

to shift the result. Once the second MSB is generated, it's used to shift the result again, and so on. However, there is a gap at the beginning of the anticipation process before any bits are calculated. This gap is filled with complementing the vectors, and a part of the most significant bits addition.

After the normalization shifting is done, the addition can start. The most significant 51 bits are sent to the adder to produce sum, and sum+1. In parallel to this adder, the rounding decision is made using the least significant 3 bits. Once the addition is done, and the rounding decision is made, the correct result can be selected.

2.5 Binary/Decimal FMA

The only binary/decimal FMA was proposed by Monsson in 2008 [4], unfortunately it was not fully functional according to the standard. The top level architecture is shown in Figure 2.7.

2.5.1 Multiplier Tree

In the multiplier tree used in this design, each multiplier digit selects two multiples from the set of (0,A,2A,4A,5A) where A is the multiplicand. Two multiplexers are used to generate the partial products corresponding to each multiplier digit as follows:

$$mux_i = (0, 2A, 4A, 4A)\, where\, i = 2b_3 + b_1 \tag{2.2}$$

$$mux_j = (0, A, 4A, 5A)\, where\, j = 2(b_3 \oplus b_2) + b_0 \tag{2.3}$$

where $b_3b_2b_1b_0$ are the bits of the Multiplier digit. A simple Wallace tree of binary CSAs are used, and a correction logic is enforced if a carry comes out of the edge of any digit.

2.5.2 Alignment

Alignment is done in parallel with the reduction tree. A two way shifter is used to align the addend with the multiplication result. If the exponent of the addend is larger than the exponent of the multiplication result, then it's shifted to the left. And if not, it's shifted to the right. Finally, there is a limitation on the shifting

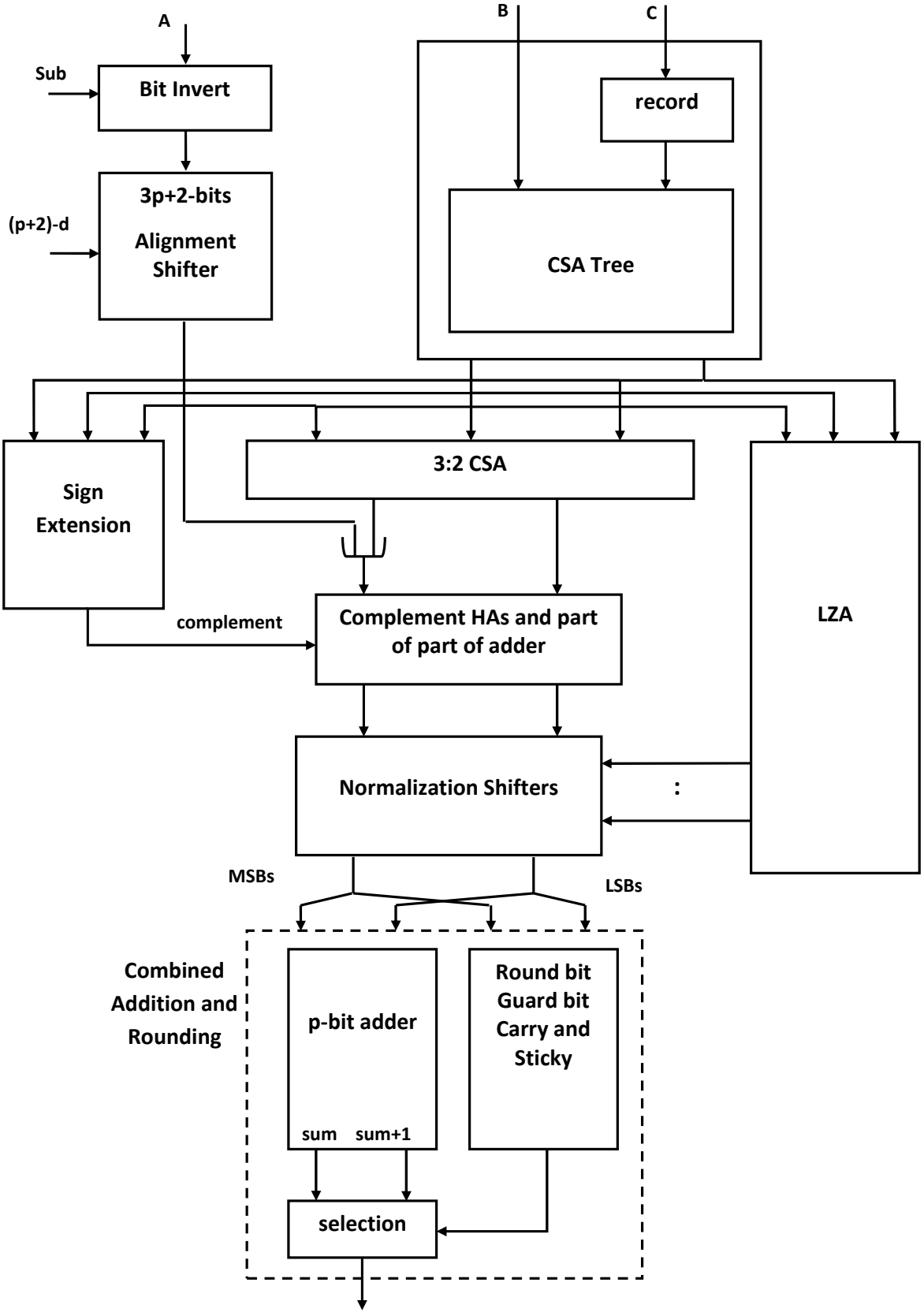

Figure 2.6: T. Lang's FMA

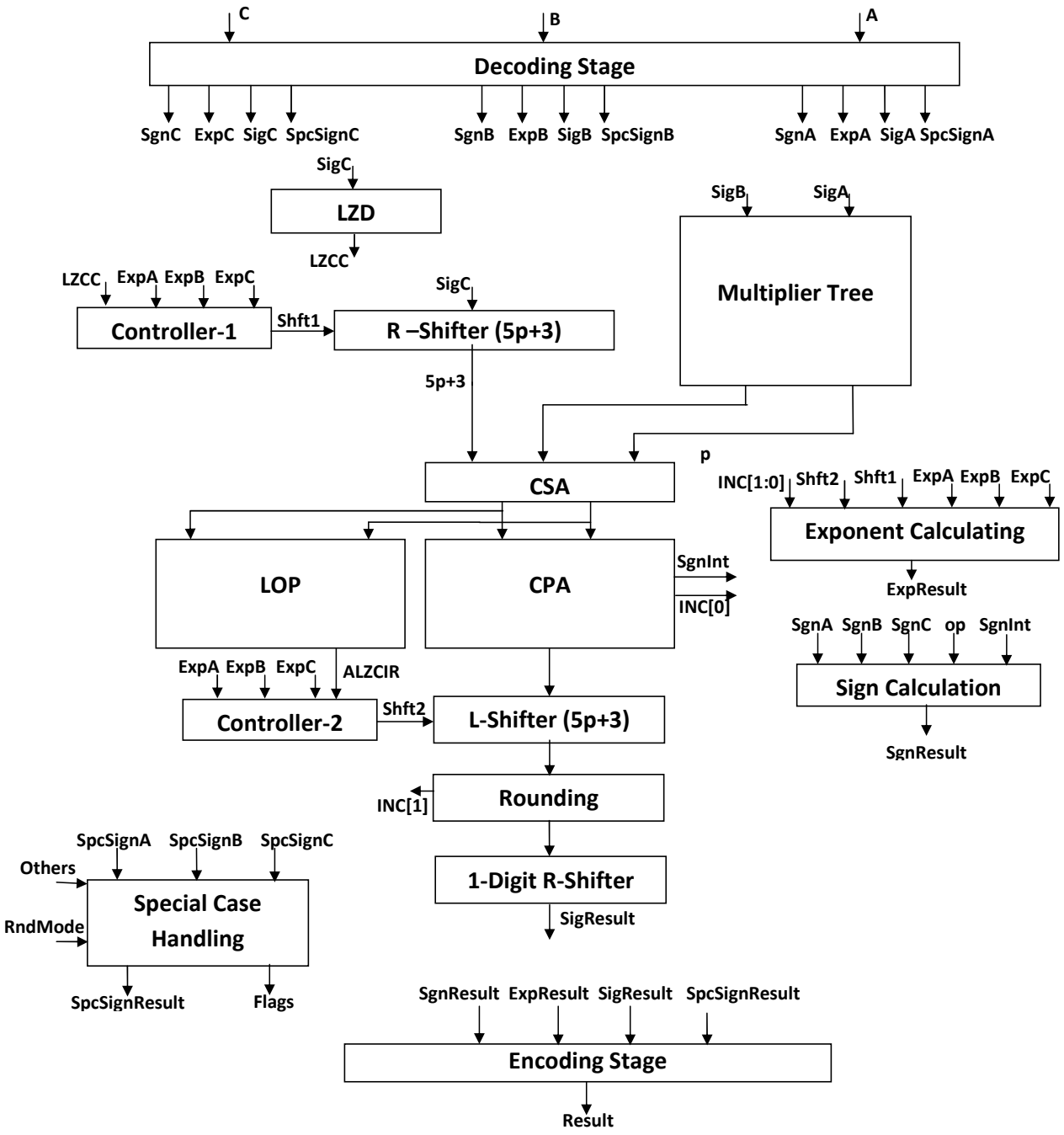

Figure 2.7: Monsson's Binary/Decimal FMA

amount: the addend should be shifted to the left until its LSB and the guard and round digits are to the left of the multiplication result.

2.5.3 Addition:

Addition is performed over two steps: CSA stage, CPA stage, with some correction logic for decimal. The addition is straight forward in binary, a simple 3:2 CSA is used to reduce the vectors from 3 to 2, then a binary CPA is used to perform the addition of these two vectors. Decimal addition can also be performed using a binary two's complement adder if the operands are pre-corrected. For effective addition, one operand must be pre-corrected by adding each digit to +6 which is already done by the multiplier. For effective subtraction one operand must be nine's complemented and pre-corrected by adding each digit to +6. This is the same as getting the fifteen's complement which is a simple bit inversion. A row of 4-bit CPAs, with +6 post-correction for each 4-bits if their value is larger than 9 or they produce a carry out, is used to reduce the number of decimal inputs to two from three; instead of the binary CSA. Finally the simple 2's complement adder is then used for both binary and decimal. But in case of decimal a post-correction is performed.

2.5.4 Leading Zeros Anticipation (LZA):

The location of leading one (non zero) digit is anticipated from the two inputs of the adder. For both binary and decimal, prediction strings are generated for both binary and decimal in parallel, and the leading one in these strings are calculated.

2.5.5 Rounding

Rounding is done in two steps: rounding decision, and increment. The decision is made by some logic gates depending on the rounding direction. A CPA incrementer is used to add the increment to the final result.

2.5.6 Design Functionality

Monsson himself states that the design doesn't work according to the standard in many cases. Also he tried only 30 test cases on his FMA which is a tiny number of test vectors to check such a massive input space of an FMA.

2.6 Conclusion

In this chapter we explored the main blocks required to build a functional FMA such as the multiplier tree, the LZA, and how the final addition and rounding could be done. We also presented most of the previously published FMAs, and highlighted the main differences in their implementation. In the next chapters we are going to present our combined binary/decimal FMA, give the details of implementing each block, and compare the synthesis results of our FMA with the previous FMAs presented in this chapter.

Chapter 3

Proposed Binary/Decimal Design

In this chapter we are going to propose a new 64-bit FMA that can work as a binary or decimal unit. The main idea is to re-use as much hardware as possible between binary and decimal to save area and hence static power. The most expensive units -area wise- are the multiplier and the adder, so we chose both the multiplier and the adder such that each of them can work as a binary or decimal multiplier and adder.

Our multiplier is based on the multi operand adder proposed by L. Dadda in [15], and the adder is based on the adder K. Yahia proposed in his master's thesis [16] .

A selection bit called (bd) is used to choose between binary and decimal architectures. When this signal is high the design operates as a binary FMA, and if it's low, the design operates as a decimal FMA. The block diagram of the presented FMA is shown in Figure 3.2.

3.1 Decoding The Inputs

In case of binary the inputs are in binary-64 format which is shown in Figure 3.1. The decoding is pretty much straight forward, we just split the most significant bit as the sign bit, the following 11 bits as the biased exponent, and the least significant 52 bits are the trailing significand. The trailing significand is then concatenated to the hidden one unless the number is subnormal or zero. The exponent of the number (Exp) is defined as the biased exponent - 1023.

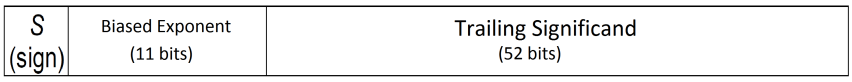

Figure 3.1: Binary-64 Format

Exponent	Mantissa	Value of the number
11111111111	$=0$	NAN
11111111111	$\neq 0$	(-1)$^{Sign}* \infty$
00000000000	$=0$	0
00000000000	$\neq 0$	$(-1)^{Sign} * 0.Trailing\,Significand * 2^{-1022}$

Table 3.1: Special Cases in Binary-64

Hence the numerical value of the binary double precision normalized number is $(-1)^{Sign} * 1.Trailing\,Significand * 2^{Exp}$. Special cases, such as subnormal numbers, infinities and NANs are shown in Table 3.1.

In case of decimal the three operands (OpA, OpB, and OpC) are read in the IEEE 754-2008 decimal-64 format with decimal encoding. The decimal encoding uses the Densely Packed Decimal (DPD) code. Each operand is decoded into its sign bit, exponent, significand, and flags for special values (NaN or infinity). The significand is then decoded to BCD-8421 format, where each digit of the 16 digits is represented in 4-bits. As shown in Figure 3.3, decimal numbers are decoded such that their sign is the MSB. The next 13-bits are the combination field, which is decoded according to Table 3.2 to get the MSD and the exponent of the decimal number, and to determine if the number is a special number (infinity, quite NaN or signaling NaN). The trailing significand which is 15 digits is decoded from the trailing part. Each successive 10-bits are decoded according to Table 3.3 to get three significand digits.

<table>
<tr><th colspan="2">Combination Filed (G)
$G_{12}G_{11}\cdots\cdots G_7$</th><th colspan="2">Special Case or
Exponent and MSD</th></tr>
<tr><td colspan="2">$0xxxxx$</td><td colspan="2">$MSD = 0G_5G_6G_7$,$E_{biased} = G_{12}G_{11}G_6G_5\cdots G_0$</td></tr>
<tr><td colspan="2">$10xxxx$</td><td colspan="2">$MSD = 0G_5G_6G_7$,$E_{biased} = G_{12}G_{11}G_6G_5\cdots G_0$</td></tr>
<tr><td colspan="2">$110xxx$</td><td colspan="2">$MSD = 100G_7$,$E_{biased} = G_9G_8G_6G_5\cdots G_0$</td></tr>
<tr><td colspan="2">$1110xx$</td><td colspan="2">$MSD = 100G_7$,$E_{biased} = G_9G_8G_6G_5\cdots G_0$</td></tr>
<tr><td rowspan="3">$1111xx$</td><td>$11110x$</td><td rowspan="3">$MSD = 0000$
$E_{biased} = 00\cdots0$</td><td>Infinity</td></tr>
<tr><td>111110</td><td>Signaling NaN</td></tr>
<tr><td>111111</td><td>Quiet NaN</td></tr>
</table>

