The Dissertation Committee for Jongwook Sohn

certifies that this is the approved version of the following dissertation:

Improved Architectures for Fused Floating-Point Arithmetic Units

Committee:

Earl E. Swartzlander, Jr., Supervisor

Lizy K. John

Andreas Gerstlauer

Nur A. Touba

Michael J. Schulte

Improved Architectures for Fused Floating-Point Arithmetic Units

by

Jongwook Sohn, B.S.E.; M.S.E.

Dissertation

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

Doctor of Philosophy

The University of Texas at Austin

May 2013

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 my committee members, Professor Lizy K. John, Professor Andreas Gerstlauer, Professor Nur A. Touba and Dr. Michael J. Schulte for their valuable advices and helpful suggestions.

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 – Jae Wook Lee, Suk-joon Hong, Bong Wan Jun, Kyoungtae Lee, Huesung Kim, Sunghyun Koh, Sang Y Lee, Jongyoon Choi, Hangkyu Lee, Hyun-Sun Um, Jason H Doh, Dongwoon Kim, Doochul Shin, Joonsoo Kim, Joon-Sung Yang, Junyoung Park and Keytaek Lee. I also would like to thank my friends – Sanghyun Chi, Hyungman Park, Seonpil Jang, Dam Sunwoo, Ikhwan Lee, Sangmin Lee, Jung-Su Lee, Seungyun Nam, Eunho Yang, Jae Hong Min, Sungpil Yang, Min Kyu Jeong, Jinsuk Chung, Minsoo Rhu, Dongwook Lee, Youngkyu Lee, Jaehyun Ahn, Changhyuk Kim, Jinhan Kwon and Donghyi Koh 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

May 2013

Improved Architectures for Fused Floating-Point Arithmetic Units

Jongwook Sohn, Ph.D.

The University of Texas at Austin, 2013

Supervisor: Earl E. Swartzlander, Jr.

Most general purpose processors (GPP) and application specific processors (ASP) use the floating-point arithmetic due to its wide and precise number system. However, the floating-point operations require complex processes such as alignment, normalization and rounding. To reduce the overhead, fused floating-point arithmetic units are introduced. In this dissertation, improved architectures for three fused floating-point arithmetic units are proposed: 1) Fused floating-point add-subtract unit, 2) Fused floating-point two-term dot product unit, and 3) Fused floating-point three-term adder. Also, the three fused floating-point units are implemented for both single and double precision and evaluated in terms of the area, power consumption, latency and throughput.

To improve the performance of the fused floating-point add-subtract unit, a new alignment scheme, fast rounding, two dual-path algorithms and pipelining are applied. The improved fused floating-point two-term dot product unit applies several optimizations: a new alignment scheme, early normalization and fast rounding, four-input leading zero anticipation (LZA), dual-path algorithm and pipelining. The proposed fused floating-point three-term adder applies a new exponent compare and significand alignment scheme, double reduction, early normalization and fast rounding, three-input LZA and pipelining to improve the performance.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

This chapter presents the motivation of the research on the fused floating-point arithmetic units. Then, the approach and methodology of the research and a brief overview of the dissertation are presented.

1.1 Motivation

The computer arithmetic units in modern microprocessors 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 extremely huge numbers. The floating-point numbers consist of three parts (sign, exponent and significand) so that the operations require complex procedures. For example, the operations frequently require the normalization, which causes an increased logic delay. Therefore, improving the performance of 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 units have been introduced: Fused floating-point multiply-add unit [2] – [5], fused

floating-point add-subtract unit [6], [7], fused floating-point two-term dot product unit [7] [8], and fused floating-point three-term adder [9], [10]. The fused floating-point operations not only improve the performance, but also reduce the area and power consumption compared to a combination of traditional floating-point units. This dissertation presents improved architecture designs and implementations for the fused floating-point units. Many digital signal processing (DSP) applications such as fast Fourier transform (FFT) and discrete cosine transform (DCT) butterfly operations have been developed to utilize the fused floating-point units [7], [11], [12]. Therefore, the improved fused floating-point units will contribute to the next generation floating-point arithmetic and DSP application development.

1.2 Approach and Methodology

In this dissertation, several fused floating-point units are investigated to improve the performance as well as reduce the area and power consumption. To design and implement the improved fused floating-point units, several optimization techniques are applied. For the fused floating-point add-subtract unit, a new alignment and fast rounding scheme, dual-path algorithm and pipelining are applied. The fused floating-point two-term dot product unit applies the optimizations: a new alignment scheme, early normalization and fast rounding, four-input leading zero anticipation (LZA), dual-path algorithm and pipelining. The fused floating-point three-term adder applies the optimizations: a new exponent compare and significand alignment scheme, double

reduction, early normalization and fast rounding, three-input LZA and pipelining. Although the basic concepts of the optimizations for all three fused floating-point units are similar, the designs and implementation details to apply those optimizations to each fused floating-point unit are different from each other.

To design and implement the improved fused floating-point units, a general VLSI circuit design and implementation methodology is used. Figure 1 shows the general research flow for the VLSI circuit design and implementation. The detailed steps for the VLSI circuit design and implementation are

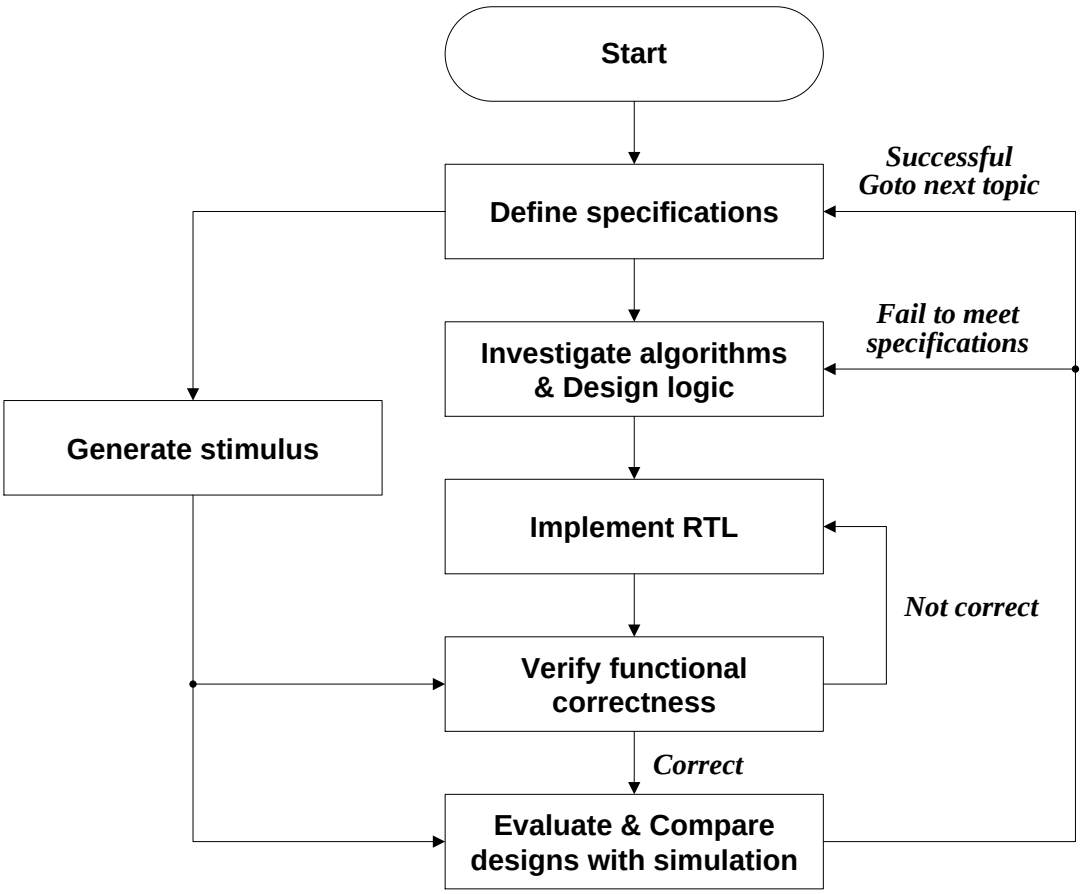


Figure 1. General VLSI Circuit Design and Implementation Flow.

1) Define the specifications: The design specification indicates the high level design concept, the purpose of the design, and the goal of the design such as target frequency, area and power consumption.
2) Investigate the algorithms and design the logic: To achieve the target specifications, various algorithms are investigated such as fast rounding, dual path and pipelining. Based on the selected algorithms, the basic logic is designed with pseudo code and logic diagrams.
3) Implement RTL: Once the logic design is completed, RTL is implemented with Verilog-HDL. In this step, all the design rules and corner cases must be considered to achieve the functionally correct circuit.
4) Verify functional correctness: To verify the functional correctness, test vectors are generated based on the specifications. The test vectors include all the functions and corner cases such as special inputs and the cases of equal exponents. The test vectors feed the inputs to the RTL implementation and compare the output with the expected results, which are stored in advance. ModelSim is used for the functional verification. If all the functionalities are correct, go to next step; otherwise, go back to the step 3) to correct the RTL implementation.
5) Evaluate and compare the designs with simulation: Once the functional verification is completed, the implementation is evaluated with simulations. The RTL implementation is compiled by Synopsys Design Compiler and synthesized with the 45nm CMOS standard cell library. Using the Design

Compiler report tools, area, power consumption and latency are estimated. The estimated results are compared with the target specifications. If the results meet the specifications, the evaluation is successful; otherwise, go back to step 2) or 3) to improve the design and implementation. The evaluation results are also compared with the other designs to verify how the design and implementation impacts the performance.

1.3 Dissertation Overview

This dissertation is divided into 6 chapters. Chapter 2 provides an introduction to the IEEE-754 floating-point standard and the fundamentals of floating-point arithmetic. Also, traditional floating-point units are introduced as a previous work including the basic floating-point adder and multiplier, multiply-add unit, add-subtract unit, two-term dot product unit and three-term adder. Chapter 3 presents improved architecture designs and implementations for a fused floating-point add-subtract unit. A new alignment scheme, fast rounding, two dual-path algorithms, and pipelining are applied for the improved fused floating-point add-subtract unit. Chapter 4 presents architecture designs and implementations for a fused floating-point two-term dot product unit. A new alignment scheme, early normalization, four-input LZA, dual-path algorithm, and pipelining are applied for the improved fused floating-point two-term dot product unit. Chapter 5 presents the improved architecture designs and implementations for a fused floating-point three-term adder. A new exponent compare and significand alignment

scheme, double reduction, early normalization, three-input LZA and pipelining are applied for the improved fused floating-point three-term adder. Finally, Chapter 6 concludes the dissertation by summarizing the designs and implementation results and suggests several ideas for future work.

Chapter 2

Background

This chapter provides an introduction to the IEEE-754 floating-point standard which governs the fundamentals of the floating-point arithmetic covered in the dissertation. Then, previous work on floating-point units is presented: 1) Floating-point adder/multiplier, 2) Fused floating-point multiply-add unit, 3) Fused floating-point add-subtract unit, 4) Fused floating-point two-term dot product unit, and 5) Fused floating-point three-term adder.

2.1 The IEEE-754 Floating-Point Standard

The IEEE-754 floating-point standard provides a discipline for performing floating-point computation [1]. In this section, an overall introduction to the floating-point standard is presented: 1) Floating-point number system, 2) Rounding modes, 3) Special values, and 4) Exceptions.

2.1.1 Floating-Point Number System

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. Since it is the most commonly used format, only the binary format is covered in this dissertation. 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

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

where

sign = 0 or 1

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

significand = $d_{p-1}d_{p-2} \ldots d_2d_1d_0$ (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

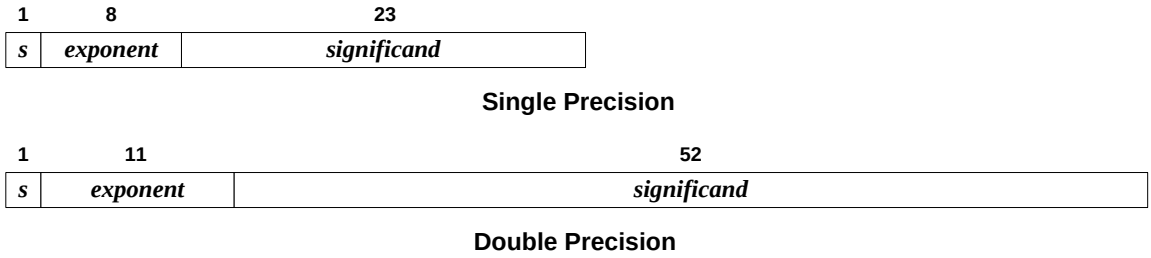


Figure 2. IEEE-754 Floating-Point Single and Double Precision Formats.

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 2 shows the bit partitions for the single and double precision formats. In this dissertation, both single and double precision implementations are covered.

2.1.2 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 near to two numbers (i.e., ties), the number with an even LSB or the number with the larger magnitude is selected, respectively. Generally, the nearest rounding modes

are more precise than the directed rounding modes. The fused floating-point units presented in this dissertation support all five rounding modes.

2.1.3 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 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.

2.1.4 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 ±∞. The overflow flag is set when the result of the operation exceeds the representable range and it returns ±∞. 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. The fused floating-point units presented in this dissertation support the three exception cases: overflow, underflow and inexact.

2.2 Basic Floating-Point Arithmetic Units

The floating-point adder and floating-point multiplier are the most fundamental units in floating-point arithmetic. Most fused floating-point units are designed and implemented based on the basic floating-point units. Therefore, the algorithms and optimization techniques for basic floating-point units can be applied to the fused floating-point units.

2.2.1 Floating-Point Adder

The floating-point adder takes two input operands and produces a rounded sum result. In contrast to fixed-point units, a floating-point adder is more complex than a floating-point multiplier due to the alignment and normalization procedure. Figure 3 shows a basic floating-point adder. The basic floating-point addition is executed as

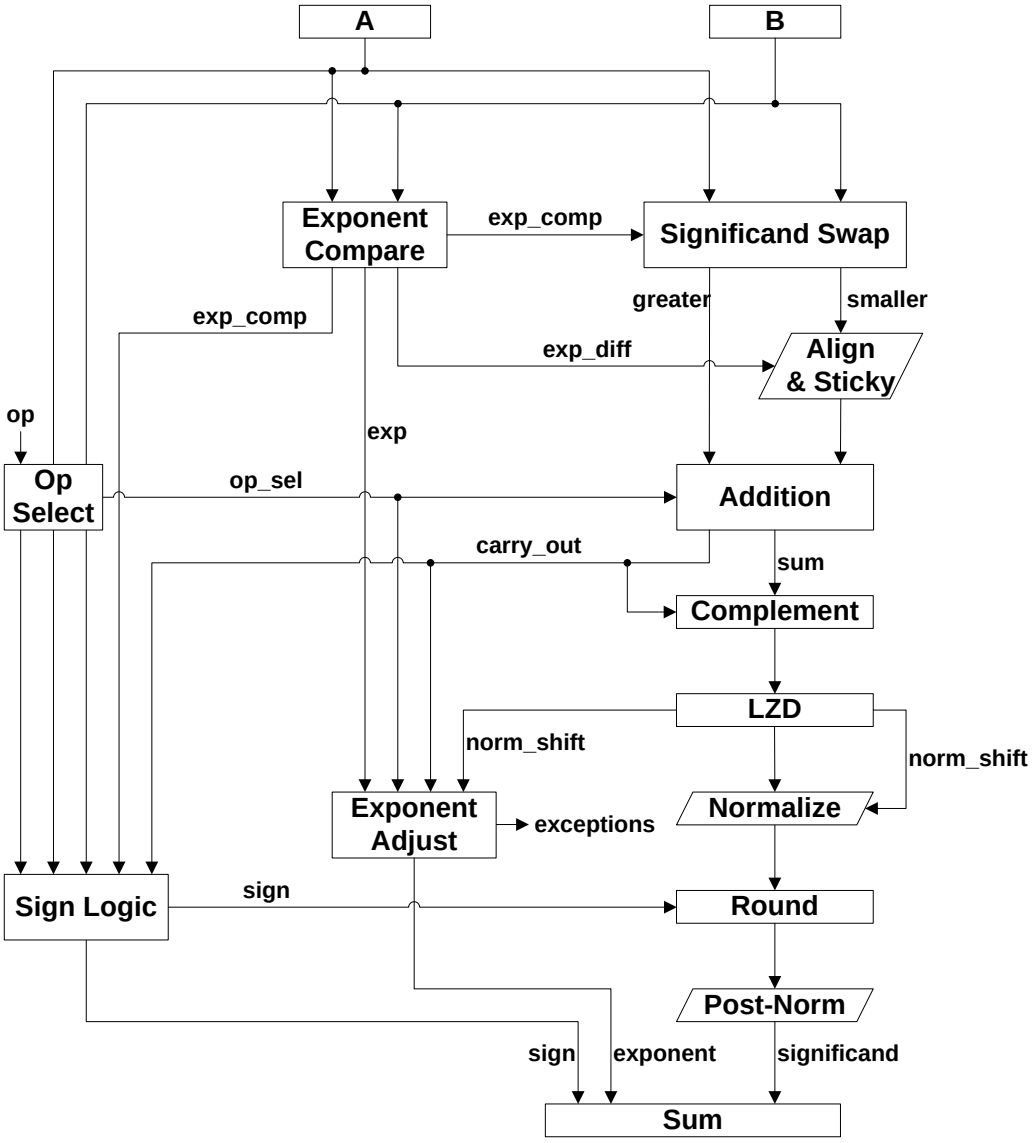


Figure 3. Basic Floating-Point Adder.

1) The exponent compare logic compares the two exponents to determine which is greater. The comparison result and the difference are passed to the significand swap, alignment and sign logic. Also the greater exponent is passed to the exponent adjust logic.

2) The significand swap logic takes the two significands and determines the significand of the greater and smaller operand based on the exponent comparison. The two significands are passed to the alignment and sticky logic. The significand of the smaller operand is shifted by the amount of the exponent difference and the LSBs of the shifted significand are discarded by the sticky logic.
3) The operation selection logic takes the two signs and the op code, and generates the effective operation. The significand addition takes the aligned two significand and operation, and computes the addition or subtraction depending on the operation. If the operation is subtraction and the carry-out is positive, indicating the sum of the significands is negative, the sum is complemented to convert it to a positive number. Since the carry-out indicates the significand comparison in the case of subtraction, it is passed to the sign logic.
4) Leading zero detection (LZD) is performed to determine the position of the MSB and the shift amount that is needed to normalize the significand when cancellation occurs during subtraction. The normalization logic shifts the significand by the amount of the LZD result. The shift amount is also passed to the exponent adjust logic.
5) The exponent adjust logic adjusts the exponent by adding the carry-out of the significand addition or subtracting the shift amount for the normalization depending on the operation. Also, the exponent adjust logic sets the exception

flags (i.e., overflow, underflow and inexact) based on the adjusted exponent. The sign logic takes the two sign bits, op code, exponent comparison and significand comparison, and generates the sign bit of the sum. 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 bit is passed to the round logic.

6) The round logic rounds or truncates the significand sum depending on the rounding modes specified in IEEE-754 Standard [1]. Then, the rounded significand sum is shifted by 1 bit for the post-normalization.

In order to improve the basic procedure, several techniques can be applied: 1) Compound addition and fast rounding [13] – [16], 2) Leading zero anticipation (LZA) for fast normalization [18] – [21], and 3) Dual-path algorithm [14] – [17]. Figure 4 shows a dual-path floating-point adder which applies the three optimizations. The compound significand addition generates a rounded result and an unrounded result simultaneously. The round logic is performed in parallel with the significand addition[1] and it selects an appropriate significand result for fast rounding. The leading zero anticipation (LZA) logic is performed with the significand addition to predict the amount of the cancellation in a constant time so that the significand result is immediately normalized. The dual-path consists of a far path and a close path and the path is selected based on the exponent difference. The far path is selected if the exponent difference is greater than 1. In this

[1] For the significand addition, the Kogge-Stone adder [22], which is one of the fastest prefix adders [23], is used in this dissertation.

case, massive cancellation does not occur during subtraction so that normalization is unnecessary. The close path is selected if the difference of the two exponents is 0 or 1. Since the significands in the close path are shifted by at most 1 bit, the large significand alignment and rounding are not required [24]. The significand alignment and normalization are the bottlenecks of the floating-point adder. Therefore, the dual-path algorithm improves the performance by skipping unnecessary logic in each path.

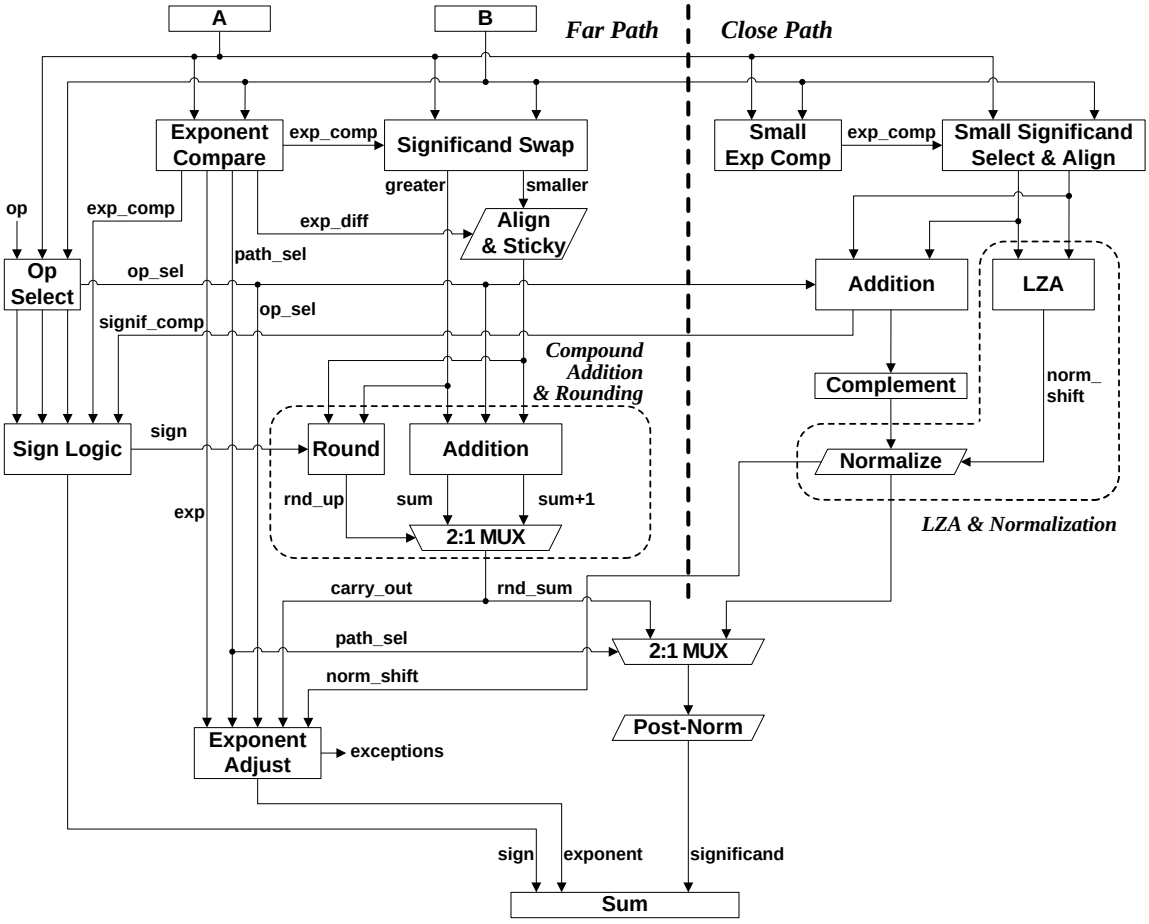


Figure 4. Dual-Path Floating-Point Adder.

2.2.2 Floating-Point Multiplier

The floating-point multiplier takes two input operands and produces a rounded product result. Although the floating-point multiplier is simple in terms of overall structure, it requires more logic area and power consumption compared to the floating-point adder.

