The Report Committee for Jongwook Sohn
Certifies that this is the approved version of the following report:

Improved Architectures for a Fused Floating-Point Add-Subtract Unit

APPROVED BY
SUPERVISING COMMITTEE:

Supervisor:

Earl E. Swartzlander, Jr.

Lizy K. John

Improved Architectures for a Fused Floating-Point Add-Subtract Unit

by

Jongwook Sohn, B.S.E.

Report

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

December 2011

Dedicated to my family with all my heart.

Acknowledgements

Most of all, I would like to express my sincere gratitude to my supervisor, Professor Earl E. Swartzlander, Jr. for his support, advice and encouragement on my graduate studies. I believe it is a great fortune for me to work with him, the foremost authority in my research area. I also deeply thank Professor Lizy K. John for her help as a reader of the Master's report.

I want to express my best gratefulness to my former teacher and a mentor, Professor Seon Wook Kim in Korea University. His guidance and training throughout my undergraduate studies have formed the cornerstones of my current research. I also would like to thank Professor Youngsun Han for his help in my undergraduate school life as a colleague, and as a friend.

I especially would like to express my appreciation to Yonghyun Kim who helped me to work in Intel, a wonderful place to be with good colleagues – Bong Wan Jun, Jae Wook Lee, Suk-joon Hong, Joon Sung Yang, Dongwoon Kim, Joonsoo Kim and Jae Hong Min. I also would like to thank my friends – Sanghyun Chi, Yeojoon Kim, Ikhwan Lee, Dam Sunwoo, Sangmin Lee, Seyoung Kim, Ick-Jae Yoon, Jaeyong Chung, Jungho Jo, Seungyun Nam, Donghyuk Shin, Eunho Yang, Min Kyu Jeong, Jinsuk Chung, Minsoo Rhu, Hyungman Park, Dongwook Lee, Youngkyu Lee, Sungpil Yang, Jaehyun Ahn and Changhyuk Kim who have been making my happy life in Austin.

I wish to express my deepest thanks to my family. I am grateful to my mother for rightly raising me up, my father for being my role model. I am also grateful to my parents-in-law with the same amount of respect to my parents. I would like to thank my brother Jinho Sohn, brother-in-law Kyungsoo Lee and his wife Jinju Han for their kind

considerations. I would like to thank my cousins Tae Eun Kang and Taewoo Kang for always giving me the inspiration and motivation. I also would like to thank my wife's cousins Yerin Lee, Yejin Lee and Taewon Lee for their constant encouragement. Finally, I would like to express all my love to my wife Soojin Lee and my daughter Eunsuh Sohn.

Jongwook Sohn

The University of Texas at Austin

December 2011

Abstract

Improved Architectures for a Fused Floating-Point Add-Subtract Unit

Jongwook Sohn, M.S.E.

The University of Texas at Austin, 2011

Supervisor: Earl E. Swartzlander, Jr.

This report presents improved architecture designs and implementations for a fused floating-point add-subtract unit. The fused floating-point add-subtract unit is useful for DSP applications such as FFT and DCT butterfly operations. To improve the performance of the fused floating-point add-subtract unit, the dual path algorithm and pipelining technique are applied. The proposed designs are implemented for both single and double precision and synthesized with a 45nm standard-cell library. The fused floating-point add-subtract unit saves 40% of the area and power consumption and the dual path fused floating-point add-subtract unit reduces the latency by 30% compared to the traditional discrete floating-point add-subtract unit. By combining fused operation and the dual path design, the proposed floating-point add-subtract unit achieves low area, low power consumption and high speed. Based on the data flow analysis, the proposed fused floating-point add-subtract unit is split into two pipeline stages. Since the latencies of two pipeline stages are fairly well balanced the throughput of the entire logic is increased by 80% compared to the non-pipelined implementation.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

This chapter presents recent issues that arise with floating-point arithmetic and its applications. It also includes an introduction to the fused floating-point add-subtract unit as a solution of some of those issues, and a brief overview of the report.

1.1 Motivation

The computer arithmetic units in a modern microprocessor execute advanced applications such as 3D graphics, multimedia, signal processing and various scientific computations that require complex mathematics. The binary fixed-point number system is not sufficient to handle such complex computations. In contrast, the binary floating-point notation, which is specified in IEEE-754 Standard floating-point arithmetic [1], represents a wide range of numbers from tiny fractional numbers to infinitely huge numbers so that it avoids overflow and underflow. However, the floating-point operations require complex procedures. Since the floating-point numbers consist of sign, exponent and significand parts, the operations should consider the normalization, which causes the increased logic delay. Therefore, improving the performance of the floating-point operations has long been a research topic in the computer arithmetic field.

To improve the performance of floating-point arithmetic, several fused floating-point operations have been introduced: Fused Multiply-Add (FMA) [2], [3], [4], Fused Add-Subtract [5], and Fused Two-Term Dot-Product [6]. The fused floating-point operations not only improve the performance, but also reduce the area and power consumption compared to the combination of traditional floating-point implementations.

This report presents improved architecture designs and implementations for a fused floating-point add-subtract unit. Many DSP applications such as FFT and DCT butterfly operations have been developed to utilize the fused floating-point add-subtract unit [7], [8]. Therefore, the improved fused floating-point add-subtract unit will contribute to the next generation floating-point arithmetic and DSP application development.

1.2 Approach

The proposed fused floating-point add-subtract unit takes two normalized floating-point operands and generates their sum and difference simultaneously. It supports all five rounding modes specified in IEEE-754 Standard [1]. Several techniques are applied to achieve low area, low power consumption and high speed floating-point arithmetic:

1) Instead of executing two identical floating-point adders, the fused floating-point add-subtract unit shares much of the common logic to generate the sum and difference simultaneously. Therefore, it saves significand area and power consumption compared to a discrete floating-point add-subtract unit. Also, it reduces the latency by simplifying the control signals.
2) A dual path algorithm is applied for high speed floating-point operation. The dual path logic consists of a far path and a close path. In the far path, the addition, subtraction and rounding logic are performed in parallel. By aligning the significands to the minimal number of bits, the addition, subtraction and rounding logic are simplified. There are three cases for the close path depending on the difference of the

exponents. For each case, addition, subtraction and leading zero anticipation (LZA) are performed in parallel and rounding is not required. Therefore, the dual path design simplifies much of the logic in the critical path.

3) To increase the total throughput, pipelining is applied. Based on data flow analysis, the proposed fused floating-point add-subtract unit is split into two pipeline stages. By properly arranging the components, latencies of the two pipeline stages are well balanced so that the throughput of the entire design is increased.

1.3 Report Overview

This report is divided into 6 chapters. Chapter 2 provides the introduction to the IEEE-754 floating-point standard, which is a fundamental of the floating-point arithmetic covered in the report. Chapter 3 presents fused floating-point add-subtract unit designs. Serial and parallel discrete floating-point add-subtract unit designs are introduced. Then, naïve and improved fused floating-point add-subtract unit designs are presented and compared with the discrete designs. Chapter 4 presents a dual path floating-point add-subtract unit design. To achieve a low area, low power and high performance floating-point add-subtract unit, this report proposes a dual path design and provides implementation details. Chapter 5 analyzes the data flow of the proposed dual path fused floating-point add-subtract unit to properly construct pipeline stages and arrange the logic components into the pipeline stages for increasing the throughput. The designs presented in Chapters 3 to 5 are implemented for both single and double precision formats. The double precision implementation can be done by extending the single precision

implementation. For simplicity, only single precision implementation details are described and the implementation results for both precisions are presented in the implementation results sections in each chapter. Finally, Chapter 6 concludes the report by summarizing the designs and implementation results and suggests several ideas for future work.

Chapter 2: Background

This chapter provides a brief introduction to the IEEE-754 floating-point standard which is a fundamental aspect of the floating-point arithmetic covered in the report.

2.1 The IEEE-754 Floating-Point Standard Formats

The IEEE-754 floating-point standard provides a discipline for performing floating-point computation [1]. In any precision, the floating-point number consists of three parts: 1) Sign, 2) Exponent, and 3) Significand. The floating-point number system is classified as a sign-magnitude representation, which means the MSB represents the sign bit – "0" indicates a positive number and "1" indicates a negative number. The exponent bits represent a multiplier, which is an exponential form with a base of 2 for binary or 10 for decimal format. As the most commonly used format, only the binary format is covered in this report. The exponent is biased by the half the maximum exponent so that it can represent both positive and negative exponents. The significand bits represent a fraction that is multiplied by the exponent term. The significand is normalized so that the MSB is implicitly set to "1", which increases the significand precision by 1. The sign, exponent and significand represent a binary floating-point number as following:

$$(-1)^{sign} \times 2^{exponent} \times significand$$

where

$sign$ = 0 or 1

$exponent = e - e_{bias} + 1$ (e = any integer between 0 and $2^{\#\text{ of exponent bits}}$)

$significand = d_0.d_1d_2...d_{p-1}$ (d_i = 0 or 1, p = significand precision)

The IEEE-754 floating-point standard provides the parameters for the equations as shown in Table 1.

Table 1. IEEE-754 Floating-Point Single and Double Precision Specifications

Format	Single Precision	Double Precision
Sign	1	1
Exponent	8	11
Significand	23	52
Total	32	64
Exponent Bias	127	1023
Exponent Range	$2^{-126} - 2^{127}$	$2^{-1022} - 2^{1023}$
Significand Precision	24	53

The IEEE-754 floating-point standard defines the single precision format has 1 sign bit, 8 exponent bits, and 23 significand bits, which adds up to 32-bits. The double precision format extends it to 64-bits that include 1 sign bit, 11 exponent bits, and 52 significand bits. Figure 1 shows the bit partitions for the single and double precision formats. In this report, both single and double precision implementations are covered. Since the double precision implementation can be done by extending the single precision, only single precision implementation details are described, although the implementation results for both precisions are presented.

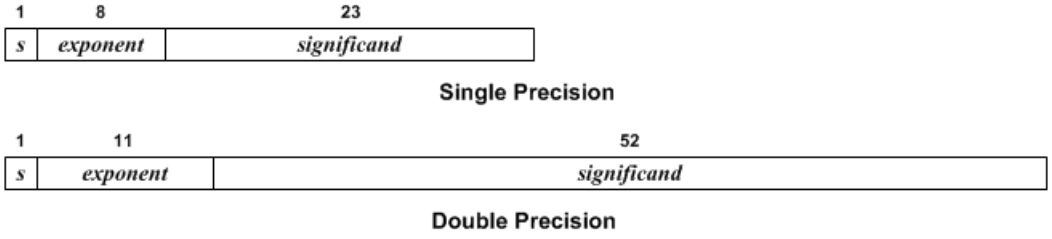

Figure 1. IEEE-754 Floating-Point Single and Double Precision Formats

2.2 Rounding Modes

The IEEE-754 floating-point standard defines five rounding modes: 1) Round to positive infinity, 2) Round to negative infinity, 3) Round to zero, 4) Round to nearest even, and 5) Round to nearest away from zero. The first three modes round the number to the certain direction that are positive infinity, negative infinity, and zero, respectively. The other two modes select a direction to round the number to the nearest. If the number is equally nearest to two numbers (i.e., ties), the number with an even LSB and the number with a larger magnitude is selected, respectively. Generally, the nearest rounding modes are more precise than the directed rounding modes. The fused floating-point add-subtract unit proposed in this report supports all five rounding modes.

2.3 Special Values

The IEEE-754 floating-point standard specifies four kinds of special values: 1) Signed zero, 2) Subnormal numbers, 3) Infinities, and 4) NaNs (Not-a-Numbers). Since the floating-point number is a sign-magnitude representation, both positive and negative zeros exist. The two values are numerically equal, whereas some operations produce different results depending on the sign (e.g., $1 / (+0) = \infty$ and $1 / (-0) = -\infty$). A subnormal number represents a value of the magnitude which is smaller than the minimum normalized number by denormalizing the significand, which means the MSB of the significand is "0". It improves the precision of the numbers that are close to zero so that the values can be represented when underflow occurs. The infinities are represented by setting all exponent and significand bits to "1" and the positive and negative infinities are

determined by the sign bit. The infinities are returned when the values are not representable due to the overflow. The NaNs are returned when an invalid operation occurs such as $(+\infty) + (-\infty)$, $0 \times \infty$ and sqrt(–1). The exponent bits of the NaNs are all "1" and the significand bits are encoded in various ways depending on the invalid operations.

2.4 Exceptions

The IEEE-754 floating-point standard specifies five exception cases: 1) Invalid operation, 2) Division by zero, 3) Overflow, 4) Underflow, and 5) Inexact. For each exception case, the implementation generates a corresponding status flag. The invalid operation exception occurs when the result of the operation is not definable and it returns NaN. Division by zero raises the exception and returns $\pm\infty$. The overflow flag is set when the result of the operation exceeds the representable range and it returns $\pm\infty$. The underflow flag is set when the result of the operation is too small to represent and it returns zero or a subnormal number. Finally, the inexact exception occurs when the result of the operation is different from the mathematical exact value. Since only overflow, underflow and inexact occur in the floating-point addition and subtraction, the proposed floating-point add-subtract unit supports those three exception cases.