Table 3.2: Decoding The Combination Field

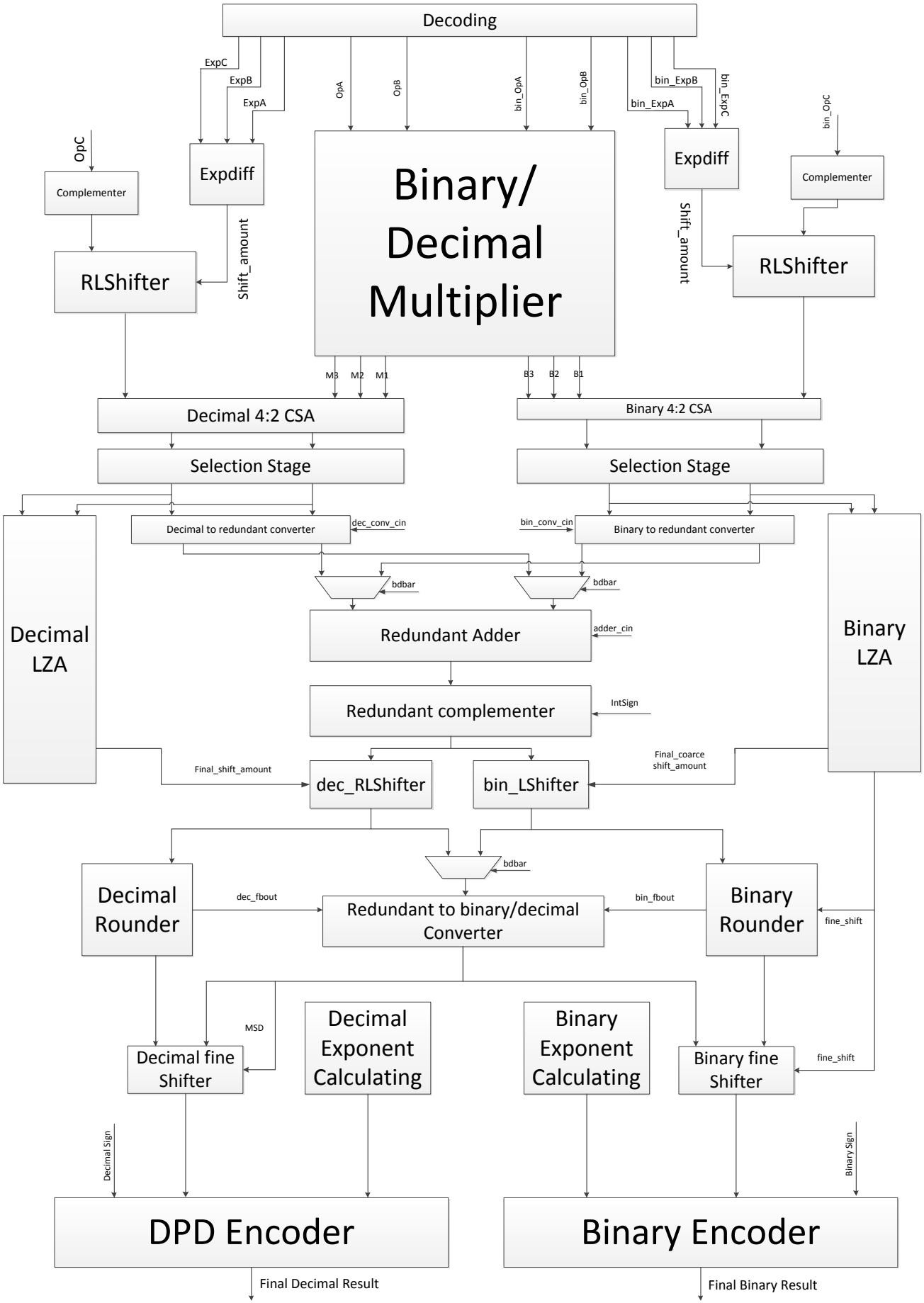

Figure 3.2: Top Level Architecture

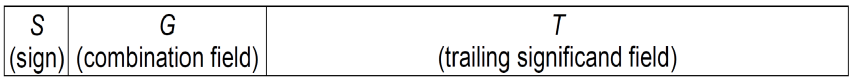

Figure 3.3: Decimal-64 DPD encoding

Declet $(I_9 \cdots\cdots I_0)$ $(I_3 I_2 I_1 I_6 I_5)$	BCD $\{O^2\}\{O^1\}\{O^0\}$ $\{O_3^2 O_2^2 O_1^2 O_0^2 O_3^1 O_2^1 O_1^1 O_0^1 O_3^0 O_2^0 O_1^0 O_0^0$
0xxxx	$\{0, I_9, I_8, I_7\}\{0, I_6, I_5, I_4\}\{1, I_2, I_1, I_0\}$
100xx	$\{0, I_9, I_8, I_7\}\{0, I_6, I_5, I_4\}\{1, 0, 0, I_0\}$
101xx	$\{0, I_9, I_8, I_7\}\{1, 0, 0, I_4\}\{1, I_2, I_1, I_0\}$
110xx	$\{1, 0, 0, I_7\}\{0, I_6, I_5, I_4\}\{0, I_9, I_8, I_0\}$
11100	$\{1, 0, 0, I_7\}\{1, 0, 0, I_4\}\{0, I_9, I_8, I_0\}$
11101	$\{1, 0, 0, I_7\}\{0, I_9, I_8, I_4\}\{1, 0, 0, I_0\}$
11110	$\{0, I_9, I_8, I_7\}\{1, 0, 0, I_4\}\{1, 0, 0, I_0\}$
11111	$\{1, 0, 0, I_7\}\{1, 0, 0, I_4\}\{1, 0, 0, I_0\}$

Table 3.3: Decoding each declet to the corresponding BCD digits

3.2 Multiplier Tree

The multiplier is the most important block of the FMA. In [8] it consumes almost 33% of the total delay, and takes about 50% of the total area. In our design it uses 39% of the total delay and 43% of the total area. So we have to start multiplication as soon as possible. The significands of the first two decoded operands (A and B) are sent right after the decoding stage to the multiplier to get the multiplication result (M).

The multiplier tree consists of two parts: partial products generation, and partial product reduction. In the generation stage, the partial products are generated, the partial products array is formed and the sign extension is handled and reduced to minimize the size of the partial product array. In the reduction stage these partial products are reduced to only three vectors in order to be added to the prepared addend (C).

3.2.1 Multiplicand Multiples Generation

The SD Radix-5 architecture proposed in [9] is used for decimal, but instead of generating the partial products in 4221, and 5211 format they are generated in BCD-8421 format in order to fit the used reduction tree, which uses the inputs in BCD-8421 format. This simplifies the Multiplicand Multiples Generation , making it faster with less area and power.

In SD Radix-5, each multiplier digit is recoded from the normal digit set $B \in \{0,1,2,...9\}$ to the Radix-5 encoding: $B_i = 5 \times B_i^U + B_i^L$, where $B_i^U \in \{0,1,2\}$ and $B_i^L \in \{-2,\cdots,2\}$.

The following multiplicand multiples needs to be generated ($\pm$A, $\pm$2A, 5A, and 10A) in BCD-8421 format. It should be noted that all these multiples are easy decimal multiples that can be obtained with no carry propagation, but with a simple O(1) logic with a few gate delays, using the properties explained in Section 1.1, as explained below:
The A BCD multiplicand: is the same input generated by the input decoder.

Multiple 2A: Each BCD digit is first recoded to the (5421) format then shifted one bit to the left, obtaining the 2A multiple in BCD-8421 format.

Multiple 5A: Each BCD digit is first recoded to the (4221) format then shifted three bits to the left with resultant digits coded in (5211). Finally each Digit is Converted back to BCD-8421 format.

Multiple 10A: The BCD digits are just shifted one decimal location (four bits) to the left obtaining the 10A multiple in the required format.

Negative Multiples: For negative multiples needed in the Lower partial products, we get the 9's complement with a simple two gate delay logic. For 10's complement, a (+1) is added at the least significant digit position of the corresponding Upper partial product.

It can be noted that in the 10A multiple, the least significant bit is always zero, so the (+1) is inserted in that bit. Also in 5A multiple, and after the three-bit left shift, the least significant bit is also zero, so the (+1) increment can be inserted in that bit, before the final conversion to BCD-8421. So in case of negative B_i^L multiple, the (+1) increment is inserted in the corresponding B_i^U multiple.

For binary, the SD Radix-4 architecture presented in [7] is used. The SD Radix-4 decodes each 4-bits of the multiplier $B \in 0,1,2,...15$, as well as a carry-in from the lower significant 4-bits, into two digits : upper and lower digit, and a carry out (Cout) to the next four bits. This carry out is generated directly from the input and doesn't depend on the value of Cin as shown in Equation 3.5. Hence it doesn't cause any carry propagation delay overhead. $B_i + Cin = 16 \times Cout + 4 \times B_i^U + B_i^L$,

where the upper digit $B_i^U \in \{-2,-1,0,1,2\}$ and the lower digit $B_i^L \in \{-2,\cdots,2\}$. We need to generate the following multiplicand multiples ($\pm$A, $\pm$2A, $\pm$4A, and $\pm$8A). All the positive multiples can be generated by a simple left shifting of the multiplicand bits.

For example, $B_i = 6$ is decoded into $B_i^L = 2$, $B_i^U = 1$, and Cout = 0, where $6 = 16\times 0 + 4\times 1 + 2$. Also $B_i = 13$ is decoded into $B_i^L = 1$, $B_i^U = -1$, and Cout = 1, where $13 = 16\times 1 - 4\times 1 + 1$.

In order to generate the negative multiples without extra delay, we get the one's complement of the corresponding positive multiple by inverting each bit, and the (+1) required for the two's complement is added in a separate vector (the I-vector in the right bottom corner of the graph) as shown in Figure 3.6.

3.2.2 Partial Products Generation

As mentioned above, for decimal, in the used Radix-5 encoding each multiplier digit is recoded into two digits: lower and upper digits, and a sign bit, each of these digits selects a partial product. The lower digit selects from {0,A,2A}, and the upper digit selects from {0,5A,10A}. The sign bit negates the partial product selected by the lower digit as mentioned above.

In order to simplify the partial product selection process, each digit Y_i^U is represented as two signals $\{y1_i^U, y2_i^U\}$. Also each digit Y_i^L is represented as four signals $\{y(+2)_i^L, y(+1)_i^L, y(-1)_i^L, y(-2)_i^L\}$ and a sign bit ys_i. The truth table for these seven signals are shown in Table 3.4. By examining the different possibilities, these signals can be obtained directly from the BCD multiplier digits Y_i using the following logical expressions, where $Y_i = \{y_{i,3}, y_{i,2}, y_{i,1}, y_{i,0}\}$is the multiplier digit being recoded.

$$(Y_i^U) \quad \begin{cases} y2_i^U = y_{i,3}; \\ y1_i^U = y_{i,2} + (y_{i,1}.y_{i,0}); \end{cases} \tag{3.1}$$

$$(Y_i^L) \quad \begin{cases} y(+2)_i^L = y_{i,1}\,.\,((y_{i,2}\,.\,y_{i,0}) + (\overline{y_{i,2}}\,.\,\overline{y_{i,0}})); \\ y(+1)_i^L = (\overline{y_{i,3}}\,.\,\overline{y_{i,2}}\,.\,\overline{y_{i,1}}\,.\,y_{i,0}) + (y_{i,2}\,.\,y_{i,1}\,.\,\overline{y_{i,0}}); \\ y(-1)_i^L = (y_{i,3}\,.\,y_{i,0}) + (y_{i,2}\,.\,\overline{y_{i,1}}\,.\,\overline{y_{i,0}}); \\ y(-2)_i^L = (y_{i,3}\,.\,\overline{y_{i,0}}) + (\overline{y_{i,2}}\,.\,y_{i,1}\,.\,y_{i,0}); \\ ys_i = y(-2)_i^L + y(-1)_i^L; \end{cases} \tag{3.2}$$

After selecting the partial products, we have 32 decimal partial products, each of 17 digits as shown in Figure (3.4). Some of them are negative and need their sign to be extended.

In the case of binary, the multiplier upper digit selects from {0,4A,8A} and the lower digit selects from {0,A,2A}. The following signals are generated to simplify the selection of the multiplicand multiples$\{y(+2)_i^L, y(+1)_i^L, y(-1)_i^L, y(-2)_i^L\}$ for the lower digit, and $\{y(+8)_i^U, y(+4)_i^U, y(-4)_i^U, y(-8)_i^U\}$ for the upper digit. Also $ys_i^U$and ys_i^L, which are the sign of the upper and lower digits respectively, are generated.

The truth table for these signals are shown in Table 3.5 and the equations are shown below, where cin is the Cout from the hexadecimal digit i-1, and Y_i = $\{y_{i,3}, y_{i,2}, y_{i,1}, y_{i,0}\}$is the multiplier digit being recoded.:

$$(Y_i^L) \quad \begin{cases} y(+1)_i^L = (\overline{y_{i,1}} \cdot \overline{y_{i,0}} \cdot cin) + (\overline{y_{i,1}} \cdot y_{i,0} \cdot \overline{cin}); \\ y(+2)_i^L = \overline{y_{i,1}} \cdot y_{i,0} \cdot cin; \\ y(-1)_i^L = (\overline{y_{i,0}} \cdot y_{i,1} \cdot cin) + (y_{i,1} \cdot y_{i,0} \cdot \overline{cin}); \\ y(-2)_i^L = \overline{y_{i,0}} \cdot y_{i,1} \cdot \overline{cin}; \\ ys_i^L = (y_{i,1} \cdot \overline{y_{i,0}}) + (y_{i,1} \cdot \overline{cin}); \end{cases} \tag{3.3}$$

$$(Y_i^U) \quad \begin{cases} y(+4)_i^U = (\overline{y_{i,3}} \cdot \overline{y_{i,2}} \cdot y_{i,1}) + (\overline{y_{i,3}} \cdot y_{i,2} \cdot \overline{y_{i,1}}); \\ y(+8)_i^L = \overline{y_{i,3}} \cdot y_{i,2} \cdot y_{i,1}; \\ y(-4)_i^U = (\overline{y_{i,2}} \cdot y_{i,3} \cdot y_{i,1}) + (y_{i,3} \cdot y_{i,2} \cdot \overline{y_{i,1}}); \\ y(-8)_i^U = \overline{y_{i,2}} \cdot y_{i,3} \cdot \overline{y_{i,1}}; \\ ys_i^U = (y_{i,3} \cdot \overline{y_{i,2}}) + (y_{i,3} \cdot \overline{y_{i,1}}); \end{cases} \tag{3.4}$$

$$Cout_i = y_{i,3} \tag{3.5}$$

The sign bit inverts the partial products to get the one's complement and the (+1) increment for each partial product is inserted in a separate vector. Three zeros were added to the left of the multiplier 53 bits in order to generate the proper values for the partial products, these 56 bits are divided into 14 groups of 4 bits each. Hence, in the case of binary, the partial product array consists of 28 partial products plus extra partial product for the (+1) increments. These 28 partial products may be negative and need to properly extend their sign.

Digit	$y(+1)^L$	$y(+2)^L$	$y(-1)^L$	$y(-2)^L$	$Y1^U$	$Y2^U$	ys
0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0
2	0	1	0	0	0	0	0
3	0	0	0	1	1	0	1
4	0	0	1	0	1	0	1
5	0	0	0	0	1	0	0
6	1	0	0	0	1	0	0
7	0	1	0	0	1	0	0
8	0	0	0	1	0	1	1
9	0	0	1	0	0	1	1

Table 3.4: Generated Signals For Multiplier Recoding in Decimal SD-Radix5

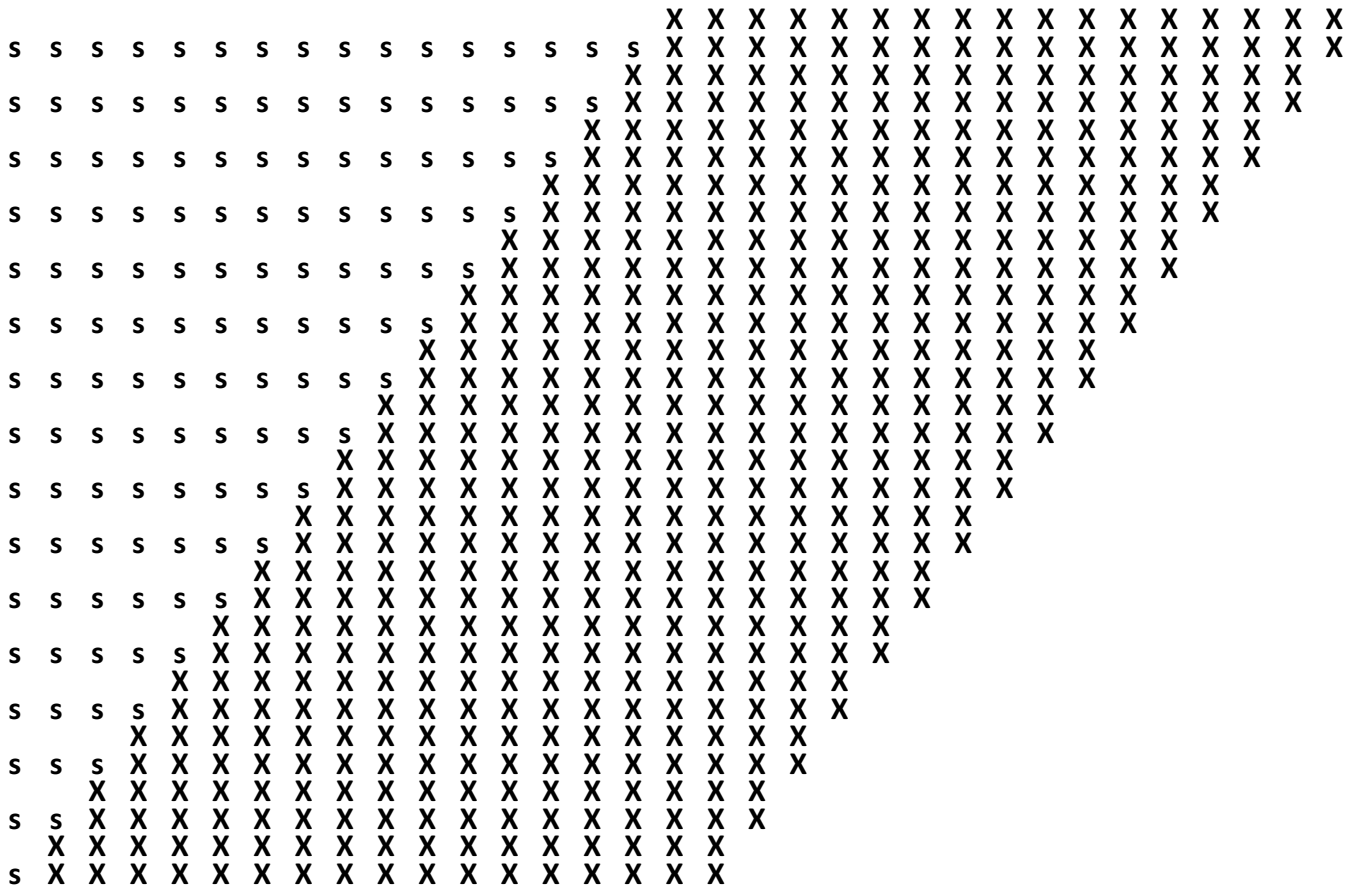

Figure 3.4: Decimal Partial Product Array before offline reduction

Y	*cin*	$y(1)^L$	$y(2)^L$	$y(-1)^L$	$y(-2)^L$	ys^L	$y(4)^U$	$y(8)^U$	$y(-4)^U$	$y(-8)^U$	ys^U	*cout*
0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	1	0	0	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	0	0	0	0
1	1	0	1	0	0	0	0	0	0	0	0	0
2	0	0	0	0	1	1	1	0	0	0	0	0
2	1	0	0	1	0	1	1	0	0	0	0	0
3	0	0	0	1	0	1	1	0	0	0	0	0
3	1	0	0	0	0	0	1	0	0	0	0	0
4	0	0	0	0	0	0	1	0	0	0	0	0
4	1	1	0	0	0	0	1	0	0	0	0	0
5	0	1	0	0	0	0	1	0	0	0	0	0
5	1	0	1	0	0	0	1	0	0	0	0	0
6	0	0	0	0	1	1	0	1	0	0	0	0
6	1	0	0	1	0	1	0	1	0	0	0	0
7	0	0	0	1	0	1	0	1	0	0	0	0
7	1	0	0	0	0	0	0	1	0	0	0	0
8	0	0	0	0	0	0	0	0	0	1	1	1
8	1	1	0	0	0	0	0	0	0	1	1	1
9	0	1	0	0	0	0	0	0	0	1	1	1
9	1	0	1	0	0	0	0	0	0	1	1	1
10	0	0	0	0	1	1	0	0	1	0	1	1
10	1	0	0	1	0	1	0	0	1	0	1	1
11	0	0	0	0	0	1	0	0	1	0	1	1
11	1	0	0	0	0	0	0	0	1	0	1	1
12	0	0	0	0	0	0	0	0	1	0	1	1
12	1	1	0	0	0	0	0	0	1	0	1	1
13	0	1	0	0	0	0	0	0	1	0	1	1
13	1	0	1	0	0	0	0	0	1	0	1	1
14	0	0	0	0	1	1	0	0	0	0	0	1
14	1	0	0	1	0	1	0	0	0	0	0	1
15	0	0	0	1	0	1	0	0	0	0	0	1
15	1	0	0	0	0	0	0	0	0	0	0	1