Figure 5 shows a floating-point multiplier. The floating-point multiplication is executed as

1) The exponent sum logic generates the sum of the two exponents. The result is passed to the exponent adjust logic.
2) The multiplier tree[2] takes the two significands and performs the reduction tree to generate the sum and carry. The significand pair is aligned to the number of final significand bits including round, guard, and sticky bits to reduce the significand addition.
3) The exponent adjust logic adjusts the exponent by adding the carry-out from the significand addition. Also, the exponent adjust logic sets the exception flags (i.e., overflow, underflow and inexact) based on the adjusted exponent. The sign logic takes the two sign bits and generates the sign bit of the product. The sign bit is passed to the round logic.
4) The compound significand addition produces rounded and unrounded sums simultaneously and the round logic determines the correct result for fast

[2] For the significand multiplication, the simple partial product generation and Dadda tree [25], which is known as the fastest algorithm [26], are used in this dissertation.

rounding. Then, the rounded significand sum is shifted by 1 bit for the post-normalization.

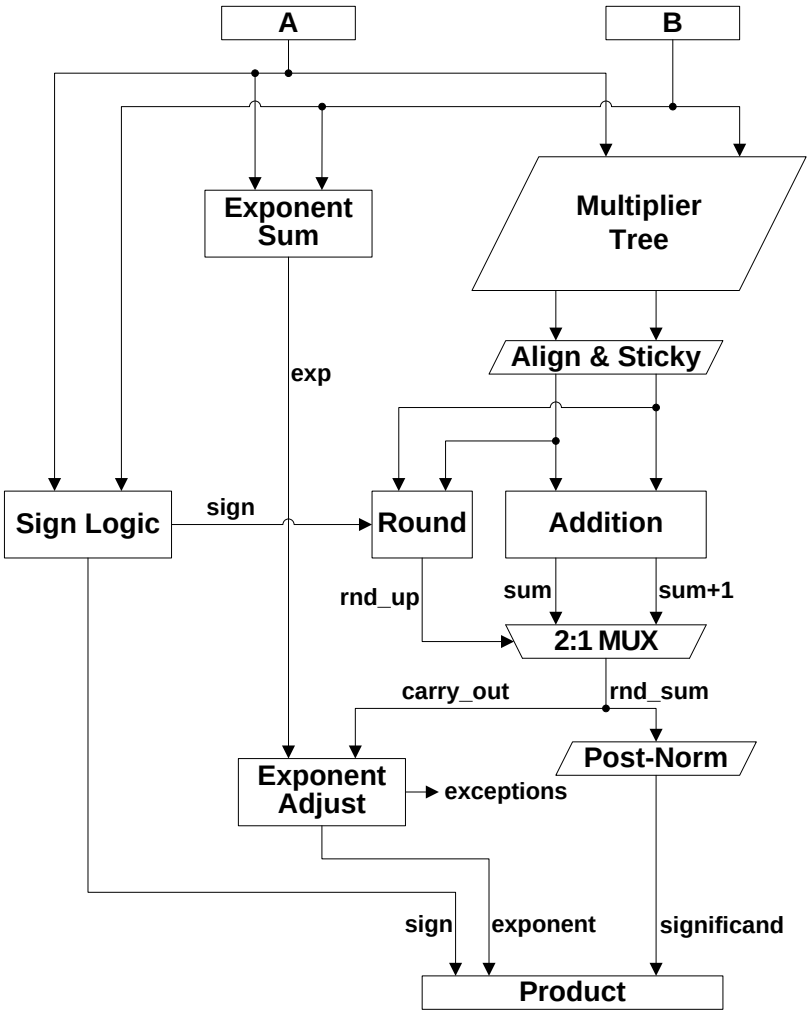


Figure 5. A Floating-Point Multiplier

2.3 Floating-Point Multiply-Add Unit

In 1990, IBM published two papers on the design of a floating-point fused multiply-add unit [2], [3]. The fused multiply-add unit takes three operands A, B, C and produces (A × B) + C, which has the advantages over the discrete multiplier and adder: 1) The logic area and latency is reduced by sharing the logic, 2) The precision is increased by performing the rounding process only one time, and 3) The number of input/output ports, register file and control logic are reduced. In this section, several designs for fused multiply-add units are presented.

2.3.1 Fused Floating-Point Multiply-Add Unit with Reduced Latency

The most significant improvement for the fused multiply-add unit has been achieved by the paper on the design of the floating-point fused multiply-add unit with reduced latency [4]. The paper combines the significand addition and round logic to increase the performance. Although the combination of addition and rounding is widely used for floating-point adders [13] – [16], it requires a more complex process to employ it for the fused multiply-add unit.

In order to perform the addition and rounding simultaneously, the proposed design performs the LZA and normalization prior to the significand addition and the round logic as shown in Figure 6. The three significands are reduced to two by the partial addition and normalized by the shift amount from the LZA. To reduce the delay of the normalization, the LZA generates the shift amount from the MSB so that the LZA logic is overlapped with the normalization shift. The normalized significands are passed to the

dual adder and round logic. The dual adder produces both the rounded and unrounded sums and the round logic selects the correct result. The proposed design is estimated to improve the performance by 15 – 20% compared to the traditional fused multiply-add unit [4].

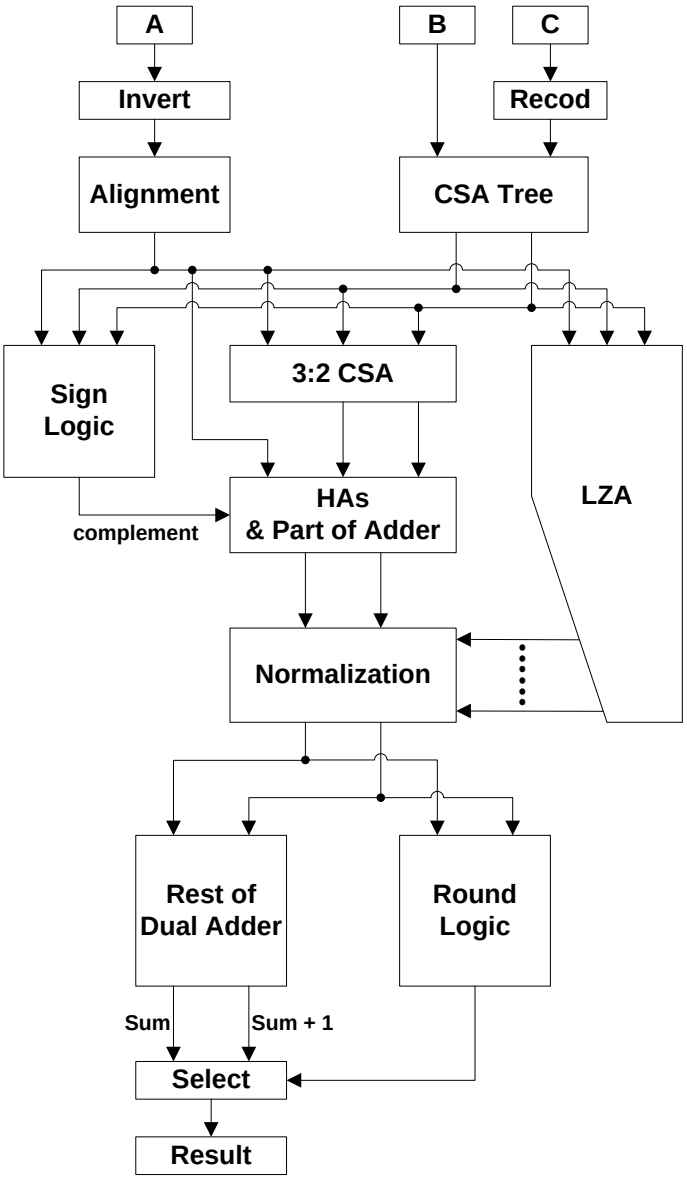


Figure 6. Fused Floating-Point Multiply-Add Unit with Reduced Latency (After [4]).

2.3.2 Dual-Path Fused Floating-Point Multiply-Add Unit with Reduced Latency

Based on the design of the fused floating-point multiply-add unit with reduce latency, a dual-path fused floating-point multiply-add unit is proposed to improve the performance [5]. The dual-path consists of far path and close path logic based on the exponent difference. The close path is selected if the exponent difference is 2, 1, 0 or –1 and the far path is selected for the rest of the cases. In the far path, massive cancellation during the subtraction does not occur so that the large LZA and normalization are unnecessary. In the close path, the exponent difference is small so that a large significand alignment is unnecessary. Since the significand alignment and normalization are the bottlenecks of the floating-point multiply-add unit, the dual-path approach can improve the performance by skipping one of them depending on the path selection. For both paths, the normalization is performed prior to the significand addition so that the significand addition size is reduced and it is performed in parallel with the rounding which is the advantage inherited from the single-path fused floating-point multiply-add unit with reduced latency. Figure 7 shows the dual-path fused multiply-add unit with reduced latency. The proposed dual-path design is estimated to improve the performance by 30% compared to the single-path fused multiply-add unit [5].

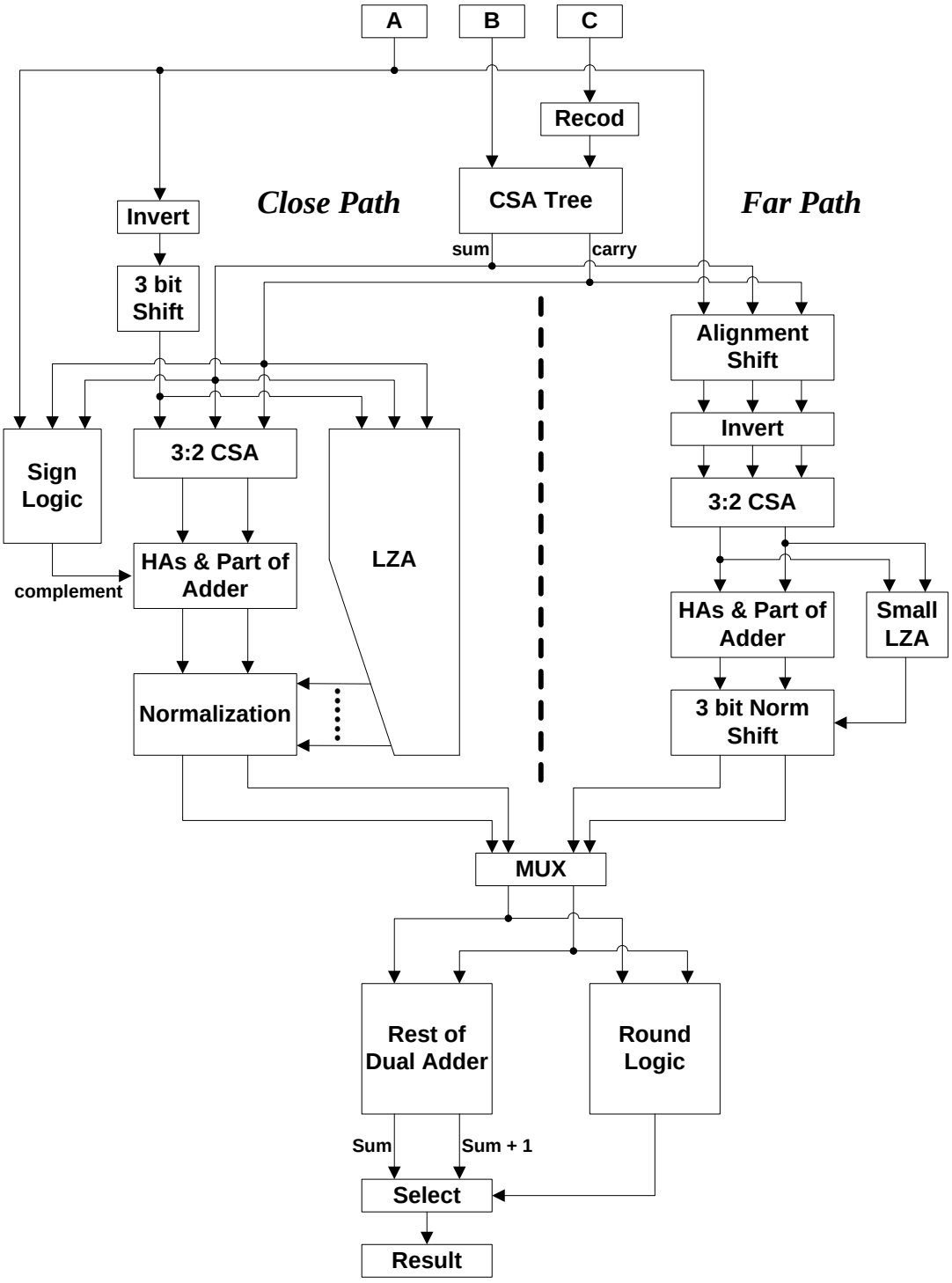


Figure 7. Dual-Path Fused Multiply-Add Unit with Reduced Latency (After [5]).

2.3.3 Three-Path Fused Floating-Point Multiply-Add Unit

The dual-path algorithm significantly improves the performance for the fused floating-point multiply-add unit as described in the previous section. However, its advantage is limited due the large latency of the alignment in the far path logic. In order to increase the performance, the three-path fused floating-point multiply-add unit is proposed [27], [28]. The three-path fused floating-point multiply-add unit splits the data path following the multiplier tree into three paths as shown in Figure 8. The three paths are independently executed and the correct path is selected based on the exponent difference.

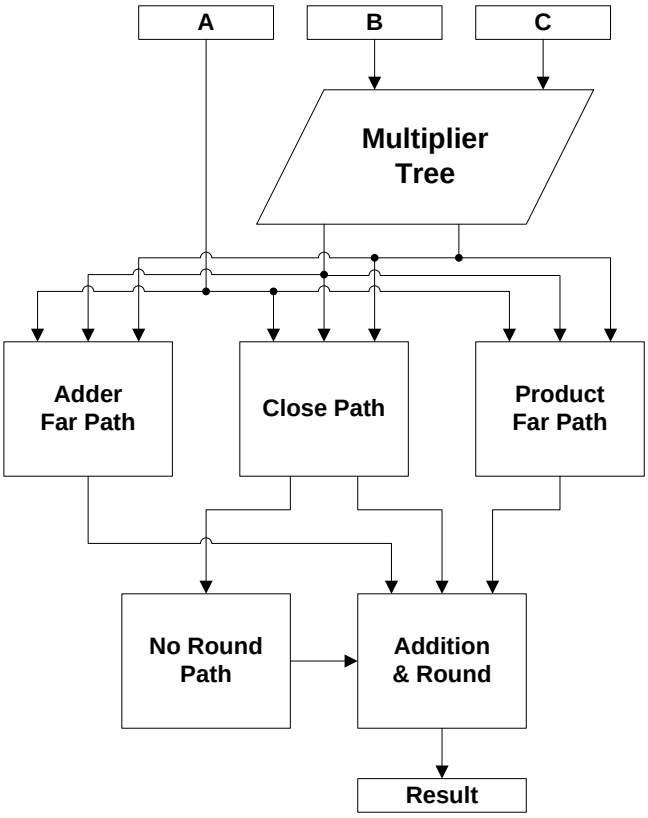


Figure 8. Three-Path Fused Floating-Point Multiply-Add Unit [27].

The three-path fused floating-point multiply-add unit consists of two far paths and a close path. In order to reduce the overhead of the large amount of alignment in the far path, the far path is split into two paths: the adder far path and the product far path. Figures 9 and 10 show the adder and product far path logic, respectively. The adder far path is selected if the exponent difference determines that the addend is larger than the product. In this case, the sum and carry from the multiplier tree are aligned and inverted. The three significands are reduced to two by the 3:2 CSA and normalized for the overflow adjustment. The product far path is selected if the product is larger than the addend. In this case, the addend is aligned and inverted. Similar to the adder far path, the three significands are reduced to two and adjusted.

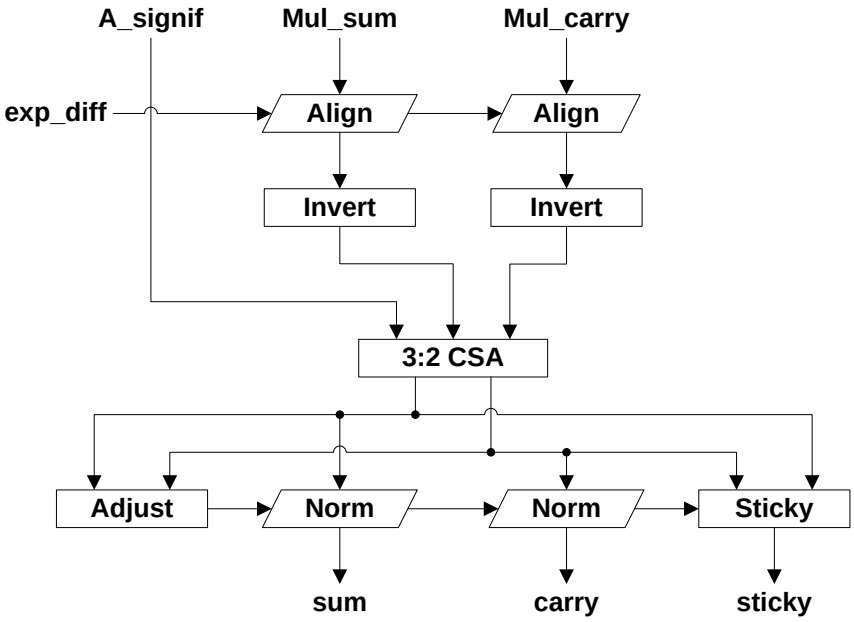


Figure 9. Adder Far Path for a Three-Path Fused Multiply-Add Unit [27].

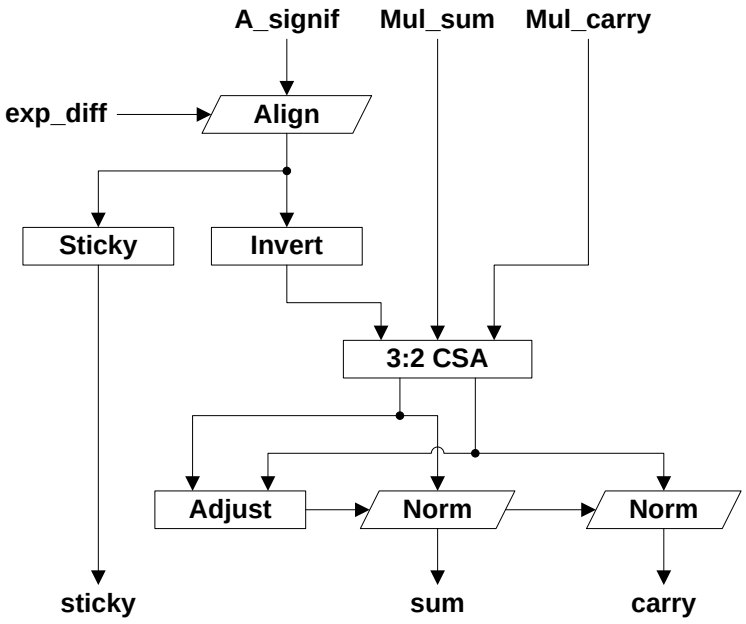


Figure 10. Product Far Path for a Three-Path Fused Multiply-Add Unit [27].

The close path is selected if the exponent difference is small so that massive cancellation may occur during subtraction. Figure 11 shows the close path logic. Since the exponent difference does not detect which is larger, two inversion cases are performed and the correct result is selected after the significand comparison. For fast normalization, LZA predicts the shift amount for the massive cancellation during subtraction. The LSBs of the significands are reserved to be controlled as the no round path, which is used for the post-normalization in the add/round logic.

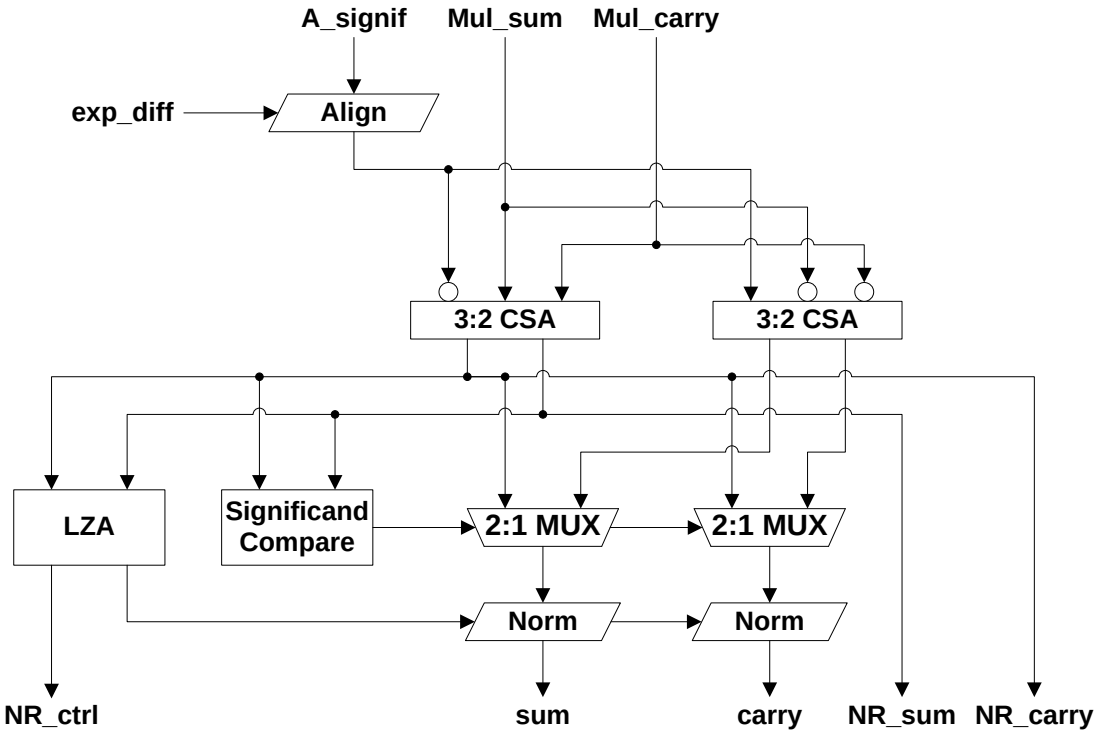


Figure 11. Close Path for a Three-Path Fused Multiply-Add Unit [27].

Among the three paths, a path is selected by path select logic and the significands are passed to the addition and round logic. Figure 12 shows the addition and round logic. The compound adder produces rounded and unrounded sums simultaneously and the round logic selects the correct result. The no round path computes the LSBs of the significands from the close path and the result is used for post-normalization. The three-path fused multiply-add unit reduces the latency and power consumption by 10 – 15% with 40% increased logic area [27], [28].

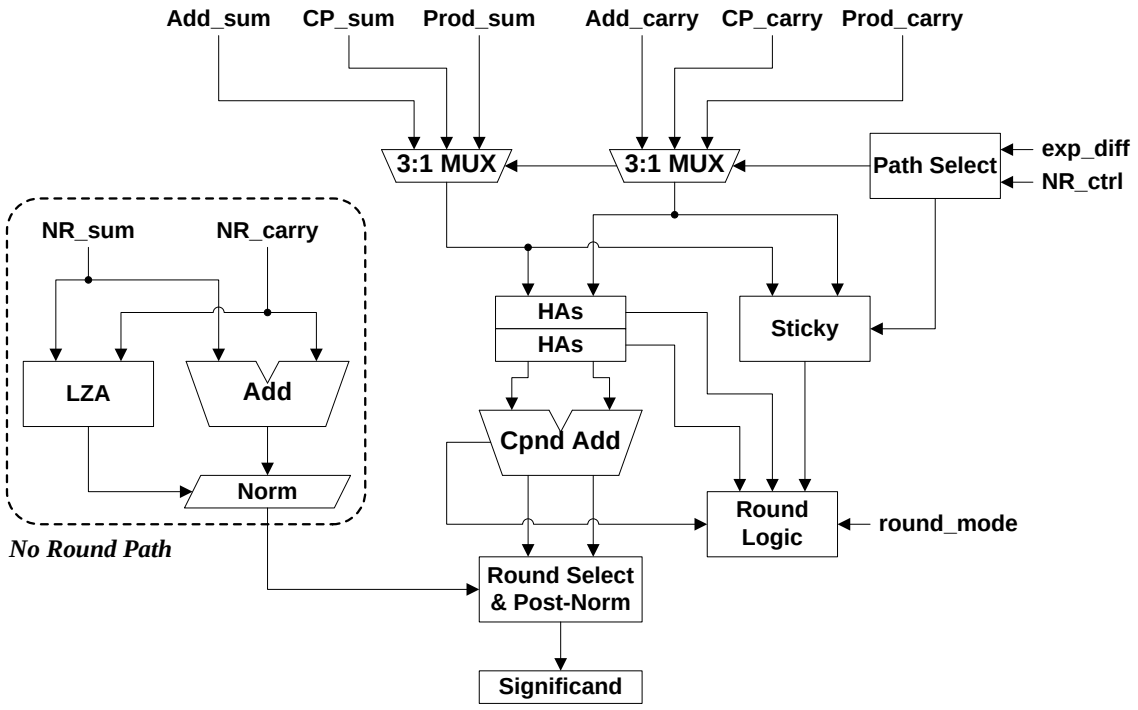


Figure 12. Add/Round Logic for a Three-Path Fused Multiply-Add Unit [28].