Chapter 3: A Fused Floating-Point Add-Subtract Unit

In this chapter, a fused floating-point add-subtract unit design is presented. Traditional floating-point add-subtract units have been implemented by executing two discrete floating-point adders either in serial or parallel. A fused floating-point add-subtract unit produces the results of addition and subtraction simultaneously with a single operation. It provides reduced latency compared to the serial discrete implementation and reduced area and power consumption compared to the parallel discrete implementation.

3.1 Traditional Floating-Point Add-Subtract Unit

A direct way to implement a floating-point add-subtract operation is to execute two identical floating-point adders either in serial or parallel. The serial implementation performs the addition with a floating-point adder and performs the subtraction after the addition is completed. The addition result is temporarily stored in a flip-flop and released when the subtraction is completed so that the sum and difference results are produced simultaneously as shown in Figure 2.

The parallel implementation uses two identical floating-point adders in parallel as shown in Figure 3. One of those adders performs the addition and the other performs the subtraction to produce the sum and difference results simultaneously.

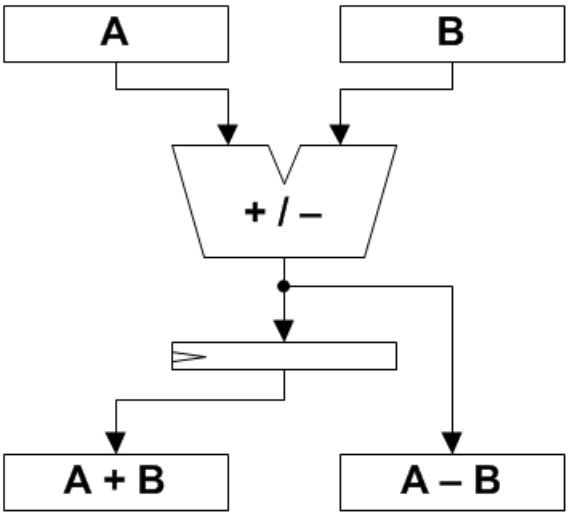

Figure 2. Serial Discrete Floating-Point Add-Subtract Unit

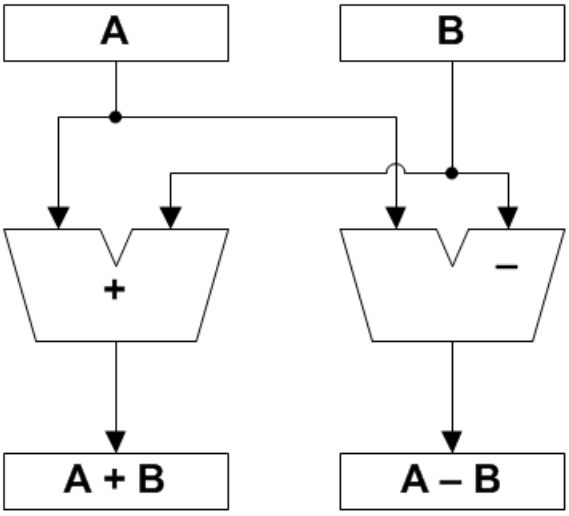

Figure 3. Parallel Discrete Floating-Point Add-Subtract Unit

A traditional floating-point adder [9], [10] such as that of Figure 4 can be used for each operation. The steps to execute the floating-point addition are:

1) Exponent compare logic compares the exponents of the two operands to determine which exponent is greater and calculates their difference.
2) The exponent comparison results are used for the significand swap logic. When the exponents are equal, the significands are compared to identify the smaller significand. The significand of the smaller operand is shifted by the amount of the exponent difference (if any) for the alignment and the guard, round and sticky bits are attached to the LSB.
3) Since some of rounding modes specified in IEEE-754 Standard [1] require knowing the sign (i.e., round to positive and negative infinity), the sign logic must be performed prior to the round logic. The sign logic provides the sign of the sum and the operation decision bit to the round logic and significand adders, respectively.
4) The two significands are passed to the significand adder, round logic and LZA simultaneously. The significand adder performs the addition or subtraction of the two significands depending on the operation. It produces rounded and unrounded results and the round logic selects one of them for a fast rounding. The LZA generates the amount of cancellation during the subtraction in a constant time so that the subtraction result is immediately normalized [11]. The overflow of the significand adder and the shift amount from the LZA are passed to the exponent adjust logic.

5) Using the shift amount, the exponent adjust logic generates the exponent of the sum. In this step, inexact, overflow and underflow of the exponent (if any) are detected for setting the exception flags.

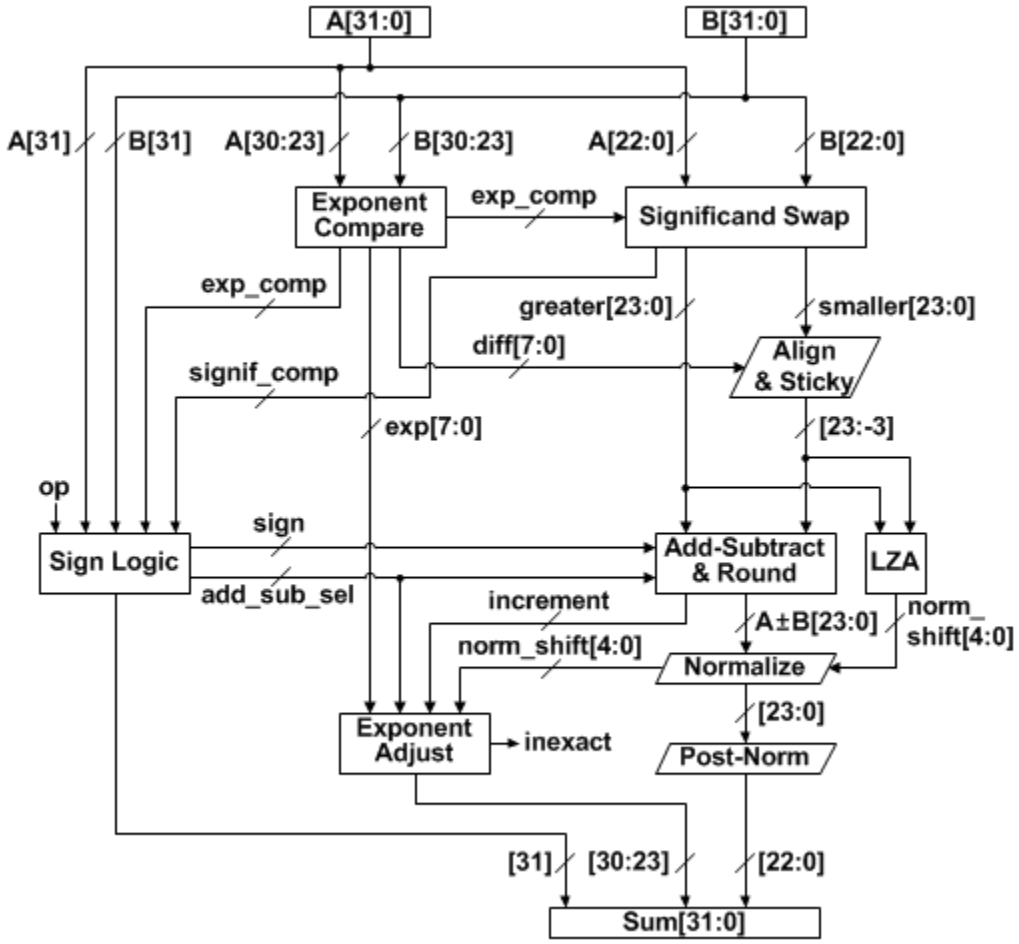

Figure 4. Traditional Floating-Point Adder (After [9], [10])

3.2 A Fused Floating-Point Add-Subtract Unit

The traditional discrete floating-point add-subtract unit produces the sum and difference results simultaneously by executing two identical floating-point additions.

However, much of the logic such as exponent comparison, significand swap and alignment in the two floating-point adders is nearly the same for the two operations.

In order to reduce the overhead, the fused floating-point add-subtract operation was introduced in [5]. The fused floating-point unit produces the sum and difference results simultaneously by sharing the common logic for the two operations as shown in Figure 5.

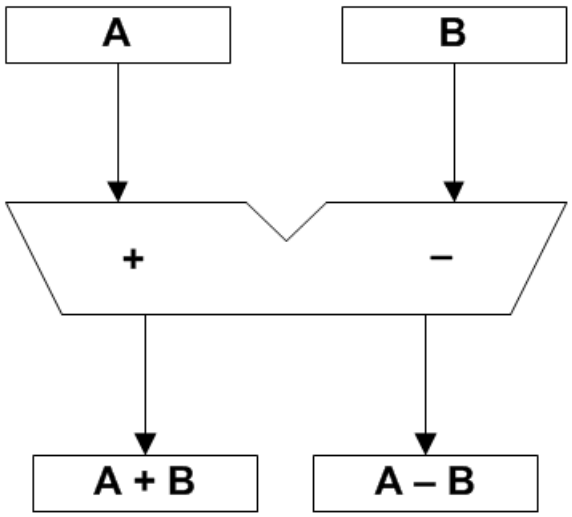

Figure 5. A Fused Floating-Point Add-Subtract Unit

3.2.1 Naïve Fused Floating-Point Add-Subtract Unit Design

A fused floating-point add-subtract unit produces sum and difference results simultaneously by sharing common logic for two operations. Both floating-point addition and subtraction require exponent comparison, significand swapping, alignment, sticky bit formation, sign decision, and exponent adjustment. The fused floating-point add-subtract unit performs those operations only one time so that it saves much of the logic area and

power consumption.

Figure 6 shows a naïve design for a fused floating-point add-subtract unit. The procedure for comparing the two exponents and swapping the significands are the same as for the discrete operation which is explained in the previous section. The sign logic produces two sign bits for sum and difference and operation decision bit. Two integer adders, round logics and LZAs are used for significand addition and subtraction. One

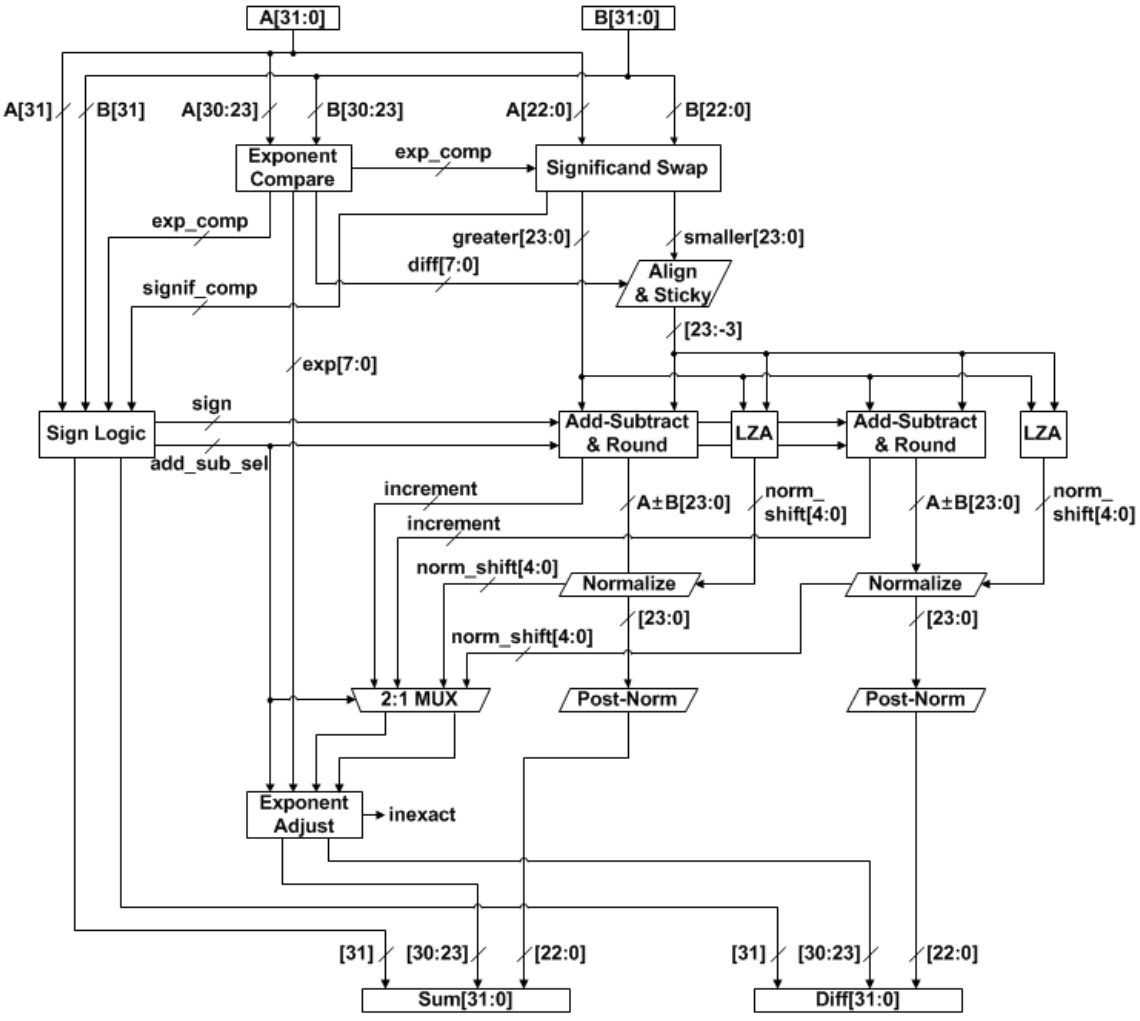

Figure 6. Naïve Fused Floating-Point Add-Subtract Unit (After [5])

significand adder performs the addition and the other performs the subtraction depending on the operation. Each significand adder produces rounded and unrounded results and the round logic selects one of them for a fast rounding. Two overflows and shift amounts are generated for the normalization and exponent adjustment. Since the overflow is used for an addition increment and the shift amount is used for subtraction cancellation, one of the two results is selected by the operation and the other is ignored.