Table 3.5: Generated Signals For Multiplier Recoding in Binary SD-Radix4

	Case(1)	Case(2)	Case(3)	Case(4)
$S_1 \cdots S_1 S_1$	$0 \cdots 00$	$0 \cdots 00$	$9 \cdots 99$	$9 \cdots 99$
$S_2 \cdots S_2 S_2$	$0 \cdots 00$	$9 \cdots 99$	$0 \cdots 00$	$9 \cdots 99$
$S_R \cdots S_R S_R$	$0 \cdots 00$	$9 \cdots 99$	$9 \cdots 99$	$9 \cdots 98$

Table 3.6: The four possibilities of two sign digit vectors

	Case(1)	Case(2)	Case(3)	Case(4)
$Q_1 \cdots Q_1 Q_1$	$0 \cdots 01$	$0 \cdots 01$	$0 \cdots 00$	$0 \cdots 00$
$Q_2 \cdots Q_2 Q_2$	$9 \cdots 99$	$9 \cdots 98$	$9 \cdots 99$	$9 \cdots 98$
$S_R \cdots S_R S_R$	$0 \cdots 00$	$9 \cdots 99$	$9 \cdots 99$	$9 \cdots 98$

Table 3.7: equivalent sign digit values

3.2.3 Sign Extension

3.2.3.1 Decimal Sign Extension

As mentioned above, for decimal, we have 32 partial products in the partial product array, 16 of them are generated by B_i^U which are always positive and don't need sign extension. The other 16 partial products, which are generated by B_i^L might be negative, and need their sign to be extended. However, the part of the sign extension can be reduced offline. Since the sign digit (S) can only be (9) or (0), the possibilities of each two successive sign vectors are explored in order to reduce the part of the sign extension in the array, and hence the area and power of the reduction tree needed to reduce them.

Table 3.6 shows the four possibilities of two sign digit vectors S_1 and S_2 and their sum S_R. The same S_R value can be obtained from the values of the modified sign digits (Q_1 and Q_1) shown in Table 3.7.

The four equivalent values of $Q_1$and Q_2 can be generically expressed as:

$$\begin{array}{|c|c|}\hline Q_1 \cdots Q_1 Q_1 & 0 \cdots 0(000\bar{s}_1) \\ \hline Q_2 \cdots Q_2 Q_2 & 9 \cdots 9(100\bar{s}_2) \\ \hline \end{array}, s_i = \begin{cases} 0\, S_i = 0 \\ 1\, S_i = 9 \end{cases}$$

Where Q is the sign digit after the first stage of reduction.

The same technique can be used to reduce the leading nines in each two successive vectors starting from the second sign extended row as follows:

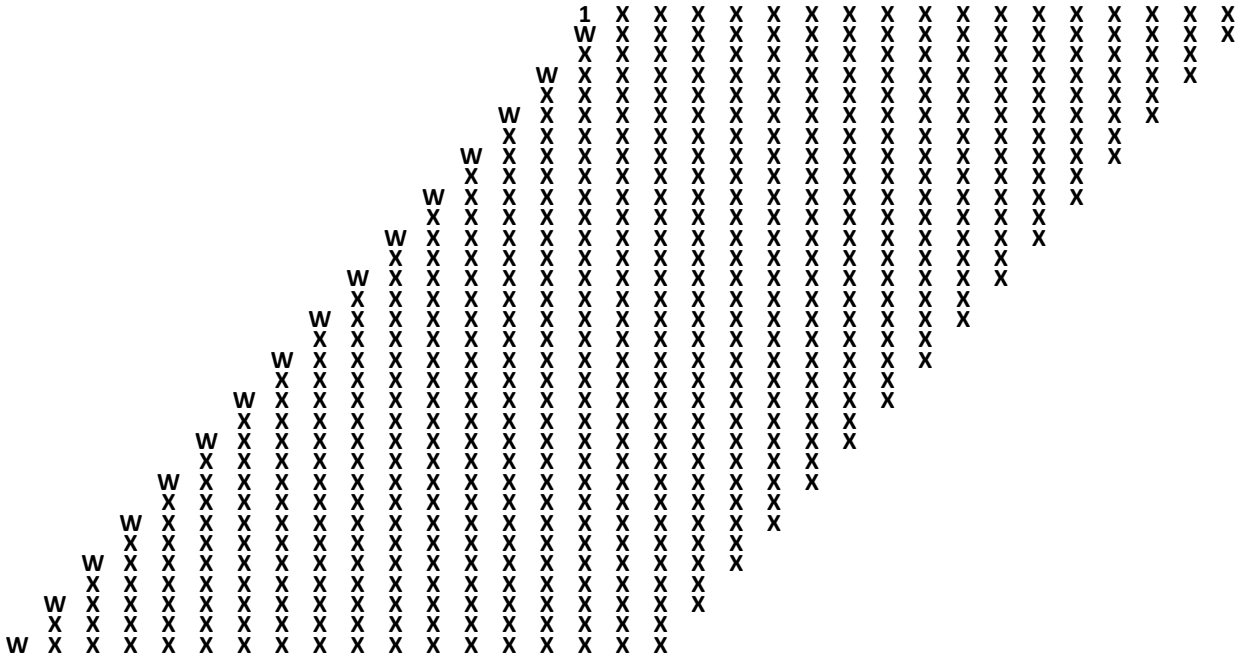

Figure 3.5: Final Decimal Partial Product Array

	Case(1)	Case(2)
$Q_1 \cdots Q_1 Q_1$	$9 \cdots 99$	$9 \cdots 99$
$Q_2 \cdots Q_2 Q_2$	$0 \cdots 09$	$0 \cdots 10$
$Q_R \cdots Q_R Q_R$	$0 \cdots 08$	$0 \cdots 09$

This can be replaced by:

	Case(1)	Case(2)
$W_1 \cdots W_1 W_1$	$0 \cdots 00$	$0 \cdots 00$
$W_2 \cdots W_2 W_2$	$0 \cdots 0(100\bar{s}_2)$	$0 \cdots 0(100\bar{s}_2)$

where W is the sign digit after the second reduction.

Now the sign extended partial product array is simplified offline, and the final array is shown in Figure (3.5).

3.2.3.2 Numerical Example

In order to explain the method shown above, imagine we have four vectors each is a 4-digit number resulting from the lower digit from the multiplier, and each number is extended by 4 digits of sign extension as shown:

7	6	5	4	3	2	1	0
S1	S1	S1	S1	X1	X1	X1	X1
S2	S2	S2	X2	X2	X2	X2	0
S3	S3	X3	X3	X3	X3	0	0
S4	X4	X4	X4	X4	0	0	0

Applying first reduction step, this will turn into:

7	6	5	4	3	2	1	0
0	0	$000\overline{S1}$	S1	X1	X1	X1	X1
9	9	$100\overline{S2}$	X2	X2	X2	X2	0
$000\overline{S3}$	S3	X3	X3	X3	X3	0	0
$100\overline{S4}$	X4	X4	X4	X4	0	0	0

Now applying second reduction step on vectors 2 and 3 starting from digit number 7 we get:

7	6	5	4	3	2	1	0
0	0	$000\overline{S1}$	S1	X1	X1	X1	X1
0	0	$100\overline{S2}$	X2	X2	X2	X2	0
0	$100\overline{S3}$	X3	X3	X3	X3	0	0
$100\overline{S4}$	X4	X4	X4	X4	0	0	0

Finally, we can look closer at digits 4 and 5 of vector 1:

	S1=1	S1=0
digits 4,5	09	10
	$0(100\overline{S1})$ +1	$0(100\overline{S1})$ +1

Finally, digits 4 and 5 of the first vector can be represented as $(0\ 100\overline{S1})$ + 1. This +1 is placed in the vector resulting from the first upper multiplier digit:

7	6	5	4	3	2	1	0
0	0	0	1	X1u	X1u	X1u	X1u
0	0	0	$000\overline{S1}$	X1	X1	X1	X1
0	0	$100\overline{S2}$	X2	X2	X2	X2	0
0	$100\overline{S3}$	X3	X3	X3	X3	0	0
$100\overline{S4}$	X4	X4	X4	X4	0	0	0

The final reduction tree is shown in Figure (3.5).

3.2.3.3 Binary Sign Extension

In the case of binary we have 28 partial products than may be negative. The sign extension bits of each partial products can be represented as $ss.....s$ where s is the sign of the partial product this can also be written as 11.....1 + $\overline{s}$. If $s = 1$ then the sign extension is 11.....1, and if $s = 0$ the sign extension is 100.....0, the same without sign extension but with extra carry out that must be taken into consideration while extending any sign later in the design, and while checking for overflow.

This is done for all the 28 partial products, and then the strings of ones of all partial products are added offline and replaced by their sum in order to reduce the hardware required to reduce them in the run time.

In order to use the same reduction hardware for binary and decimal, each binary partial product is divided into groups of four bits. During reduction each group is treated as a decimal digit. The final partial product array is shown in Figure 3.6. Where I is a vector containing the increments for all negative partial products, D is the hexa-decimal value D = 1101 in binary, and E is also the hexa-decimal value E = 1110 in binary, and Sn = $\overline{ys_i^U}$ Plus $\overline{ys_i^L}$ (As mentioned above, each partial product generates an ($\overline{s}$) to have its sign extension in the form of (111111 + $\overline{s}$), and since each upper and lower partial products are of the same weight, their $\overline{s}$ ($\overline{ys_i^U}$ and $\overline{ys_i^L}$) has to be added together to give the Sn signal).

3.2.4 Partial Products Reduction

The multi-operand decimal adder presented in [15] is used to add the digits in each column.

As shown in Figure 3.5, and 3.6 the partial product array consists of 32 columns (for binary three of them are zeros). Number of digits (N) in each column varies from 2 digits (in the last columns), up to 32 in the three middle columns.

Each column is reduced independently. The digits of each column are reduced using binary CSA tree into two binary numbers, which are then added using a binary carry look-ahead adder into the final sum of this column. The maximum number of bits in this sum is 9 bits. For binary this is the final sum and is divided again into groups of four bits (B3,B2,B1), where B2 has a weight of 16 times the weight of B1, and the weight of B3 is 16 times the weight of B2.

For decimal we need another stage to convert this sum into decimal weighted digits (units, tens and hundreds). This is done using the binary to decimal converter proposed in [15]. After conversion to decimal, the sum can be represented in two

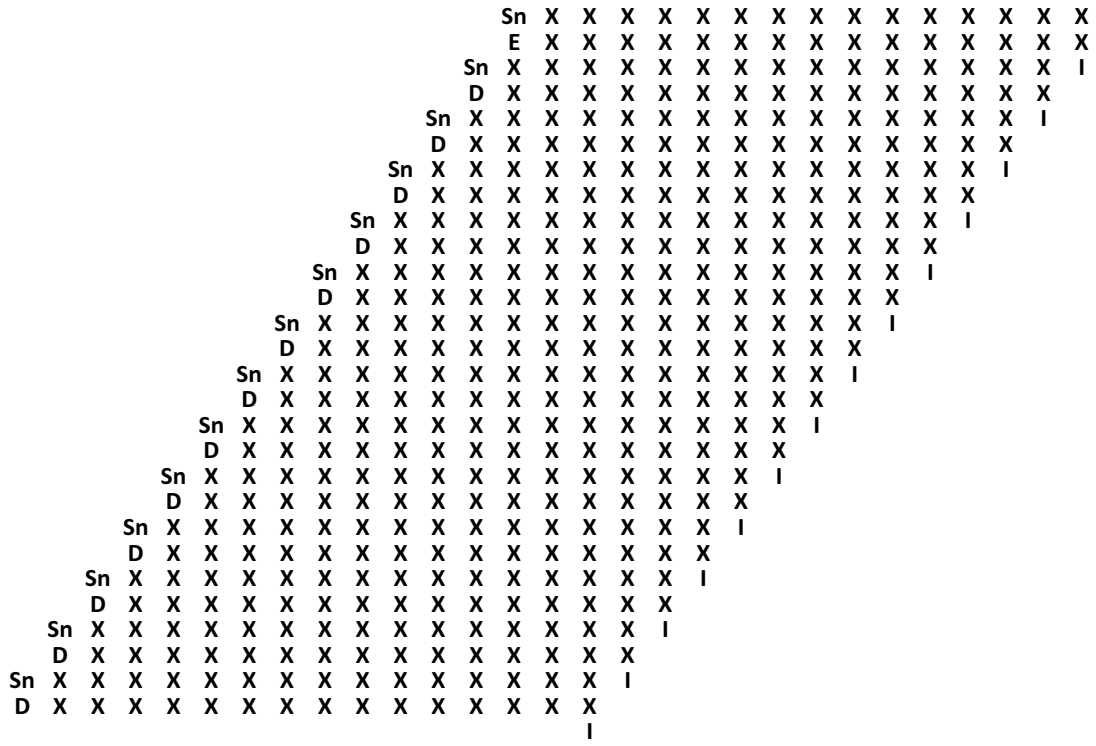

Figure 3.6: Final Binary Partial Product Array

or three digits depending on the length of the column. In the first five columns, the maximum sum of the digits (if all digits were 9's) is 90 as the number of digits per column is 10 in the fifth (longest) column of them, hence the sum of each of these columns can be represented in two digits (Units, and Tens). Also the sum of each of the last five columns are represented in two digits. Each of the middle 23 columns has the sum of its digits between 108 in the sixth column, and 288 in the longest columns, hence the sum of each of them can be represented in three digits (Units, Tens, and Hundreds).

As mentioned above, each column is reduced independently, which reduces the carry propagation between different columns, hence reduces the total switching power. The block diagram of the reduction of the longest column is shown in Figure 3.7.

3.2.5 Sign of the two resulting decimal vectors

If we look at the final column in the reduced partial product array, we can easily calculate its sum to be (w + 0) = w, which equals to either 8, or 9 depending on

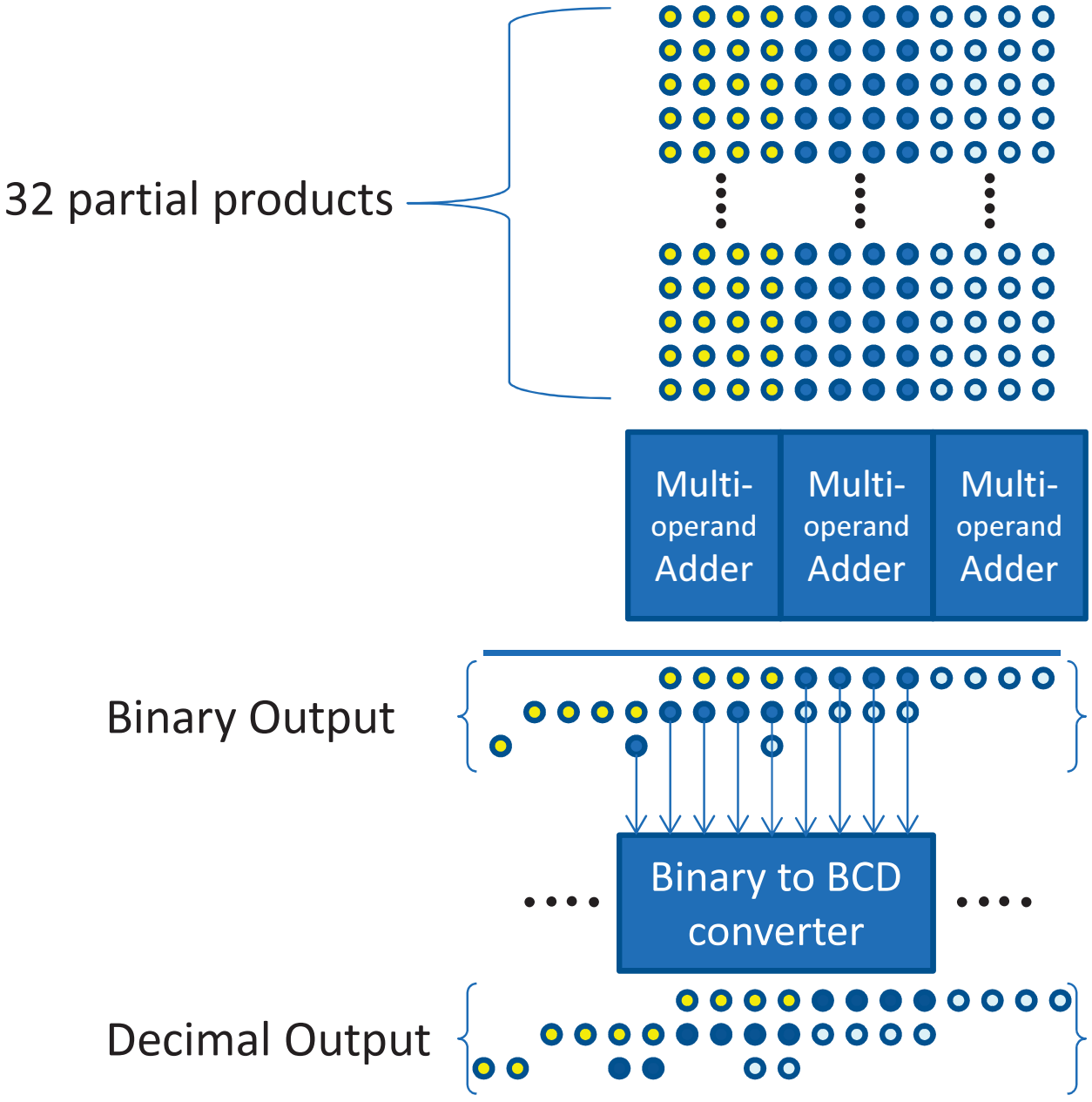

Figure 3.7: Longest Column Reduction

the sign of the 31^{st} partial product. Hence the resulting Units vector will have its last (sign) digit equals to either 8, or 9 which means it will always be negative.