2.4 Floating-Point Add-Subtract Unit

Many DSP applications such as FFT and DCT require both the sum and difference of a pair of two operands for executing butterfly operations. The floating-point add-subtract unit is useful for those applications by producing the sum and difference simultaneously. The floating-point add-subtract unit takes two operands and produces the sum and difference simultaneously. There are two approaches to design the floating-point add-subtract unit. In this section, the two design approaches for the floating-point add-subtract unit are presented: 1) Discrete floating-point add-subtract unit and 2) Fused floating-point add-subtract unit.

2.4.1 Discrete Floating-Point Add-Subtract Unit

A direct way to implement the floating-point add-subtract operation is to execute two floating-point additions in parallel. The floating-point adder introduced in the previous section can be used for the discrete floating-point add-subtract unit. The discrete floating-point add-subtract unit uses two identical floating-point adders in parallel as shown in Figure 13. One of those adders performs the addition and the other performs the subtraction to produce the sum and difference results simultaneously. Since the discrete floating-point add-subtract unit executes two floating-point adders in parallel, the area and power consumption are same as that of two floating-point adders and the latency is same as that of a single floating-point adder.

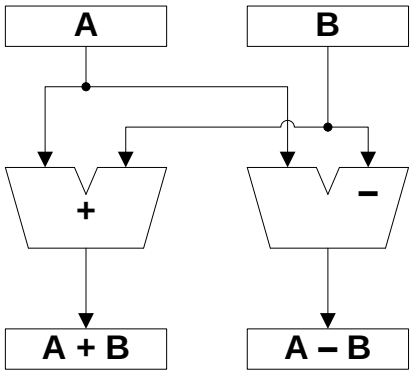


Figure 13. Discrete Floating-Point Add-Subtract Unit.

2.4.2 Traditional Fused Floating-Point Add-Subtract Unit

The 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 compare, significand swap, alignment, sign logic and

exponent adjust logic in the floating-point adder is nearly the same for the two operations. In order to reduce the overhead, a fused floating-point add-subtract unit has been introduced [6], [7]. The fused floating-point add-subtract unit produces the sum and difference results simultaneously as shown in Figure 14.

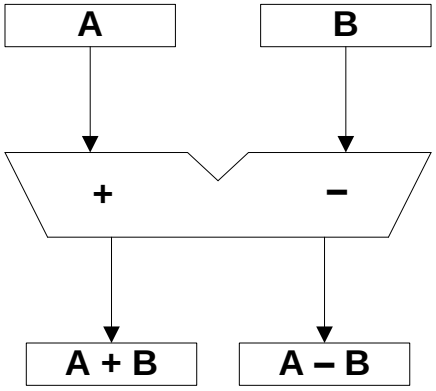


Figure 14. Fused Floating-Point Add-Subtract Unit.

Figure 15 shows the traditional fused floating-point add-subtract unit. The fused floating-point add-subtract unit produces the sum and difference results simultaneously by executing the shared logic such as the exponent compare, significand swap, alignment, sign logic and exponent adjust logic. Also, the 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 the comparison of the exponents and significands. Since two operations are explicitly performed for sum and difference results (e.g., if the addition is used for the sum, the 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 based on the operation selection bit, which is the XOR of the two sign bits. This approach simplifies the addition and subtraction operations so that the area and power consumption are reduced compared to that of the discrete design.

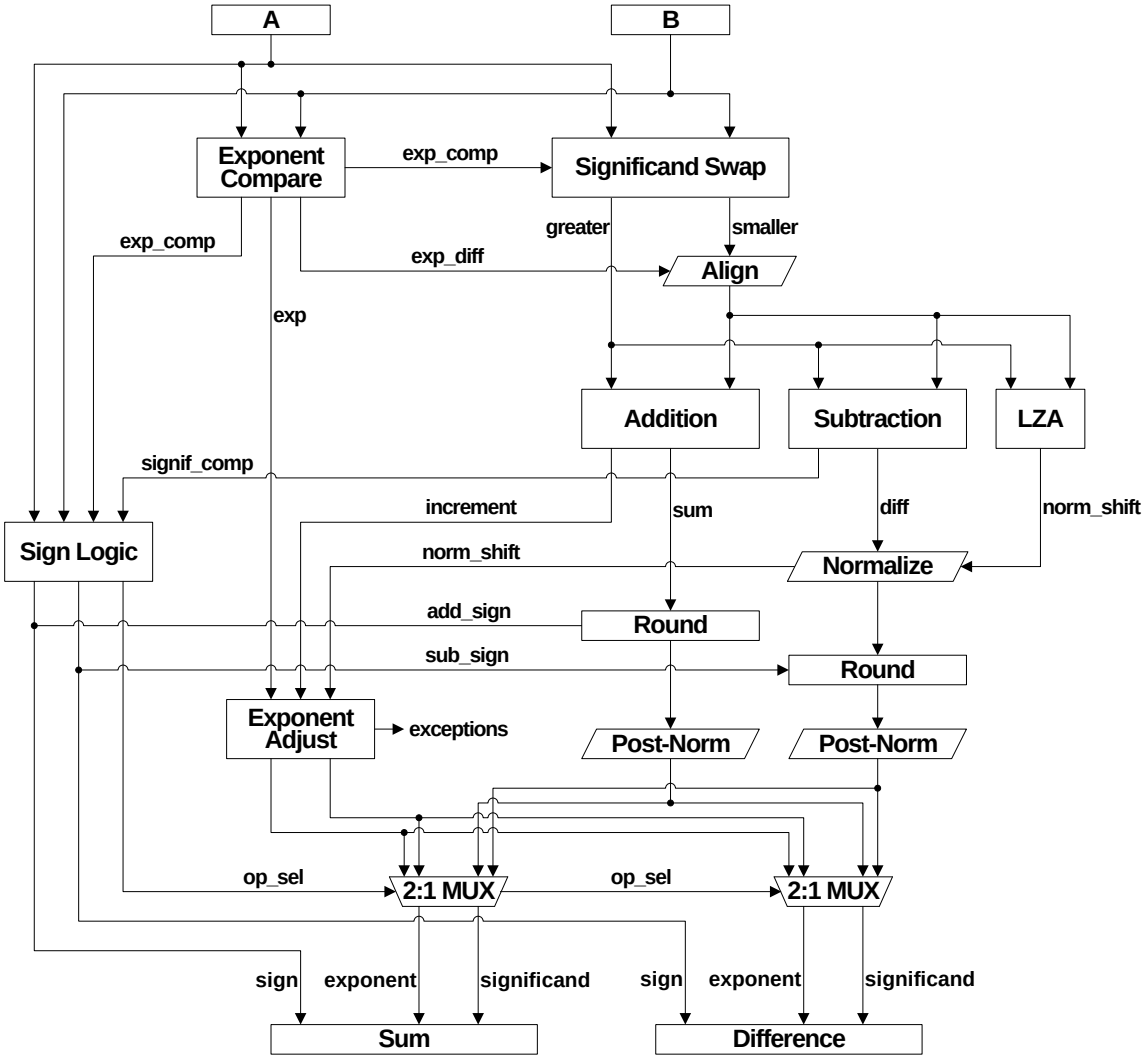


Figure 15. Traditional Fused Floating-Point Add-Subtract Unit (After [6], [7]).

Table 2. Sign Decision Table [29].

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|)

2.5 Floating-Point Two-Term Dot Product Unit

The floating-point two-term dot product is a common operation used for DSP applications such as complex multiplications and FFT and DCT butterfly operations. The floating-point two-term dot-product unit takes four operands and computes the dot product result. The two-term dot product operation requires an addition subsequent to two multiplications. There are two approaches to design the floating-point two-term dot product unit. In this section, the two design approaches are presented: 1) Discrete floating-point two-term dot product unit and 2) Fused floating-point two-term dot product unit.

2.5.1 Discrete Floating-Point Dot Product Unit

A direct way to design the floating-point two-term dot product unit is to execute two floating-point multiplications and a floating-point addition. The floating-point multiplier and adder introduced in the previous sections can be used for the discrete

floating-point two-term dot product unit. The discrete floating-point two-term dot product unit uses two identical floating-point multipliers and a floating-point adder as shown in Figure 16. Each multiplier takes two operands and computes a product. The floating-point adder takes the two products from the two floating-point multipliers and computes the dot product result. The area and power consumption are equal to that of two floating-point multipliers and a floating-point adder. The latency is same as that of a floating-point multiplier and a floating-point adder.

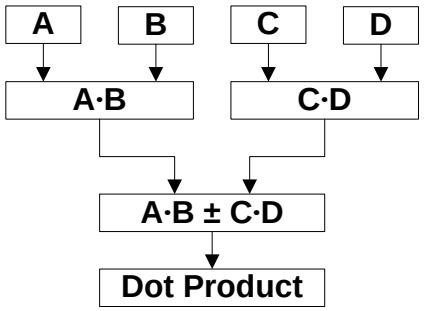


Figure 16. Discrete Floating-Point Two-Term Dot Product Unit.

2.5.2 Traditional Fused Floating-Point Dot Product Unit

The discrete floating-point two-term dot product unit simply executes two multiplications and an addition to produce the dot product result. However, it requires large logic area, power consumption and latency. Moreover, since rounding is performed three times (after each of the multiplications and after the addition), the accuracy is decreased. In order to reduce the area, power consumption and latency, and increase the accuracy, the fused floating-point dot product unit has been introduced [7], [8]. Figure 17 shows the fused floating-point two-term dot product unit. The fused floating-point two-

term dot product unit shares the common logic such as exponent compare, significand addition, exponent adjust and sign logic so that the area, power consumption and latency are reduced. Also, the fused floating-point dot product unit performs only a single rounding so that the accuracy increases.

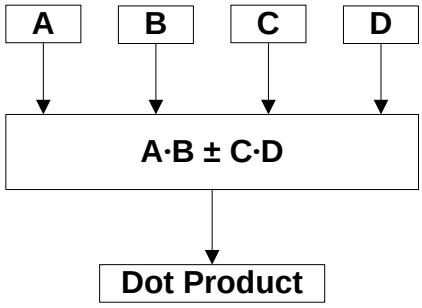


Figure 17. Fused Floating-Point Two-Term Dot Product Unit.

Figure 18 shows the traditional fused floating-point dot product unit. The traditional fused floating-point dot product unit [7], [8] is based on the fused floating-point multiply-add unit. The steps to execute the fused floating-point dot product are

1) Two multiplier trees are used to produce two pairs of sums and carries (a total of four numbers). In parallel, two sums of exponents are computed and compared to determine the greater product and the difference is computed. Also, the operation (addition or subtraction) is selected using the sign bits and op code.
2) One sum and carry pair is aligned based on the exponent difference result and inverted if the operation is subtraction. The two pairs of significands are

passed to a 4:2 reduction tree. Carry save adders are used to form the reduction tree, which reduces the four significands to two.

3) The two significand additions are performed and the sum is complemented if it is negative. The LZA is performed in parallel with the significand addition and the significand sum is shifted by the amount of the LZA result. The carry-out of the significand addition is passed to the sign logic and the exponent adjust logic.
4) The sign logic determines the sign of the product result. Since some of rounding modes specified in IEEE-754 Standard [1] require the sign (i.e., round to positive and negative infinity), the sign logic must be performed prior to the round logic.
5) The normalized significands are rounded and post-normalized. The exponent is adjusted with the significand addition carry-out and the shift amount from the LZA.

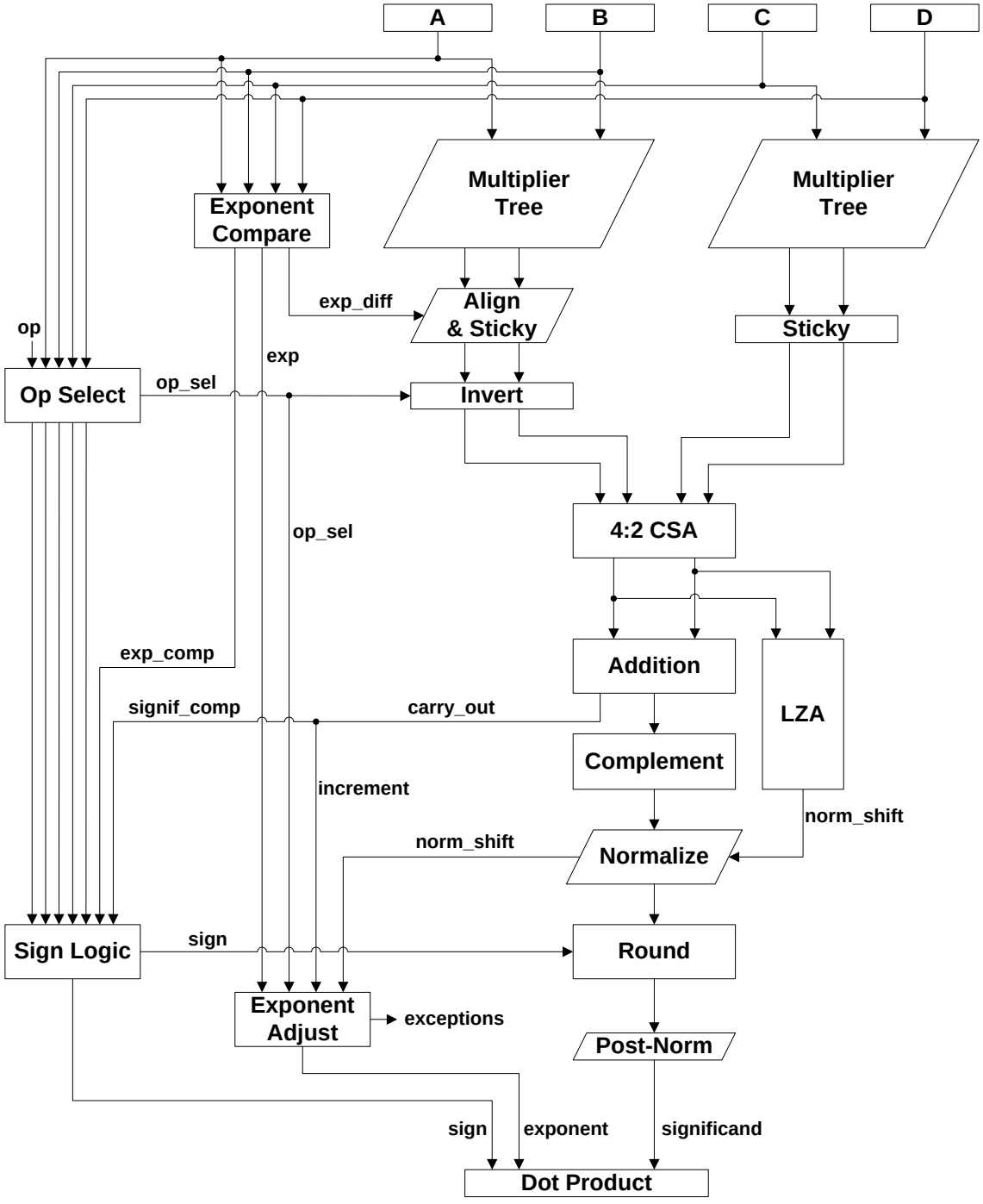


Figure 18. Traditional Fused Floating-Point Two-Term Dot Product Unit (After [7], [8]).

2.6 Floating-Point Three-Term Adder

In many DSP applications, multiple floating-point additions are executed consecutively. The floating-point multi-term adder takes multiple operands and executes multiple additions with an operation to generate a sum. The general issues on the floating-point multi-term adder design can be represented by the floating-point three-term adder designs. There are two approaches to design the floating-point three-term adder. In this section, the two design approaches are presented: 1) Discrete floating-point three-term adder and 2) Fused floating-point three-term adder.

2.6.1 Discrete Floating-Point Three-Term Adder

A direct way to design the floating-point three-term adder is to execute two floating-point additions as shown in Figure 19. The first floating-point adder takes two operands and computes an intermediate sum. Then, the second floating-point adder takes the intermediate sum and the third operand and computes the final sum. The floating-point adder introduced in the previous section can be used for the two floating-point adders. The area and power consumption are that of a floating-point adder and control logic. The latency is same as that of the two floating-point adders and control logic.

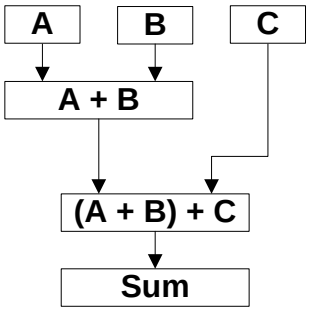


Figure 19. Discrete Floating-Point Three-Term Adder.

2.6.2 Traditional Fused Floating-Point Three-Term Adder

The discrete floating-point three-term adder simply executes the two floating-point adders in serial, which requires double area, power consumption and latency of the floating-point adder. Moreover, the serial execution of the two floating-point adders performs rounding twice, which reduces the accuracy. In order to increase both the accuracy and performance, a fused floating-point three-term adder is proposed [9], [10]. Figure 20 shows the fused floating-point three-term adder.

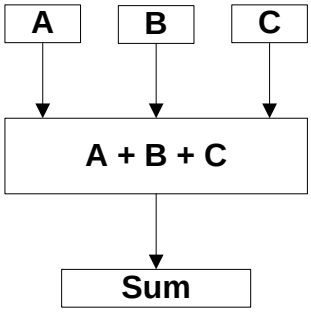


Figure 20. Fused Floating-Point Three-Term Adder.

Figure 21 shows the traditional fused floating-point three-term adder. The traditional floating-point three-term adder takes three operands and computes the two additions at once. The procedure of executing the fused floating-point tree-term adder is

1) The exponent compare logic determines the max exponent among the three exponents and computes the differences between the max exponent and each exponent. The three significands are shifted by the amount of the corresponding exponent differences.
2) The effective operations are determined based on the three sign bits and the two op codes. The aligned significands are inverted if the corresponding operations are subtraction. Then, the significands are passed to the 3:2 reduction tree. Carry save adders are used to form the reduction tree, which reduces the three significands to two.
3) The significand addition is performed and the sum is complemented if it is negative. The LZA is performed in parallel with the significand addition and the significand sum is shifted by the amount of the LZA result. The carry-out of the significand addition is passed to the sign logic and the exponent adjust logic.
4) The sign logic determines the sign of the sum result. Since some of the rounding modes specified in IEEE-754 Standard [1] require the sign (i.e., round to positive and negative infinity), the sign logic must be performed prior to the round logic.

5) The normalized significands are rounded and post-normalized. The exponent is adjusted with the significand addition carry-out and the shift amount from the LZA.

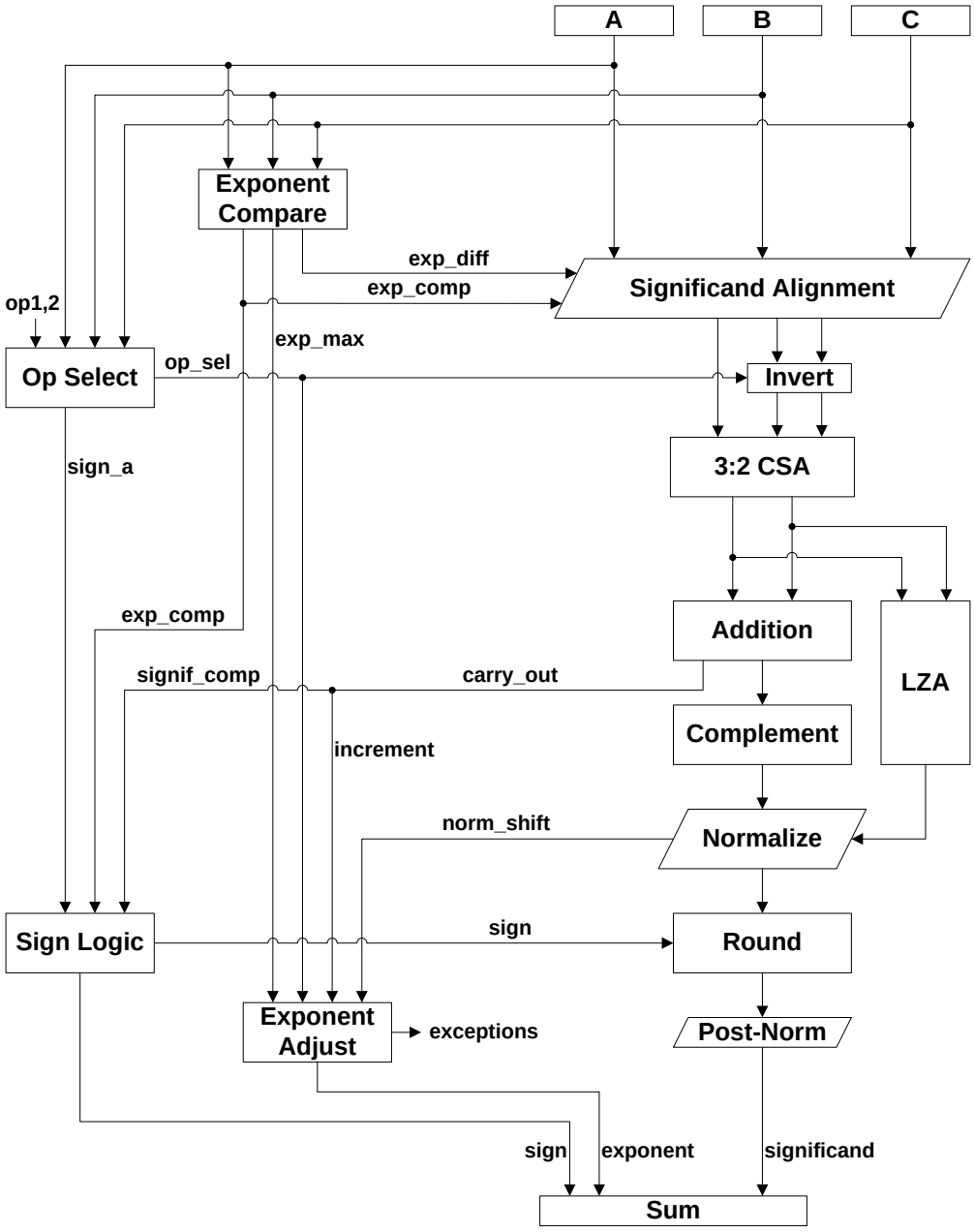


Figure 21. Traditional Fused Floating-Point Three-Term Adder (After [9], [10]).

Chapter 3

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

In this chapter, improved architecture designs and implementation details for a fused floating-point add-subtract unit are presented. Many digital signal processing (DSP) applications such as fast Fourier transform (FFT) and discrete cosine transform (DCT) butterfly operations can benefit from the fused floating-point add–subtract unit [7], [11]. Therefore, the improved fused floating-point add–subtract unit will contribute to the next generation floating-point arithmetic and DSP application development.

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]. The traditional fused floating-point add-subtract unit reduces the area and power consumption compared to the discrete floating-point add-subtract unit. In order to further improve the performance of the fused floating-point add-subtract unit several algorithms and optimization techniques can be applied as

1) For a fast significand alignment, a new alignment scheme is proposed. By performing sticky logic after the significand shift, small number of significand bits are generated, which reduces the latency of the significand addition, subtraction and round logic.