3.2.2 Improved Fused Floating-Point Add-Subtract Unit Design

The naïve fused floating-point add-subtract unit produces the sum and difference results simultaneously by sharing the common logic such as exponent compare, significand swap, exponent adjust, and sign logic. However, it requires additional control logic for determining the operation in the significand adder and exponent adjust logic, which increases the logic delay. Also, two identical significand additions, LZAs and normalizations are executed, even though one the two results is ignored depending on the operation, which causes increased logic area and power consumption.

To eliminate those inefficiencies, an improved design for a fused floating-point add-subtract unit is proposed as shown in Figure 7 [7]. The improved fused floating-point add-subtract unit performs only one significand addition and subtraction for each operation. Table 2 shows the sign decision table based on the signs of the two operands and comparison of the exponents and significands. Since two operations are explicitly performed for sum and difference results (i.e., if the addition is used for the sum, the

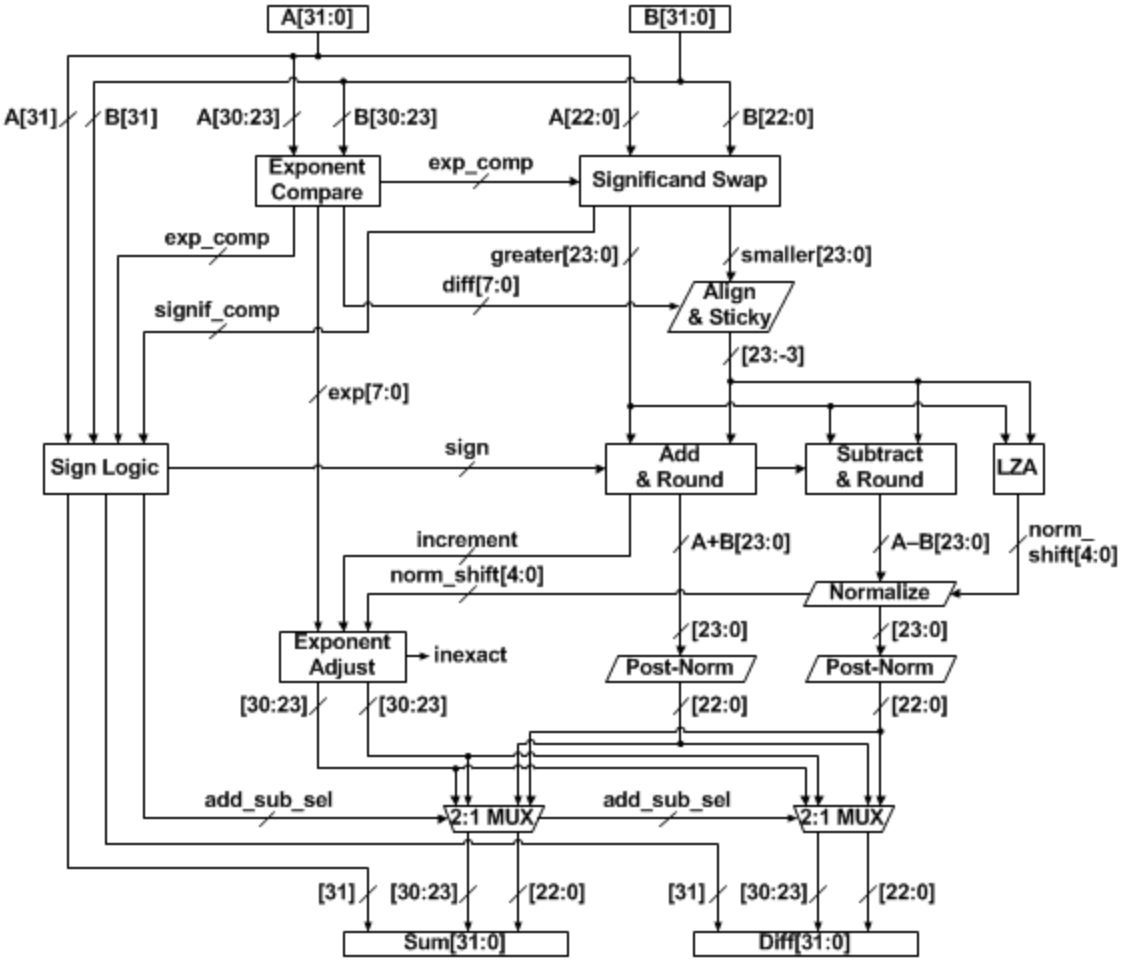

Figure 7. Improved Fused Floating-Point Add-Subtract Unit (After [7])

Table 2. Sign Decision Table

A_sign	B_sign	Comp.	Sum	Difference
+	+	$\|A\| < \|B\|$	$\|A\| + \|B\|$	$-(\|B\| - \|A\|)$
+	+	$\|A\| > \|B\|$	$\|A\| + \|B\|$	$\|A\| - \|B\|$
+	–	$\|A\| < \|B\|$	$-(\|B\| - \|A\|)$	$\|A\| + \|B\|$
+	–	$\|A\| > \|B\|$	$\|A\| - \|B\|$	$\|A\| + \|B\|$
–	+	$\|A\| < \|B\|$	$\|B\| - \|A\|$	$-(\|A\| + \|B\|)$
–	+	$\|A\| > \|B\|$	$-(\|A\| - \|B\|)$	$-(\|A\| + \|B\|)$
–	–	$\|A\| < \|B\|$	$-(\|A\| + \|B\|)$	$\|B\| - \|A\|$
–	–	$\|A\| > \|B\|$	$-(\|A\| + \|B\|)$	$-(\|A\| - \|B\|)$

subtraction is used for the difference), the addition and subtraction are separately placed and only one LZA and normalization (for the subtraction) is required. Assuming both sign bits are positive, the addition and subtraction are performed separately. Then, two multiplexers select the sum and difference results with the operation decision bit, which is the XOR of the two sign bits. More details of the sign logic are provided in Section 4.2.2. This approach simplifies the addition and subtraction operations. It also reduces the control signals for determining the operation in the sub-logic. Thus, the improved fused floating-point add-subtract unit achieves low area, low power consumption and high speed.

3.3 Implementation and Results

Four designs for the fused floating-point add-subtract unit are introduced: 1) Serial discrete design, 2) Parallel discrete design, 3) Naïve fused design, and 4) Improved fused design. Each design for both single and double precision is implemented in Verilog-HDL and synthesized with the Nangate 45nm technology standard cell library. In order to evaluate the designs, the area, critical path latency, throughput and power consumption are compared. Table 3 shows the evaluation results of the four implementations. All the percentages in table are ratios compared to the parallel discrete design.

Since the serial discrete design executes two operations back-to-back with one floating-point adder, it shows the half logic area and power consumption and the twice larger latency compared to the parallel discrete design. The naïve fused design reduces

the area and power consumption by about 40% compared to the parallel discrete design due to its shared logic. The improved fused design reduces the control logic by explicitly executing two operations. As a result, it reduces the area and power consumption by about 45% and reduces the latency by 8%.

Table 3. Discrete vs. Fused Floating-Point Add-Subtract Unit Comparison

Single Precision				
	Parallel Discrete	**Serial Discrete**	**Naïve Fused**	**Improved Fused**
Area (μm^2)	15,403	7,964 (52%)	9,605 (63%)	8,908 (58%)
Latency (ns)	1.32	2.68 (203%)	1.36 (103%)	1.21 (92%)
Throughput (1/ns)	0.76	0.37 (49%)	0.74 (97%)	0.83 (109%)
Power (mW)	7.77	3.98 (51%)	4.78 (62%)	4.21 (54%)
Double Precision				
	Parallel Discrete	**Serial Discrete**	**Naïve Fused**	**Improved Fused**
Area (μm^2)	34,606	18,004 (52%)	20,017 (58%)	18,534 (54%)
Latency (ns)	1.66	3.36. (202%)	1.69 (102%)	1.52 (92%)
Throughput (1/ns)	0.60	0.30 (49%)	0.59 (98%)	0.66 (109%)
Power (mW)	15.46	7.52 (49%)	8.44 (55%)	8.17 (53%)

The double precision implementations for both discrete and fused design require about twice the area and power consumption compared to the single precision implementations due to the larger addition and subtraction. By using the parallel prefix form [12], the larger addition and subtraction logarithmically increases the latency so that the latency for double precision increases by only 25%. The benefits of the fused design are shown in both single and double precision. In terms of logic area and power consumption, the double precision implementation of the improved fused design is about 5% better than the single precision implementation.

Chapter 4: A Dual Path Fused Floating-Point Add-Subtract Unit

Most high speed floating-point adders employ the dual path algorithm [13], [14]. Since the fused floating-point add-subtract unit is based on the floating-point adder, the dual path algorithm is applied to achieve the high performance. In this chapter, a dual path approach for a high performance fused floating-point add-subtract unit is presented. Also, the implementation details of dual path design and sub-modules are presented.

4.1 Dual Path Fused Floating-Point Add-Subtract Unit Design

To achieve a high performance fused floating-point add-subtract unit, the dual path algorithm is employed. Figure 8 shows the dual path floating-point add-subtract unit. The dual path algorithm skips the normalization step depending on the exponent difference. Since the normalization after the subtraction is one of the bottlenecks in the fused floating-point add-subtract unit, the dual path approach improves the performance.

The dual path approach consists of far path and close path logic. The far path takes the significands if the difference of the exponents is larger than 1 and the close path takes the significands if the difference of the exponents is 0 or 1. The next two sub-sections present the implementation details of far path and close path logic.

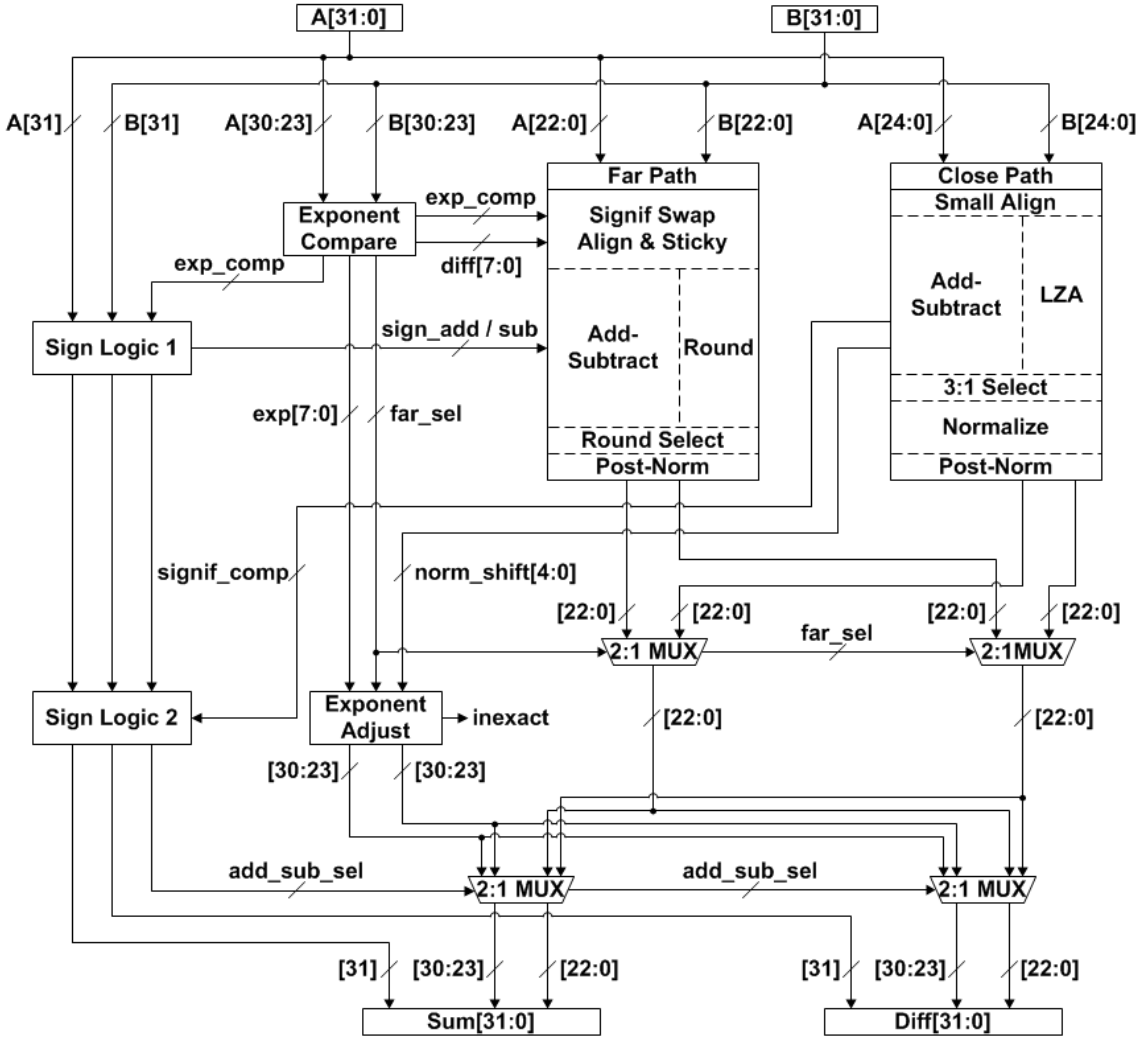

Figure 8. A Dual Path Fused Floating-Point Add-Subtract Unit

4.1.1 Far Path Logic

The far path logic is active when the exponent difference is larger than 1. In this case, massive cancellation does not occur during the subtraction so that the LZA is unnecessary. The far path logic is implemented similar to the front end of the traditional floating-point adder as shown in Figure 9.