If we look at the column before the last one, we can calculate its maximum sum as (w + 0 + x + x ∈[8,27]), which means that the last (sign) digit of the Tens vector will be either 0, 1, or 2 which means that the Tens vector will always be positive.

The Hundreds vector is also positive, since the last few columns will produce a sum less than a hundred, which leads to some leading zeros in the Hundreds vector.

After the first decimal 3:2 compressor in the CSA stage, which is explained in Section 3.4, the resulting sum vector (M1) will be negative, and the carry vector (M2) will always be positive. The sign digit of the inputs to this 3:2 compressor are shown in the following table:

Vector	4221 representation
U	111x (either 8 or 9)
T	0000
H	0000
M1	111x
M2	0000

3.3 Addend Preparation

The decimal addend is prepared to be added to the multiplication result in parallel with the Multiplier tree. First it's converted to (BCD-4221) format to simplify adding it with the result of the multiplier tree using decimal 3:2 compressor as will be shown below. To avoid adding more delay in the critical path by shifting the multiplication result, the addend is shifted in parallel with the multiplier tree. It is shifted to the right or to the left depending on the sign of the exponent difference ($ExpDiff$) between the result of the multiplication and the addend as shown in Equations (3.8),(3.9). Where $ExpA$, $ExpB$, $ExpC$, and $ExpM$ are the exponents of the multiplier, multiplicand, addend, and the multiplication result respectively and Bias is the exponent bias defined by the IEEE standard.

$$ExpM = ExpA + ExpB - Bias \tag{3.6}$$

$$ExpDiff = |ExpM - ExpC| \tag{3.7}$$

$$ShiftR = Max(ExpDiff, 2p+1) \quad ExpM > ExpC \tag{3.8}$$

$$ShiftL = Max(ExpDiff, 3p+1) \quad ExpM \leq ExpC \tag{3.9}$$

The maximum amount to shift the addend to the right is (2p+1) as shown in equation (3.8), other wise the addend only contributes to the sticky (unless the multiplication result (M)=0, yet this case is considered as an exception of the default data path) as it will be out of the selected operating width. In case of shifting to the left the maximum shift amount to be considered is (3p+1), otherwise the multiplication result only contributes to the sticky (unless the addend (C)=0, yet this case is also considered as an exception of the default data path) as it will be out of the selected operating width. Also the 9's complement of the addend is needed in case of effective subtraction. The 9's complement is easily calculated by inverting every bit in the recoded addend (since it's now in BCD-4221 format). We do that using one level of Xor gates that xor the addend with the effective operation (eop), where:

$$eop = signA \oplus signB \oplus signC \oplus op \tag{3.10}$$

where op is an input signal which defines either the operation is $A \times B + C$ or $A \times B - C$.

In case of effective subtraction, the 10's complement is needed, so a (+1) increment needs to be added to the 9's complement. This increment is added later in the conversion to redundant stage.

In case of binary, the addend is also prepared in parallel with the multiplier. First we get the exponent difference according to equations 3.6, and 3.7. While the exponent difference is calculated, the addend is complemented if the effective operation (eop) is subtraction. Then the addend is shifted to the right or to the left depending on the sign of the ExpDiff. The shift amount is calculated as follows:

$$binShiftR = Max(ExpDiff, 2P+1) \quad ExpM > ExpC \tag{3.11}$$

$$binShiftL = Max(ExpDiff, P+1) \quad ExpM \leq ExpC \tag{3.12}$$

Case	Condition	MSD
1	$ExpM > ExpC$	3p+1
2	$ExpC - LZC - p < ExpM \leq ExpC$	3p+1
3	$ExpC - LZC - p \leq ExpM \leq ExpC - LZC - 2p$	4p+1
4	$ExpC - LZC - p > ExpM$	5p+1

Table 3.8: Four Cases of Reduced Data path

The maximum right shift amount is calculated such that the addend is completely shifted out of the multiplication result by at least one bit, so that it contributes only to the sticky. The same concept applies in case of left shifting, the maximum shift amount is chosen such that the multiplication result is shifted out of the addend by at least one bit, hence the multiplication result contributes only to the sticky. This is shown in Figures 3.8 and 3.10.

3.4 Selection and Carry Save Adder

3.4.1 Case of Decimal

Instead of working with the full width (4p+1) digits, the width of the significand path is reduced in the selection stage. The selection stage determines the position of the most significant digit (MSD) and ignores the higher significant digits. The four cases of the position of the MSD are shown in Figure 3.8. The selection is done by the selection lines generated from the data path control unit. This unit also determines the amount of shifting the addend $Shift1$, and its direction. Table 3.8. also lists the four cases of the MSD position.

After the digits to the left of the MSD are removed, the selected digits of the aligned addend and the three resulting vectors of the multiplier tree (U, T, and H in) are added using a 4:2 Decimal Carry Save Adder (DCSA) that uses the properties of the BCD-4221 encoding to simplify and speed-up the addition. The structure of the DCSA is shown in Figure 3.9, the blocks inside the dotted area performs the $\times 2$ operation. The resulting sum and carry vectors of the DCSA are converted back to BCD-8421 as it's easier to use in the next stages that require carry generation networks as will be shown.

The selected operating width is 3p+1, as in case 1 the multiplication result might have some leading zeros up to (2p-1) zeros if it's not zero. In that case we will need extra p digits to the right of the multiplication result. Hence the selected width is 3p+1 digits and we will operate with that width in all cases.

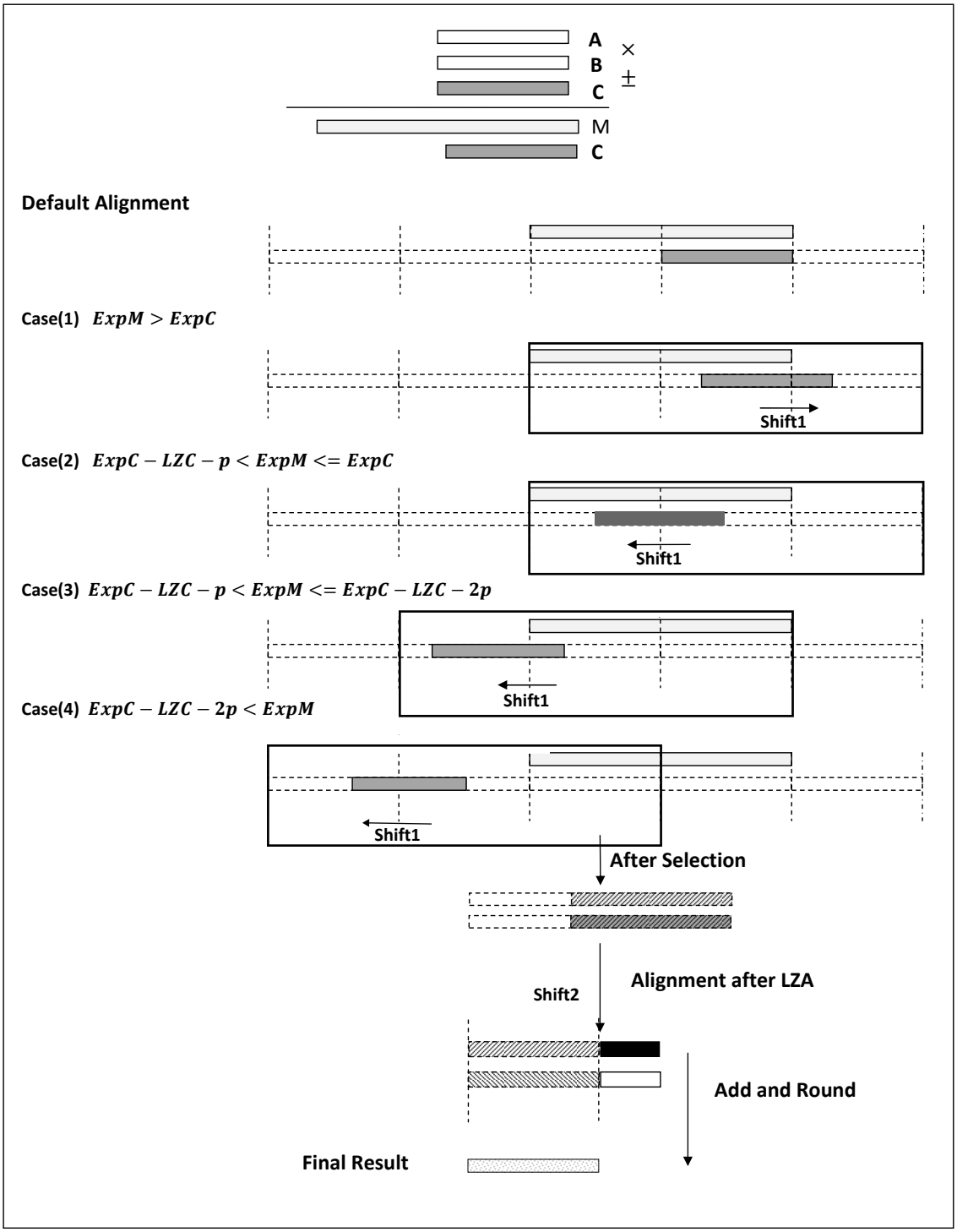

Figure 3.8: Decimal Operating Width Selection

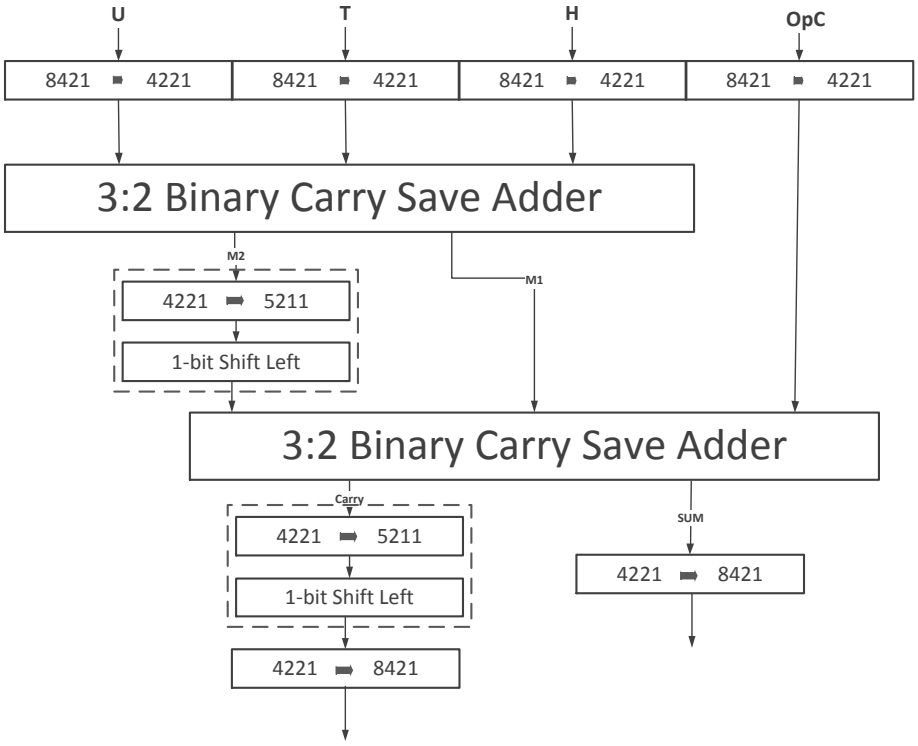

Figure 3.9: Decimal Carry Save Adder

On the other hand, as mentioned before, one of the vectors resulting from the first 3:2 DCSA (the sum vector M1) is negative and the other is positive. If we explored the two possibilities of the addend in case of effective addition and subtraction and the possible carry in at the sign digit, we can prove that, in all cases one of the resulting two vectors will be positive and the other will be negative.

3.4.2 Case of Binary

Unlike the case of decimal, the default alignment of the addend is not at the right most of the multiplication result. The default alignment is shown in Figure 3.10. In the binary data path, we have only two cases for selecting the location of the MSB in order to reduce the data path width to minimize area and delay.

Case1: Right Shift: In case of right shift (if ExpC $<$ ExpM), the multiplication result can't be subnormal as its exponent is larger than ExpC, i.e there is at least a leading one in the most significant two bits of the multiplication result. Hence the location of the MSB of the result is at the MSB of the multiplication result i.e at bit number (3p+1) and the intermediate exponent of the result (ExpRi) is ExpM.

Case2: Left Shift: In case of left shift (if ExpC $>$ ExpM) the location of the MSB of the result is at the MSB of the addend i.e at bit location 4p+2 and the intermediate exponent of the result (ExpRi) is ExpC + 54.

Special Cases: - The case of left shift by one bit or no shift at all: In that case, the MSB of the addend is aligned with the MSB of the multiplication result and huge cancellation can occur if the effective operation is subtraction, so if we treat it as Case2, we will need 3P bits as the operating width. However, in that case the MSB of C didn't Exceed the MSB of M, and it can be treated as Case1. This way the required width is also 2p+1.

-Case of SubnormalM and ZeroC, in that case and irrespective of the shifting amount or direction, the MSB is chosen at 4p+2 and the exponent is ExpM + 54 . This is done to avoid shifting the result to the right after the addition to normalize the exponent. Then it can be treated like Case2 but with different exponent.

So in All cases the width of the next hardware is limited to 2p+1. Unlike the case of decimal, where the required operating width is 3p+1, the case that forced us to work with 3p+1 width in decimal can't happen in binary, as in case of right shift, the multiplication result can't be subnormal so it can't have any leading zeros.

After selecting the operating width, the three resulting vectors from the multiplier tree (B1, B2, and B3) as well as the addend are reduced using a 4:2 binary CSA into only two vectors which will be added using the redundant adder as will be shown below.

3.4.3 Sign extension of the resulting vectors of the CSA

In cases 3,4 in decimal, and in case 2 in binary, where the location of the MSD is to the left of the MSD of the multiplication result (and the result from the CSA), the resulting vectors of the CSA has to be properly sign-extended. As mentioned before in the case of binary, there is an extra carry inside the sign extension of the multiplication result that will always propagate to the higher bits as the result of the multiplication must be positive. The same for decimal, one of the vectors from the multiplication result has to be negative and the other one is positive, hence a carry out will occur. This carry out, in both cases, can appear in the CSA stage, or after the final addition, and in both cases this is not considered as an overflow. If it didn't appear in the CSA stage, one of the resulting vectors has to be concatenated by leading nines (ones) in order to keep proper sign extension. And if this carry appeared in the CSA stage, the digits to the left of the CSA output are (999...9999)

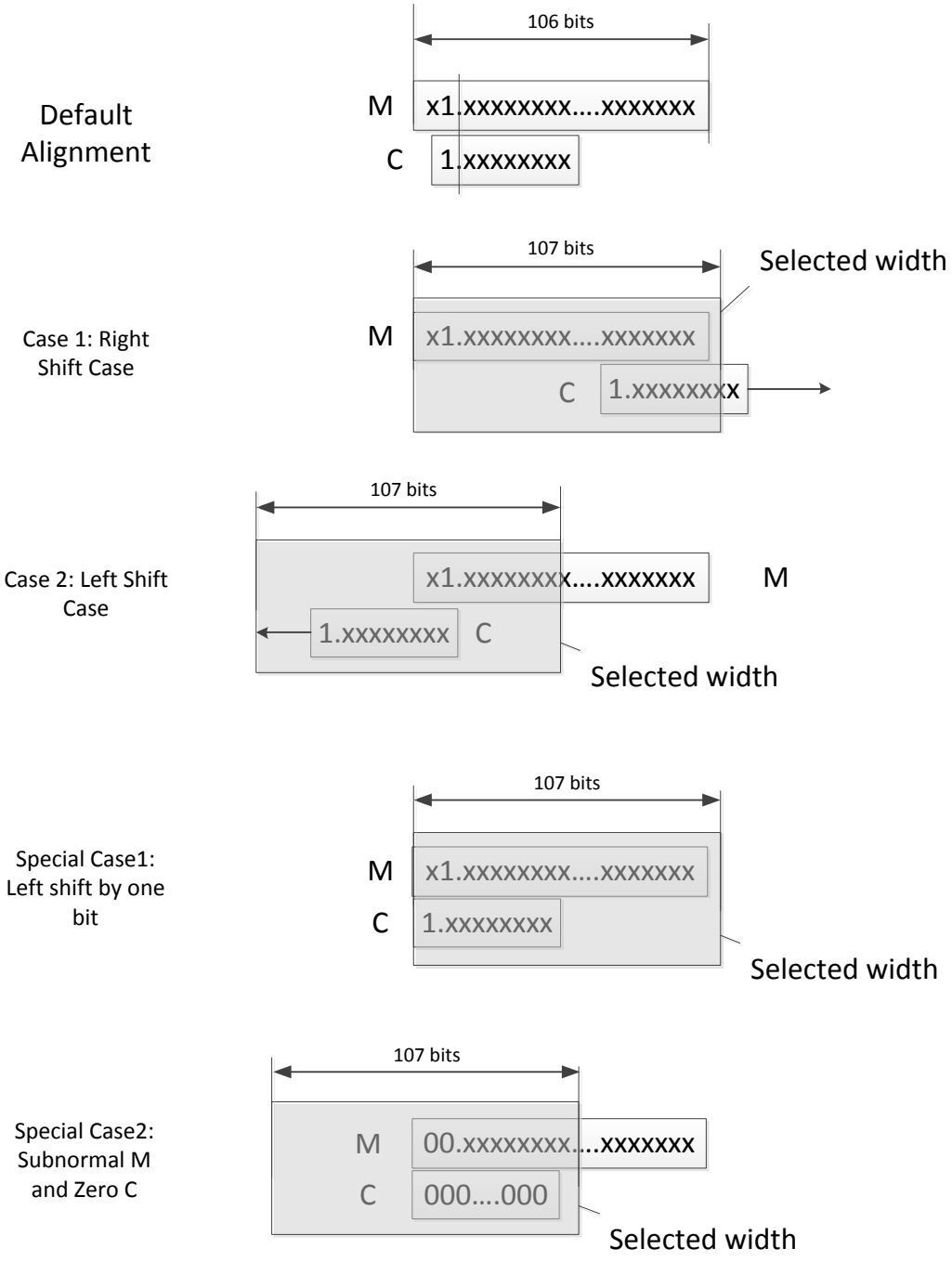

Figure 3.10: Binary Operating Width Selection

for proper sign extension, and now this carry is added to these nines making them all zeros. Hence in case a carry out appeared from the CSA, the resulting vectors of the CSA are concatenated with a string of leading zeros, and in case there was no carry from the CSA, one of the resulting vectors is concatenated with a string of leading nines (ones in case of binary).