2) For fast rounding, compound addition, subtraction and rounding logic are performed in parallel. The compound addition and subtraction computes both the rounded and unrounded sum and difference, respectively, and the result of the round logic selects the correct result so that the latency of the round logic is hidden.
3) A dual-path algorithm is applied to improve the performance. The dual-path logic consists of a far path and a close path. In the far path, massive cancellation does not occur during subtraction so that the leading zero anticipation (LZA) and normalization are not required. In the close path, the significands are shifted by only two bits at most so that the large significand alignment and rounding are not required. By skipping the unnecessary logic in each path, the dual-path design reduces the latency of the critical path.
4) To increase the throughput, pipelining is applied. Based on a data flow analysis, the proposed dual-path design is split into two pipeline stages. By properly arranging the components, latencies of the two pipeline stages are fairly well balanced so that the throughput of the entire design is increased.

3.1 Enhanced Floating-Point Add-Subtract Unit

Traditional fused floating-point add-subtract unit described in Chapter 2 results in a 40% reduction in the area and a 3% increase in the latency compared to the discrete add-subtract unit [6], [7]. However, it is an initial design so that the performance can be

further improved by applying several optimizations. Figure 22 shows the modified design for an enhanced fused floating-point add-subtract unit. In this section, two optimizations for the enhanced fused floating-point add-subtract unit are introduced: 1) New alignment scheme and 2) Compound addition and fast rounding scheme.

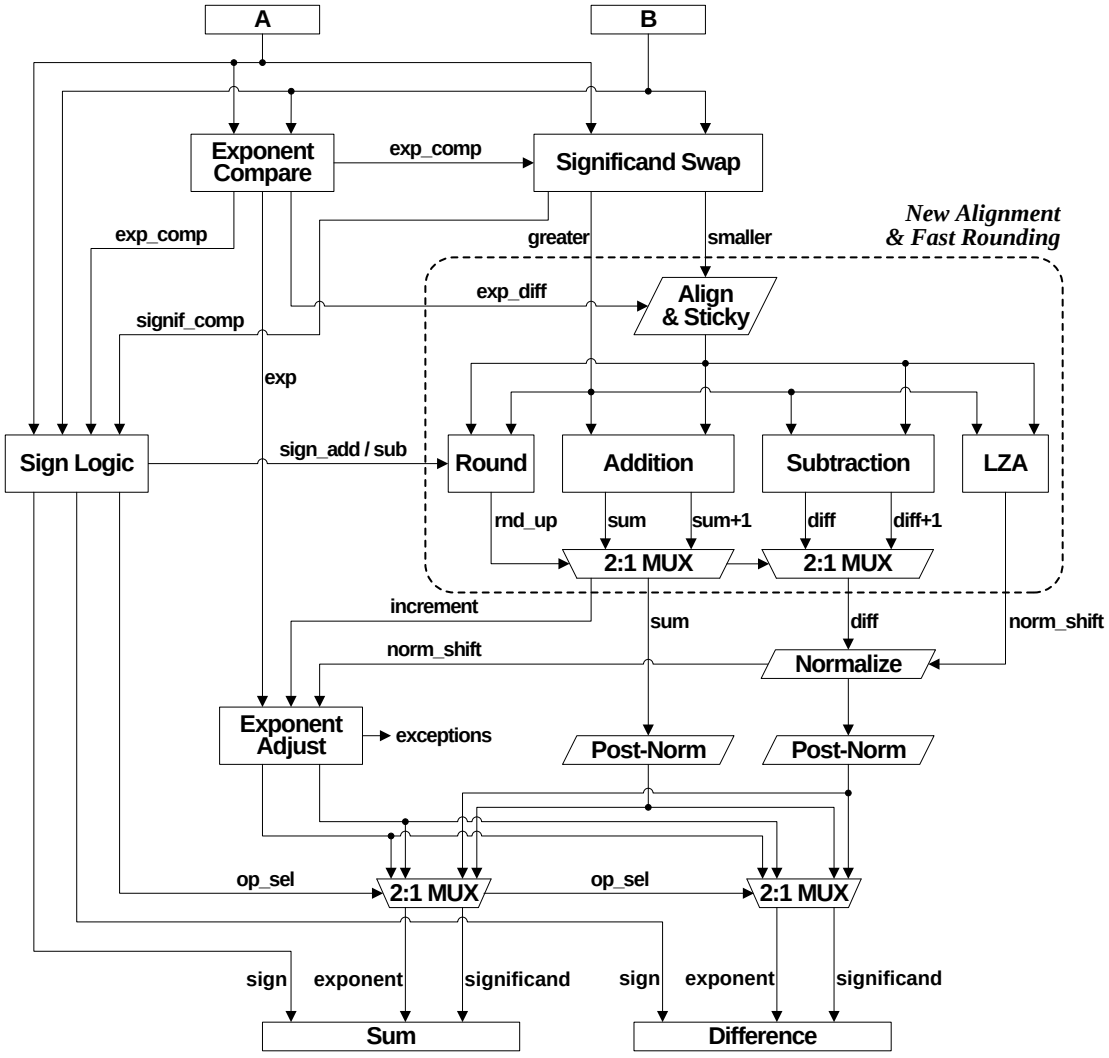


Figure 22. Enhanced Fused Floating-Point Add-Subtract Unit [29].

3.1.1 New Alignment Scheme

The traditional alignment scheme shifts the smaller significand by the amount of the exponent difference and passes it to the significand addition as shown in Figure 23. Since the significand can be shifted by up to the number of significand bits, the aligned significands are 2f bits, where f is the number of the significand bits. The large significands are passed to the significand addition and subtraction, resulting in a large delay.

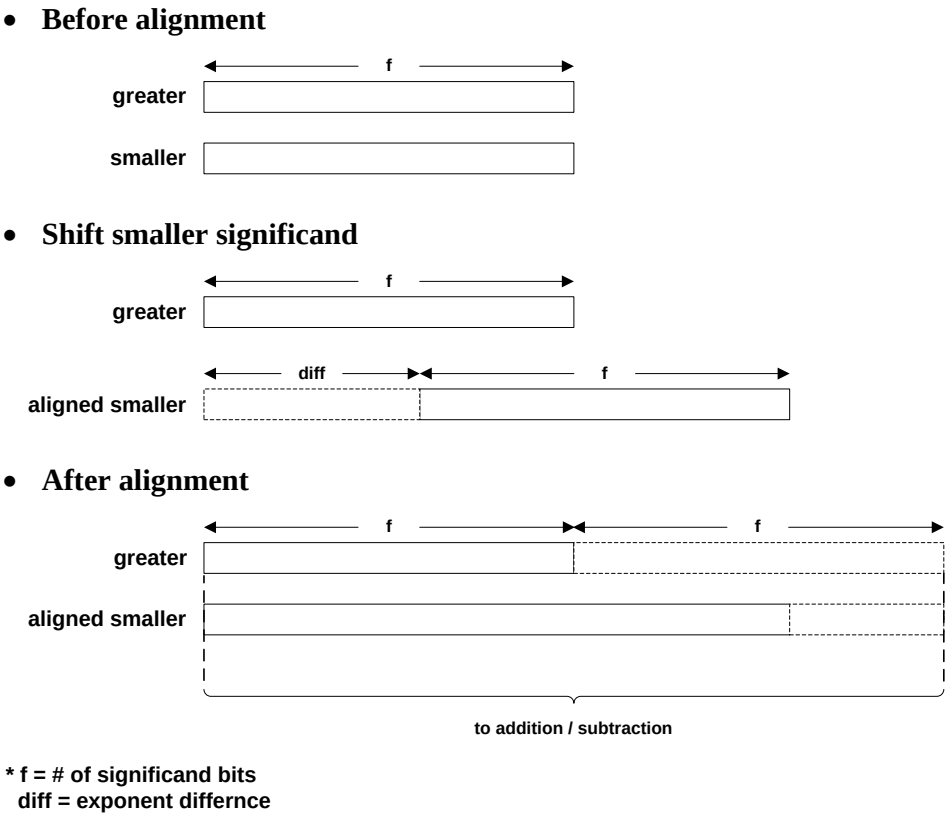
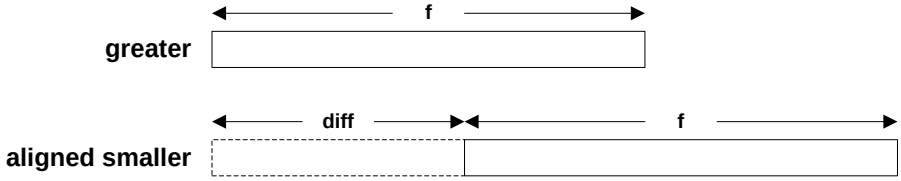


Figure 23. Traditional Alignment Scheme for a Fused Add-Subtract Unit [29].

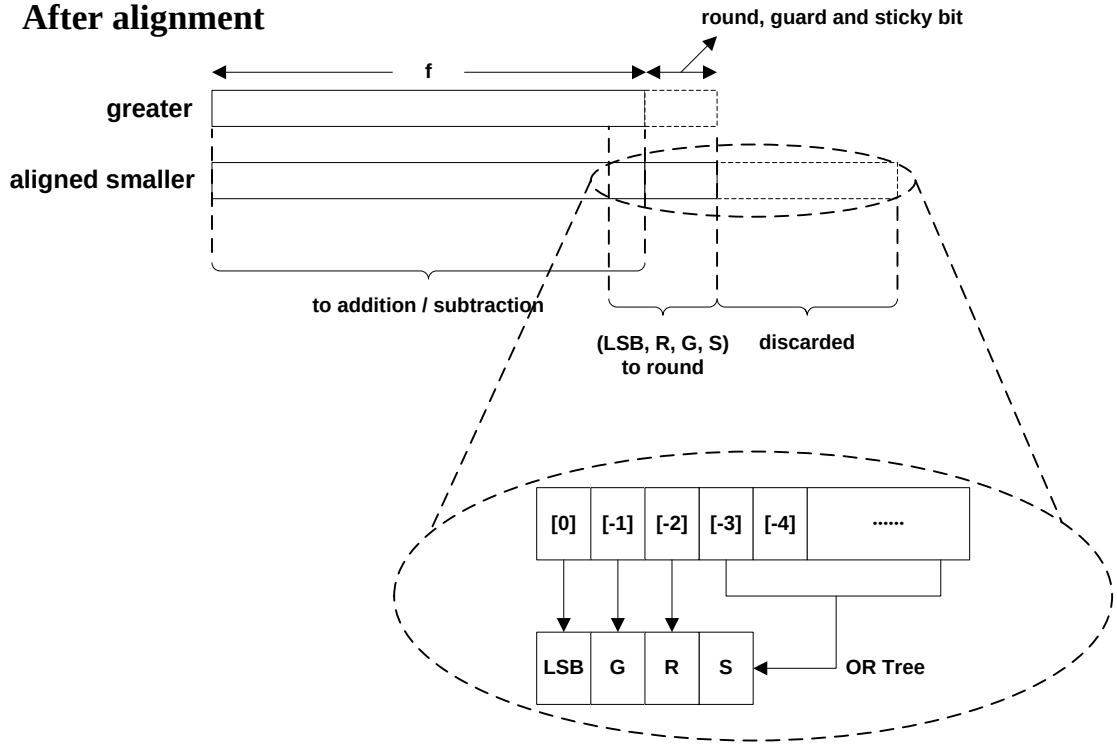
In order to reduce the overhead, a new alignment scheme is proposed, which performs the sticky logic after the smaller significand is shifted as shown in Figure 24. The sticky logic generates round, guard and sticky bits. The 1^{st} and 2^{nd} bits under the LSB become the guard and round bits and the sticky bit is set if at least one bit of the rest of the LSBs is 1, which can be implemented with OR tree. The four bits including the LSB, guard, round and sticky bits are used for the round logic to simplify the round logic and the rest of the LSBs are discarded. Using the new alignment scheme, only f bits are passed to the significand addition and subtraction so that the delay is reduced.

- **Before alignment**

- **Shift smaller significand**



- **After alignment**



*** f = # of significand bits**
diff = exponent differnce

Figure 24. New Alignment Scheme for a Fused Add-Subtract Unit [29].

3.1.2 Compound Addition and Fast Rounding Scheme

The aligned significands are passed to the significand addition, subtraction and round logic. For fast rounding, the proposed design uses compound addition and subtraction, and performs the round logic in parallel. The compound addition and subtraction produce both rounded and unrounded sums and differences simultaneously and the round logic selects the correct result as

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

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

The round logic takes the LSB, 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 rounding modes specified in IEEE-754 Standard [1] as shown in Table 3.

Table 3. Round Table [29].

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	+	1
		–	0
Round to negative infinity (010)	x000	x	0
	else	+	0
		–	1
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 result of 4 bit add/subtract.
** x means don't care.

3.2 Dual-Path Fused Floating-Point Add-Subtract Unit

To achieve a high performance fused floating-point add-subtract unit, dual-path algorithms are proposed [29], [30]. The dual-path algorithms skip the normalization in case of the far path and skip the large significand alignment in case of the close path. The path is selected based on the exponent difference. Since the normalization and the significand alignment are the bottlenecks of the fused floating-point add-subtract unit, the dual-path algorithms, which enable to skip one of them, improve the performance. In this section, two designs for a dual-path fused floating-point add-subtract unit are presented: 1) Low power design and 2) High-speed design.

3.2.1 Low Power Dual-Path Fused Floating-Point Add-Subtract Unit

The dual-path for the low power design consists of the far path and close path logic. Figure 25 shows the low power dual-path fused floating-point add-subtract unit. The far path logic contains the significand swap, alignment and sticky logic and the close path logic contains the 2 bit exponent compare, 1 bit significand alignment, significand compare logic, LZA and normalization. One of the two paths is selected based on the exponent difference. The far path skips the LZA and normalization and the close path skips the significand alignment. The low power dual-path design performs the normalization in the close path prior to the significand addition and subtraction, while the enhanced design performs the normalization after the significand subtraction. The rest of logic is designed similar to the enhanced design including the compound addition, subtraction and rounding.

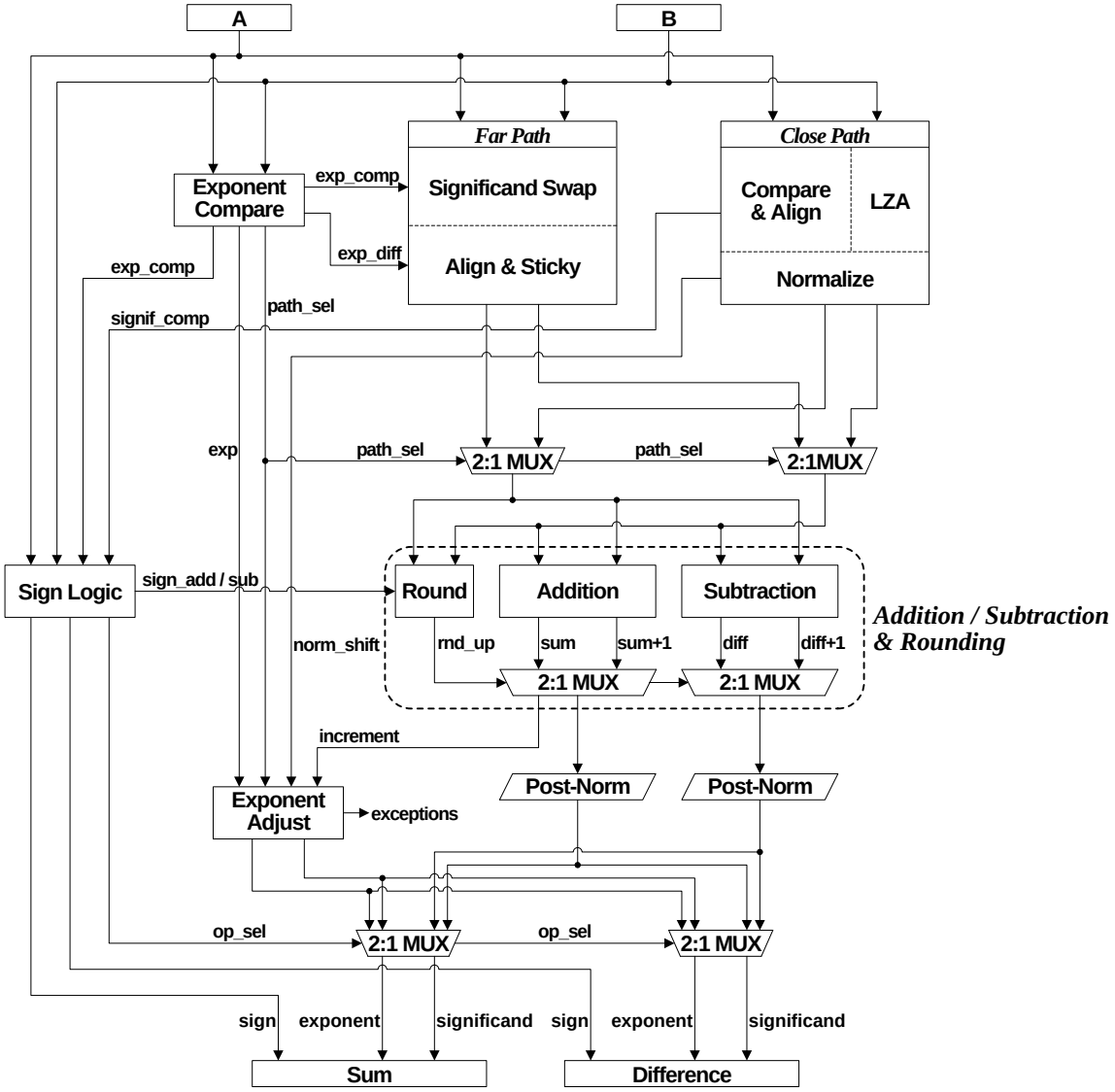


Figure 25. Low Power Dual-Path Fused Floating-Point Add-Subtract Unit [30].

3.2.1.1 Far Path Logic

The far path is selected if the exponent difference is greater than 1. In this case, massive cancellation does not occur during the subtraction so that the LZA is unnecessary. The far path logic is designed similar to the front end of the traditional

floating-point adder as shown in Figure 26. The far path logic consists of the significand swap, alignment and sticky logic.

The greater and smaller significands are determined by swapping the two significands based on the exponent comparison as

$$greater_{signif} = \begin{cases} (1, \mathrm{A}_{signif}) & \text{if } \mathrm{A}_{exp} > \mathrm{B}_{exp} \\ (1, \mathrm{B}_{signif}) & \text{if } \mathrm{A}_{exp} < \mathrm{B}_{exp} \end{cases}$$

$$smaller_{signif} = \begin{cases} (1, \mathrm{B}_{signif}) \gg diff_{exp} & \text{if } \mathrm{A}_{exp} > \mathrm{B}_{exp} \\ (1, \mathrm{A}_{signif}) \gg diff_{exp} & \text{if } \mathrm{A}_{exp} > \mathrm{B}_{exp}, \end{cases}$$

where $diff_{exp}$ is the exponent difference. The two significands are aligned with a 1 attached to the MSB forming normalized significands. With the significands, the new alignment and sticky logic are performed that are described in Section 3.1.1. Since the far path requires at most a 1 bit normalization shift, the large normalization shift is skipped.

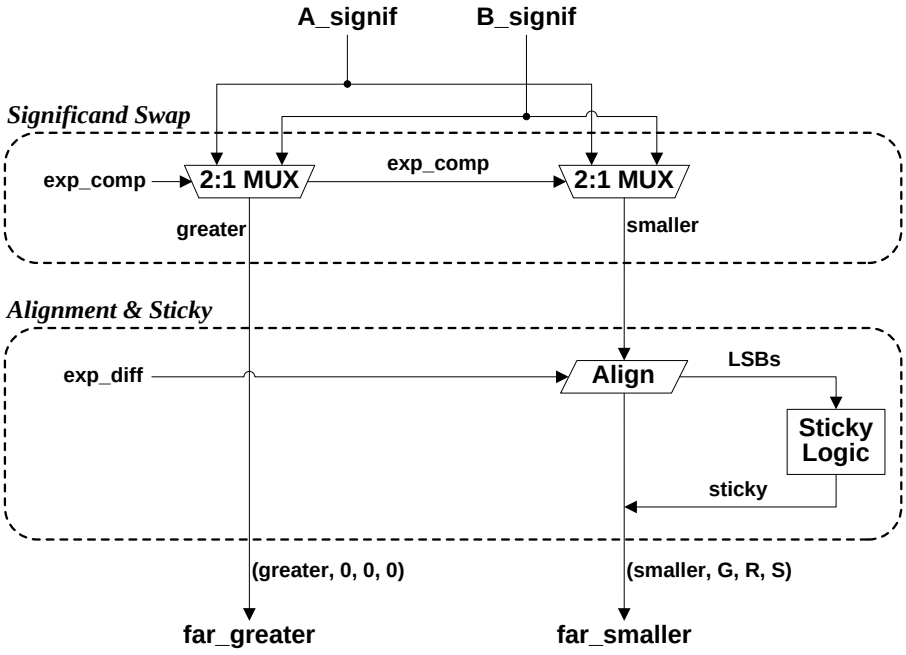


Figure 26. Far Path for a Low Power Dual-Path Add-Subtract Unit [30].

3.2.1.2 Close Path Logic

The close path is selected if the difference of the two exponents is 0 or 1. Figure 27 shows the close path logic for the low power dual-path design. The close path requires the LZA and normalization to handle the cancellation shift during the subtraction. In the close path, the exponent difference is 0 or 1, the 2 bit exponent comparison and 1 bit significand alignment are sufficient which enables skipping the large significand alignment. There are three cases of the 1 bit significand alignment depending on the difference of the exponents as

$$\mathrm{A}_{aligned\_signif} = \begin{cases} (1, \mathrm{A}_{signif}, 0) & \text{if } \mathrm{A}_{exp} - \mathrm{B}_{exp} = 1 \\ (1, \mathrm{A}_{signif}, 0) & \text{if } \mathrm{A}_{exp} - \mathrm{B}_{exp} = 0 \\ (01, A_{signif}) & \text{if } \mathrm{A}_{exp} - \mathrm{B}_{exp} = -1 \end{cases}$$

$$\mathrm{B}_{aligned\_signif} = \begin{cases} \left(01, \mathrm{B}_{signif}\right) & \text{if } \mathrm{A}_{exp} - \mathrm{B}_{exp} = 1 \\ (1, \mathrm{B}_{signif}, 0) & \text{if } \mathrm{A}_{exp} - \mathrm{B}_{exp} = 0 \\ (1, \mathrm{B}_{signif}, 0) & \text{if } \mathrm{A}_{exp} - \mathrm{B}_{exp} = -1. \end{cases}$$

The aligned significands are passed to the two multiplexers to determine the greater significand. The greater significand is determined based on the exponent comparison. If the exponent difference is 0, significand comparison determines the greater significand. The significand comparison is also passed to the sign logic to determine the signs.