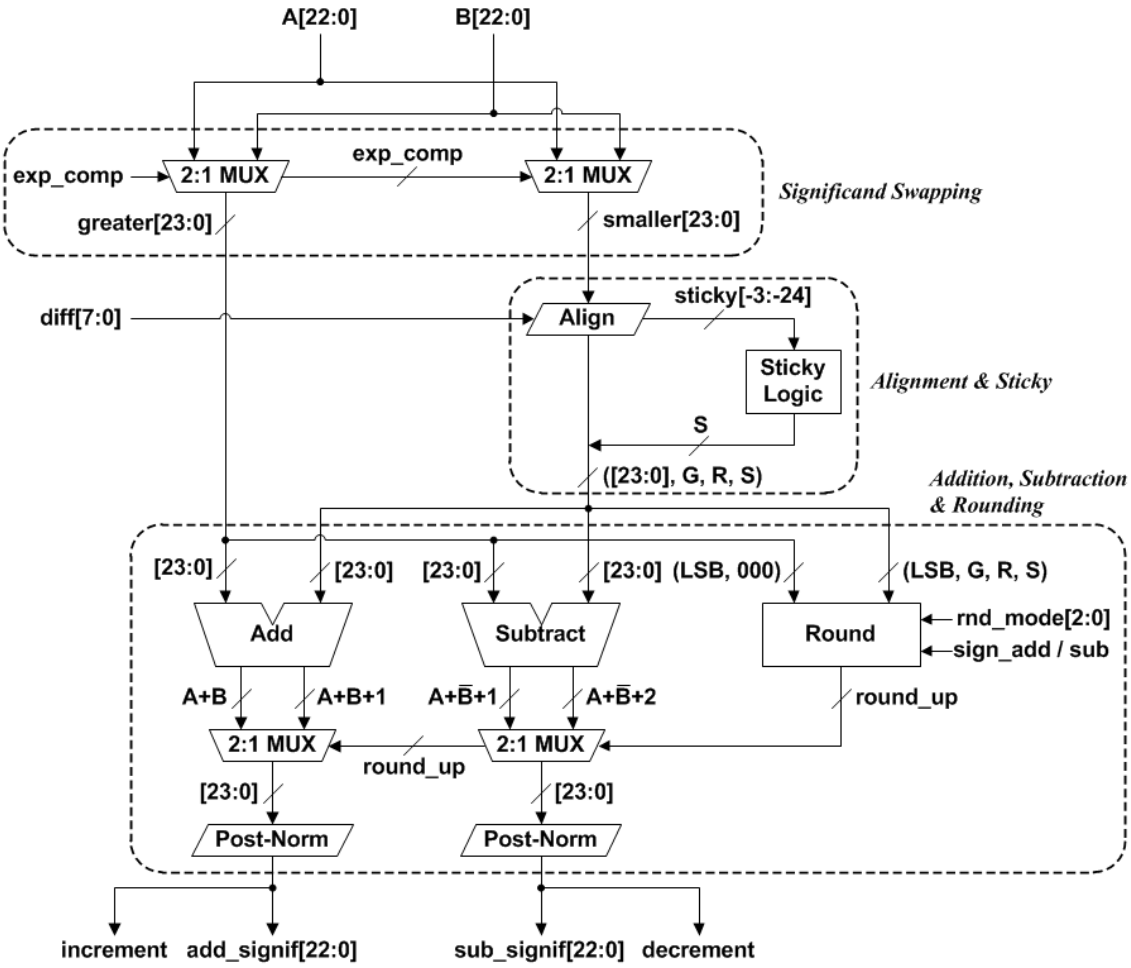

Figure 9. Far Path Logic for a Dual Path Fused Floating-Point Add-Subtract Unit

The far path logic consists of three parts: 1) Significand swapping, 2) Significand alignment and sticky logic, and 3) Significand addition, subtraction and rounding. The greater and smaller significands are determined by swapping two significands based on the exponent comparison:

$$greater\_signif = \begin{cases} (1, \mathrm{A}[22:0]) & \text{if } \mathrm{A}_{exp} > \mathrm{B}_{exp} \\ (1, \mathrm{B}[22:0]) & \text{if } \mathrm{A}_{exp} < \mathrm{B}_{exp} \end{cases}$$

$$smaller\_signif = \begin{cases} (1, \mathrm{B}[22:0]) \gg diff_{exp} & \text{if } \mathrm{A}_{exp} > \mathrm{B}_{exp} \\ (1, \mathrm{A}[22:0]) \gg diff_{exp} & \text{if } \mathrm{A}_{exp} < \mathrm{B}_{exp} \end{cases}$$

where $diff_{exp}$ is the exponent difference. The two significands are aligned with a 1 attached to the MSB end to make 24-bit normalized significands. By aligning the two significands to 24-bits, significand addition and subtraction are simplified, resulting in a reduction in the logic area and delay.

The sticky bit is set if at least one bit of the 22 LSBs is a 1 and the 23rd and the 24th LSBs become the round and guard bits, respectively as shown in Figure 10. Since the significand of the larger operand is not shifted, the 24-bit significand is kept as it is without guard, round and sticky bits.

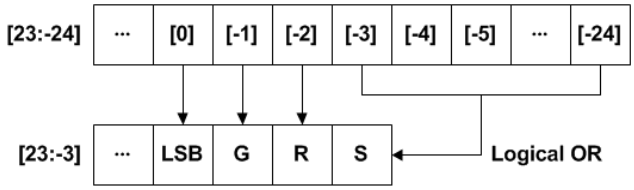

Figure 10. Setting Guard, Round and Sticky bits

The greater and smaller significands are passed to the addition and subtraction units. For the fast integer addition and subtraction, the Kogge-Stone parallel prefix approach is used [12]. More details of the integer adder implementation are described in Section 4.2.3. The addition and subtraction produce the rounded and unrounded results and one of them is selected by the round logic:

$$Add_{signif} = \begin{cases} A + B + 1 & \text{if } round\_up = 1 \\ A + B & \text{otherwise} \end{cases}$$

$$Sub_{signif} = \begin{cases} A + \overline{B} + 2 & \text{if } round\_up = 1 \\ A + \overline{B} + 1 & \text{otherwise} \end{cases}$$

The round logic takes the LSBs, guard, round and sticky bits of the two significands and performs 4-bit addition and subtraction to determine if the result is rounded up or not for each operation. Also, it requires the sign bits of the addition and subtraction to support all five round modes specified in IEEE-754 Standard [1] as shown in Table 4. Since the far path requires at most a 1-bit normalization shift for both addition and subtraction, it avoids a large normalization procedure.

Table 4. Round Table

Round mode [2:0]	*(LSB, G, R, S)	Sign	Round_up
Round to zero (000)	**xxxx	x	0
Round to positive infinity (001)	x000	x	0
	else	+	0
		–	1
Round to negative infinity (010)	x000	x	0
	else	+	1
		–	0
Round to nearest even (011)	≤ 0100	x	0
	> 0100		1
Round to nearest away from zero (100)	< 0100	x	0
	≥ 0100		1

* (LSB, G, R, S) is the 4-bit result of add/sub.
** x means don't care.

4.1.2 Close Path Logic

The close path logic is active when the exponent difference is 0 or 1. Figure 11

shows the close path logic. In contrast to the far path, the close path requires the LZA and normalization to handle the cancellation shift amount during the subtraction. Since the close path is selected only in three cases, three addition, subtraction and LZAs are performed in parallel for a fast significand addition.

The close path logic consists of three parts: 1) Small exponent comparison and significand alignment, 2) Significand addition, subtraction and LZA, and 3) Normalization. Since the exponent difference is small, 2-bit exponent comparison and significand alignment are sufficient for the pre-process of significand addition. There are three cases for each significand alignment depending on the exponent difference:

$$\mathrm{A}_{signif}[23:-1] = \begin{cases} (1, \mathrm{A}[22:0], 0) & \text{if } \mathrm{A}_{exp} - \mathrm{B}_{exp} = 1 \\ (1, \mathrm{A}[22:0], 0) & \text{if } \mathrm{A}_{exp} - \mathrm{B}_{exp} = 0 \\ (01, \mathrm{A}[22:0]) & \text{if } \mathrm{A}_{exp} - \mathrm{B}_{exp} = -1 \end{cases}$$

$$\mathrm{B}_{signif}[23:-1] = \begin{cases} (01, \mathrm{B}[22:0]) & \text{if } \mathrm{A}_{exp} - \mathrm{B}_{exp} = 1 \\ (1, \mathrm{B}[22:0], 0) & \text{if } \mathrm{A}_{exp} - \mathrm{B}_{exp} = 0 \\ (1, \mathrm{B}[22:0], 0) & \text{if } \mathrm{A}_{exp} - \mathrm{B}_{exp} = -1 \end{cases}$$

For each case, addition, subtraction and LZA are performed simultaneously. The leading zero anticipation (LZA) with concurrent correction is used for a fast normalization [15], [16]. More details of the LZA implementation is described in Section 4.2.4. One of the three results is selected based on the small exponent comparison, which compares the two LSBs of the exponents. In contrast to the far path, the significands are not swapped to avoid a large significand comparison. When the subtraction result is negative, a two's complement operation is performed to convert the result to a positive

value. The carry-out of the subtraction indicates a significand comparison, which is passed to the sign logic, to determine the sign bits when the two exponents are equal.

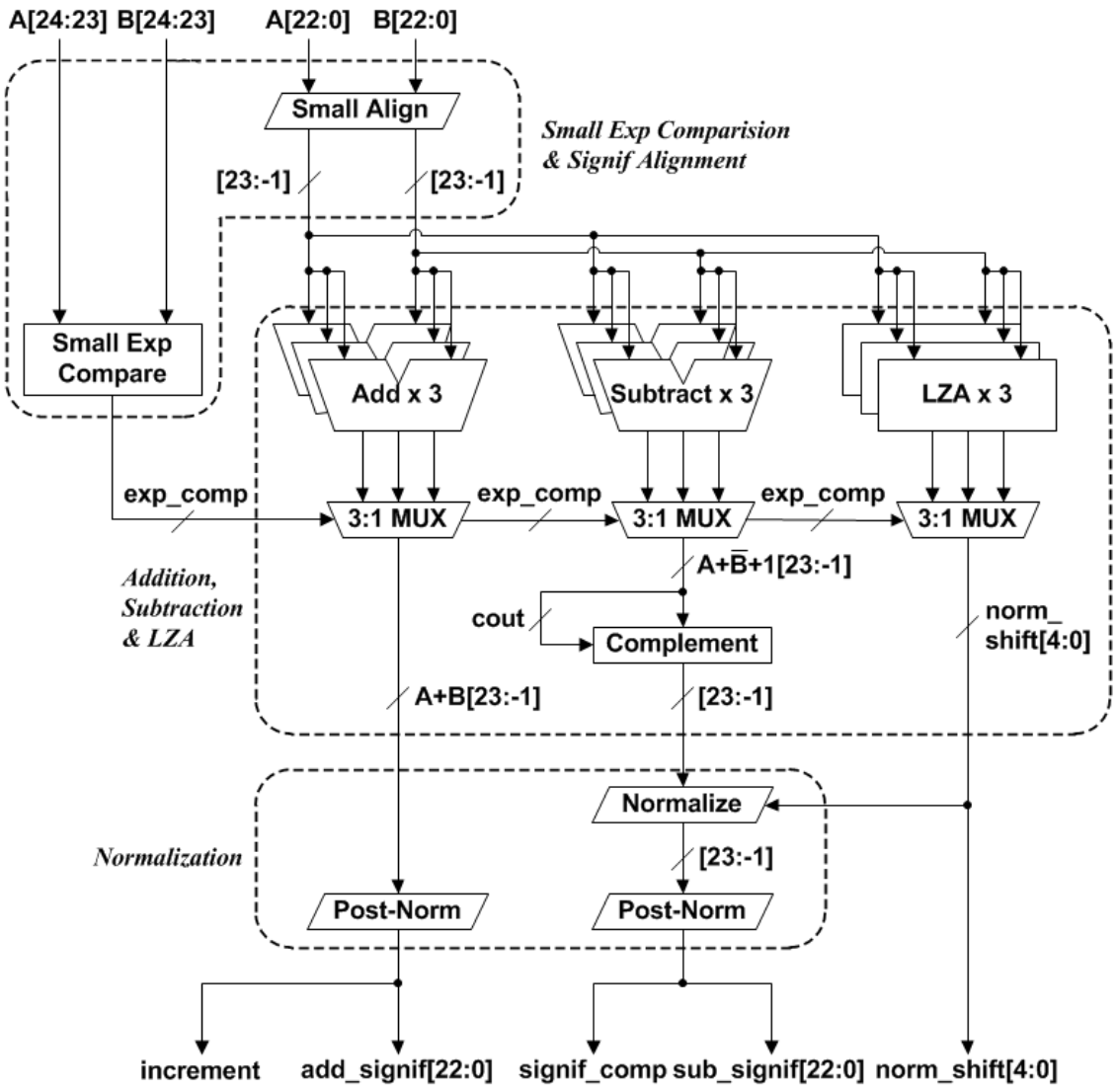

Figure 11. Close Path Logic for a Dual Path Fused Floating-Point Add-Subtract Unit