3.5 Leading Zeros Anticipation

3.5.1 Decimal LZA

In parallel with the adder, that will be explained in Section 4, we have to anticipate the leading zeros count in the result from the two output vectors of the DCSA before adding them. This is done using an LZA as follows:

In general there are two types of LZAs: Exact and Inexact LZAs. Exact LZAs produce the exact leading zeros count directly, but it takes long delay and big area. Inexact LZAs produce the correct Leading Zeros Count (LZC) with an error of one. The inexact LZA requires another stage of correcting this error of one digit in the anticipation in order to produce the correct leading zeros count. In our work we used an inexact LZA for both binary and decimal. In binary a correction stage is used to correct the LZA error. In case of decimal the correction takes huge area and delay, hence the correction circuit is removed and the error in the anticipation is handled in the rounding stage.

The LZA consists of two main stages: the preliminary anticipation which anticipates the leading zeros with a possible error of one digit, and the correction unit that corrects that error. In this design there is no need for the correction stage, because the decimal rounder can handle this one digit error. That reduces the area and the delay of the LZA.

Next, the technique for leading zero anticipation in both effective addition and subtraction operations will be explained.

Since the inputs to the LZA in this design is almost in the same format as the inputs to to LZA in [8], we will use the same LZA used in [8].

3.5.1.1 Inputs to the LZA

The inputs to the LZA stage are the most significant 2p+1 digits of the two resulting vectors of the DCSA. As stated before, one of the two vectors is positive and the other one is negative.

In case of effective subtraction, the addend is negated in the 4221 format but the (+1) increment is not added yet, so the negative vector can be considered in 9's complement format.

In case of effective addition, the addend is positive, and the negative vector comes from the multiplier tree which is in the correct 10's complement format. and in this case there is no need for the (+1) increment.

3.5.1.2 Effective Subtraction Case

In effective subtraction case, the preliminary leading zero anticipator proposed in [17] is used with small modifications as explained below: The inputs to the LZA in this case are A and $\overline{B}$ (the nine's complement of B), and we want to anticipate the leading zeros in $(A-B)$. To anticipate the leading zeros we apply the following steps:

1) Encode the two operands digit wisely to the following signals (g9, g2, g1, zero, s1, s2, s9). The meaning of each of these signals is shown in Table 3.9, where A_i is the i^{th} digit in operand A, and B_i is the i^{th} digit in operand B.

Signal	Condition	Signal	Condition
$g9_i$	$A_i = 9, B_i = 0$	$s9_i$	$A_i = 0, B_i = 9$
$g2_i$	$A_i \geq B_i + 2$	$s2_i$	$A_i \leq B_i - 2$
$g1_i$	$A_i = B_i + 1$	$s1_i$	$A_i = B_i - 1$
$zero_i$		$A_i = B_i$	

Table 3.9: Signals used for Leading Zeros Anticipation

2) Explore different digit pattern sets and get leading zero count of each pattern.

3) Extract a binary string P that has leading zeros equivalent to the leading zeros count in the result of $(A-B)$ with an error of one digit. Note that the string P will consist of 33 bits.

4) Use an LZD detect the leading zeros in the binary string (P), we get the leading zeros count as follows (where LZR is the correct leading zeros count):

$$LZA\{A,\overline{B}\} = PLZR\{A-B\} = LZR \; or \; LZR-1 \tag{3.13}$$

3.5.1.3 Effective Addition Case

In case of effective addition, the previously proposed decimal leading zero anticipator in [17] is not the best design for our case as will be discussed below.

In the design proposed in [17] the preliminary leading zero anticipation in case of addition is calculated as $PLZR_{effadd} = Min\{LZCOp1,\ LZCOp2\}$ where $LZCOp1,\ LCZOp2$ are the leading zero count of the added operands. This gives the leading zeros count with a possible error of (+1) digit. Although the leading zeros count of the operands of the LZA are not known up to this stage and has to be calculated in the critical path, This can be avoided by using the LZC of the initial operands that needs to be added. The first operand is the addend , its LZC can be detected in parallel with the Multiplier tree and it doesn't add any extra delay in the critical path. The other operand is the Multiplication result (M). It can be anticipated with a possible error of (-1) digit error as follows : $PLZM = LZCA + LZCB$, for $M = A \times B$. If we used this anticipation to get the PLZC of the result such that $PLZR_{effadd} = Min\{PLZCM,\ LZCC\}$, this will result in one of the three possibilities: $PLZR_{effadd} = LZCR - 1$ *or* $LZCR$ *or* $LZCR + 1$. This will complicate the decimal rounder since we will have three possible positions for rounding.

Thus the proposed technique in [17] is not suitable for this architecture and the problem of the leading zero anticipation in case of effective addition is reformulated to overcome this issue. As previously stated, in case of effective addition, one of the two operands is positive and the other is negative in the 10's complement format. Using this fact, the problem can be reformulated as follows:

$$LZA\{A, \overline{B} + 1\} = LZA\{A, \overline{B + 1}\}$$

$$= PLZCR\{(A - (B + 1))\} = PLZCR\{(A - 1) - B)\}$$

Hence the problem of getting the PLZC of (A,B) in case of effective addition is reformulated to be an LZC problem in case of effective subtraction for (A-1,B). The effect of the (-1) is examined in the different digit patterns in order to get the correct PLZC, and it's been proven in [8] that the same hardware as the case of effective subtraction is needed in this case to produce either the correct LZC or the correct LZC-1, and as in the case of effective subtraction, this (-1) can be tolerated in the rounding stage.

3.5.1.4 Leading Zeros Detector

The LZD used here is the one proposed in [18].

The main block in this LZD tree is the LZD of 2-bit string. This block takes two bits $B1,\ B0$ where $B1$ is the most significant bit and $B0$ is the least significant

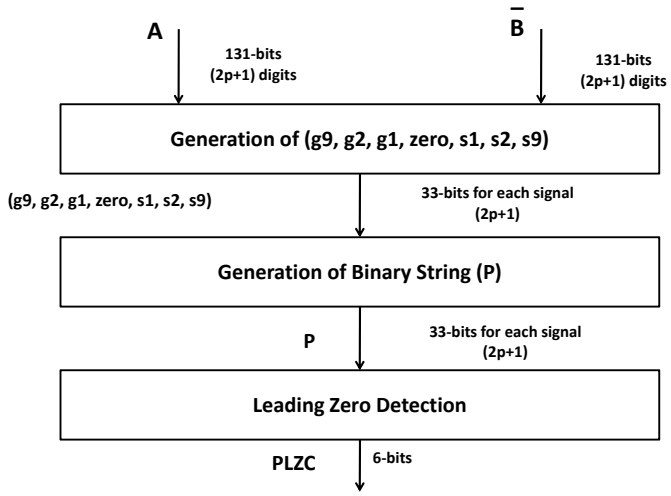

Figure 3.11: LZA block diagram

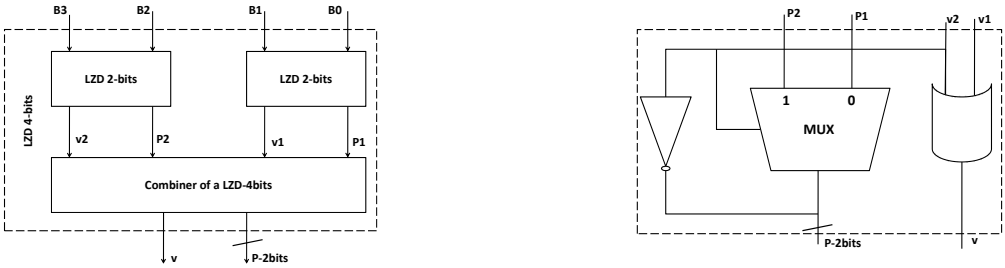

Figure 3.12: (a)LZD for 4-bit Binary String (b)Internal Structure of LZD4

one. Then, it generates two signals the valid signal (v) and the LZC signal (P) such that:

$$v = B0 \,|\, B1 \tag{3.14}$$

$$P = \overline{B1} \tag{3.15}$$

The valid signal indicates if the two bits are zeros or not and the leading zero count signal 'P' indicates if there is a leading zero bit or not. Two of these blocks can be used to detect the LZC signal in a four bit binary string such as Figure 3.12. As shown in Figure 3.12 (b), the two bit signal (P) counts the number of leading zeros in the binary string ($B_3B_2B_1B_0$) and the signal $v = 0$ if all the bits in the string are zeros.

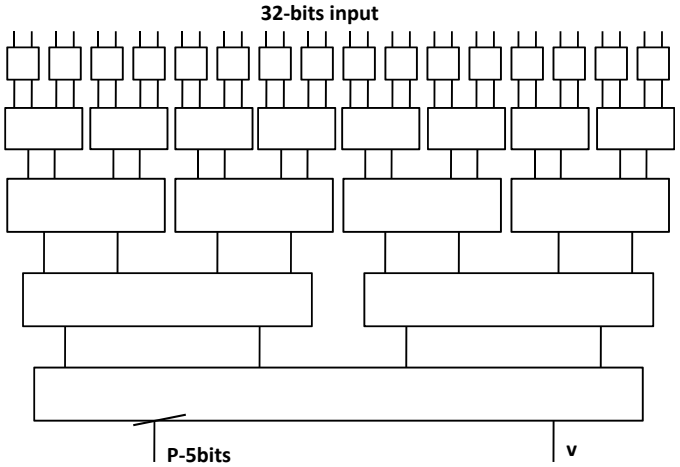

Figure 3.13: Leading Zero Detector of 32-bit Binary String

This can be generalized for any number of bits. Hence, a 32 LZD can be implemented as 5 levels starting with 16 LZDs of 2-bits as shown in Figure 3.13.

3.5.2 Binary Leading Zeros Anticipation

3.5.2.1 Inexact LZA

Same as the case of decimal, we have to anticipate the leading zeros in the result from the two resulting vectors of the CSA in order to perform proper normalization shift of the intermediate result after the addition. Since the intermediate result may be positive or negative, we have to count the leading zeros or the leading ones depending on the sign of the intermediate result. The Inexact LZA proposed in [19] is used. The main idea is based on a property that if two normalized vectors are added then a string of leading zeros will be found if and only if this pattern is detected starting from the MSB: $p^i g k^j$ In that case the leading zeros count is $i+j+1$, where i and j are the longest values that this formula holds for. In order to detect such a pattern a string x is generated as shown in Equations 3.16, and 3.17. The leading zeros count in the string x is one less than the number of leading zeros in the result, and is detected using a leading zeros detector as explained below.

$$x_0 = p_0 \tag{3.16}$$

$$x_i = \overline{p_i}\,\overline{k_{i-1}} \qquad , i > 0 \tag{3.17}$$

3.5.2.2 Leading Zeros Detector

As will be shown later, the result of the addition has to be shifted while in the redundant format. In the redundant system used in this work, each redundant digit (4-bits) contains the information of one octal digit (3-bits). So the minimum shifting step is one redundant digit, which is three bits. Hence we have to generate the shift amount (the LZC) as a multiple of 3. Usually the LZDs generate the LZC as a binary string. Hence previously published LZDs are not suitable for this design and have to be modified.

A new LZD is proposed here, the main advantage of this LZD is that its output is in base-3 format, since the weights of the base-3 digits (3,9,27,81,... etc.) are multiples of 3 then there will be no problem while shifting the redundant result. The only problem is with the least significant base-3 digit, which is of weight 1. But this problem is solved in the rounding block as will be explained in Section 4.8.1.

3.5.2.3 Base-3 Leading Zero Detector

First the 2P +1 (107) bits, are divided into 36 groups of 3 bits each. The leading zeros of each group is calculated using a small 3-bit LZD cell.

Basic Base-3 LZD cell: The Basic LZD cell generates the LZC of its 3-bit input as well as a *valid* signal that is set if the input is not all zeros indicating that the string of zeros has been terminated inside these bits and the LZC of the next blocks (the ones to the right) is not needed to determine the LZC of the input string. The truth table of the basic cell is shown in Table 3.10,where v is the *valid* signal, and X2,X1,X0 are the input three bits. A simple 2-gate delay logic is used to determine the 2-bit (one base-3 digit) LZC and the *valid* signal for each group of three bits as shown in Equation 3.18.

$$\begin{aligned} LZC[0] &= X1\,\overline{X2} \\ LZC[1] &= \overline{X1}\,\overline{X2} \\ valid &= X0+X1+X2 \end{aligned} \tag{3.18}$$

Logarithmic Tree: The Next step is to take the results of the Basic 3-bit LZC cells and count the LZC in the input string. This is done by grouping the results of

X2	X1	X0	LZC	v
0	0	0	xx	0
0	0	1	10	1
0	1	0	01	1
0	1	1	01	1
1	0	0	00	1
1	0	1	00	1
1	1	0	00	1
1	1	1	00	1

Table 3.10: Truth Table for the Base-3 LZD basic Cell

each three cells together and find the LZC of this 9-bit string, which can be represented in two base-3 digits, and generate these digits as well as a *valid* signal if any of the 9 bits is non-zeros. The design equations for this tree are shown in Equations 3.19,3.20, and 3.21. This step is repeated again using the same equations to find the LZC in each consecutive 27-bit strings.

$$LZCi = \begin{cases} LZC2 & if\ (v2 = 1) \\ LZC1 & if\ (v2 = 0)\ and\ (v1 = 1) \\ LZC0 & f\ (v2 = 0)\ and\ (v1 = 0) \end{cases} \tag{3.19}$$

$$LZC = \{\overline{v1 + v2}, v1\ \overline{v2}, LZCi\} \tag{3.20}$$

$$valid = v0 + v1 + v2 \tag{3.21}$$

Instead of grouping three 27-bits LZC to form an 81-bits LZC, in the final step four 27-bits LZCs are combined to form a 108-bits LZC according to the following equations:

$$LZCi = \begin{cases} LZC3 & if\ (v3 = 1) \\ LZC2 & if\ (v3 = 0)\ and\ (v2 = 1) \\ LZC1 & f\ (v3 = 0)\ and\ (v2 = 0)\ and\ (v1 = 1) \\ LZC0 & f\ (v3 = 0)\ and\ (v2 = 0)\ and\ (v1 = 0) \end{cases} \tag{3.22}$$

$$LZCf = \{\overline{v1 + v2 + v3}, v1\ \overline{v2}\ \overline{v3}, v2\ \overline{v3}, LZCi\} \tag{3.23}$$

where LZC3, LZC2, LZC1, LZC0 are the leading zeros from each 27-bits block, LZCf is the final LZC count, and v3,v2,v1,v0 are the corresponding valid bits.

Correction Network: Since the least significant base-3 digit of the LZC is not a multiple of 3, then there is three different locations of rounding already, as will be explained later. So using just the inexact LZA will add more complications to the rounding module. Hence correction stage is needed.

As mentioned in [19], a (+1) correction is needed if and only if there was no carry to the right most location of string of zeros in x. The idea is to determine if there is a carry resulting from the two added vectors at the leading one location in x. In order to determine that, a 107 bits carry look-ahead network is used to get the carry from each bit location. Then we define the vector Co that contains the 107 carries from each bit location. and we define the vector $\hat{x}$, where $\hat{x}_i = x_i \, \overline{x_{i-1} + x_{i-2} + ...x_0}$. $\hat{x}$ contains all zeros except at the location of the leading 1 of x. Finally the correction is needed if the vector $corr = Co\,\hat{x}$ is all zeros. It has to be pointed out that the delay of the correction tree is less than the delay of the redundant adder, hence this carry look ahead network doesn't add any extra delay in the critical path.

3.5.2.4 Leading Ones Anticipator

In case of negative result, the leading ones are needed instead of the leading zeros. To find the leading ones count (LOC), the one's complement of the two input vectors is obtained and the same LZA explained above is used to find the LZC of the inverted inputs. Note that $LOC(A,B) = LZC(\overline{A},\overline{B})$. This can be easily proven if we can prove the property that if all of the inputs of a full adder is inverted, the output sum and carry will be inverted as well. If this is correct, then the leading ones in $A+B$ will be leading zeros in $\overline{A}+\overline{B}$. The proof of this property is shown below:

if (a+b+Cin = S+2Cout)
then
(1-a) + (1-b) + (1-Cin) = 3 - (a+b+Cin)

= 3 - (S + 2Cout) = (1-S) + 2(1-Cout).

Hence inverting the inputs of the full adder, will result in inverted outputs.