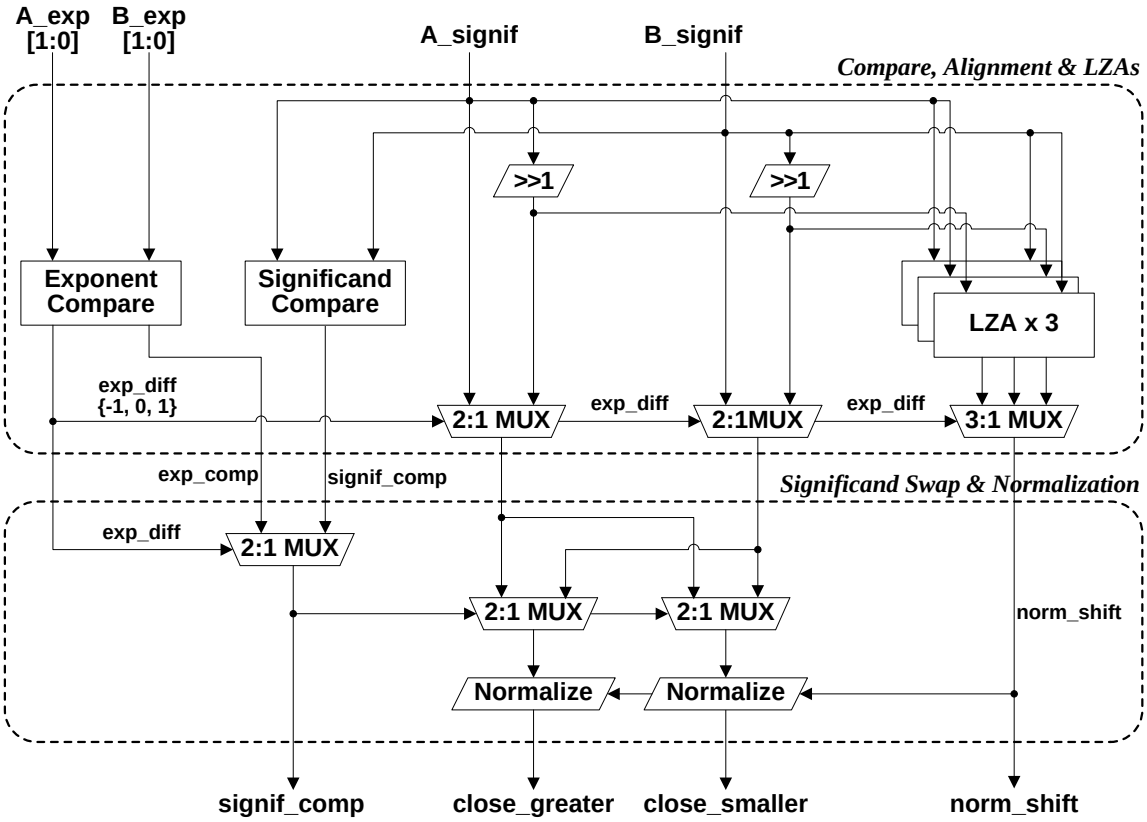


Figure 27. Close Path for a Low Power Dual-Path Add-Subtract Unit [30].

For fast normalization, three LZAs performed in each case of the significand alignment, which computes the shift amount for the normalization. The LZA with concurrent correction is used for fast normalization [19] – [21][3]. More details of the LZA logic are presented in Section 3.2.2.5. One of the three LZA results is selected based on the exponent difference. Then, the aligned significands are normalized by the shift amount from the LZA.

[3] The error correction logic in [19] is modified by [20] and [21] to improve the accuracy and eliminate the redundancy, respectively.

3.2.2 High-Speed Dual-Path Fused Floating-Point Add-Subtract Unit

The dual-path for the high-speed design consists of the far path and close path with further optimizations. Figure 28 shows the high-speed dual-path fused floating-point add-subtract unit. The far path consists of the significand swap, alignment, sticky logic, significand addition, subtraction, rounding. The close path consists of the 2 bit exponent compare, 1 bit significand alignment, three additions, subtractions and LZAs, and normalization. The significand additions, subtractions, rounding and LZAs are performed in parallel in each path so that the area and power consumption are increased compared to the low power dual-path design. However, the logic components are more parallelized so that the latency is reduced compared to the low power design.

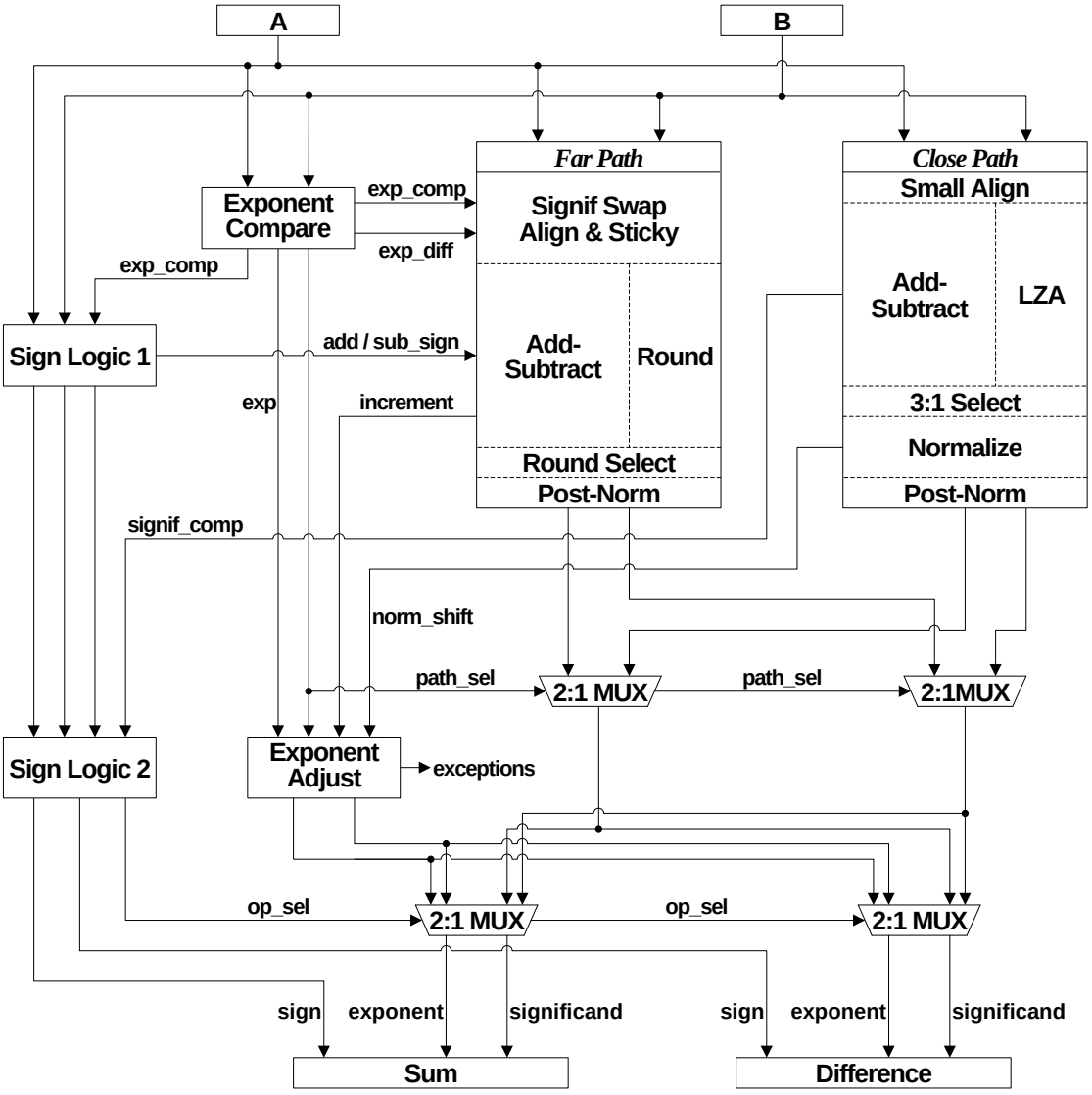


Figure 28. A High-Speed Dual-Path Fused Floating-Point Add-Subtract Unit [29].

3.2.2.1 Far Path Logic

The front end of the far path logic for the high-speed dual-path design is same as that of the low power dual-path design. It includes the significand swap, alignment and sticky logic. Figure 29 shows the far path logic for the high-speed dual-path fused

floating-point add-subtract unit. The aligned significands are passed to the significand addition, subtraction and rounding. For fast rounding, the compound addition, subtraction and round logic are performed in parallel as described in Section 3.1.2.

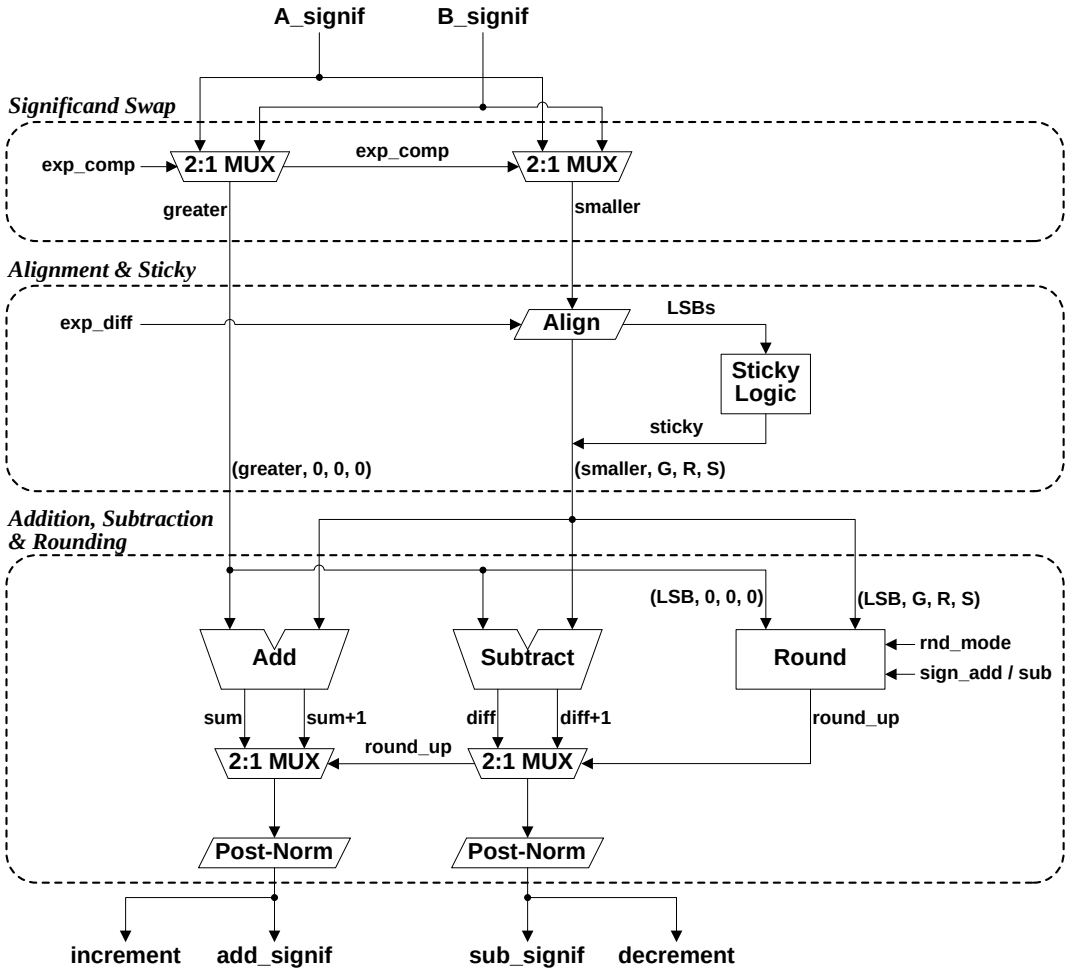


Figure 29. Far Path for a High-Speed Dual-Path Add-Subtract Unit [29].

3.2.2.2 Close Path Logic

The front end of the close path logic for the high-speed dual-path design is same as that of the low power dual-path design. It includes 2 bit exponent compare and 1 bit

significand alignment. Figure 30 shows the close path logic for the high-speed dual-path fused floating-point add-subtract unit. There are three cases of the 1 bit significand alignment depending on the difference of the exponents as described in the low power dual-path design. Three significand additions, subtractions and LZAs are performed simultaneously in each case of the significand alignment.

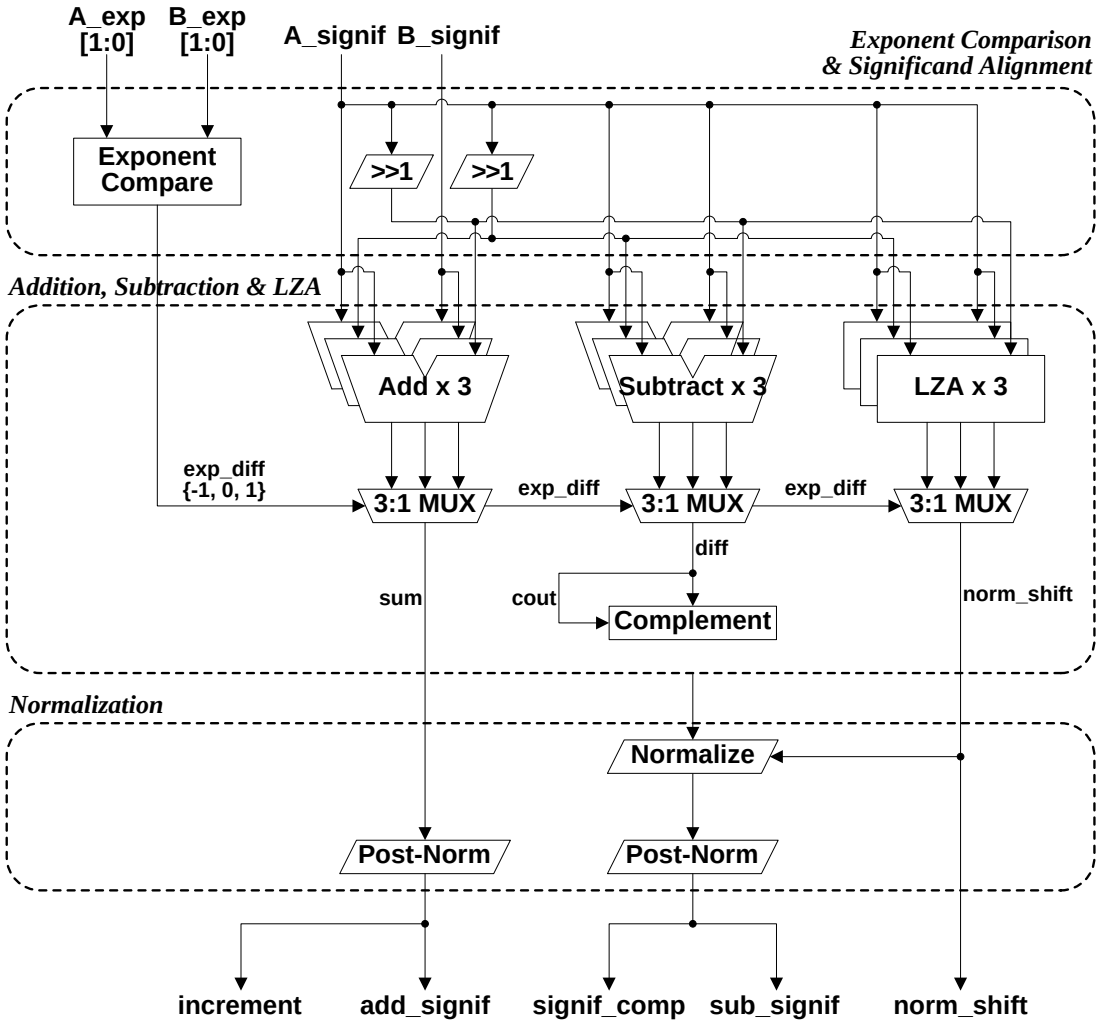


Figure 30. Close Path for a High-Speed Dual-Path Fused Add-Subtract Unit [29].

One of the three results is selected based on the exponent comparison, which compares the two LSBs of the exponents. In contrast to the far path, the significands are not swapped to avoid the 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. Since the significands in the close path are shifted by at most 1 bit, rounding is not required [24]. The addition result is normalized by 1 bit overflow, while the subtraction result is normalized using the shift amount from the LZA.

3.2.2.3 Exponent Compare Logic

The exponent compare logic computes the difference of the two exponents and determines which is greater as shown in Figure 31. 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 as

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

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

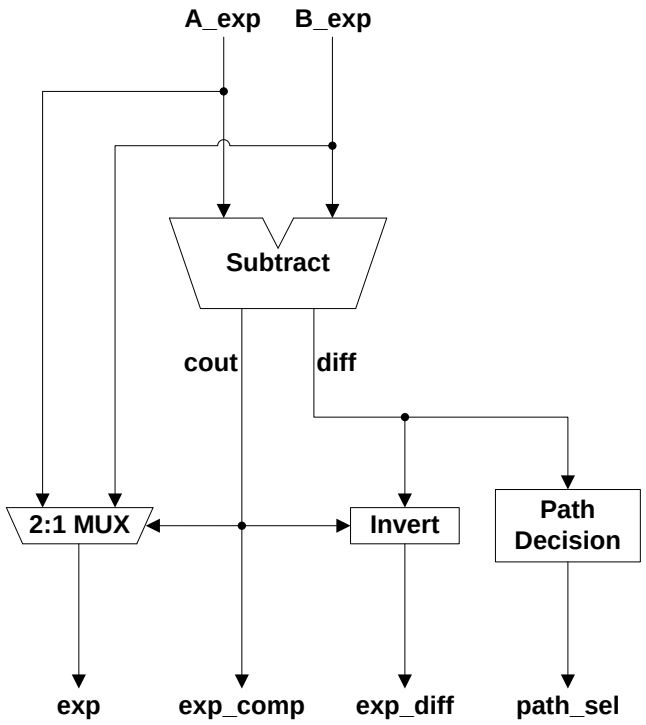


Figure 31. Exponent Compare for a Dual-Path Fused Add-Subtract Unit [29].

3.2.2.4 Significand Addition / Subtraction

The dual-path fused floating-point add-subtract unit requires several integer adders for significand addition and subtraction, exponent compare and exponent adjust logic. Since the integer adder accounts for a large amount of area, latency and power consumption in the dual-path fused floating-point add-subtract unit, the addition scheme affects the performance of the entire design. In order to achieve a high performance design, Kogge-Stone adders are used for the integer additions. As well known, the Kogge-Stone adder is one of the fastest integer adders using parallel prefix form [22], [23]. The parallel prefix adder is a carry-look-ahead style architecture that uses basic carry operators such as AOI/OAI and NOR/NAND. Figure 32 shows the structure of the

24 bit Kogge-Stone adder, which is mainly used for the significand additions. The propagate/generate (PG) generators for the parallel-prefix form are shown in Figure 33.

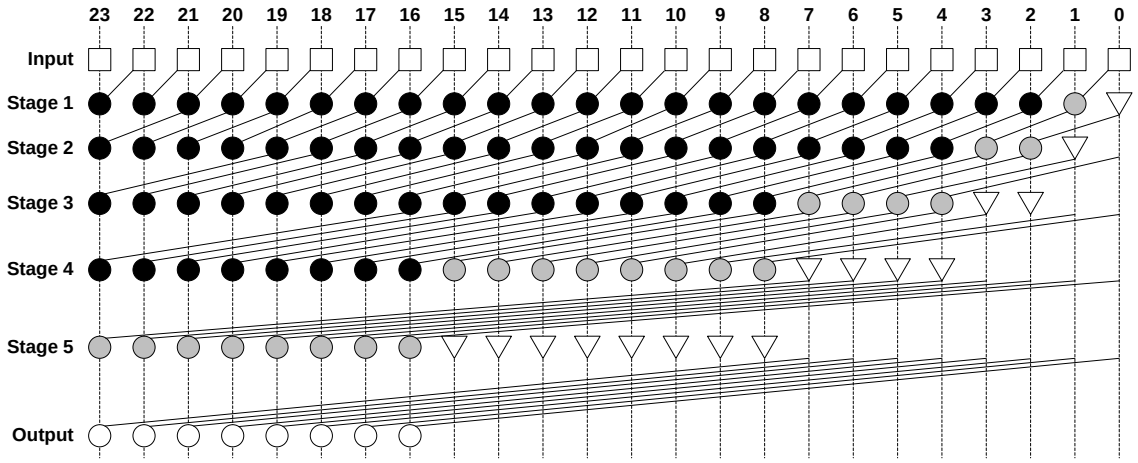


Figure 32. 24 bit Kogge-Stone Adder (After [22]).

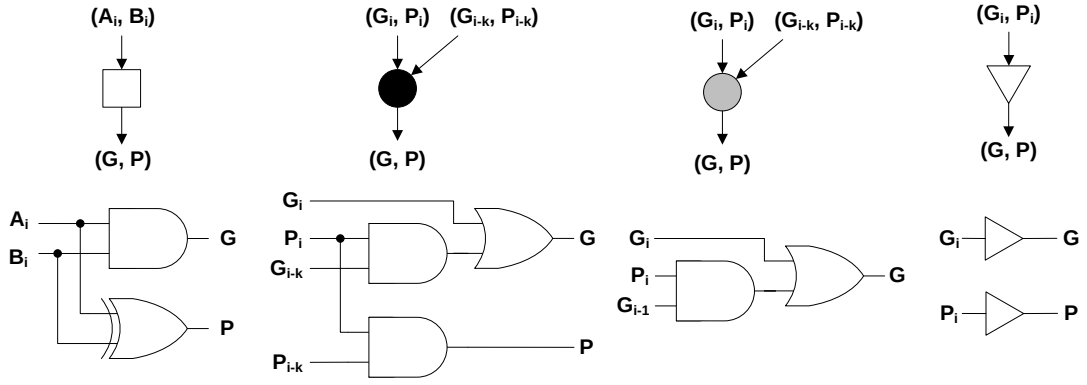


Figure 33. PG Generators for a Parallel Prefix Adder (After [22]).

3.2.2.5 Leading Zero Anticipation (LZA)

In floating-point subtraction, it is required to normalize the significand after the significand subtraction in case cancellation occurs. Figure 34 shows an example of

cancellation and normalization for the significand subtraction. If the MSB of the subtraction result is not 1, it is required to be left shifted until the MSB becomes 1, which is the normalization.

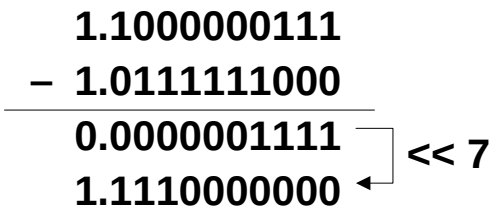


Figure 34. Example of Cancellation and Normalization.

The leading zero detection (LZD) logic determines the MSB location after the significand subtraction [31], which increases the latency of the critical path. To eliminate the delay, leading zero anticipation (LZA) is proposed [18], which is performed in parallel with the significand addition. The LZA logic predicts the MSB location of the subtraction result in constant time so that it hides the delay for detecting the shift amount. For some input patterns, however, the shift amount from the LZA is required to be corrected based on the carry-out of the subtraction, which increases the critical path latency. To avoid the correction logic after the subtraction, concurrent correction logic is proposed [19] – [21][4]. Figures 35 and 36 show the LZA with and without concurrent correction logic, respectively.

[4] The error correction logic in [19] is modified by [20] and [21] to improve the accuracy and eliminate the redundancy, respectively.

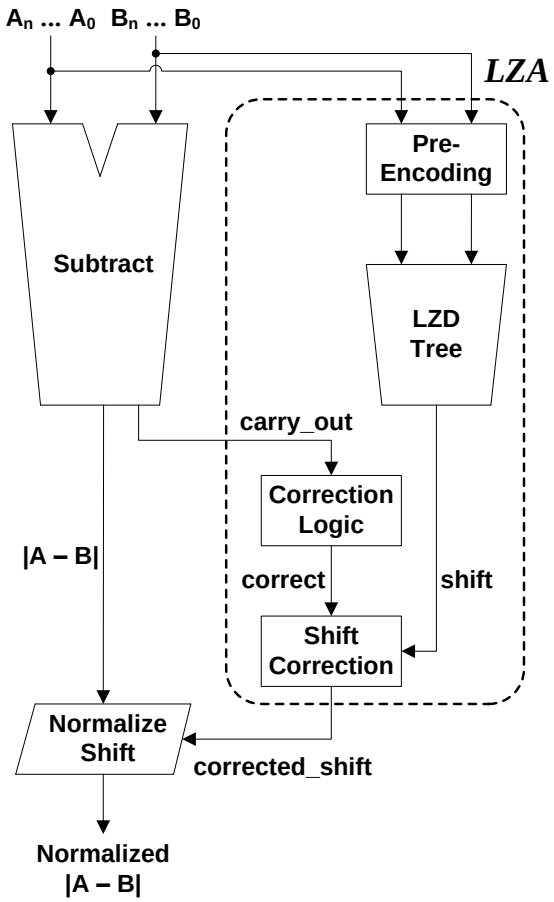


Figure 35. LZA without Concurrent Correction [19].