Since the significands in the close path are mis-aligned by at most 1-bit, rounding is not required [15]. The addition result is normalized by 1-bit overflow, while the subtraction result is normalized by up to 5-bits using the shift amount from the LZA.

4.2 Sub-Modules for a Dual Path Floating-Point Add-Subtract Unit

In this section, the critical sub-modules and implementation details for a dual path floating-point add-subtract unit are presented: 1) Exponent compare logic, 2) Sign logic, 3) Significand adder, 4) Leading-zero anticipator (LZA), and 5) Exponent adjust logic.

4.2.1 Exponent Compare Logic

The exponent compare logic shown in Figure 12 computes the difference of the two exponents and determines which is greater, those are the same functions required for the traditional logic.

The carry-out from the subtraction determines which exponent is greater and the greater exponent is passed to the exponent adjust logic. The exponent subtraction result is complemented if it is negative and passed to the significand swap logic in the far path logic. Also, the subtraction result is used for the path decision between the far path and close path:

$$far\_sel = \begin{cases} 1 & \text{if } \mathrm{A}_{exp} - \mathrm{B}_{exp} \in \{-1, 0, 1\} \\ 0 & \text{otherwise} \end{cases}$$

The path decision bit is passed to the two multiplexers for selecting the addition and subtraction results between the far path and close path.

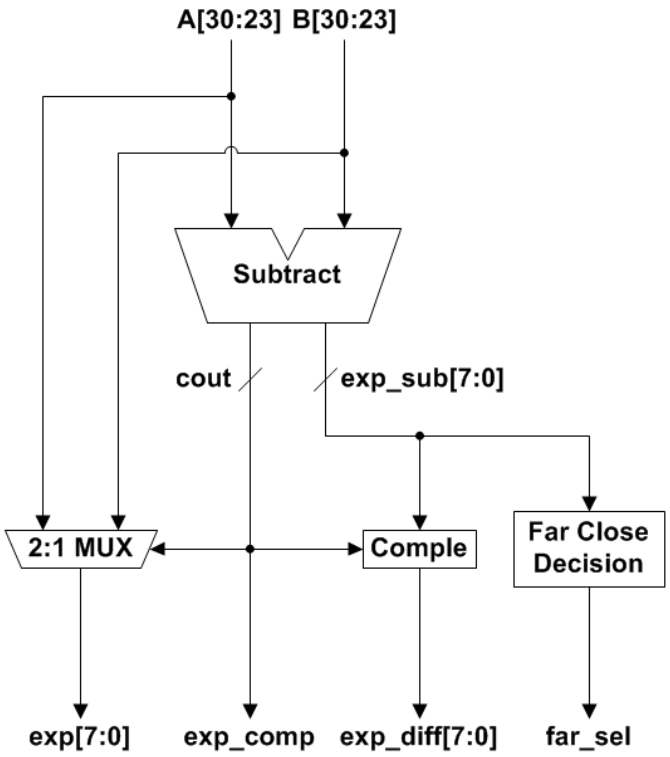

Figure 12. Exponent Compare Logic

4.2.2 Sign Logic

The sign logic consists of two parts as shown in Figure 13. The first sign logic generates two sign bits of the addition and subtraction for the rounding in the far path and the second part generates the sign bits of the sum and difference and an operation decision bit.

In the far path case, the exponent difference is large enough to determine the sign bits with the exponent comparison. Since the round logic in the far path requires the sign bits, the sign bits are passed to the far path logic. The close path, however, requires significand comparison for the case of equal exponents. Therefore, the sign bits of the

sum and difference are generated after the significand comparison bit is provided by the close path. The sign logic for sign bits and an operation decision bit are:

$$add_{sign} = \mathrm{A}_{sign}$$

$$sub_{sign} = \mathrm{A}_{sign} comp_{exp} + \overline{\mathrm{A}}_{sign} \overline{comp}_{exp}$$

$$sum_{sign} = \mathrm{A}_{sign}\, comp_{exp} + \mathrm{A}_{sign} comp_{signif} + \mathrm{B}_{sign} \overline{comp}_{exp} \overline{comp}_{signif}$$

$$diff_{sign} = \mathrm{A}_{sign} comp_{exp} + \mathrm{A}_{sign} comp_{signif} + \overline{\mathrm{B}}_{sign}\, \overline{comp}_{exp} \overline{comp}_{signif}$$

$$add\_sub\_sel = \mathrm{A}_{sign} \oplus \mathrm{B}_{sign}$$

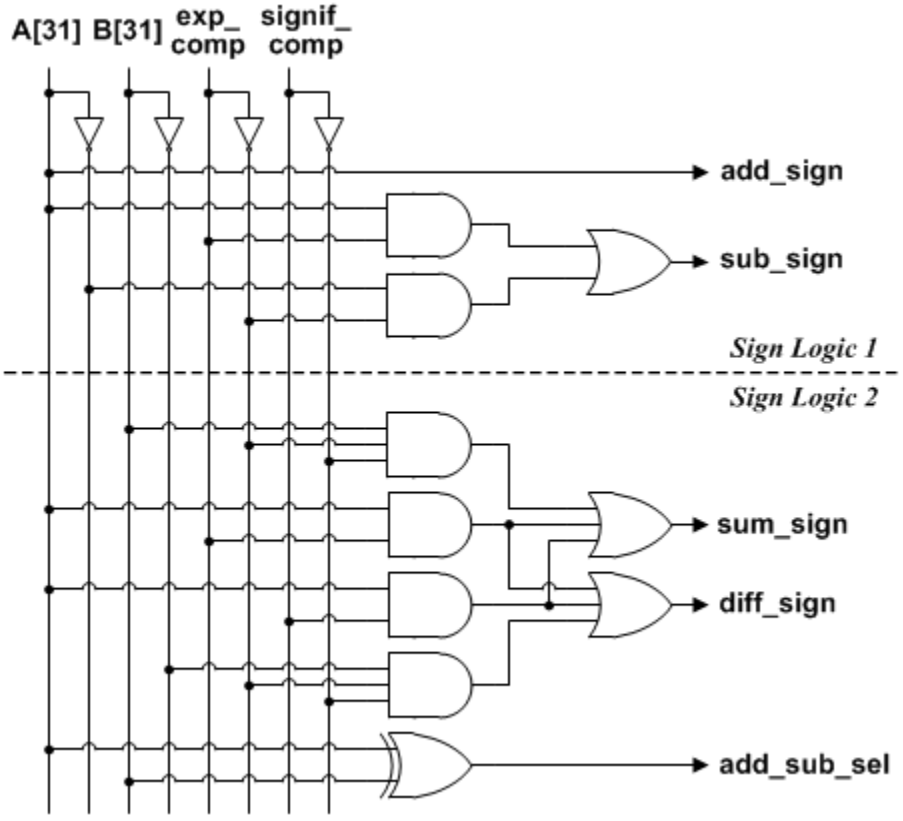

Figure 13. Sign Logic

Once the operation decision bit is generated, it is passed to the two multiplexers for selecting the sum and difference.

4.2.3 Significand Adder

The proposed fused floating-point add-subtract unit requires several integer adders for the significand additions, exponent subtraction, and exponent adjustment. Since the additions account for a large amount of logic area, power consumption and critical path latency, the adder scheme affects to the entire design performance. To achieve a high performance design, the Kogge-Stone adder is employed for all the integer additions for the proposed fused floating-point add-subtract unit. As well known, the Kogge-Stone adder is one of the fastest integer adders using parallel prefix form [17]. The parallel prefix adder is a carry-look-ahead style architecture that uses basic carry operators such as AOI/OAI and NOR/NAND. Figure 14 shows the structure of the 24-bit Kogge-Stone adder [12], which is mainly used for the significand additions. The propagate/generate (PG) generators for the parallel-prefix form is shown in Figure 15.

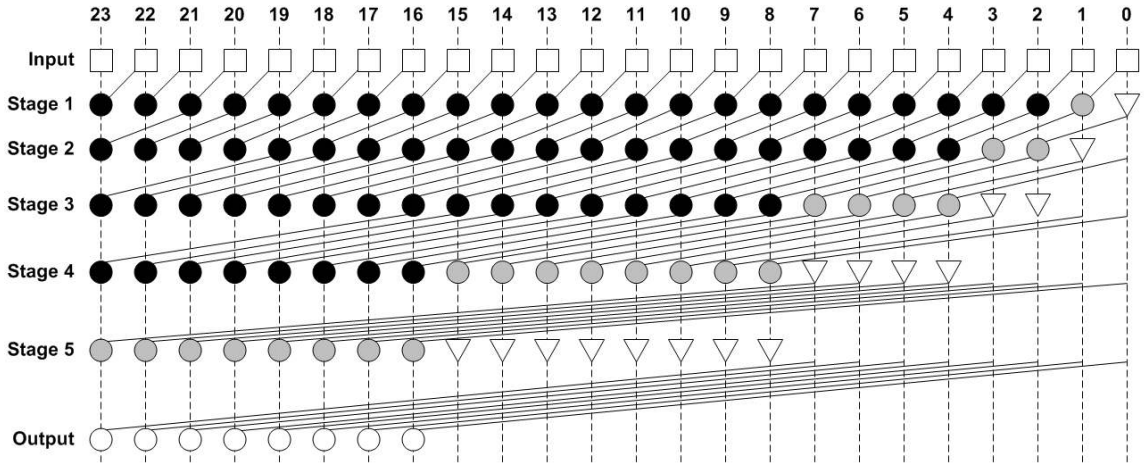

Figure 14. 24-bit Kogge-Stone Adder (After [12])

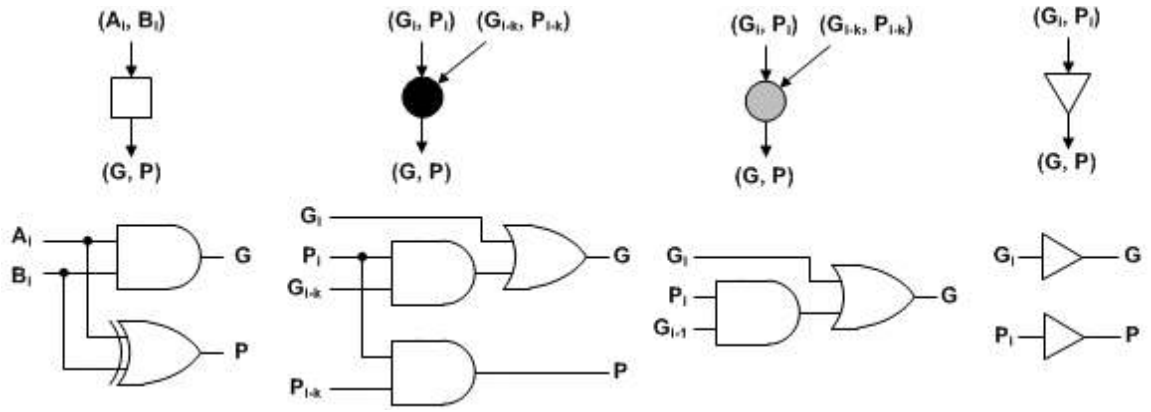

Figure 15. PG Generators for a Parallel Prefix Adder (After [12])

4.2.4 Leading-Zero Anticipator (LZA)

Leading-zero anticipation is used in floating-point adders to eliminate the delay for detecting the MSB location. When massive cancellation occurs during the subtraction, the MSB is left shifted for the normalization. The leading-zero detection (LZD) algorithm computes the shift amount after the subtraction [18], which increases the critical path latency. To eliminate the delay, the LZA is introduced [11], which is performed in parallel with the subtraction. The LZA predicts the MSB location with two operands in a constant time (generally shorter than the subtraction time) so that it hides the delay for detecting the shift amount. For specific input patterns, however, the shift amount from the LZA is required to be corrected after the subtraction, which increases the critical path latency. To avoid the correction logic after the subtraction, concurrent correction logic is proposed [15], [16][1]. Figure 16 and 17 shows the LZA with and without concurrent correction logic, respectively.