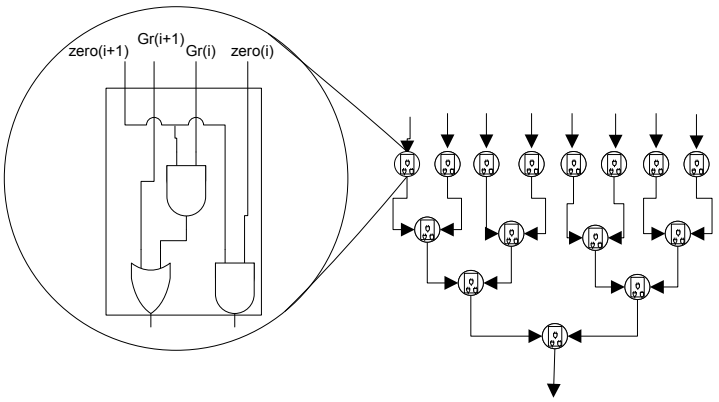

Figure 3.14: Intermediate Sign Detector For 16-Digit Decimal Operands

3.6 Intermediate Sign detection

3.6.1 Decimal Intermediate Sign Detection

We propose a simple sign detection tree that operates in parallel to the preliminary anticipation. We use the $zero_i$ and $Gr_i = g1_i \,|\, g2_i$ signals generated in the LZA to detect the intermediate sign in case of effective subtraction. Where $Gr_i = 1$ indicates that digit A_i is greater than digit B_i. This vector Gr and the $zero$ vector are used as inputs to the sign detection tree shown in Figure 3.14.

3.6.2 Binary Intermediate Sign Detection

In case of effective addition, the intermediate sign is always positive. However, in the case of effective subtraction, the intermediate sign depends on whether C is larger than M or not. In that case we are performing two's complement addition, hence the intermediate result is positive if there is a carry out from the addition and it's negative it there is no carry out. The carry out is detected in parallel with the LZA using a look-ahead carry tree.

Note that we have the addend in the one's complement format so far as the increment is not yet added. However, this increment is ready at this stage, and is taken into consideration as a carry-in to the look-ahead carry network. It has to be pointed out that the delay of this carry look ahead tree is smaller than the delay of the adder, and the intermediate sign is ready before the addition result, where it

is needed, hence it doesn't add any extra delay in the critical path. Also it's not a complete adder, so the area is not that huge either.

3.7 Final Alignment

3.7.1 Final Alignment Control

3.7.1.1 Case of Decimal

After the PLZC is calculated, it is used to determine the required final shift of the two operands. The shifting at the default data path will always be to the left. The shifting will take place after the addition as will be shown below, where cases 1 to 4 are the cases explained in Table 3.8.

$$Shift2 = \begin{cases} Min\{PLZC, ExpDiff\} & case(1) \\ Min\{PLZC, p+1\} & case(2) \\ PLZC & case(3,4) \end{cases} \tag{3.24}$$

Finally, the 3p+1 digits are sent to the redundant adder to be added and rounded, and the final result is obtained.

It is important to note that, the underflow case limits the value of final shift or shift the operands to the right instead. However, it is considered as an exceptional data path case.

3.7.1.2 Case of Binary

After obtaining the LZC using the LZA or the LOA explained above, the final normalization shift amount is obtained and the result is shifted after the addition. It has to be said here that the shifting is always to the left in all cases, including in the case of subnormalM and ZeroC. As in the case of submormalM and ZeroC the location of the MSB is chosen to the far left (4P+2) hence the exponent already is biased by 54, so we can shift the result to the left up to 54 bits without reaching the minimum exponent. Note that we can't need more left shift than 54 as in that case the result should be an inexact zero.

The only limit on the shifting amount is the underflow. Underflow will occur if the biased exponent of the result reaches 0 ($ExpR = 0$), and since $ExpR = ExpRi - normshifting$, Hence the maximum normalization shifting amount is reached when $0 < ExpRi - normshifting$, i.e $normshifting = ExpRi - 1$. Hence the normalization shifting amount is obtained as shown in Equation :

$$normshifting = Min\{LZC\,,\, ExpRi - 1\}$$

Chapter 4

Redundant Addition, Normalization, and Rounding

The next step is to add the two vectors resulting from the selection stage together to get the un-normalized sum. To be able to use the same adder for both binary and decimal, both binary and decimal are first converted to the same redundant format. The sum is then obtained by adding these two vectors.

The redundant system proposed in [16] is used. This redundant system uses the digit-set [-6, 6] encoded in the two's complement format instead of the conventional representation ([0,9] in the case of decimal, and [0,7] in the case of octal) which does not allow for a carry-free addition/subtraction.

4.1 Conversion from Binary/Decimal to Redundant

In case of decimal, each conventional decimal digit is converted to the redundant digit set [-6,6]. In case of binary, each three bits are grouped together to form an octal digit [0,7], then this octal digit is converted to the redundant format [-6,6].

The conversion procedure is shown below:

if (input$i$ > 5)

Input	output	OTD
0000	0000(0)	0
0001	0001(1)	0
0010	0010(2)	0
0011	0011(3)	0
0100	0100(4)	0
0101	0101(5)	0
0110	1100(-4)	1
0111	1101(-3)	1
1000	1110(-2)	1
1001	1111(-1)	1

Table 4.1: Conversion from decimal to redundant

{

INT_i = $input_i$ - radix;

OTD_i = 1;

}

else

{

INT_i = $input_i$;

OTD_i = 0;

}

$output_i$ = INT_i+ ITD_i;

where $ITD_i = OTD_{i-1}$, $input_i$ is the input digit at location i, INT_i is the intermediate sum, radix = 10 for decimal, and 8 for octal, OTD, and ITD are the output and input transfer digits respectively.

Input	output	OTD
0000	0000(0)	0
0001	0001(1)	0
0010	0010(2)	0
0011	0011(3)	0
0100	0100(4)	0
0101	0101(5)	0
0110	1110(-2)	1
0111	1111(-1)	1

Table 4.2: Conversion from octal to redundant

4.2 Redundant Addition

After the two vectors are converted to the redundant representation, they are ready to be added using the redundant adder. The redundant adder used is the one presented in [16], but with small modification to enhance the delay of the binary/decimal adder cell.

The adder takes two inputs in the redundant format shown above, adds them and puts the result in the same redundant format. First the two digits are added together using a simple 4-bit carry look-ahead adder to get the intermediate sum. Instead of waiting until the intermediate sum is ready to calculate the OTD from the intermediate sum, the OTD is accurately estimated from the two inputs using a 8-input combinational circuit in parallel with the 4-bit adder. After the OTD is ready, which should be ready a little before the addition result, a correction digit is formed, this correction digit contains the radix addition or subtraction, and the value of the ITD as shown in the algorithm below. Finally the correction digit is added to the intermediate sum to get the final redundant result. The algorithm is shown below:

if ($sum_i > 5$)

{

$INT_i = sum_i$ - radix;

$OTD_i = 1$;

}

if ($sum_i < -5$)

{

$INT_i = sum_i + radix;$

$OTD_i = -1;$

}

else

{

$INT_i = input_i$;

$OTD_i = 0;$

}

$final\_sum_i = INT_i + ITD_i;$

where $ITD_i = OTD_{i-1}$, $input_i$ is the input digit at location i, INT_i the intermediate sum, radix = 10 for decimal, and 8 for octal, OTD, and ITD are the output and input transfer digits respectively.

The OTD is represented in two bits: otdn, and otdp. If the transfer digit is negative, otdn is raised. And if it's positive, otdp is raised. Hence the numerical value of the transfer digit is OTD = otdp - otdn. Also the numerical value of the input transfer digit is : ITD = itdp - itdn.

4.2.1 Correction Digit Generation

The different values of correction digits are studied and limited to four possibilities I1,I2,I3, and I4.

In order to calculate the values of these vectors, we first define the vector O, where O = $\{itdn \,.\, \overline{itdp}, itdn \,.\, \overline{itdp}, itdn \,.\, \overline{itdp}, itdn \oplus itdp\}$. The four vectors are defined as follows:

I1 = O.

I2 = $\{0,1,itdp \,.\, \overline{itdn}, itdn \oplus itdp\}$.

I3 = $\{\overline{O[3]}$, O[2:0]$\}$.

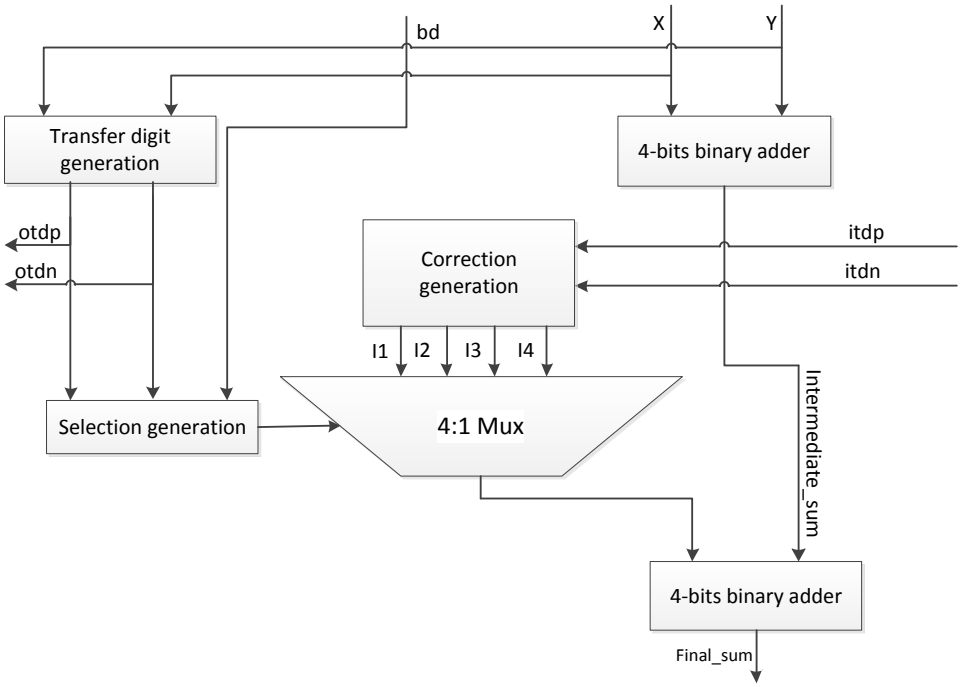

Figure 4.1: Redundant Adder Cell

I4 = {1,0,$itdp$. $\overline{itdn}$,$itdn \oplus itdp$}.

The correction digit is chosen as one of the above four possibilities according to the two-bit signal selection, where selection = {$\overline{otdp}$. $otdn + bd$. $\overline{otdn}$. $otdp$, $\overline{bd}$. ($otdn \oplus otdp$)}. The selection is done using a four to one Mux as shown in Figure 4.1.

4.2.2 Adder Carry in

The transfer to the least significant digit of the adder is assumed to be zero, as there is no digits to the right of that location that can produce an output transfer digit. This input transfer digit can be used to inject a carry-in to the adder without any overhead in the delay or the area.

In case 2 of binary selection and case 4 in decimal selection, the least significant bit of the selected width lies in the middle of the multiplication result. It's assumed that there is no carry to the right location of the selected width as the least significant part of multiplication result, the part which lies outside the selected width, should be added to a string of zeros because the addend is shifted to the left of the multiplication result. Unfortunately, a carry to the most significant half of the multiplication result may occur (while adding the three resulting

vectors of the multiplication result to each other) due to the used sign extension in the multiplier tree. This carry is calculated in parallel with the second stage of the CSA and is injected as a carry-in to the redundant adder.

4.3 Result Complementation

After obtaining the redundant sum from the adder, it may need to be complemented if the intermediate sign is negative. The complementation is easily done without any carry propagation, since each digit can be positive or negative, the 2's complement of each digit is obtained separately and in parallel with each other.

4.4 Size of the redundant vectors

To determine the width of the redundant vectors, we have to know the width of the inputs to the adder.

As mentioned above, the width of the decimal vectors after selection stage is 3p+1, which is 49 digits, after conversion to redundant, we will have 50 digits, as we may have OTD from the final digit.

In case of binary, the width is 107 bits, which is cascaded with one zero to the left to form 36 octal digits, which will be 37 redundant digits.

So to work for binary and decimal, the width of the redundant operands are selected to be 50, and 13 zero digits are concatenated to the left of binary operands.

4.5 Normalization Shifting

As mentioned above, the result from the adder may contain some leading zeros that has to be shifted out. The result has to be normalized (shifted to the left) by the leading zeros count calculated from the LZA, unless the preferred exponent is reached in decimal, or the exponent becomes subnormal in binary. The calculation of the shift amount is explained in Section 3.7.1. A simple barrel left shifter is used to perform the coarse normalization of the result.

4.5.1 Binary Shifting

As mentioned above, each four bits (one digit) of the redundant sum contains the information of three bits of the binary output. This will cause a problem while

shifting if the shift amount is in binary format. For example, if the shift amount is (1000), so we have to shift the result by eight binary bits, to know how much digits should be shifted we have to divide the number by 3, hence the result has to be shifted to the left by two digits, and two bits. Since the division by 3 is a complicated operation, we have to find another way to solve this problem. Instead of changing the way we perform the shift, we can change the format of the shift amount, in order to represent it as a multiple of 3. The chosen format is to represent it as a base-3 number. Since the shifting amount is limited to the width of the operand, we need only five base-3 digits to represent the shifting amount, with the most significant digit (the one with the weight 81) can't exceed 1, as the maximum shifting amount can't be more than 107 bits. Now we have five base-3 digits with weights of (81,27,9,3,1) and any multiple of them, is divisible by 3, except the least significant digit. Hence the shifting process is divided into two stages: coarse and fine shifting as explained below.

Coarse shifting Coarse shifting is done using the most significant four digits of the shifting amount. It takes place right after the complementation stage after the redundant addition. It's done while the result is still in the redundant format. The problem of the shifting amount not being divisible by 3 discussed above will not happen in this case, because the shifting amount is represented as multiples of 3 (in base-3 format). Table 4.3 shows how the shifting amount is calculated.

Fine Shifting Fine shifting is done using the least significant digit of the base-3 shifting amount. It has to be done after converting the result to binary, i.e after the rounding. Which implies that rounding has to be done while the result could be shifted for one or two bits to the left, or not shifted at all. This complicates the rounding stage as we have three different locations of rounding. However, this problem doesn't add extra delay in the critical path and rounding is done in parallel with the conversion to binary with no extra delay as will be shown in Section 4.8.1.

4.5.2 Decimal Shifting

In case of decimal, we don't have the problem we faced in binary because each decimal digit is encoded in one redundant digit. However, the decimal LZA doesn't always give the correct shifting amount, the actual shifting amount may be one larger than the PLZC obtained from the decimal LZA. The correction of

S4	S3	S2	S1	shifting amount (binary bits)	shifting amount (redundant digits)
00	00	00	00	0	0
00	00	00	01	3	1
00	00	00	10	6	2
00	00	01	00	9	3
00	00	01	01	12	4
00	00	01	10	15	5
00	00	10	00	18	6
00	00	10	01	21	7
00	00	10	10	24	8
00	01	00	00	27	9
00	01	00	01	30	10
00	01	00	10	33	11
00	01	01	00	36	12
00	01	01	01	39	13
00	01	01	10	42	14
00	01	10	00	45	15
00	01	10	01	48	16
00	01	10	10	51	17
00	10	00	00	54	18
00	10	00	01	57	19
00	10	00	10	60	20
00	10	01	00	63	21
00	10	01	01	66	22
00	10	01	10	69	23
00	10	10	00	72	24
00	10	10	01	75	25
00	10	10	10	78	26
01	00	00	00	81	27
01	00	00	01	84	28
01	00	00	10	87	29
01	00	01	00	90	30
01	00	01	01	93	31
01	00	01	10	96	32
01	00	10	00	99	33
01	00	10	01	102	34
01	00	10	10	105	35
01	01	00	00	108	36

Table 4.3: Calculating the redundant shifting amount from base-3 shifting amount

this error in the LZA shift amount, can be done by checking the MSD after the conversion to decimal, if it's zero then it has to be shifted one digit more to the left, unless the preferred exponent is reached. And since we also don't exactly know the shifting amount before rounding, there is two different locations of rounding. The same solution applied to the binary is applied in decimal, and rounding is done in parallel with the conversion to decimal and doesn't add extra delay in the critical path.

4.6 Sticky Generation

There are three parts that can contribute to the sticky:

1) The left-out bits in the selection stage (selection_sticky): The bits to the right of the selected bits in the selection stage explained in Section 3.4.

2) In case decimal, if underflow occurs, and right shifting is needed (dec_underflow_sticky): The bits that has been shifted out of to the right of the result.

3) The redundant sticky: The redundant digits that doesn't contribute to the final result, i.e. the digits to the right of the least significant redundant digit used in the rounding explained in Section 4.6.1.

The first two contributors are unsigned, so we need to know only if they exist or not. On the other hand, since the redundant sticky is signed, and may be negative, it can generate a borrow to the higher significant digits which may affect the rounding decision. So in case of redundant sticky we need to know if there is a sticky or not, and if the sticky exist what its sign is.

Hence, the sticky is represented in two bits: sticky, and sticky_effective_sign. The equations of these two bits are shown below:

$$sticky = redundant\_sticky + selection\_sticky + dec\_underflow\_sticky$$

$$\begin{aligned} sticky\_effective\_sign &= redundant\_sticky\_sign \\ &+ \overline{redundant\_sticky}\,.\,IntSign\,.\,selection\_sticky \end{aligned}$$

Note that the sticky_effective_sign is not the actual sticky sign, it's the sign relative to the IntSign. In other words, if the sticky_effective_sign is zero this

means that this sticky (if exists) will be added to the absolute value of the result, and if sticky_effective_sign was one, it means that the sticky (it has to be existent in that case) will be subtracted from the absolute value of the result. And since the dec_underflow_sticky has always a sign that is similar to the result (as it is obtained after the complementation of the intermediate result), its contribution to the sticky_effective_sign will always be positive. Now, to calculate these two signals we have to define the sticky first, then find if it exists and detect its sign using a sticky sign detector as will be shown below.