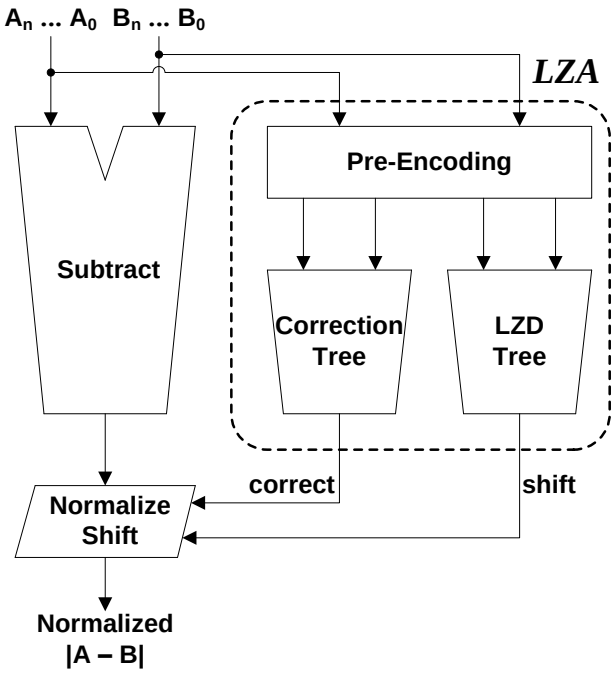


Figure 36. LZA with Concurrent Correction [19].

The pre-encoding logic performs bitwise operation with two operands to generate the W vector as

$$W = A - B$$

$$w_i = a_i - b_i, \qquad w_i \in \{-1, 0, 1\},$$

where a_i, b_i are the i^{th} bits from the MSB of the two significands. The W vector is represented by one of three symbols z_i, p_i and n_i as

$$z_i = \overline{a_i \oplus b_i} \quad (z_i = 1 \quad if \quad a_i = b_i)$$

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

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

The pre-encoding patterns that terminate the leading zeros and the corresponding leading zeros for W > 0 are shown in Table 4. The number of leading zeros is computed with the three symbols as

$$f_i(pos) = z_{i+1}p_i\bar{n}_{i-1} + \bar{z}_{i+1}n_i\bar{n}_{i-1} \text{ for } W > 0$$

Similarly, for the bit patterns when W < 0,

$$f_i(neg) = z_{i+1}n_i\bar{p}_{i-1} + \bar{z}_{i+1}p_i\bar{p}_{i-1} \text{ for } W < 0.$$

Combining the two equations, the F vector is generated as

$$f_i = z_{i+1}(p_i\overline{n}_{i-1} + n_i\overline{p}_{i-1}) + \overline{z}_{i+1}(p_i\overline{p}_{i-1} + n_i\overline{n}_{i-1}).$$

Table 4. LZA Pre-Encoding Patterns for W > 0 [19].

W vector	Leading Zeros	Pre-encoding Pattern
$0^k11(x)$	k	$z_{i+1}p_ip_{i-1}$
$0^k10(1\ or\ 0)$	k	$z_{i+1}p_ip_{i-1}$
$0^k10^l(\bar{1})$	$k+1$	$z_{i+1}p_iz_{i-1}$*
$0^k1\bar{1}^l1(x)$	$k+l$	$\bar{z}_{i+1}n_ip_{i-1}$
$0^k1\bar{1}^l0(1\ or\ 0)$	$k+l$	$\bar{z}_{i+1}n_iz_{i-1}$
$0^k1\bar{1}^l0^m(\bar{1})$	$k+l+1$	$\bar{z}_{i+1}n_iz_{i-1}$*

* Correction needed

Figure 37 shows the pre-encoding logic which generates F vector. The F vector is passed to the LZD tree, which computes the shift amount for the normalization. The traditional LZD tree generates the shift amount from the LSB [31], which requires the normalization to start the shift after the LZA logic is completed. To reduce the delay, a new LZD tree is proposed. The new LZD tree generates the shift amount from the MSB so that the normalization shift is overlapped with the LZD by starting the shift from the MSB generated from the LZD [4]. Figure 38 shows the 25 bit LZD tree which is used for

the single precision fused floating-point add-subtract unit. The correction tree is performed in parallel with the LZD tree to adjust the possible 1 bit error. The traditional correction tree performs the two trees for positive and negative cases [19], which require redundant binary representations. To reduce the redundancy, a new correction tree with one side tree is proposed [21]. The new correction tree merges the two trees with the same level of gate delay and fewer gates. Thus, the new correction tree reduces the area and power consumption while it maintains a competitive delay. Figure 39 shows the new correction tree node. The correction bit is determined at the root of the tree by (a $\oplus$ b) $\oplus$ c and zero is determined by $\overline{b + c}$.

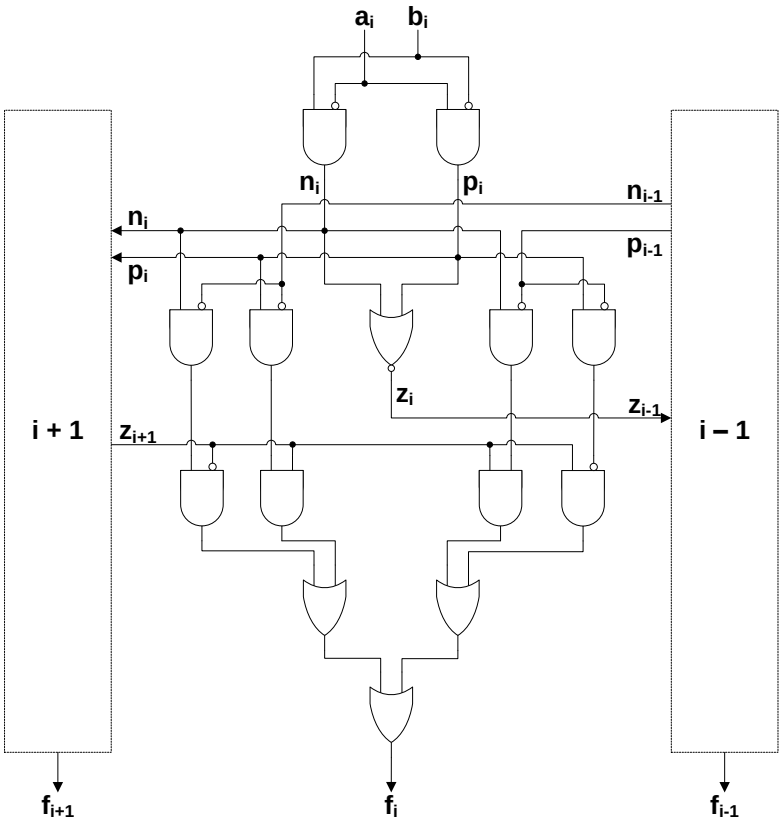


Figure 37. Pre-Encoding Logic of the LZA (After [19]).

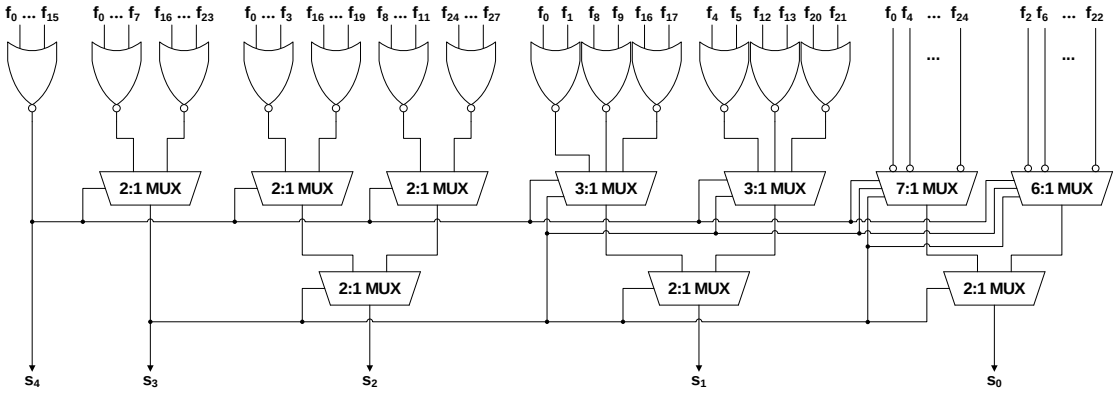


Figure 38. 25 bit Leading Zero Detection Tree (After [4]).

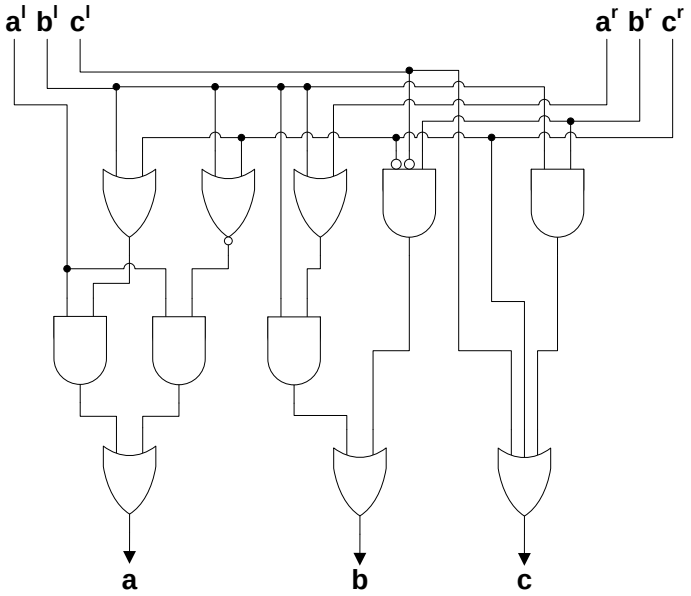


Figure 39. Correction Tree for the LZA with Concurrent Correction (After [21]).

3.2.2.6 Sign Logic

The sign logic for a dual-path fused floating-point add-subtract unit consists of two parts as shown in Figure 40. The first sign logic generates two sign bits of the addition and subtraction to be used for rounding in the far path and the second part

generates the sign bits of the sum and difference and an operation decision bit.

In case of the far path, 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 generated in the first sign logic are passed to the far path logic. The close path, however, requires a 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 significand comparison in the close path logic. The sign logic for sign bits and an operation decision bit are

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

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

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

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

$$op\_sel = \mathrm{A}_{sign} \oplus \mathrm{B}_{sign}.$$

The operation selection bit is passed to the two multiplexers for selecting the sum and difference.

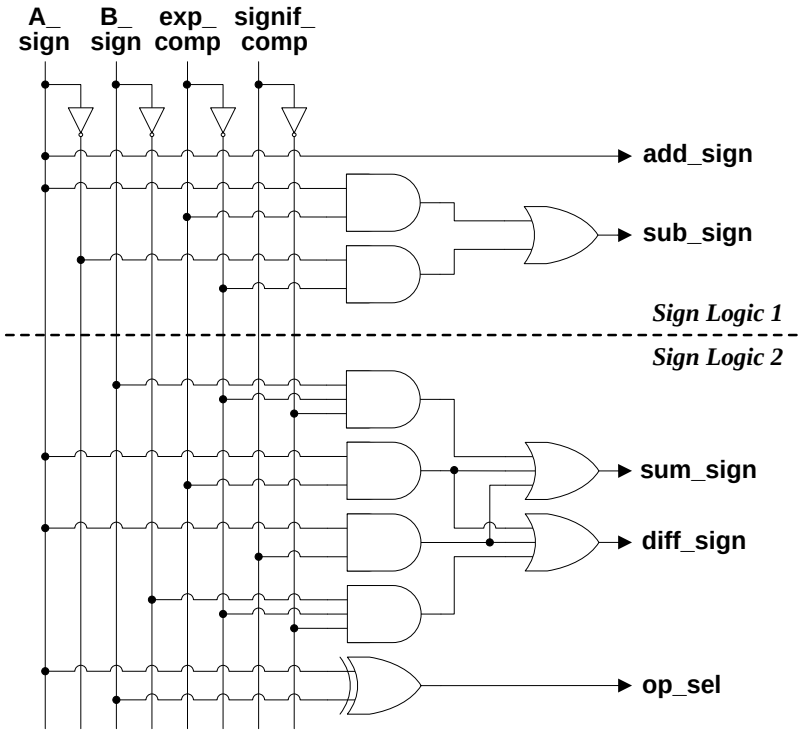


Figure 40. Sign Logic for a Dual-Path Fused Add-Subtract Unit [29].

3.2.2.7 Exponent Adjust Logic

The exponent adjust logic performs an addition and subtraction to adjust the exponents by the amount that the significands are shifted as shown in Figure 41. The exponent adjust logic produces two exponent results simultaneously. In the case of addition, one of the increment values between the far path and the close path 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 the 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 subtraction.

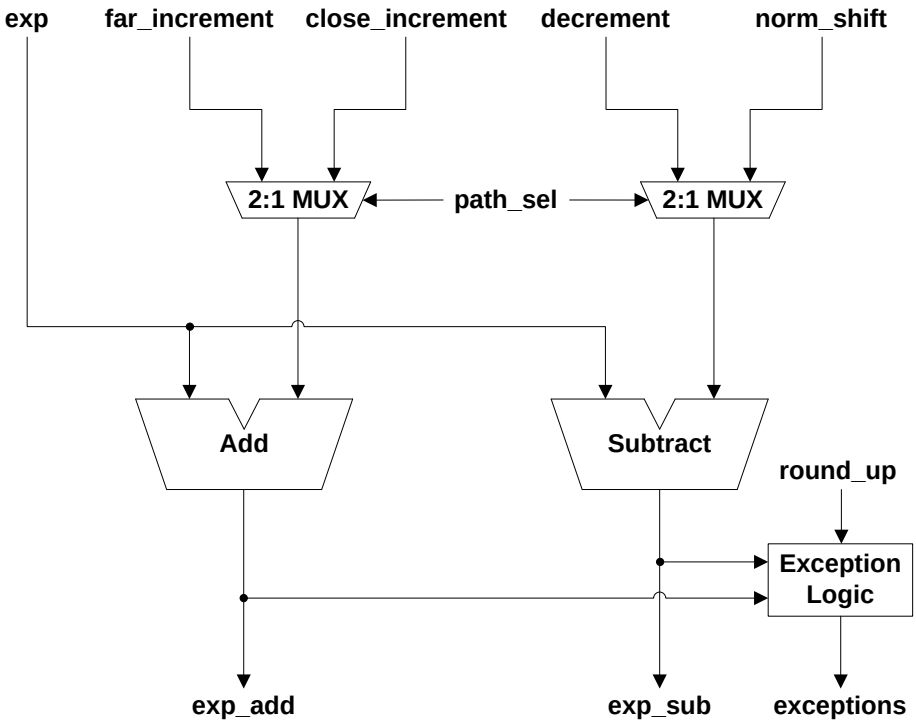


Figure 41. Exponent Adjust for a Dual-Path Fused Add-Subtract Unit [29].

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

$$overflow = \begin{cases} 1 & if\ exp \geq max\_exp \\ 0 & otherwise \end{cases}$$

$$underflow = \begin{cases} 1 & if\ exp \leq 0 \\ 0 & otherwise \end{cases}$$

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

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 [32]. The inexact flag is set if the rounded significand result is not exact, which is the case either of the rounding bit, the overflow flag or the underflow flag is set.

3.3 Pipelined Fused Floating-Point Add-Subtract Unit

As is well known, proper pipelining increases the throughput of floating-point adders [14] – [16], [33]. The floating-point adders can be split into two pipeline stages so that the results are produced every cycle. In the pipelined logic, the slowest stage latency determines the maximum throughput. If the stage latencies are not well balanced, the stages must wait until the slowest stage is completed, which increases the total logic delay. Therefore, it is important to properly arrange the logic components so that the latencies of the stages are well balanced. This section presents a data flow analysis to arrange the logic components of the fused floating-point add-subtract unit and to determine the composition of each pipeline stage. To achieve a high performance pipelined fused floating-point add-subtract unit, the high-speed dual-path design is used.

3.3.1 Data Flow Analysis

In order 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 45nm CMOS technology

standard-cell library. The latencies of the various elements of the single precision dual-path fused floating-point add-subtract unit are listed in Table 5.

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 dual-path fused floating-point add-subtract unit 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 42 shows the data flow and the critical path of the pipelined dual-path fused floating-point add-subtract unit.

Table 5. Component Latencies in a Dual-Path Fused Add-Subtract Unit [29].

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		

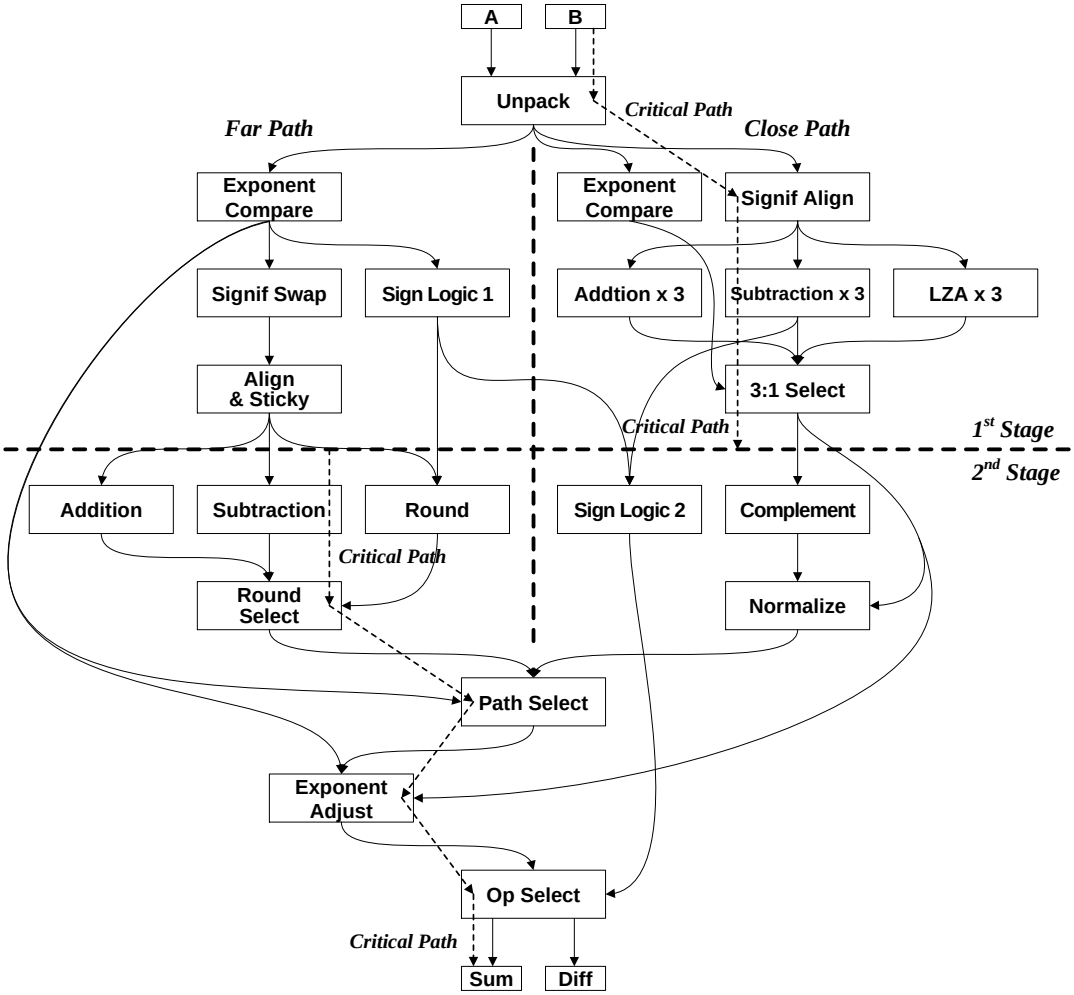


Figure 42. Data Flow of a Pipelined Dual-Path Fused Add-Subtract Unit [29].

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

Based on 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

3.3.2.1 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 29 and 30. 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.

3.3.2.2 The Second 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. 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.

3.4 Implementation and Results

Previous sections introduced the designs of the three advanced fused floating-point add-subtract units: 1) Enhanced fused floating-point add-subtract unit, 2) Dual-path fused floating-point add-subtract unit, and 3) Pipelined dual-path fused floating-point add-subtract unit. Each design is implemented for both single and double precision in Verilog-HDL and synthesized with the Nangate 45nm CMOS technology standard cell library. To evaluate the proposed designs, the logic area, critical path latency, throughput and, power consumption of the three implementations are compared with the discrete design and the traditional fused design as shown in Table 6. All the percentages in the table are ratios compared to the discrete design.

The traditional fused design reduces the area and power consumption by about 40% compared to the discrete design due to the shared logic. The enhanced fused design applies the new alignment and fast rounding schemes. As a result, it reduces the area and power consumption by 45% and reduces the latency by 8% compared to the discrete design. Since the dual-path designs execute two independent logic components including four additions and subtractions, they require more area and power consumption compared to the single-path designs. However, the dual-path designs skip the normalization in the far path, and the large significand alignment and rounding in the close path, respectively.

Table 6. Floating-Point Add-Subtract Unit Design Comparison [29].

Single Precision						
	Discrete	**Traditional Fused**	**Enhanced Fused**	**Low Power Dual-Path**	**High-Speed Dual-Path**	**High-Speed Dual-Path + Pipeline**
Area (μm^2)	15,403	9,605 (63%)	8,908 (58%)	9,876 (64%)	11,342 (74%)	13,497 (88%)
Latency (ns)	1.32	1.36 (103%)	1.21 (92%)	1.08 (82%)	0.92 (70%)	1.00 (76%)
Throughput (1/ns)	0.76	0.73 (97%)	0.83 (109%)	0.93 (122%)	1.09 (144%)	1.92 (254%)
Power (mW)	7.77	4.78 (62%)	4.21 (54%)	4.83 (62%)	4.91 (63%)	5.22 (67%)
Double Precision						
	Discrete	**Traditional Fused**	**Enhanced Fused**	**Low Power Dual-Path**	**High-Speed Dual-Path**	**High-Speed Dual-Path + Pipeline**
Area (μm^2)	34,606	20,017 (58%)	18,534 (54%)	20,522 (59%)	23,430 (68%)	27,586 (80%)
Latency (ns)	1.66	1.69 (102%)	1.52 (92%)	1.35 (81%)	1.12 (68%)	1.22 (74%)
Throughput (1/ns)	0.60	0.59 (98%)	0.66 (109%)	0.74 (123%)	0.89 (148%)	1.56 (259%)
Power (mW)	15.46	8.44 (55%)	8.17 (53%)	8.73 (56%)	9.03 (59%)	10.58 (68%)

As a result, the low power dual-path fused design reduces 20% of the critical path latency compared to the discrete design with a minimum increased area and power consumption. The high-speed dual-path fused design is more optimized to improve the performance so that it reduces latency by 30% compared to the discrete design.

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 [23] logarithmically increases the latency, the latency for the double precision increases by only 20%. The benefits of the new alignment, fast rounding schemes and dual-path algorithm are shown in both single and double precision.

The proposed pipelined dual-path fused floating-point add-subtract unit contains two stages. Each pipeline 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 7. 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 Stages for a Dual-Path Fused Add-Subtract Unit [29].

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

Chapter 4

Improved Architectures for a Fused Floating-Point Two-Term Dot Product Unit

In this chapter, improved architecture designs and implementation details for a fused floating-point two-term dot product unit are presented. The fused floating-point two-term dot product unit is useful for many digital signal processing (DSP) applications [7], [11], [12]. Therefore, the improved fused floating-point two-term dot product unit will contribute to the next generation of floating-point unit designs and DSP application development.

The proposed fused floating-point two-term dot product unit takes four normalized operands and computes the sum or difference of the two products as

$$P = AB \pm CD.$$

It supports all five rounding modes specified in the IEEE-754 Standard [1]. Several algorithms and optimization techniques are applied not only to improve the performance but also to reduce the area and power consumption:

1) For fast alignment, a new alignment scheme is proposed. By swapping the significands and shifting only the smaller significands, the shift amount is reduced so that the area and latency are reduced.
2) Early normalization is applied, which was proposed to reduce the latency of the fused floating-point multiply-add unit [4]. By performing the

normalization prior to the addition, the length of significands can be reduced using sticky logic, reducing the addition size by half. The sign is also determined prior to the addition so that the addition and rounding can be performed together, which significantly reduces the latency.