[1] [16] modified the logic to correct the error of the logic in [15].

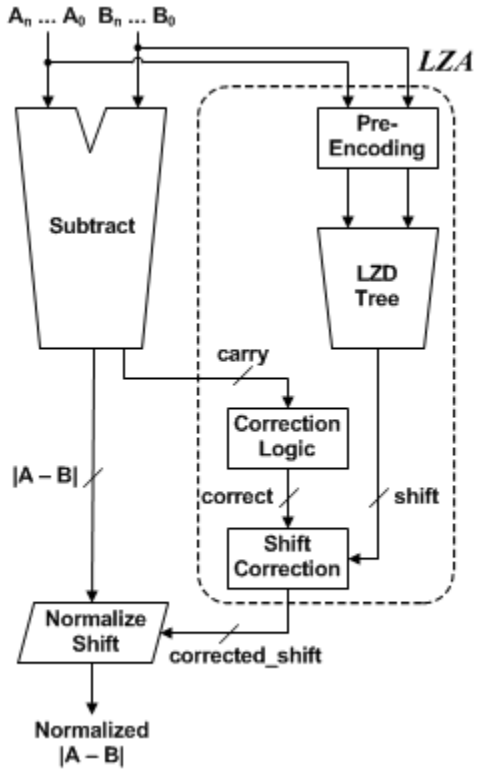

Figure 16. LZA without Concurrent Correction [15]

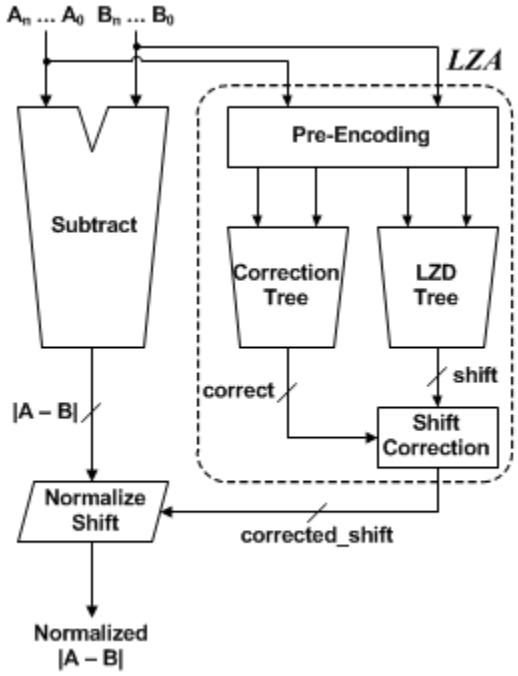

Figure 17. LZA with Concurrent Correction [15]

The pre-encoding logic for the LZA performs bitwise logic with two operands to generate the input pattern of leading-zero detection (LZD) and concurrent correction:

$$e_i = \overline{a_i \oplus b_i} \qquad (e_i = 1 \quad \text{if } a_i = b_i)$$

$$g_i = a_i \overline{b}_i \qquad (g_i = 1 \quad \text{if } a_i > b_i)$$

$$s_i = \overline{a}_i b_i \qquad (s_i = 1 \quad \text{if } a_i < b_i)$$

For F, $\quad f_i = e_{i+1}(g_i \overline{s}_{i-1} + s_i \overline{g}_{i-1}) + \overline{e}_{i+1}(g_i \overline{g}_{i-1} + s_i \overline{s}_{i-1})$

For G_p, $\; p_i = (g_i + e_{i+1}s_i)\overline{s}_{i-1} \qquad n_i = e_{i+1}s_i \qquad z_i = \overline{p_i + n_i}$

For G_n, $\; n_i = (s_i + e_{i+1}g_i)\overline{g}_{i-1} \qquad p_i = e_{i+1}g_i \qquad z_i = \overline{p_i + n_i}$

where $0 \leq i \leq n - 1$. Figure 18 shows the pre-encoding logic of the LZD and concurrent correction. Encoded result F is passed to the LZD tree and G_p and G_n are passed to the correction tree. The shift correction logic increments the shift amount by 1 depending on the concurrent correction. Using the concurrent correction, the logic delay of the LZA does not exceed that of the subtraction so that the LZA logic delay is eliminated.

The LZD tree generates the shift amount with the pre-encoded bits F. Figure 19 shows the 24-bit LZD tree and Figure 20 shows the LZD tree nodes that are used for the proposed design. The LZD tree consists of several levels of LZD tree nodes. Each of the nodes generates *level + 1* bits and passes the bits to the next level node until the number of bits becomes $\lceil log\, n \rceil + 1$, where n is the number of input bits. In the LZD tree nodes, the l_x and r_x represent the inputs left and right from the previous level, respectively.

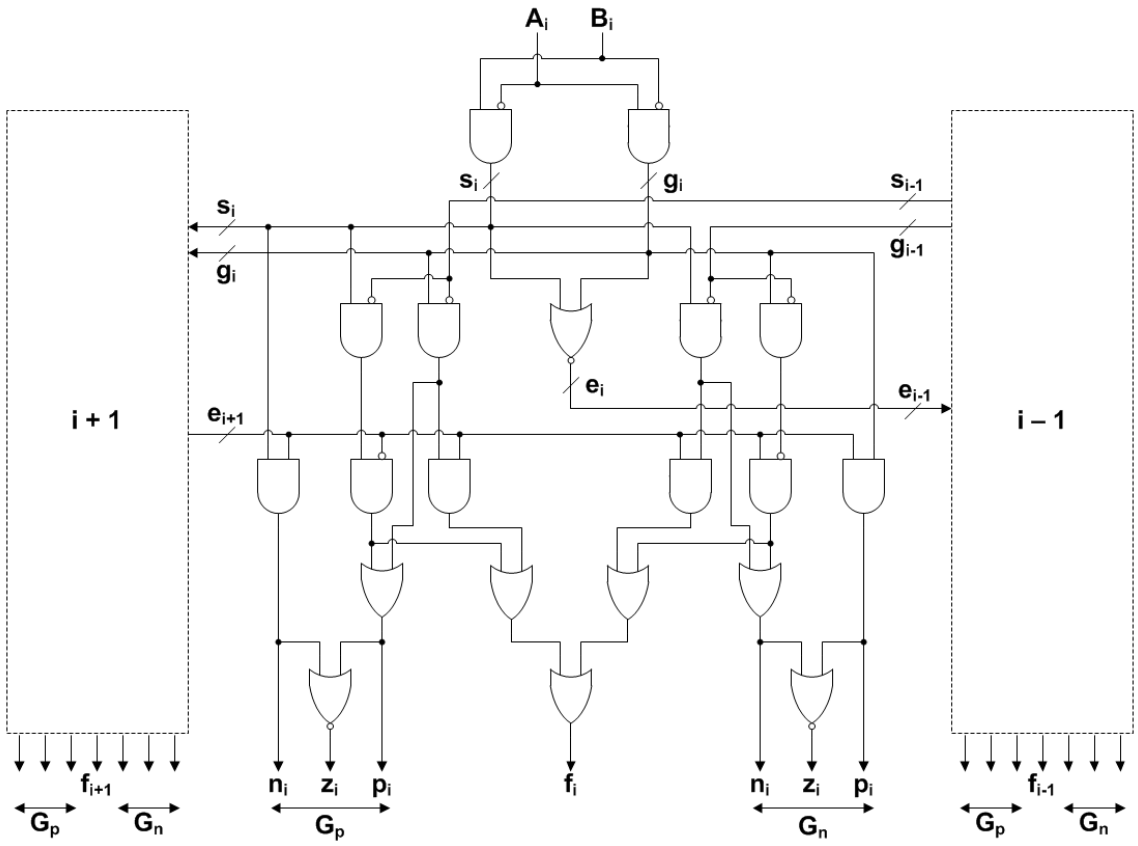

Figure 18. Pre-Encoding Logic of the LZD and Concurrent Correction [16]

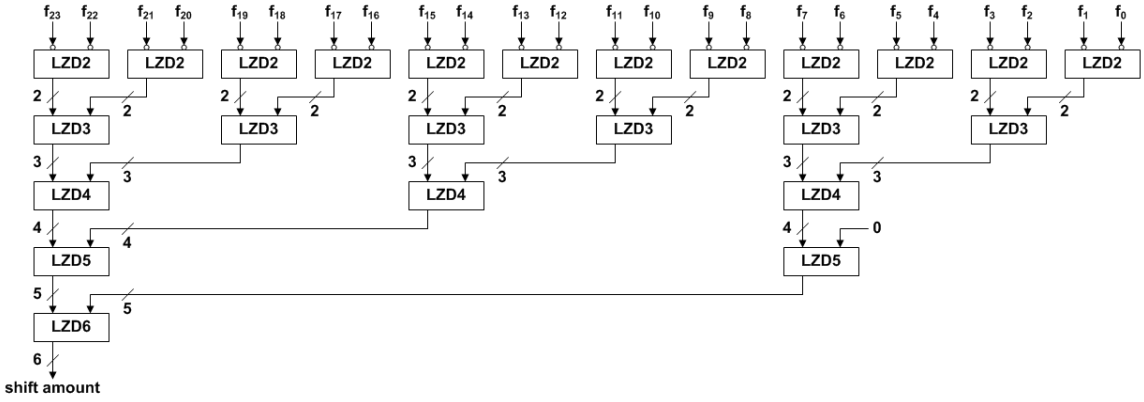

Figure 19. Leading-Zero Detection (LZD) Tree (After [18])

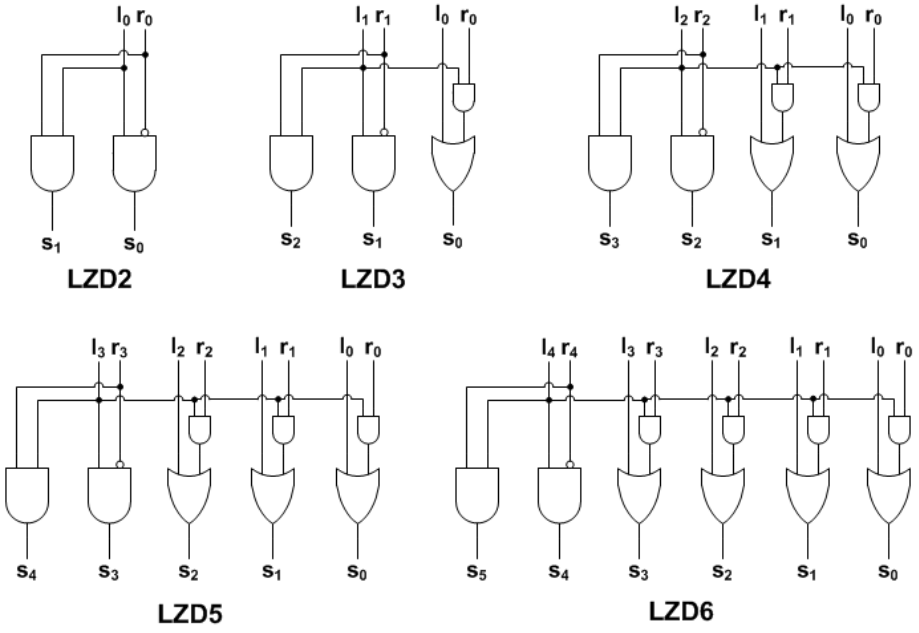

Figure 20. Leading-Zero Detection (LZD) Tree Nodes (After [18])

The concurrent correction logic consists of positive and negative correction trees. The two trees search the patterns that are required to be corrected for two cases: 1) Positive (A > B) and 2) Negative (A < B), respectively. The two trees take pre-encoded G_p(Z, P, N) and G_n(Z, P, N), respectively and each tree generates (Z, P, N, Y) using the binary tree algorithm. The pattern search logic equations are:

For positive,

$$Z = Z^l Z^r \quad P = Z^l P^r + P^l Z^r \quad N = N^l + Z^l N^r \quad Y = Y^l + Z^l Y^r + P^l N^r$$

For negative,

$$Z = Z^l Z^r \quad P = P^r + Z^l P^r \quad N = Z^l N^r + N^l Z^r \quad Y = Y^l + Z^l Y^r + N^l P^r$$

where (Z^l, P^l, N^l, Y^l) and (Z^r, P^r, N^r, Y^r) represent the left and right input from the previous tree node, respectively. *Y* indicates the incorrect pattern and the correct bit is set, if an incorrect pattern is found from one of the two trees. Figure 21 shows the concurrent correction logic and Figure 22 shows the two correction tree nodes.