4.6.1 Separating the Sticky

As mentioned above, the result is represented in 50 redundant digits.

In case of decimal, the redundant result contains one extra digit due to a possible OTD from the MSD of the decimal input, then 16 digits which are the significand digits, then a guard digit as we may have one extra left shift, and finally the round digit. So the most significant 19 redundant digits may contribute to the result and the rounding decision. the remaining 31 digits are the sticky digits that we need to find out if they are all zeros or not, and to detect their effective sign.

In binary, there is 13 zero digits, then the extra redundant digit due to the OTD from the last octal digit, followed by 18 digits that carries the 53 significant bits of the results. then we need two guard bits, as we might have to shift the result by 1 or 2 bits according to the fine_shift amount, and a round bit, hence we need an extra digit (three bits). which makes the least significant 17 digits the sticky digits.

4.6.2 Sticky Sign Detector

For the sticky sign detector to work for both binary and decimal, it is designed to accept 31 digits of sticky. The output of this block is two bits: $sticky$ which equals 1 if there is any non-zero redundant digit in the sticky, and $sticky\_effective\_sign$ which is raised if the sticky exists, and its sign is negative.

In order to generate these two bits, two signals (sign, and zero) are generated from each redundant digit. *sign* signal is 1 if this redundant digit is negative, and *zero* signal is 1 if the digit is zero. These two signals are calculated according to the following equations:

$$
\begin{aligned}
sign_i &= X_i[3] \\
zero_i &= \overline{X_i[3] + X_i[2] + X_i[1] + X_i[0]}
\end{aligned}
$$

where $X_i[j]$ is the jth bit of the ithredundant digit.

It may be clear now that the problem of calculating $sticky\_effective\_sign$ is exactly the problem of calculating the carry out of a 31-bit binary adder. A carry look-ahead tree is used to determine the final sign, using the $zero_i$ signal as a propagate signal, and $sign_i$ as the generate signal. Also the $sticky$ signal is calculated from the $zero_i$ signals as follows:

$$sticky = \overline{zero_0 \,.\, zero_1 \ldots\ldots\ldots zero_{30}}$$

It has to be mentioned here that the part of calculating the sign of the sticky is done in parallel with the first part of the rounder where the sticky effective sign is not needed. The two paths has almost the same delay, so the choice of the carry look ahead tree doesn't add any delay overhead to the critical path.

4.7 Conversion Back to Binary/Decimal

Conversion to Binary/Decimal is the exact opposite of conversion to redundant. But it includes carry propagation.

4.7.1 Previous technique

The previously proposed techniques was to separate the positive digits in a vector, and the negative digits in a separate vectors. Then add these two vectors using a carry propagate adder. The separation takes a MUX delay, and the carry propagate adder needs correction if its result is negative.

In this work we propose a new simpler conversion technique as shown in the next section.

4.7.1.1 Our proposed technique

To derive the new technique, we can start the way we started in the conversion to redundant process. If the redundant digit is positive or zero, then it's representable in the non-redundant format, hence it doesn't need any correction. Otherwise,

if it's negative, it needs to be corrected by adding the radix (10 if we convert to decimal, and 8 of we convert to octal (binary)) and a borrow should be "generated" to the next digit. This borrow will only affect the higher significant digit, unless this higher significant digit is zero. In that case the borrow is "propagated" to the next digit. We can easily see the similarity between our case of calculating the borrow-in of each redundant digit, and the case of calculating the carry-in in case of performing a simple binary addition. Hence a look-ahead carry tree is used to quickly calculate the borrow-in of each digit. The propagate signal of each digit is raised if this digit is zero with a simple four-input NOR gate. and the generate signal is raised if its sign bit = 1 with no gate delays.

4.7.2 Converting back to Decimal

In case of decimal, and as mentioned in Section 4.6.1 the most significant redundant digit (digit 50) is discarded, the next 16 redundant digits (49-34) contain the result significand and the digit (33) is the rounding digit if there is no error in the anticipation of leading zeros. In case there was an error in the anticipation, one more digit (digit 49) will be discarded, and the significand will be in the 16 digits following the two most significant digits (digits 48-33) and the rounding digit is digit (32).

Since we don't know yet whether we have an error in the anticipation or not, digits 34,33, and 32 are sent to the decimal rounder, in order to perform rounding in the two possible locations simultaneously.

The remaining 15 significand digits (digits 49-35) are converted back to decimal using the redundant to decimal/binary converter explained above. Also we don't know the value of the borrow to the right to these 15 digits, which depends on the rounding decision as will be shown in Section 4.8.2, so the conversion process is done using the concept of carry select; these 15 digits are converted once assuming input borrow = 0, and in parallel these 15 digits are converted assuming input borrow = 1. Once the rounding decision is made, which is done in parallel with the conversion, we can select the correct result using the correct borrow from the rounder.

4.7.3 Converting back to Binary

In case of binary, and as mentioned in Section 4.6.1 The most significant 13 redundant digits (50-38) are all zeros, the next redundant digit (digit 37) is discarded,

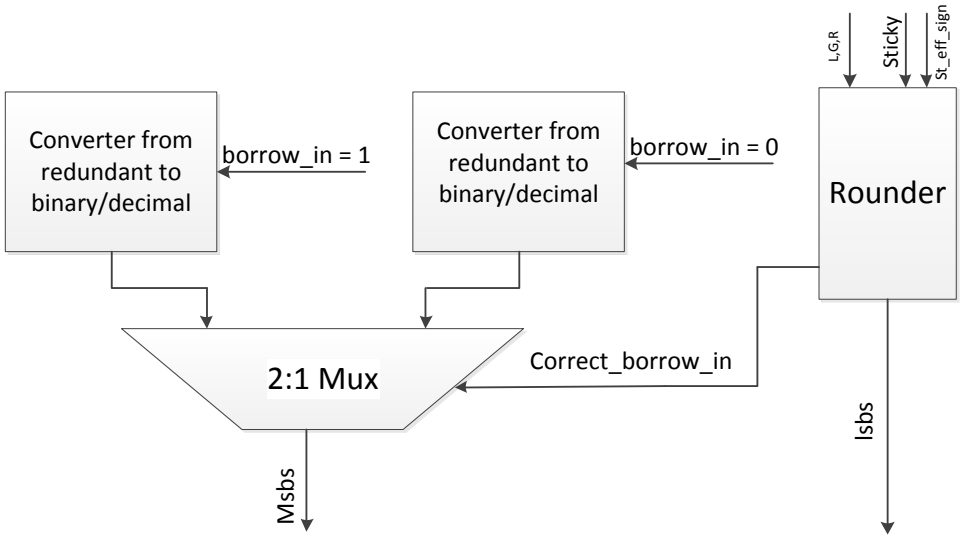

Figure 4.2: Rounding and Conversion Block Diagram

and the next 18 digits (36-19) contains the result significand and the digit (18) contains the guard and round bits for different values of the fine shift.

Since we can't perform the fine shift yet, digits 19, and 18 are sent to the binary rounder, in order to perform rounding in the three possible bit locations simultaneously.

The remaining 17 redundant digits that contain all of the binary significand except the least significant three, four, or five bits depending on the fine shift amount (digits 36-20) are converted back to binary using the redundant to decimal/binary converter explained above. Also we don't know the value of the borrow to the right to these 17 digits, which depends on the rounding decision as will be shown in Section 4.8.1, so the conversion process is also done using the concept of carry select; these 17 digits are converted once assuming input borrow = 0, and in parallel these digits are converted assuming input borrow = 1. Once the rounding decision is made, which is done in parallel with the conversion, we can select the correct result using the correct borrow from the rounder.

4.8 Rounding

Rounding is very critical for both accuracy, and correctness of the design, and also for the total delay. It can be done after converting the number to the non-redundant format, but in that case we will add a logarithmic carry propagation delay in the critical path. Or it can be done in parallel with the conversion, this way we remove the rounding delay completely from the critical path. In our design, rounding is done in parallel with the conversion for both binary and decimal.

Rounding while in the redundant format is one of the big challenges of using a redundant format. In the next three sections, we will present how the rounding is done with no overhead in the delay, although we don't actually know the exact location of the rounding yet.

4.8.1 Rounding in binary

As mentioned before, in case of binary, two redundant digits are sent to the binary rounder. First these two digits are converted to binary using a redundant to binary converter to get six bits (lsbs[5:0]), as well as an initial borrow signal $fbout\_local$. This is not the final borrow out signal that is used to choose between the results of the two redundant to binary converters mentioned above, as the rounding decision may generate a carry out, which cancels the $fbout\_local$ signal. Also the rounding increment may be negative as will be shown below, and generate a borrow out signal even if $fbout\_local$ was zero.

In order to correctly round the result, we have to define these variables and find their values:

1) LSB: is the least significant bit of the result, it's needed in the rounding to nearest, ties to even.

2) RB: it's the bit in the right to the LSB, it determines whether the value of the bits right to the LSB is smaller that half LSB, or not.

3) Sticky: if there is any non-zero bit to the right of the rounding bit, this should be raised indicating that this is not a TIE case.

4) sticky_effective_sign: the effective sign of the sticky to the right of the rounding bit.

5) Sign: the sign of the number is needed in the rounding towards positive, or negative infinity.

6) $rounding\_mode$: three bits that determines the rounding direction. Our design supports the five rounding directions specified in the IEEE-754 standard,

rounding_mode	rounding direction
000	Round to nearest, Ties to even
001	Round away from zero
010	Round towards positive infinity
011	Round towards negative infinity
100	Round towards zero
101	Round to nearest, Ties away from zero
110	Round to nearest, Ties towards zero

Table 4.4: supported rounding directions

fine shift	00	01	10
LSB	lsbs[3]	lsbs[2]	lsbs[1]
RB	lsbs[2]	lsbs[1]	lsbs[0]
sticky	stin + lsbs[1] + lsbs[0]	stin + lsbs[0]	stin
sticky_effective_sign	$\overline{lsbs[1]+lsbs[0]}$. st_sign	$\overline{lsbs[0]}$. st_sign	st_sign
Sign	Sign	Sign	Sign

Table 4.5: Values of the parameters needed for rounding for each value of fine shift

as well as two extra rounding directions. The supported rounding directions are shown in Table 4.4

Now we have to define these bits for each value of the fine shift. This is shown in Table 4.5. Where stin is the input sticky which is the sticky to the right of the converted two redundant digit, i.e. the value of the sticky calculated by the sticky sign detector. And st_sign is the sticky effective sign also to the right of these two digits, i.e. the one calculated by the sticky sign detector.

After obtaining the values of LSB, RB, sticky, sticky_effective_sign, and Sign they are sent to the rounding cell to take the rounding decision and generate the incp, and incn signals, which are the positive and negative increments respectively.

In order to save time from the critical path, we don't wait for the values of incp and incn to be calculated, we start preparing the result in each case. Two vectors are prepared in parallel with the rounding cell, lsbs_p and lsbs_n. Where lsbs_p = lsbs[5:i] + 1, and lsbs_n = lsbs[5:i] -1. where i=3-final_shift. Carry look-ahead adder is used to perform these two additions in parallel with the rounding cell. Once the rounding cell has made the decision the correct LSBs can be chosen according to the following equation:

$$lsbs[5:i] = \begin{cases} lsbs\_p & if\ incp = 1 \\ lsbs\_n & if\ incn = 1 \\ lsbs[5:i] & if\ incp = incn = 0 \end{cases}$$

The final borrow signal is calculated as:
$fbout = fbout\_local\,.\,cout + \overline{lsbs[i-1] + lsbs[i-2]...lsbs[0]}\,.\,incn$
Where cout is the carry out from the carry look-ahead adder.

4.8.2 Rounding in decimal

As mentioned before, in case of decimal, three redundant digits are sent to the decimal rounder. First these three digits are converted to decimal using a redundant to decimal converter to get the corresponding three decimal digits: LSD,GD, and RD, as well as an initial borrow signal $fbout\_local$. As in the case of binary rounding, this is not the final borrow out signal that is used to choose between the result of the two redundant to decimal converters mentioned earlier, as the rounding decision may generate a carry out, which cancels the $fbout\_local$ signal. Also the rounding increment may be negative as will be shown below, and generate a borrow out signal even if $fbout\_local$ was zero.

In order to use the same rounding cell used in binary, we have to correctly map the decimal parameters to the inputs of the rounding cell. In decimal we have only two possible locations of rounding, since the LZA can give the leading zeros count with an error of one. The signals sent to the rounding cell for both cases of shift are shown in Table 4.6. The extra shift can be determined by checking the MSD of the resulting number, and if this MSD = 0 we have to shift the result one digit to the left.

In binary, the RB is raised if the bits to the right of the LSB are half, or more. The sticky bit is raised if the bits to the right of the LSB are either more than half, or nonzero and less than a half. In order to correctly fit decimal to the binary rounding cell, three signals are generated for each of the GD, and the RD. These signals are $notequaltofive$, $greaterthanorequaltofive$, and $nonzero$, indicating if the RD, or the GD are not equal to exactly five (half), greater than or equal to five (more than a half), and if they are not zero. The use of these signals to generate the rounding cell parameters are shown in table 4.6.

The rest of the rounder is the same as the binary rounder. In parallel with the rounding cell, two BCD adders are generating the correct digits in case of positive and negative increments respectively. Finally the rounding decision chooses the correct result and cout signals and the final borrow is obtained the same way as the binary case.

	MSD >0	MSD = 0
LSB	LSD[0]	GD[0]
RB	GD_greaterthatorequattofive	RD_greaterthatorequattofive
sticky	(GD_notequaltofive . GD_nonzero) + RD_nonzero + stin	(RD_notequaltofive . RD_nonzero) + stin
sticky_ effective_sign	$(\overline{GD\_notequaltofive}$ +$\overline{GD\_iszero})$. $\overline{RD\_iszero}$. st_sign	$(\overline{RD\_notequaltofive}$+$\overline{RD\_iszero})$. st_sign
Sign	Sign	Sign

Table 4.6: Values of the parameters needed for rounding for both cases of shift

LSB	RB	sticky	sticky_effective_sign	incp	incn
0	0	0	0	0	0
0	0	0	1	x	x
0	0	1	0	0	0
0	0	1	1	0	0
0	1	0	0	0	0
0	1	0	1	x	x
0	1	1	0	1	0
0	1	1	1	0	0
1	0	0	0	0	0
1	0	0	1	x	x
1	0	1	0	0	0
1	0	1	1	0	0
1	1	0	0	1	0
1	1	0	1	x	x
1	1	1	0	1	0
1	1	1	1	0	0

Table 4.7: Rounding to Nearest, Ties to Even

4.8.3 Rounding Cell

The rounding cell is the thinking mind of the rounding logic. It calculates the positive and negative increments. The inputs to the rounding cell are LSB, RB, sticky, sticky_effective_sign, and sign.

The truth table of the positive increment (incp) and the negative increment (incn) in different rounding directions are shown in Tables 4.7, 4.8, 4.9, 4.10, 4.11, 4.12, and 4.13.

RB	sticky	sticky_effective_sign	incp	incn
0	0	0	0	0
0	0	1	x	x
0	1	0	1	0
0	1	1	0	0
1	0	0	1	0
1	0	1	x	x
1	1	0	1	0
1	1	1	1	0

Table 4.8: Rounding Away from Zero

Sign	RB	sticky	sticky_effective_sign	incp	incn
0	0	0	0	0	0
0	0	0	1	x	x
0	0	1	0	1	0
0	0	1	1	0	0
0	1	0	0	1	0
0	1	0	1	x	x
0	1	1	0	1	0
0	1	1	1	1	0
1	0	0	0	0	0
1	0	0	1	x	x
1	0	1	0	0	0
1	0	1	1	0	1
1	1	0	0	0	0
1	1	0	1	x	x
1	1	1	0	0	0
1	1	1	1	0	0

Table 4.9: Rounding towards positive infinity

Sign	RB	sticky	sticky_effective_sign	incp	incn
0	0	0	0	0	0
0	0	0	1	x	x
0	0	1	0	0	0
0	0	1	1	0	1
0	1	0	0	0	0
0	1	0	1	x	x
0	1	1	0	0	0
0	1	1	1	0	0
1	0	0	0	0	0
1	0	0	1	x	x
1	0	1	0	1	0
1	0	1	1	0	0
1	1	0	0	1	0
1	1	0	1	x	x
1	1	1	0	1	0
1	1	1	1	1	0

Table 4.10: Rounding towards negative infinity

RB	sticky	sticky_effective_sign	incp	incn
0	0	0	0	0
0	0	1	x	x
0	1	0	0	0
0	1	1	0	1
1	0	0	0	0
1	0	1	x	x
1	1	0	0	0
1	1	1	0	0

Table 4.11: Rounding towards zero

RB	sticky	sticky_effective_sign	incp	incn
0	0	0	0	0
0	0	1	x	x
0	1	0	0	0
0	1	1	0	0
1	0	0	1	0
1	0	1	x	x
1	1	0	1	0
1	1	1	0	0

Table 4.12: Rounding to Nearest, Ties away from zero

RB	sticky	sticky_effective_sign	incp	incn
0	0	0	0	0
0	0	1	x	x
0	1	0	0	0
0	1	1	0	0
1	0	0	0	0
1	0	1	x	x
1	1	0	1	0
1	1	1	0	0

Table 4.13: Rounding to Nearest, Ties towards zero

Chapter 5

Sign, Flags, and Exceptional Data paths

5.1 Final Sign Calculation

The sign is calculated as follows:

$$SignR1 = \overline{SignIR}\,.\,signM \,|\, signIR\,.\,signCeff \tag{5.1}$$

where $SignIR$: is the intermediate result sign.
and $SignCeff = eop \oplus signC$ is the effective sign of the addend.
and $SignM = signA \oplus signB$ is the multiplication result sign.