3) Since the normalization is performed prior to the addition, the leading zero anticipation (LZA) and normalization shift are on the critical path. In order to reduce the latency, a four-input LZA is proposed, which hides the delay of the 4:2 reduction trees.
4) The dual-path algorithm is applied to improve the performance. The dual-path logic consists of a far path and close path. Based on the exponent difference, a path is selected. In the far path, massive cancellation does not occur so that LZA and normalization are unnecessary. In the close path, only a two bit significand alignment is required so that the large significand shifter is unnecessary. By removing the unnecessary logic in each path, the latency is reduced.
5) In order to increase the throughput, pipelining can be applied. Based on the data flow analysis, the proposed dual-path fused floating-point two-term dot product unit is split into three stages. Since the latencies of three stages are fairly well balanced, the throughput is improved.

4.1 Enhanced Fused Floating-Point Two-Term Dot Product Unit

The traditional fused floating-point two-term dot product unit reduces the area, latency and power consumption compared to the discrete floating-point two-term dot product unit by sharing the common logic [7], [8]. However, it is an initial design so that optimizations can be applied to improve the performance [34]. Figure 43 shows the modified design for the enhanced fused floating-point two-term dot product unit. In this section, three optimizations for the enhanced fused floating-point two-term dot product unit are introduced: 1) New alignment scheme, 2) Early normalization and fast rounding and 3) Four-input LZA.

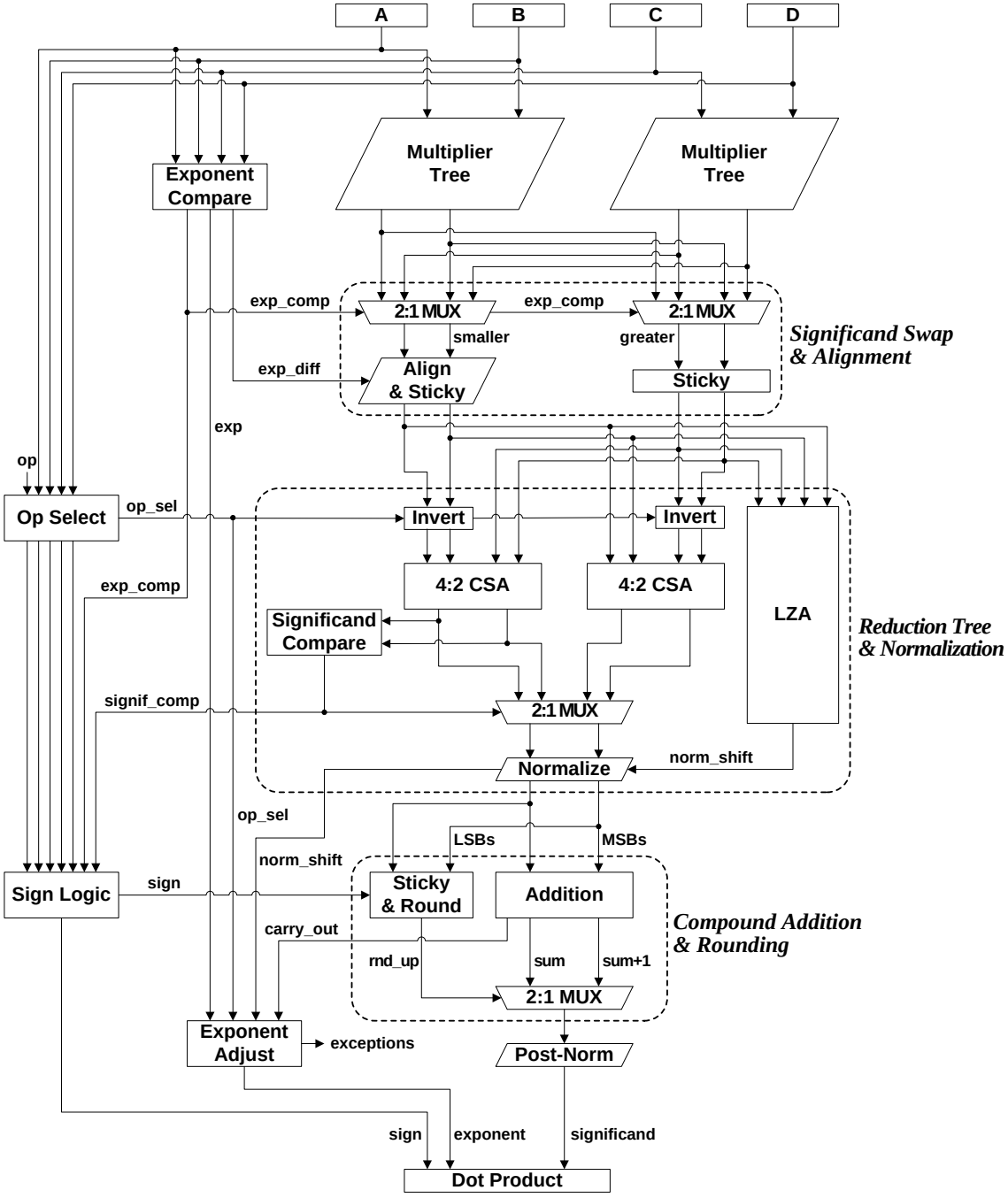


Figure 43. Enhanced Fused Floating-Point Two-Term Dot Product Unit [34].

4.1.1 New Alignment Scheme

The traditional fused floating-point two-term dot product unit performs the significand alignment on a single side significand pair (sum and carry) as shown in Figure 44. The one way alignment requires a large shift amount, which increases the latency of the critical path.

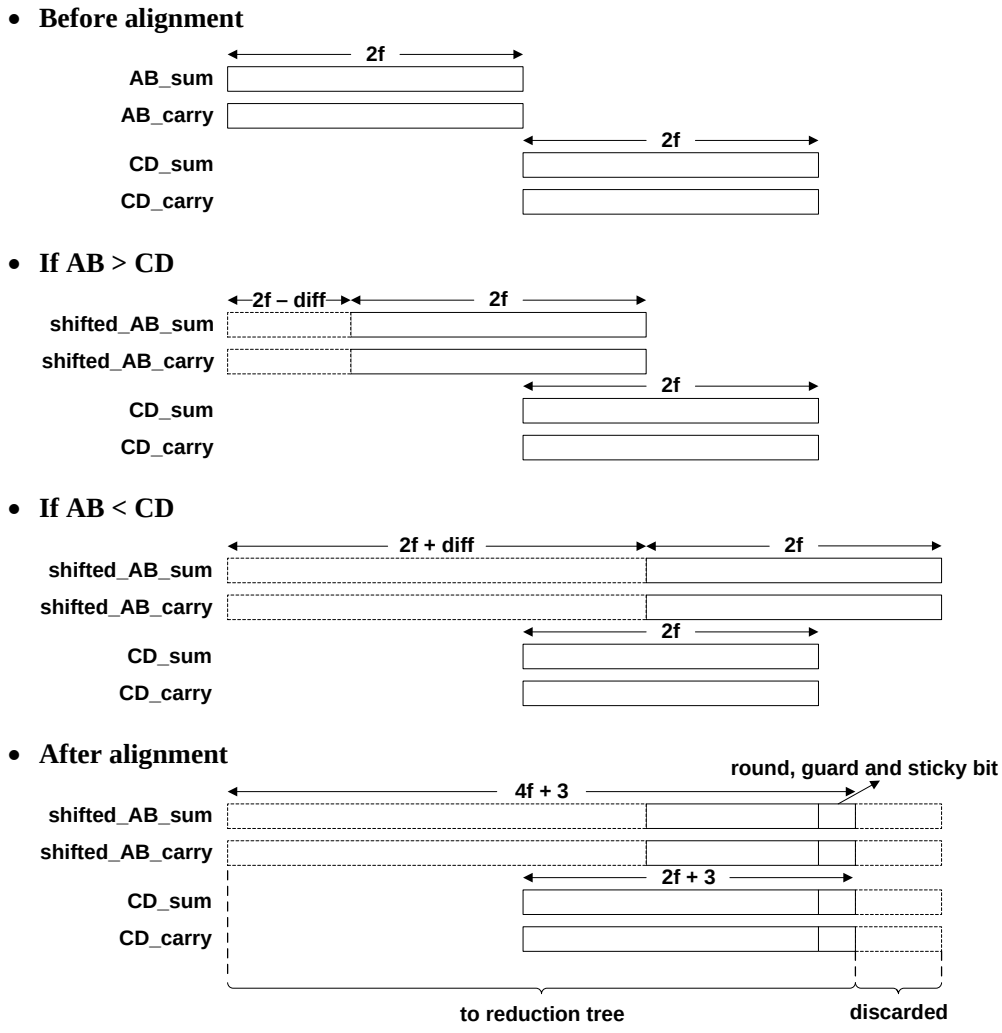


Figure 44. Traditional Alignment Scheme for a Fused Two-Term Dot Product Unit [34].

In order to reduce the latency of the alignment, the new alignment scheme swaps the significands to shift the smaller significand pair so that the shift amount is reduced as shown in Figure 45. Also, the sticky logic is performed to generate the round, guard and sticky bits. Then, the LSBs under the sticky bit can be discarded so that the length of the significand pairs is reduced. If the exponent difference is larger than 2, massive cancellation does not occur so that the discarded bits are not affected by the normalization. If the exponent difference is 2 or less, the shifted bits are maintained by the round, guard and sticky bits. Due to the reduced shift amount and sticky logic, smaller significand pairs are generated compared to the traditional alignment, resulting in reduced area and power consumption for the following logic.

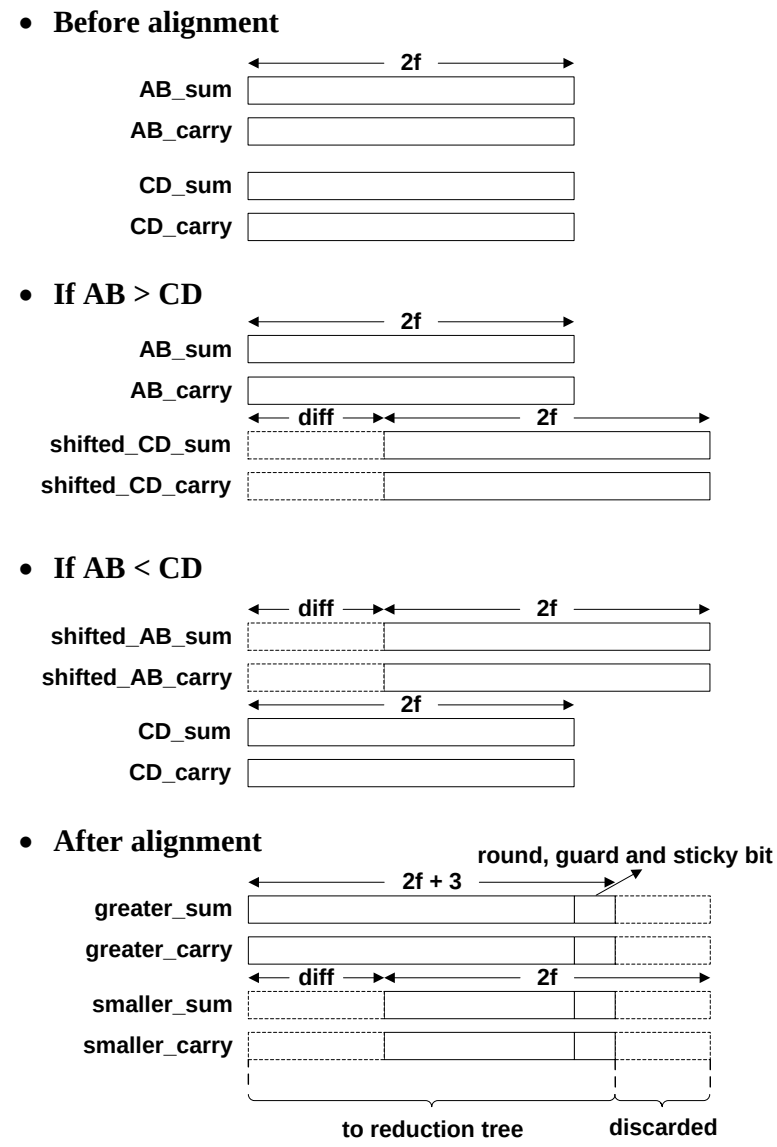


Figure 45. New Alignment Scheme for a Fused Two-Term Dot Product Unit [34].

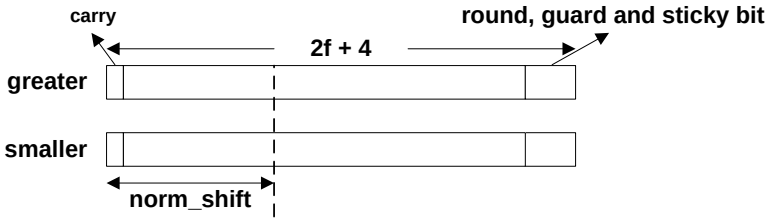
4.1.2 Early Normalization and Fast Rounding Scheme

If the effective operation is subtraction, the smaller significand pair is required to be inverted to be subtracted. If the exponent difference is 0, however, the smaller significand pair is not determined, so the two significand reduction trees are used and the inverted and non-inverted significand pairs are passed to each 4:2 reduction tree. The two reduction trees generate two significand pairs and one of the pairs is passed to the significand compare logic. The two significands are compared and the comparison result selects the one so that the significands do not need to be complemented after the significand addition. Also, the significand comparison result is used in the sign logic.

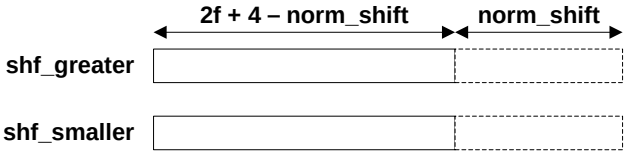
The reduced significand pair is passed to the normalization. The traditional fused floating-point two-term dot product unit performs the normalization after the significand addition, which requires a large significand adder and compliment followed by the round logic. For fast significand addition and rounding, early normalization is applied, which was previously proposed for the fused multiply-add unit [4]. By normalizing the significands prior to the significand addition, f + 1 bits are used for the significand and the round logic can be performed in parallel, where f is the number of significand bits. Figure 46 shows the early normalization and sticky logic. The MSBs of the normalized significands are passed to the addition and the LSBs are passed to the sticky and round logic. The sticky logic is performed again to generate round, guard and sticky bits. The first and second bits under the LSB become the guard and round bits and the sticky bit is set if at least one bit of the rest of the LSBs is 1, which can be implemented with an OR

tree. The four bits including the LSB, guard, round and sticky bits are used for the round logic to simplify the round logic and the rest of the LSBs are discarded.

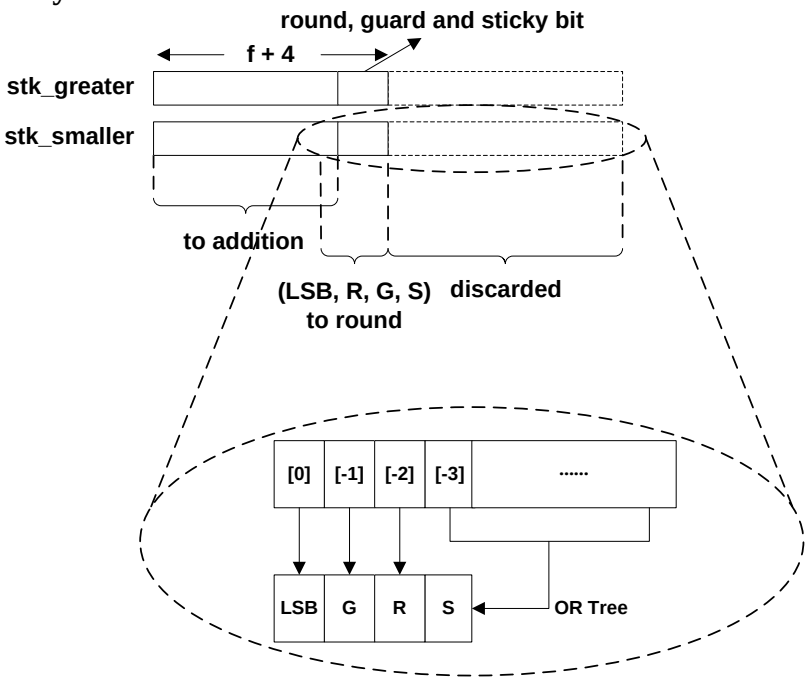
- **After reduction tree**



- **Normalization**



- **Sticky**



*** f = number of significand bits**

Figure 46. Early Normalization for a Fused Two-Term Dot Product Unit [34].

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 significand comparison result from the partial addition is used for the sign logic, if the exponent difference is zero. The sign bit is passed to the final result as well as the round logic. For fast rounding, compound addition is used, which produces the rounded and unrounded sums together and the round logic selects the correct result. By performing the significand addition and rounding together, the latency is significantly reduced.

4.1.3 Four-Input LZA

Since the normalization is performed prior to the significand addition, the LZA and normalization is placed on the critical path. To use the traditional two-input LZA for the fused floating-point two-term dot product unit, a 4:2 reduction tree is required prior to the LZA. The four-input LZA reduces the overhead of the reduction tree by encoding the four inputs at once. Figure 47 shows the comparison of the two-input LZA. By encoding the four inputs at once, four-input LZA hides the delay of the 4:2 reduction, which significantly reduces latency.

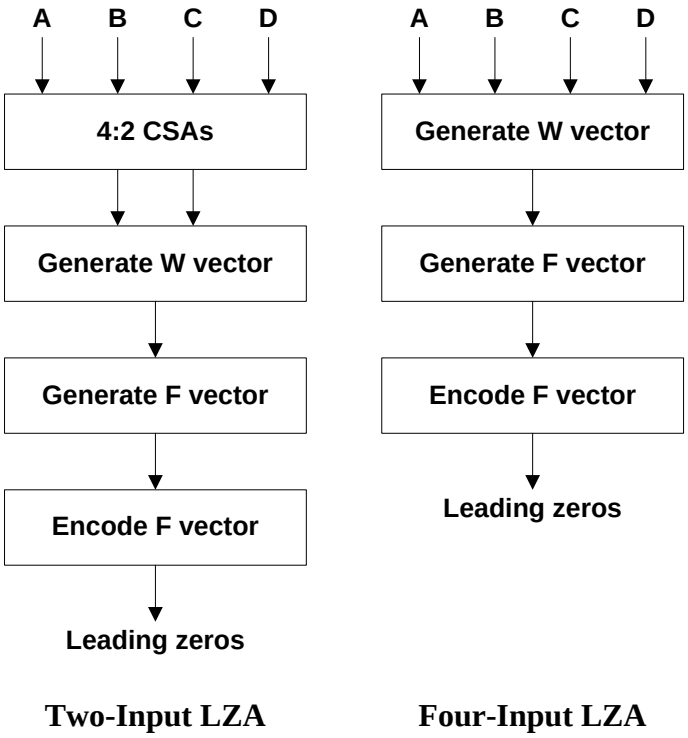


Figure 47. Two-Input LZA and Four-Input LZA Comparison.

The four-input LZA can be implemented by extending the traditional two-input LZA [19]. In order to encode four inputs, the W vector is generated with bitwise operations as

$$W = A + B - C - D$$

$$w_i = a_i + b_i - c_i - d_i, \qquad w_i \in \{-2, -1, 0, 1, 2\},$$

where a_i, b_i, c_i, d_i are the i^{th} bits from the MSB of the four significands. The W vector can be represented by one of the five elements, $\bar{2}_i, \bar{1}_i, 0_i, 1_i$ and 2_i, indicating that w_i is equal to –2, –1, 0, 1 and 2, respectively. The W vector is pre-encoded into three symbols, p_i, z_i and n_i as

$$p_i = 1 \; if \; w_i = 1$$

$$z_i = 1 \; if \; w_i = 0$$

$$n_i = 1 \; if \; w_i = \bar{1}.$$

To handle the cases if w_i is equal to –2 or 2, two consecutive bits are involved for pre-encoding. For example, bit pattern 0_i2_{i-1} and $1_i\bar{2}_{i-1}$ are considered as 1_i0_{i-1} and 0_i0_{i-1}, respectively. Thus, the three symbols are represented as

$$p_i = 2_i(2_{i-1} + \bar{2}_{i-1}) + 1_i(1_{i-1} + 0_{i-1} + \bar{1}_{i-1}) + 0_i2_{i-1}$$

$$\begin{aligned} z_i = {} & 2_i(1_{i-1} + 0_{i-1} + \bar{1}_{i-1}) + 1_i(2_{i-1} + \bar{2}_{i-1}) \\ & + 0_i(1_{i-1} + 0_{i-1} + \bar{1}_{i-1}) + \bar{1}_i(2_{i-1} + \bar{2}_{i-1}) \\ & + \bar{2}_i(1_{i-1} + 0_{i-1} + \bar{1}_{i-1}) \end{aligned}$$

$$n_i = 0_i\bar{2}_{i+1} + \bar{1}_i(1_{i+1} + 0_{i+1} + \bar{1}_{i+1}) + \bar{2}_i(2_{i+1} + \bar{2}_{i+1}).$$

The pre-encoding patterns that terminate the leading zeros and the corresponding leading zeros for W > 0 are shown in Table 8. The number of leading zeros is computed with the three symbols as

$$f_i(pos) = z_{i+1}p_i\bar{n}_{i-1} + \bar{z}_{i+1}n_i\bar{n}_{i-1} \text{ for } W > 0$$

Similarly, for the bit patterns when W < 0,

$$f_i(neg) = z_{i+1}n_i\bar{p}_{i-1} + \bar{z}_{i+1}p_i\bar{p}_{i-1} \text{ for } W < 0.$$

Combining the two equations, the F vector is generated as

$$f_i = z_{i+1}(p_i\overline{n}_{i-1} + n_i\overline{p}_{i-1}) + \overline{z}_{i+1}(p_i\overline{p}_{i-1} + n_i\overline{n}_{i-1}).$$

This is essentially the same equation as that of the traditional two-input LZA [19]. The F vector is encoded with the leading zero detector (LZD) to obtain the leading zeros, which is the shift amount of the normalization. For fast normalization, the MSBs of the shift

amount are generated so that the LZD tree and the normalization shifter are overlapped [4].

Table 8. LZA Pre-Encoding Patterns for W > 0 [19].

W vector	Leading Zeros	Pre-encoding Pattern
$0^k11(x)$	k	$z_{i+1}p_ip_{i-1}$
$0^k10(1\ or\ 0)$	k	$z_{i+1}p_ip_{i-1}$
$0^k10^l(\bar{1})$	$k+1$	$z_{i+1}p_iz_{i-1}$*
$0^k1\bar{1}^l1(x)$	$k+l$	$\bar{z}_{i+1}n_ip_{i-1}$
$0^k1\bar{1}^l0(1\ or\ 0)$	$k+l$	$\bar{z}_{i+1}n_iz_{i-1}$
$0^k1\bar{1}^l0^m(\bar{1})$	$k+l+1$	$\bar{z}_{i+1}n_iz_{i-1}$*

* Correction needed

Like most of the two-input LZAs that are inexact due to a possible 1 bit error, the proposed four-input LZA also requires correction logic. For fast error detection and correction, concurrent error correction logic can be used, which was previously proposed [19] – [21][5]. In the cases of the bit patterns[6] $0^k10^l\bar{1}$ and $0^k1\bar{1}0^l\bar{1}$ for W > 0 and $0^k\bar{1}0^l1$ and $0^k\bar{1}10^l1$ for W < 0, correction is required by adding 1. More details on the correction logic are described in Section 3.2.2.5.

[5] The error correction logic in [19] is modified by [20] and [21] to improve the accuracy and eliminate the redundancy, respectively.

[6] The notation x^k denotes a bit string of k consecutive bits, where $x \in \{\bar{1}, 0, 1\}$ and $k \geq 0$.

4.2 Dual-Path Fused Floating-Point Two-Term Dot Product Unit

In order to further improve the performance of the fused floating-point two-term dot product unit, the dual-path algorithm is applied. The dual-path fused floating-point two-term dot product unit consists of a far path and close path as shown in Figure 48. The path is determined based on the exponent difference. The far path skips the LZA and normalization and the close path skips the significand swap and alignment. Since these two processes are the bottlenecks of the traditional fused floating-point two-term dot product unit, the dual-path algorithm improves the performance. In this section, the dual-path design for the fused floating-point two-term dot product unit is introduced.