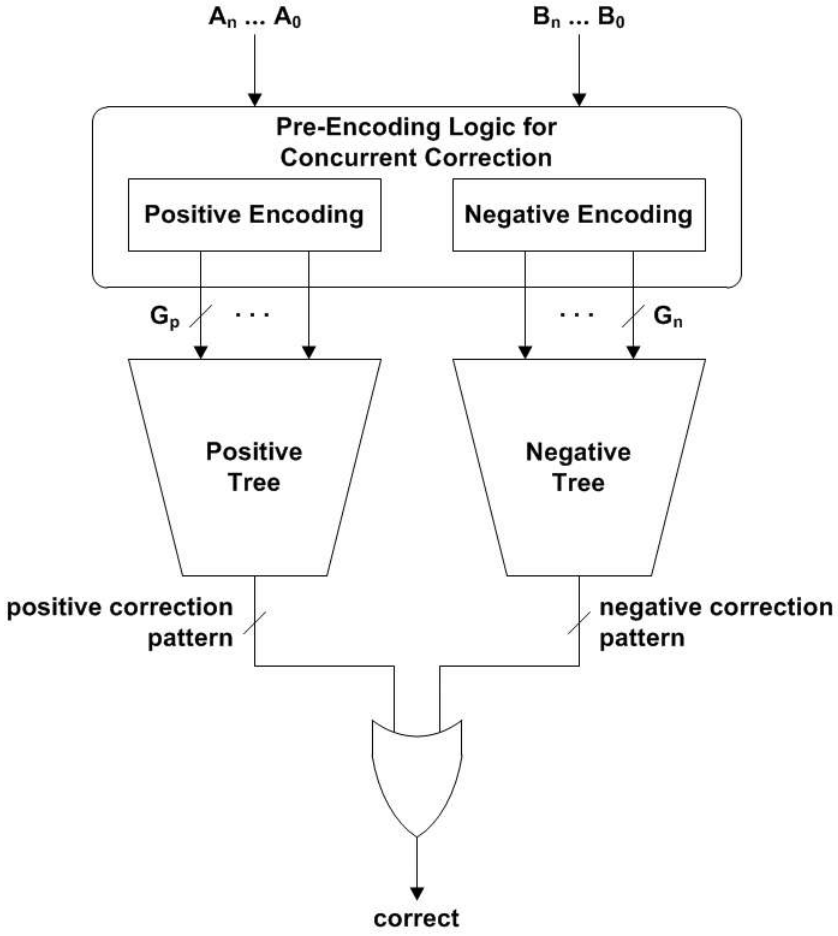

Figure 21. Concurrent Correction Logic [15]

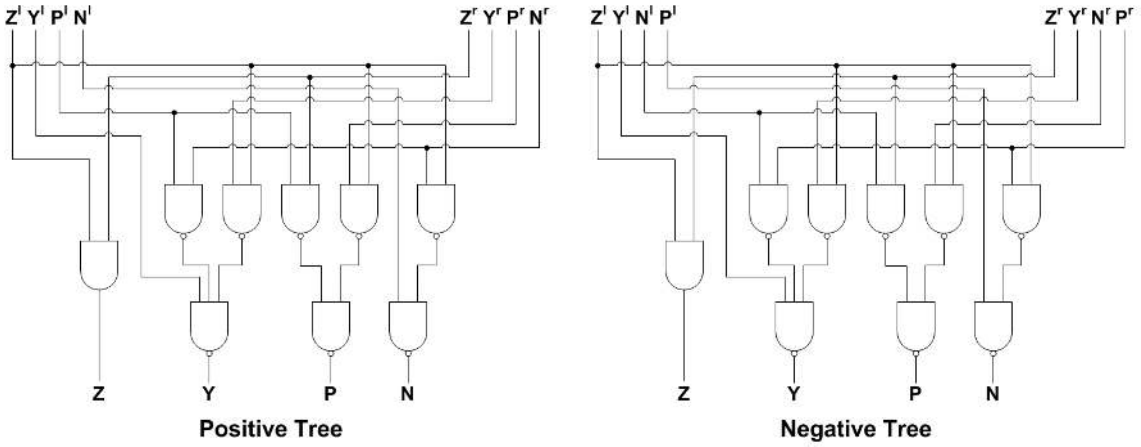

Figure 22. Positive and Negative Correction Tree Nodes [15]

4.2.5 Exponent Adjust Logic

The exponent adjust logic shown in Figure 23 performs addition and subtraction to adjust the exponents by the amount that the significands are shifted. The exponent adjust logic produces two exponent results simultaneously. In the case of addition, one of the increment values is added depending on the path decision that is the overflow from the significand addition. In the case of subtraction, if the far path is selected, the decrement value is subtracted that is underflow from the significand subtraction. If the close path is selected, the normalization shift value is subtracted that is the shift amount of the massive cancellation that occurred during the subtraction.

The two adjusted exponents are passed to the exception logic. The exception logic checks three exception cases specified in IEEE-754 Standard [1]:

$$overflow = \begin{cases} 1 & \text{if } exp[7:0] \geq \text{8xFF} \\ 0 & \text{otherwise} \end{cases}$$

$$underflow = \begin{cases} 1 & \text{if } exp[7:0] \leq 8\text{x}00 \\ 0 & \text{otherwise} \end{cases}$$

$$inexact = round\_up \,||\, overflow \,||\, underflow$$

where *round_up* is the rounding decision of the significand result. The overflow flag is set if the exponent exceeds the maximum value that can be represented such as positive and negative infinity. The underflow flag is set if the exponent is too small to be represented and inexact such as zero and subnormal values. Overflow only occurs in addition and underflow only occurs in subtraction [18]. The inexact flag is set if the rounded significand result is not exact, which is the case if either of the rounding bit, overflow flag or underflow flag is set.

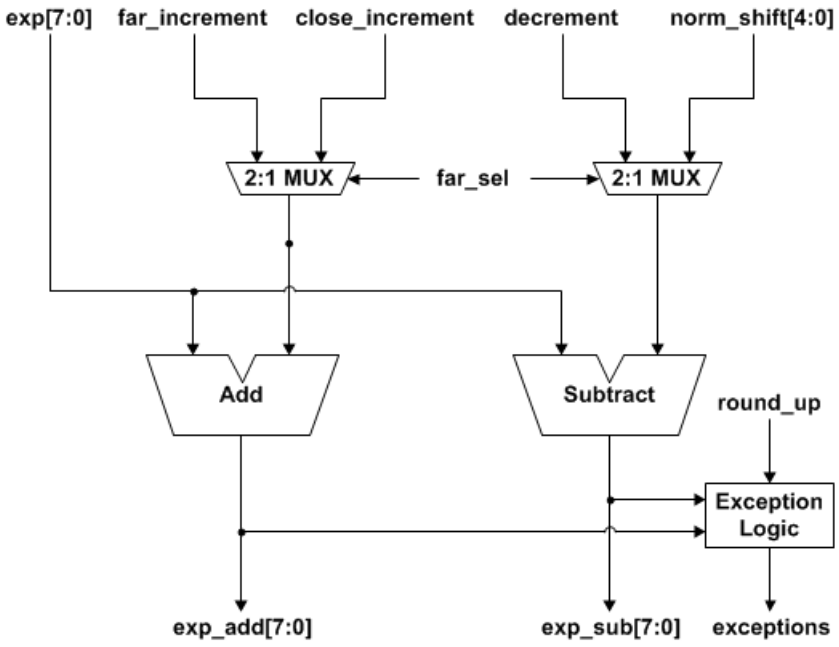

Figure 23. Exponent Adjust Logic

4.3 Implementation and Results

The dual path fused floating-point add-subtract unit for both single and double precision are implemented in Verilog-HDL and synthesized with the Nangate 45nm technology standard cell library. To verify the benefits of the dual path design, the logic area, critical path latency, throughput and power consumption are compared with discrete parallel design and improved fused design, those are described in the previous section. Table 5 shows the comparison of the implementation results. All the percentages in table are ratios compared to the parallel discrete design.

Table 5. Dual Path Fused Floating-Point Add-Subtract Implementation Results

Single Precision			
	Discrete	**Fused**	**Dual Path**
Area (μm^2)	15,403	8,908 (58%)	11,342 (74%)
Latency (ns)	1.32	1.21 (92%)	0.92 (70%)
Throughput (1/ns)	0.76	0.83 (109%)	1.09 (144%)
Power (mW)	7.77	4.21 (54%)	4.91 (63%)
Double Precision			
	Discrete	**Fused**	**Dual Path**
Area (μm^2)	34,606	18,534 (54%)	23,430 (68%)
Latency (ns)	1.66	1.52 (92%)	1.12 (68%)
Throughput (1/ns)	0.60	0.66 (109%)	0.89 (148%)
Power (mW)	15.46	8.17 (53%)	9.03 (59%)

Since the dual path design executes two independent logic components including four additions and subtractions, it requires more area and power consumption compared to the other two designs. However, the dual path design skips the normalization step in the far path, the rounding step in the close path, respectively and minimizes the exponent

comparison and significand alignment steps in the close path. As a result, the dual path fused floating-point add-subtract unit reduces the critical path latency by 30% compared to the parallel discrete floating-point add-subtract unit. By combining the fused operation and the dual path design, the proposed fused floating-point add-subtract unit achieves low area, low power consumption and high speed.

As mentioned in the previous section, the double precision implementation requires about twice as much area and power consumption as the single precision implementation due to the larger addition and subtraction. Since the addition and subtraction logic using the parallel prefix form [12] logarithmically increases the latency, the latency for the double precision increases by only 20%. The benefits of the fused design are shown in both single and double precision. By using the dual path design, the double precision implementation is about 5% better for the area and power consumption and 2% better for the latency than the single precision implementation.

Chapter 5: A Pipelined Fused Floating-Point Add-Subtract Unit

As well known, proper pipelining increases the throughput of the floating-point adders [13], [14], [20]. Those floating-point adders are split into two or three pipeline stages so that the results are produced every cycle. In the pipelined logic, all the stage latencies are set to the slowest stage latency. If the stage latencies are not well balanced, a fast stage must wait until the other stages are completed, which increases the total logic delay. Therefore, it is important to properly arrange the logic components so that the latencies the stages are well balanced. This chapter presents the data flow analysis to arrange the logic components and the composition of each pipeline stage.

5.1 Data Flow Analysis

To achieve a proper pipelined fused floating-point add-subtract unit, the latencies of the components in the proposed design are investigated. Each component is implemented in Verilog-HDL and synthesized with the Nangate 45nm technology standard-cell library. The latencies of the various elements of the single precision floating-point add-subtract unit are listed in Table 6.

Since several components are executed in parallel, they are combined to a stage and the sum of the component delays determines the latency of the stage. Considering the latencies of components and their parallel execution, the proposed design is split into two pipeline stages. Each pipeline stage is executed every cycle so that the largest latency determines the throughput of the design. Figure 24 shows the data flow, the latency of each component, and the critical path.

Table 6. Component Latencies in the Fused Floating-Point Add-Subtract Unit

Components	Latency (ns)	Components	Latency (ns)
Unpack	0.02	Small Exp Comp	0.09
Exponent Compare	0.19	Small Signif Align	0.14
Significand Swap	0.09	Add x 3	0.27
Sign Logic 1	0.06	Subtract x 3	0.29
Align & Sticky	0.16	LZA x 3	0.23
Add	0.23	3:1 Select	0.07
Subtract	0.25	Complement	0.12
Round	0.16	Normalization	0.14
Round Select	0.04	Path Select	0.04
Sign Logic 2	0.06	Exponent Adjust	0.11
Operation Select	0.04		