The intermediate result equals to $M \pm C$; therefore, if the intermediate result is negative this means that $C > M$ and the final result will follow the addend effective sign ($eop \oplus signC$). If $M > C$ then the intermediate sign will be positive and the final result will follow the multiplication result sign. In case of having $M = C$ with the intermediate result being zero, then the final sign will be zero in all cases unless the rounding mode is rounding to negative infinity. This is defined by equation 5.2. Equation 5.3 gives the final sign of the result.

$$signR2 = (RndMode == RM) \tag{5.2}$$

$$signR = \begin{cases} signR1 & ExactIntResult \neq 0 \\ signR2 & ExactIntResult = 0 \end{cases} \tag{5.3}$$

5.2 Exponent Calculating

5.2.1 Decimal Exponent

The exponent is calculated as follows:

$$expt1 = expM - Shift2 \tag{5.4}$$

$$expt2 = expt1 - dec + inc \tag{5.5}$$

where dec=1 if the preferred exponent is not reached and the most significant digit of the rounded result is zero and inc=1 if a carry out at the final digit is detected due to rounding in case of effective addition.

$$expR = expt2 + bias \tag{5.6}$$

where

$$bias = \begin{cases} p+1 & case(1,2) \\ 2p+1 & case(3) \\ 3p+1 & case(4) \end{cases} \tag{5.7}$$

The bias value compensates the initial shift due to the selection stage and case 1 to 4 are shown in Table 3.8.

5.2.2 Binary Exponent

The final binary exponent is calculated as follows

$$expt1 = expM - normshifting \tag{5.8}$$

$$expR = expt2 + bias \tag{5.9}$$

where

$$bias = \begin{cases} 0 & case(1) \\ 54 & case(2) \\ 54 & SubnormalM\&ZeroC \end{cases} \tag{5.10}$$

and case 1 to 4 are shown in Section 3.4.2.

5.3 Flag Generation

5.3.1 Inexact Flag

The inexact flag is raised if the result is rounded. It is detected from the sticky, guard and round digits.

5.3.2 Invalid Flag

The invalid flag is generated in either of these cases:

- One of the operands is sNaN.

- In case of $FMA(0, \pm\infty, c)$ or $FMA(\pm\infty, 0, c)$; where c is any DFP number including special numbers (NaNs, infinities).

The standard in this case states that it is optional to raise the invalid flag if the third operand in qNaN. In our implementation we activate the invalid flag even if the third operand in qNaN.

- In case of $FMA(|a|, +\infty, -\infty)$ or $FMA(|a|, -\infty, +\infty)$; where a is a DFP number that is not a NaN.

5.3.3 Overflow Flag

The overflow is detected after rounding. It is signaled if the final exponent exceeds the maximum exponent in the standard. If an overflow is detected, the result is rounded either to infinity or to the largest representable number according to the rounding mode and the final sign.

5.3.4 Underflow Flag

If the intermediate result is a non-zero floating point number with magnitude less than the magnitude of that format's smallest normal number, an underflow is detected. However, the underflow flag is not raised unless the result is inexact.

In case of decimal underflow a right shifter is needed to bring the exponent into range even if some precision digits are lost. Therefore right shifters used in that case in parallel with the normalization shifter right after the complementation stage. The left shifter is used in the default data path cases as explained before. However, if underflow is detected the result may be shifted to the right; hence, there is a right shifter.

5.4 Exceptional Decimal Data path

5.4.1 Zero Addend

This case is detected from the signal iszeroC that results from decoding of the addend. If the addend is zero, the default data path will not produce a correct result in all cases. If the addend is shifted at the preparation stage by a large amount, the multiplication result may be totally or partially considered in the sticky only; which is not the correct answer. Hence, in this case, the addend is not shifted. The selection signals only indicate either preferred exponent equals to ExpC or to ExpM.

The final shift amount is determined separately. Since the multiplication result is added to zero. It is only required to either shift the multiplication result to the left to reach or to approach the preferred exponent (ExpA plus ExpB) or to the right to overcome underflow. The different control blocks are reconfigured to handle this exceptional case. The sign and exponent calculation are also modified to produce a correct result in this case.

5.4.2 Zero Multiplication Result

This case is detected from the signal iszeroM = (iszeroA+iszeroB). In this case, the result should equal to the addend unless the preferred exponent is the exponent of the multiplication result. If so, the addend has to be shifted to the left to reach or approach the preferred exponent. There is no underflow in this case. The different control blocks are configured to handle this exceptional case. Also, the sign and exponent calculation are modified to produce a correct result in this case.

5.5 Exceptional Binary Data path

In binary all of the special cases are handled in the default data path. For example, if the addend is zero, the selection stage will always choose the multiplication result as the operating width and produce the correct result.

Also if the multiplication result is zero, the addend has to be shifted to the left as the biased exponent of the multiplication result in that case is zero and the addend exponent can't be less than that. Hence the selection stage selects the addend in the operating width and also produce the correct result.

The case of subnormalM and ZeroC is also explained in section 3.4.

5.6 Special Values Handling

Special values (NANs and Infinities) are handled in both binary and decimal according to the standard as explained in Chapter 1.

Chapter 6

Verification and Synthesis Results

6.1 Verification

Due to the large input space of the FMA ,192 bits of operands, 3 bits of rounding direction, a bit for binary/decimal selection, and a three bits to select the operation if it was multiplication of addition or FMA, it's impossible, using today's computational power, to try all of the possibilites of these inputs to verify full functionality in a reasonable time. However, it may be sufficient to try a large number of test vectors that was designed to test different corner cases of the design as well as the normal operation.

6.1.1 Decimal Verification

Decimal unit was verified as a multiplier, adder, and as an FMA using a large number of test vectors specially designed to check the unit in all corners. The vectors used to test the decimal functionality were presented by S. Ahmed et al in [20] and are available for download from [21]. More than 1.1 million test vectors were used to verify full decimal functionality. Our unit passed all of these test vectors which is an excellent indication of full decimal functionality. The number of test vectors used in each case is shown in Table 6.1.

Decimal Operation	Number of Test Vectors
Fused Multiply Add	927668
Addition	136340
Multiplication	96845
Total	1160853

Table 6.1: Number of test vectors applied for each decimal operation

6.1.2 Binary Verification

Unfortunately, as far as we know, there is no open source test vectors for binary available so far. Hence our design is not fully verified as a binary FMA. However, it has been tested for different cases including underflow, overflow, zero result, subnormal result, subnormal inputs, massive right and left shift, normal operation and it passed in all cases giving the correct result.

6.2 Synthesis Results

Our design was synthesized using TSMC65nmLP kit using typical process and temperature, and 1.2V supply, where the gate delay of a FO4 inverter is 35 ps, and the Area of the smallest NAND2 gate is 1.6 μm^2. Synthesis was done using Synopsys design compiler. compile_ultra command was used to get the maximum minimization effort. The target of the synthesis was the delay. Synthesis results showed a delay of 4.13 ns, area of 195,000μm^2, and power consumption of 112 mW.

It has to be pointed out that our unit ,without any modification, when operated as a binary unit has a total delay of 3.4 ns (97 FO4 delay).

6.2.1 Delay and Area contributions

Figure 6.1 shows the main contributors in the critical path delay, and Figure 6.2 shows the area of the largest blocks of the FMA.

6.2.2 Comparison With Other Units

Table 6.2 shows a comparison with the previously published FMAs. As shown in Table 6.2 our binary/decimal design is faster than any of decimal FMAs published before, and when compared to binary units, IBM Power6's binary unit is only 6.2% faster than our binary/decimal unit when it operates as a binary unit. Also when we

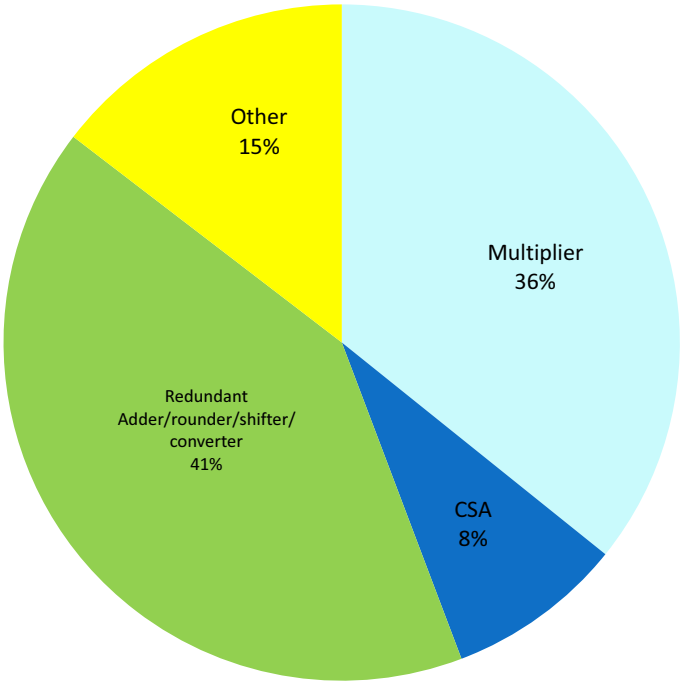

Figure 6.1: Delay of Different Blocks in the Critical Path

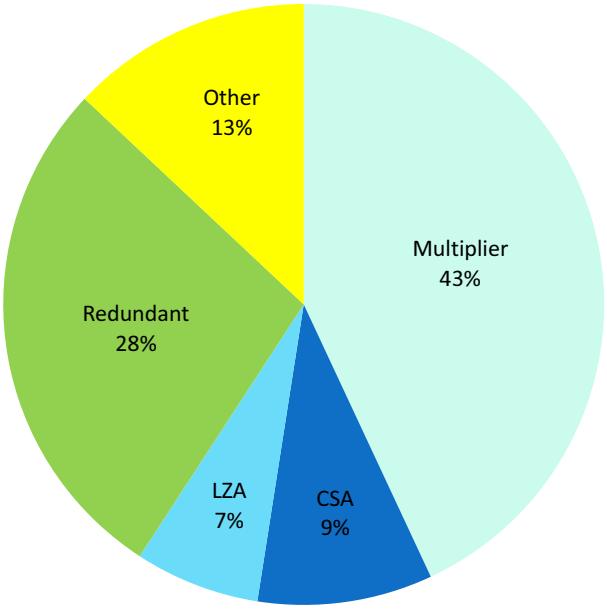

Figure 6.2: Area of Different Blocks in the FMA

	Notes	Delay (FO4)	Norm-alized	Area (NAND2)	Norm-alized
Akkas's [13]	decimal64	128.6	1.06	77,461	0.635
Ahmed El- Tantawi's [8]	decimal64	132	1.09	100,000	0.821
Rodina's [5]	dec64/128	144.4	1.19	83,421	0.684
Lang's [14]	binary64	145	1.2	NA	NA
IBM Power6 Unit [22]	binary64	91	0.75	NA	NA
Monsson's* [4]	bin/dec64	121.7	1.002	273,766	2.246
Our Binary Design	binary64	80	0.68	52,283	0.43
Our Decimal Design	decimal64	115	0.97	84,375	0.69
Our proposed design	bin/dec64	118**	1	121,875	1

*Incomplete

**When it operates as a binary unit, the total delay is 97 FO4 delay

Table 6.2: Comparison of Delay in FO4 and Area in NAND2 with Other FMAs

remove the decimal parts of our design it takes a delay which is comparable to the IBM power 6 binary unit and when we remove the binary hardware, our decimal unit is even faster than any of the decimal Units. Also, Our binary/decimal unit takes 12% less area than two stand-alone binary and decimal units.

The advantage in delay over other decimal units comes from using the redundant system, which allowed us to remove the delay of the rounding completely from the critical path.

6.2.2.1 Power consumption

To compare power consumption, we can not compare to previously published units as they didn't report the power dissipation. We can only compare to our standalone binary and decimal units. Our binary unit consumes 52 mWatts, and our decimal unit consumes 88 mWatts. The combined binary/decimal unit consumes 112 mWatts. Which means that the binary/decimal unit consumes 20% less power than the two standalone units.

Chapter 7

Conclusion

In this work we presented a new 64-bit floating point Binary/Decimal FMA. After decoding the operands, the multiplier and the multiplicand are sent to the multiplier tree to get the multiplication result. In parallel to the multiplier tree, the addend is complemented if needed and aligned by shifting it to the left or the right depending on the exponent difference. The multiplier tree used here uses the SD-radix5 and SD-radix4 recoding for decimal and binary respectively. A multi-operand adder is used to reduce the multiplication tree to only three columns. These three columns, as well as the prepared addend, are reduced using a 4:2 CSA to only two vectors that needs to be added. In parallel to the adder the LZA anticipates the leading zeros in the result for both binary and decimal. In binary a new LZA that generates its output in base-3 format to simplify the final normalization shifting is proposed. The addition is performed in a redundant system that uses the digit set [-6,6] to represent each decimal or octal digit. The two vectors are converted to this redundant format and added using a redundant adder. The result is normalized after the addition and needs to be rounded and converted back to binary/decimal. A new rounding-while-redundant technique is proposed to hide the rounding delay and perform the rounding in parallel with the conversion to binary/decimal. Finally the design was synthesized using TSMC65nmLP kit and the results were discussed and compared with other previously published units.

References

[1] "Telco Benchmark for telephone company billing application, Available online at : http://speleotrove.com/decimal/telco.html/, last accessed on May, 23rd 2014."

[2] "IEEE Standard for Floating-Point Arithmetic," *IEEE Std 754-2008*, pp. 1–58, Aug 2008.

[3] C. Hinds, "An enhanced floating point coprocessor for embedded signal processing and graphics applications," in *Signals, Systems, and Computers, 1999. Conference Record of the Thirty-Third Asilomar Conference on*, vol. 1, pp. 147–151 vol.1, Oct 1999.

[4] P. K. Monsson, "Combined binary and decimal floating-point unit," Master's thesis, Technical University of Denmark, 2008.

[5] R. Samy, H. Fahmy, R. Raafat, A. Mohamed, T. ElDeeb, and Y. Farouk, "A decimal floating-point fused-multiply-add unit," in *Circuits and Systems (MWSCAS), 2010 53rd IEEE International Midwest Symposium on*, pp. 529 –532, aug. 2010.

[6] "Sun BigDecimal Library, Available Online At : http://download.oracle.com/javase/1.5.0/docs/api/java/math/BigDecimal.html, last accessed on May, 23rd 2014 ."

[7] A. Vazquez, E. Antelo, and P. Montuschi, "A new family of high.performance parallel decimal multipliers," in *Computer Arithmetic, 2007. ARITH '07. 18th IEEE Symposium on*, pp. 195 –204, June 2007.

[8] A. EL-Tantawy, "Decimal floating point arithmetic unit based on a fused multiply add module," Master's thesis, Electrical and Electrical Communications Department, Cairo University, 2011, Available At : http://eece.cu.edu.eg/∼hfahmy/thesis/2011_08_dfma.pdf.

[9] A. Vazquez, E. Antelo, and P. Montuschi, "Improved design of high-performance parallel decimal multipliers," *Computers, IEEE Transactions on*, vol. 59, pp. 679 –693, May 2010.

[10] P. M. Kogge and H. S. Stone, "A parallel algorithm for the efficient solution of a general class of recurrence equations," *Computers, IEEE Transactions on*, vol. C-22, pp. 786 –793, Aug 1973.

[11] A. Vazquez, *High Performance Decimal Floating Point Units*. PhD thesis, Electrical and Computer Engineering, Universidade de Santiago de Compostela. Spain, 2009.

[12] A. Vazquez and E. Antelo, "A high-performance significand BCD adder with IEEE 754-2008 decimal," in *Computer Arithmetic, 2009. ARITH-19 2009. 19th IEEE Symposium on*, pp. 135 – 144, June 2009.

[13] A. Akkas and M. Schulte, "A decimal floating-point fused multiply-add unit with a novel decimal leading-zero anticipator," in *Application-Specific Systems, Architectures and Processors (ASAP), 2011 IEEE International Conference on*, pp. 43–50, Sept 2011.

[14] T. Lang and J. Bruguera, "Floating-point multiply-add-fused with reduced latency," vol. 53, pp. 988–1003, Aug 2004.

[15] L. Dadda, "Multioperand parallel decimal adder: A mixed binary and BCD approach," *Computers, IEEE Transactions on*, vol. 56, no. 10, pp. 1320–1328, 2007.

[16] K. Yehia, "A mixed decimal/binary redundant floating-point adder," Master's thesis, Electronics and Electrical Engineering Department, Cairo University, 2011, Available At : http://eece.cu.edu.eg/∼hfahmy/thesis/2011_07_d_b_fpadd.pdf.

[17] L.-K. Wang and M. Schulte, "A decimal floating-point adder with decoded operands and a decimal leading-zero anticipator," in *Computer Arithmetic, 2009. ARITH 2009. 19th IEEE Symposium on*, pp. 125 –134, June 2009.

[18] V. G. Oklobdzija, "An algorithmic and novel design of a leading zero detector circuit: Comparison with logic synthesis," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 2, pp. 124 –128, March 1994.

[19] A. Verma, A. Verma, P. Brisk, and P. Ienne, "Hybrid LZA: A near optimal implementation of the leading zero anticipator," in *Design Automation Conference, 2009. ASP-DAC 2009. Asia and South Pacific*, pp. 203–209, Jan 2009.

[20] A. Sayed-Ahmed, H. Fahmy, and M. Hassan, "Three engines to solve verification constraints of decimal floating-point operation," in *Signals, Systems and Computers (ASILOMAR), 2010 Conference Record of the Forty Fourth Asilomar Conference on*, pp. 1153–1157, 2010.

[21] "Cairo University Test vectors, Available Online At : http://eece.cu.edu.eg/∼hfahmy/arith_debug/index.htm, last accessed on May, 23rd 2014l."

[22] S. Trong, M. Schmookler, E. Schwarz, and M. Kroener, "P6 binary floating-point unit," in *Computer Arithmetic, 2007. ARITH '07. 18th IEEE Symposium on*, pp. 77–86, June 2007.