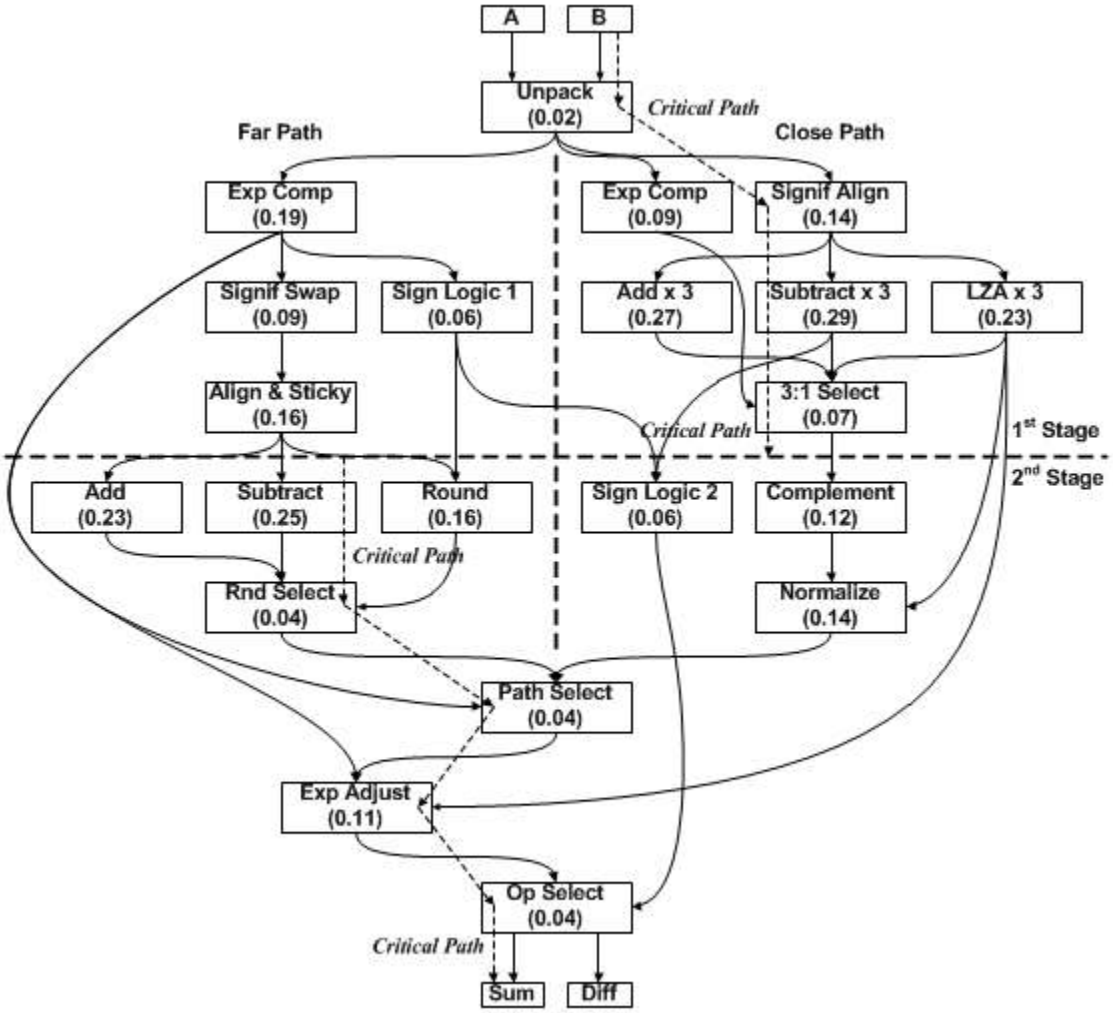

Figure 24. Data Flow of the Pipelined Fused Floating-Point Add-Subtract Unit

5.2 Pipeline Stages of a Dual Path Fused Floating-Point Add-Subtract Unit

According to the data flow analysis, the proposed fused floating-point add-subtract unit is split into two pipeline stages. The critical paths of the two pipeline stages are:

First stage: Unpack → Small Significand Align → Close Path Significand Subtraction → 3:1 Select

Second stage: Far Path Significand Subtraction → Round Select → Path Select → Exponent Adjust → Operation Select

5.2.1 The First Pipeline Stage

The first pipeline stage consists of unpacking logic and the two data paths: the far path and the close path. The two data paths are the first half of the dual path, which is described in Figures 9 and 11. The far path in the first pipeline stage contains the exponent compare, sign logic 1, significand swap, align and sticky logic. The close path in the first pipeline stage contains the small exponent compare, small significand align, three additions, subtractions and LZAs, and 3:1 select logic. Among the two data paths, the close path takes the larger latency so that it becomes the critical path. The series of components in the close path determines the latency of the first pipeline stage, which is 0.52ns.

5.2.2 The First Pipeline Stage

The second half of the dual path and the remaining logic comprise the second pipeline stage. The far path in the second pipeline stage contains the addition, subtraction, round logic and round select logic. The close path in the second pipeline stage contains the sign logic 2, complement and normalization logic. Among the two data paths, the far path takes the larger latency so that the second half of the far path logic and the remaining logic (path select, exponent adjust and operation select logic) comprise the critical path, which adds up to 0.48ns. The latencies of the two pipeline stages are well balanced so that the throughput of the design is increased. Since the latency of the first pipeline stage is slightly larger than that of the second pipeline stage, it determines the throughput of the entire design.

5.3 Implementation and Results

Previous sections introduced four fused floating-point add-subtract designs: 1) Discrete design, 2) Improve fused design, 3) Dual path design, and 4) Pipelined design. Each design is implemented for both single and double precision are implemented in Verilog-HDL and synthesized with the Nangate 45nm technology standard cell library. The logic area, critical path latency, throughput and, power consumption of all four implementations are compared in Table 7.

The proposed pipelined fused floating-point add-subtract unit contains two stages. Each stage requires latches since many data and control signals are passed from the first stage to the next. The area, latency, throughput and power consumption of each pipeline

stage are given in Table 8. The latencies of the pipeline stages are well balanced so that the throughput is increased. Although the latches and control signals in pipeline stages increase the total area, latency and power consumption, the throughput is increased by more than 80% compared to the non-pipelined implementation.

Table 7. Pipeline Stage Comparison

Single Precision			
	Area (μm^2)	Latency (ns)	Power (mW)
Stage 1	7,852 (58%)	0.52 (52%)	2.94 (56%)
Stage 2	5,635 (42%)	0.48 (48%)	2.28 (44%)
Double Precision			
	Area (μm^2)	Latency (ns)	Power (mW)
Stage 1	16,028 (58%)	0.64 (52%)	5.95 (56%)
Stage 2	11,557 (42%)	0.58 (48%)	4.63 (44%)

Table 8. Fused Floating-Point Add-Subtract Design Comparison

Single Precision				
	Discrete	Fused	Dual Path	Pipeline
Area (μm^2)	15,403	8,908 (58%)	11,342 (74%)	13,497 (88%)
Latency (ns)	1.32	1.21 (92%)	0.92 (70%)	1.00 (76%)
Throughput (1/ns)	0.76	0.83 (109%)	1.09 (144%)	1.92 (254%)
Power (mW)	7.77	4.21 (54%)	4.91 (63%)	5.22 (67%)
Double Precision				
	Discrete	Fused	Dual Path	Pipeline
Area (μm^2)	34,606	18,534 (54%)	23,430 (68%)	27,586 (80%)
Latency (ns)	1.66	1.52 (92%)	1.12 (68%)	1.22 (74%)
Throughput (1/ns)	0.60	0.66 (109%)	0.89 (148%)	1.56 (259%)
Power (mW)	15.46	8.17 (53%)	9.03 (59%)	10.58 (68%)

Chapter 6: Conclusions and Future Work

This chapter presents the conclusions on the improved architectures for a fused floating-point add-subtract unit and summaries the implementation results and comparison. Finally, the chapter ends with suggested future work on the design and implementation of fused floating-point arithmetic.

6.1 Conclusions

The floating-point add-subtract unit is useful for digital signal processing applications such as FFT and DCT butterfly operations. This report presents improved architectures which apply the dual path algorithm and pipelining to the fused floating-point add-subtract unit and compares the area, latency, throughput and power consumption with the traditional parallel implementation.

The fused floating-point add-subtract unit saves more than 40% of area and power consumption compared to the traditional discrete floating-point add-subtract unit by sharing the common logic. Also, the fused floating-point add-subtract unit reduces the latency due to its simplified control logic. The dual path fused floating-point add-subtract unit reduces the latency by 30% by performing several add-subtract operations for each case in parallel. With those two techniques, the proposed fused floating-point add-subtract unit achieves low area, low power and high speed. Additionally, a pipelined implementation to increase the throughput of the proposed fused floating-point add-subtract unit is described. It is split into two pipeline stages and the latencies are well

balanced so that the throughput is increased by about 80% compared to the non-pipelined implementation.

6.2 Future Work

The proposed fused floating-point add-subtract unit shows low area, low power and high performance. The improved architectures are expected to contribute to the DSP applications such as DCT and FFT. The implementation of those DSP applications with the new fused floating-point add-subtract unit is an interesting research topic. It involves the improved architectures to specific applications and verifying the advantages.

The floating-point architectures that are used in this report (i.e., multiple path design and pipelining) can be applied to other fused floating-point arithmetic operations such as fused floating-point add-multiply and dot-product. Combining fused operations and those improved architectures would reduce the area, power consumption and improve the speed.

Bibliography

[1] *IEEE Standard for Floating-Point Arithmetic*, ANSI/IEEE Standard 754-2008, New York: IEEE, Inc., Aug. 29, 2008.

[2] R. K. Montoye, E. Hokenek, and S. L. Runyon, "Design of the IBM RISC System/6000 Floating-Point Execution Unit," *IBM Journal of Research & Development*, Vol. 34. pp. 59-70. 1990.

[3] E. Hokenek, R. K. Montoye and P. W. Cook, "Second-Generation RISC Floating Point with Multiply-Add Fused," *IEEE Journal of Solid-State Circuits*, vol. 25, pp. 1207-1213, 1990.

[4] T. Lang and J. D. Bruguera, "Floating-Point Fused Multiply-Add with Reduced Latency," *IEEE Transactions on Computers*, Vol. 53, pp. 988-1003, 2004.

[5] H. H. Saleh and E. E. Swartzlander, Jr., "A Floating-Point Fused Add-Subtract Unit," *Proceedings of the 51st IEEE Midwest Symposium on Circuits and Systems*, 2008, pp. 519 - 522.

[6] H. H. Saleh and E. E. Swartzlander, Jr., "A Floating-Point Fused Dot- Product Unit," *Proceedings of the IEEE International Conference on Computer Design*, pp. 427-431, 2008.

[7] E. E. Swartzlander, Jr. and H. H. Saleh, "FFT Implementation with Fused Floating-Point Operations," *IEEE Transactions on Computers*, in press.

[8] E. E. Swartzlander, Jr. and H. H. Saleh, "Fused Floating-Point Arithmetic for DSP," *Proceedings of the 42nd Asilomar Conference on Signals, Systems and Computers*, 2008.

[9] M. P. Farmwald, *On the Design of High Performance Digital Arithmetic Units*, Ph.D. dissertation, Dept. Computer Science, Stanford University, 1981.

[10] N. Quach and M. Flynn, *Design and Implementation of the SNAP Floating-Point Adder.* Technical Report CSL-TR-91-501, Stanford University, 1991.

[11] E. Hokenek and R. Montoye, "Leading-Zero Anticipator (LZA) in the IBM RISC System/6000 Floating-Point Execution Unit," *IBM Journal Research and Development*, vol. 34, pp. 71–77, 1990.

[12] P. M. Kogge and H. S. Stone, "A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations," *IEEE Transactions on Computers*, vol. C-22, pp. 786 – 793, 1973.

[13] A. Beaumont-Smith, N. Burgess, S. Lefrere, and C. Lim, "Reduced Latency IEEE Floating-Point Standard Adder Architectures," *Proceedings of the 14th IEEE Symposium on Computer Arithmetic*, pp. 35-43, 1999.

[14] P. M. Seidel and G. Even. "Delay-Optimized Implementation of IEEE Floating-Point Addition," *IEEE Transactions on Computers,* vol. 53, pp. 97–113, 2004.

[15] J. D. Bruguera and T. Lang. "Leading–One Prediction with Concurrent Position Correction," *IEEE Transactions on Computers*, vol. 48, pp. 1083–1097, 1999.

[16] R. Ji, Z. Ling, X. Zeng, B. Sui, L. Chen, J. Zhang, Y. Feng, and G. Luo, Comments on "Leading One Prediction with Concurrent Position Correction," *IEEE Transactions on Computers*, vol. 58, pp. 1726–1727, 2009.

[17] G. Dimitrakopoulos and D. Nikolos, "High-Speed Parallel-Prefix VLSI Ling Adders," *IEEE Transactions on Computers,* vol. 54, pp. 225–231, 2005.

[18] V. G. Oklobdzija, "An Algorithmic and Novel Design of a Leading Zero Detector Circuit Comparison with Logic Synthesis," *IEEE Transactions on VLSI Systems,* vol. 2, pp. 124–128, 2005.

[19] X. Hong, W. Chongyang, and Y, Jiangyu, "Analysis and Research of Floating-Point Exceptions," *Proceedings of the 2nd International Conference on Information Science and Engineering*, pp. 1851–1854, 2010.

[20] A. Nielsen, D. Matula, C.-N. Lyu, and G. Even, "IEEE Compliant Floating-Point Adder that Conforms with the Pipelined Packet- Forwarding Paradigm," *IEEE Transactions on Computers,* vol. 49, pp. 33-47, 2000.

Vita

Jongwook Sohn was born in Seoul, Republic of Korea on December, 4th 1982. He received a Bachelor of Science degree in Electrical Engineering from Korea University, Seoul, Republic of Korea in 2009. He entered the Graduate School of the University of Texas at Austin in 2009. He is working on high-speed computer arithmetic and application specific processor with his supervisor, Prof. Earl E. Swartzlander, Jr. While he is continuing his studies in the Graduate School of the University of Texas at Austin, he began to work as an engineer in Atom and SOC Development Group at Intel Corporation, Austin, Texas in 2011.

Permanent address: 11500 Jollyville Rd. Apt 2313, Austin, TX 78759

This report was typed by the author.