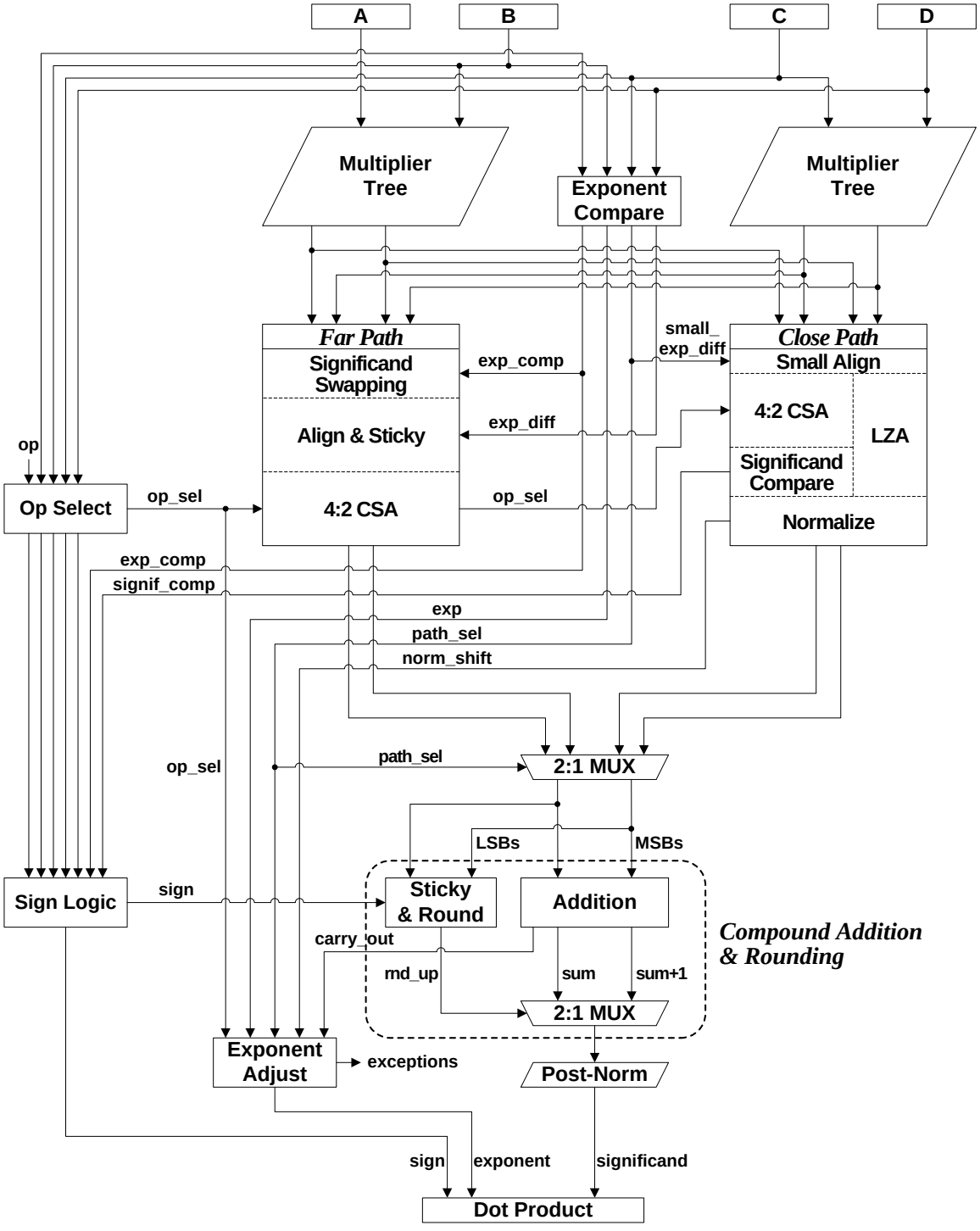


Figure 48. Dual-Path Fused Floating-Point Two-Term Dot Product Unit [34].

4.2.1 Far Path Logic

The far path logic for the dual-path fused floating-point two-term dot product unit can be implemented as the significand swap and alignment part of the enhanced fused floating-point two-term dot product unit as shown in Figure 49. The far path is selected if the exponent difference is larger than 2 or the operation is addition. Since the addition of four significands generates a carry out of up to 3, the exponent difference margin for the far path is two bits, which is 1 bit larger than that of the general dual-path floating-point adder. In this case, massive cancellation during the subtraction does not occur so that the LZA and normalization are unnecessary. Two multiplexers are used to swap the significand pairs so that only the smaller significand pair is aligned, which reduces the shift amount. The aligned significand pair is inverted if the operation is subtraction. The sticky logic is performed for both significand pairs to reduce the significand length. The significand pair for the far path is generated by the reduction tree, which reduces the four significands to two.

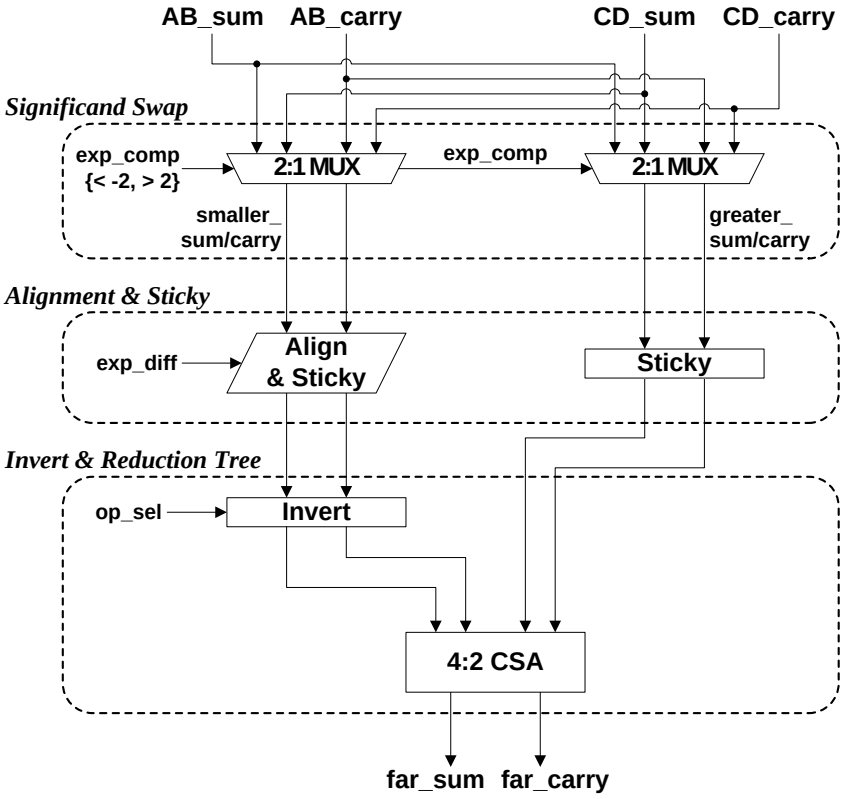


Figure 49. Far Path for a Dual-Path Fused Two-Term Dot Product Unit [34].

4.2.2 Close Path Logic

The close path is selected if the exponent difference is less than 3 and the operation is subtraction. In this case, only a two bit shifter is required for the significand alignment. The significand pairs are aligned as

$$AB_{aligned} = \begin{cases} (AB_{signif}, 00) & if\ AB_{exp} - CD_{exp} = 0, 1, 2 \\ (0, AB_{signif}, 0) & if\ AB_{exp} - CD_{exp} = -1 \\ (00, AB_{signif}) & if\ AB_{exp} - CD_{exp} = -2 \end{cases}$$

$$CD_{aligned} = \begin{cases} (CD_{signif}, 00) & if\ AB_{exp} - CD_{exp} = 2 \\ (0, CD_{signif}, 0) & if\ AB_{exp} - CD_{exp} = 1 \\ (00, CD_{signif}) & if\ AB_{exp} - CD_{exp} = 0, -1, -2. \end{cases}$$

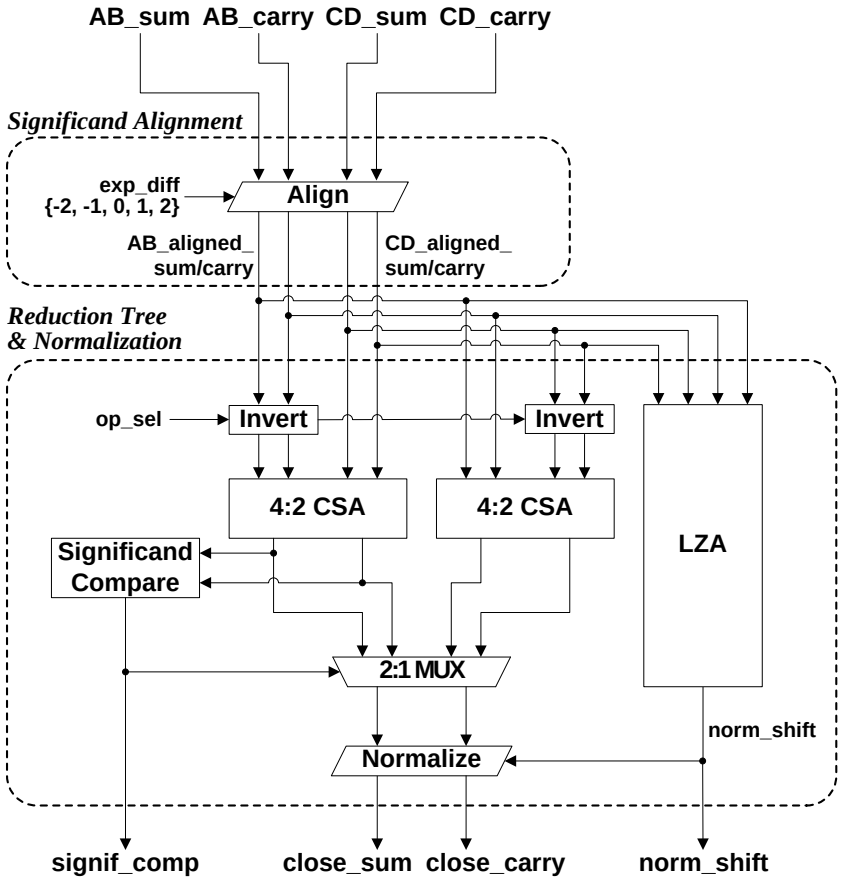


Figure 50. Close Path for a Dual-Path Fused Two-Term Dot Product Unit [34].

The rest of the close path logic can be implemented as the partial addition and normalization part of the enhanced fused floating-point two-term dot product unit as shown in Figure 50.

4.2.3 The Other Sub-Logic

Among the two significand pairs from the far path and close path, a significand pair is selected based on the exponent difference and the operation. The selected significand pair is passed to the significand addition and round logic. The significand

addition and round logic can be implemented same as the enhanced fused floating-point two-term dot product unit, which is described in the previous section. This section contains the rest of sub-logic designs for the dual-path fused floating-point two-term dot product unit: 1) Exponent compare logic, 2) Operation select logic, 3) Multiplier trees, 4) Significand reduction trees, 5) Sign logic and 6) Exponent adjust logic.

4.2.3.1 Exponent Compare Logic

The exponent compare and path select logic are shown in Figure 51. For the exponent process, two pairs of exponents are summed and a greater exponent sum is selected. Then, the bias is subtracted for the exponent result. The two exponent sums are compared to determine the greater one.

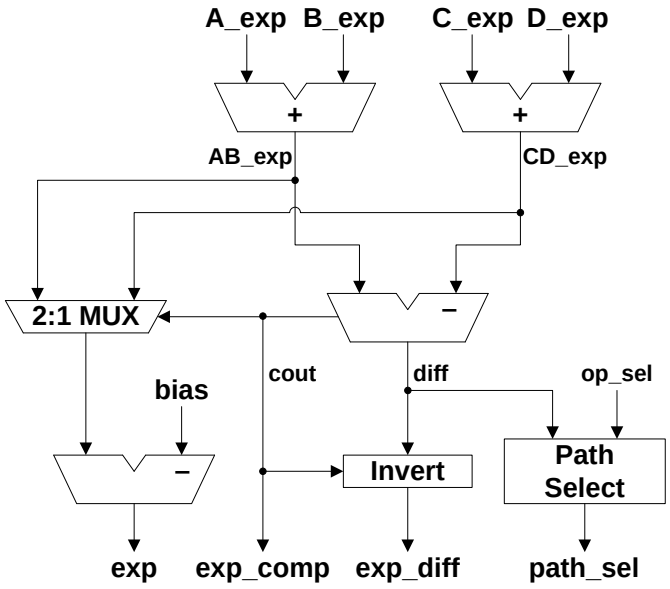


Figure 51. Exponent Compare for a Dual-Path Fused Two-Term Dot Product Unit [34].

The exponent comparison result is used for the significand swapping and the exponent difference is used for the alignment. Also, the path selection bit is determined based on the exponent difference and the operation as

$$path_{sel} = \begin{cases} 1 & if\ |AB_{exp} - CD_{exp}| \leq 2\ \ or\ \ op\_sel = 0 \\ 0 & otherwise. \end{cases}$$

4.2.3.2 Operation Select Logic

The operation select logic generates the effective operation, *op_sel* bit, which determines if the significands are inverted for the significand subtraction. Using the four sign bits and the input op code, the operation is selected as

$$op\_sel = \begin{cases} AB_{sign} \oplus CD_{sign} & if\ op = add \\ \overline{AB_{sign} \oplus CD_{sign}} & if\ op = sub, \end{cases}$$

where AB_{sign} *is* $A_{sign} \oplus B_{sign}$ and CD_{sign} *is* $C_{sign} \oplus D_{sign}$.

4.2.3.3 Multiplier Trees

Two multiplier trees are used for computing a part of the significand multiplication. Each multiplier tree takes two significands and generates a sum and carry pair using reduction tree. A simple partial product generation and a Dadda reduction tree, which is known as one of the fastest algorithms [26], is used for the significand multiplier trees. Figure 52 shows the dot-diagram of the 24 bit Dadda multiplier tree, which is used for the single precision fused floating-point two-term dot product unit. The Dadda multiplier tree uses 7 layers of reductions using full-/half-adders to generate a sum and

carry pair. The sum and carry pairs from the two multiplier trees are passed to the two multiplexers to determine the greater significands based on the exponent comparison. Then, the two sum and carry pairs are passed to the alignment and sticky logic. The new alignment scheme and sticky logic described in a previous section is used.

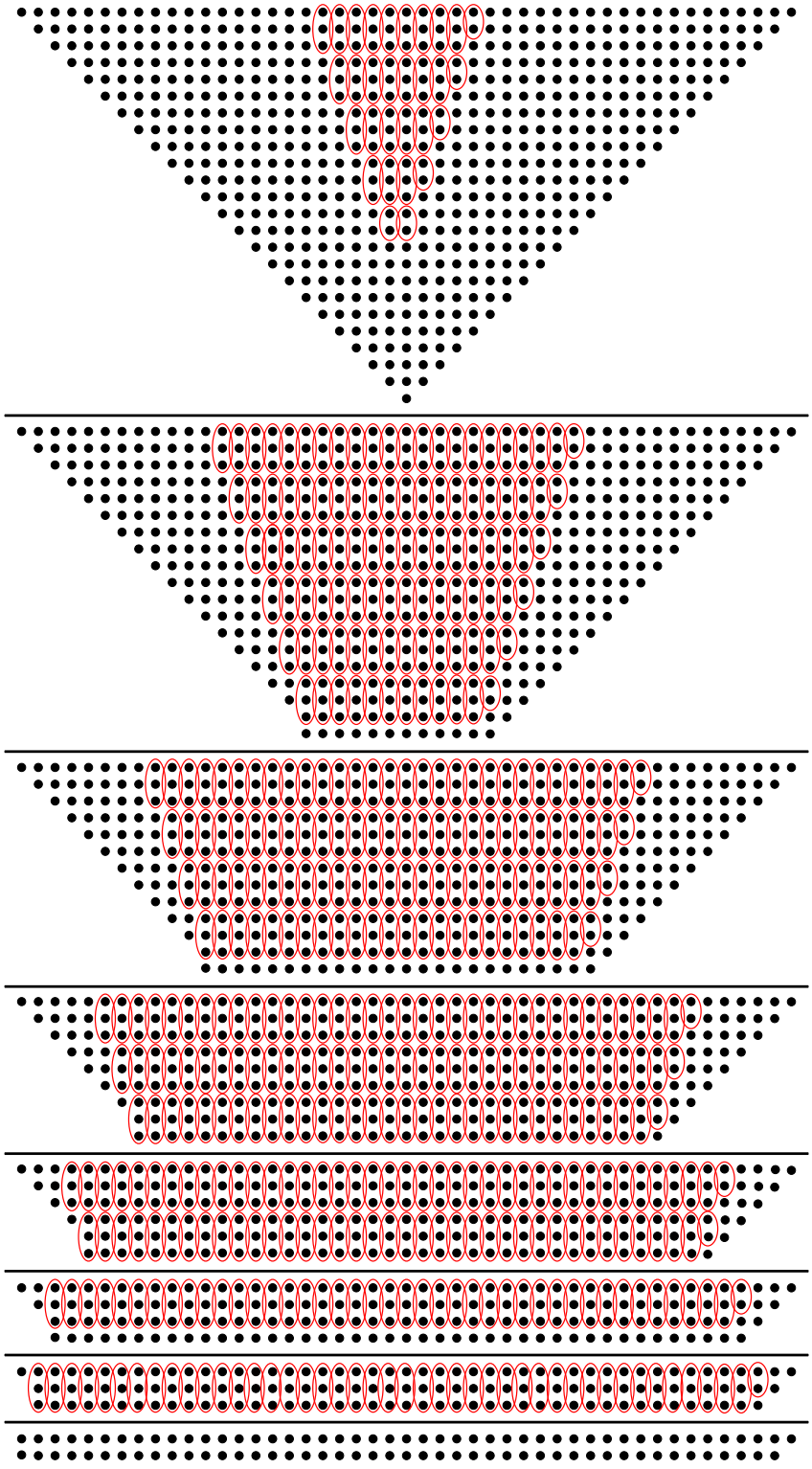


Figure 52. 24 bit Dadda Multiplier Tree.

4.2.3.4 Significand Reduction Trees

The aligned sum and carry pairs are passed to the 4:2 significand reduction trees. The two reduction trees are used for the early normalization and fast rounding as described in the previous section. Figure 53 shows the 50 bit 4:2 reduction tree using a carry save adder (CSA), which is used for the single precision fused floating-point two-term dot product unit. The 4:2 CSA takes four significands and uses two layers of reductions using full-/half-adders to generate a significand pair. The two significand pairs from the two reduction trees are passed to the multiplexer to determine the correct one based on the significand comparison.

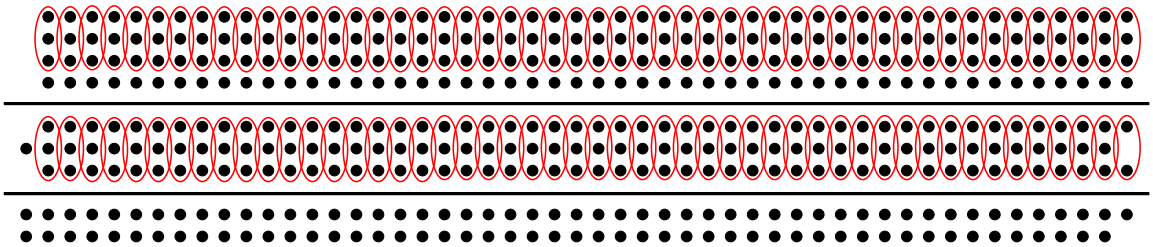


Figure 53. 50 bit 4:2 Reduction Tree using the Carry Save Adder (CSA).

4.2.3.5 Sign Logic

The sign logic determines the final sign bit that is also used for the round logic. The four sign bits of the operands, the input op code, the exponent comparison and the significand comparison are used to determine the sign bit as

$$sign = \begin{cases} AB_{sign}CD_{sign} + AB_{sign}comp_{exp} \\ \quad + AB_{sign}comp_{signif} \\ \quad + CD_{sign}\overline{comp}_{exp}\overline{comp}_{signif} & if\ op = add \\ AB_{sign}\overline{CD}_{sign} + AB_{sign}comp_{exp} \\ \quad + AB_{sign}comp_{signif} \\ \quad + \overline{CD}_{sign}\overline{comp}_{exp}\overline{comp}_{signif} & if\ op = sub. \end{cases}$$

4.2.3.6 Exponent Adjust Logic

Figure 54 shows the exponent adjust logic, which adjusts the exponent by adding or subtracting the carry-out from the significand addition. Since the four significands generate a carry-out of up to 3, two carry out bits are used for the adjustment. The normalization shift amount is subtracted in the case of massive cancellation. Using the selection bits and the carry-outs from the addition and subtraction, the exceptions are detected. The three exception cases specified in IEEE-754 Standard [1] are detected as

$$overflow = \begin{cases} 1 & if\ exp \geq max\_exp \\ 0 & otherwise \end{cases}$$

$$underflow = \begin{cases} 1 & if\ exp \leq 0 \\ 0 & otherwise \end{cases}$$

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

where *round_up* is the rounding decision of the significand result.

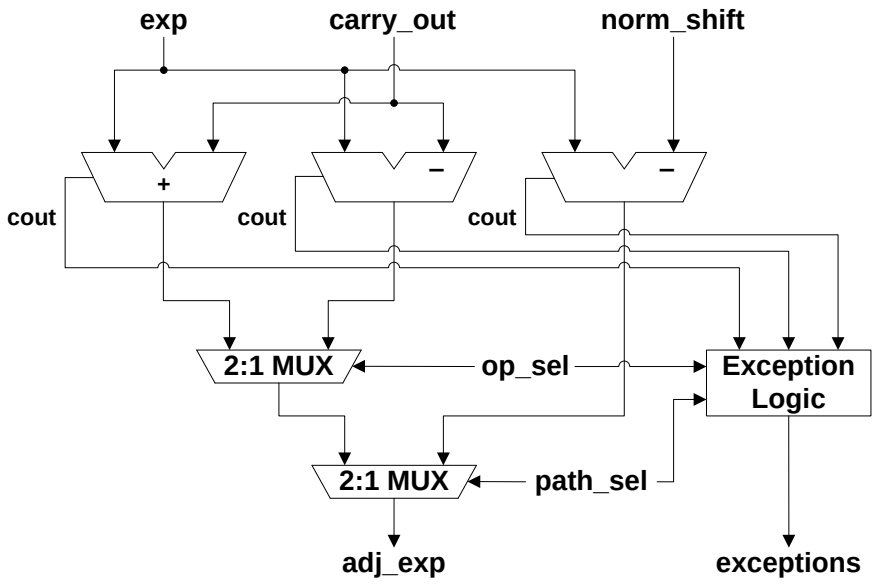


Figure 54. Exponent Adjust for a Dual-Path Fused Two-Term Dot Product Unit [34].

4.3 Pipelined Fused Floating-Point Two-Term Dot Product Unit

Pipelining is applied to improve the throughput of the fused floating-point two-term dot product unit. The proposed dual-path fused floating-point two-term dot product unit can be split into three pipeline stages so that the results are produced every cycle. The throughput of the pipelined logic is determined by the slowest pipeline stage. Therefore, it is important to properly arrange the logic components so that the latencies of the stages are well balanced. In this section, the data flow analysis to arrange the logic components for the proposed dual-path fused floating-point two-term dot product unit and the composition of pipeline stages is presented.

4.3.1 Data Flow Analysis

In order to achieve the proper pipelining for the fused floating-point two-term dot product unit, the arrangement of the components is investigated. Each component is implemented in Verilog-HDL and synthesized with the Nangate 45nm CMOS technology standard-cell library. The latencies of the various elements of the single precision dual-path fused floating-point two-term dot product unit are listed in Table 9.

Figure 55 shows the data flow and critical path of the dual-path fused floating-point two-term dot product unit. 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 dual-path fused floating-point two-term dot product unit is split into three pipeline stages. Each pipeline stage is executed every cycle so that the largest latency determines the throughput of the design.

Table 9. Component Latencies in a Dual-Path Fused Two-Term Dot Product Unit [34].

Components	Latency (ns)	Components	Latency (ns)
Unpack	0.02	Exponent Compare	0.27
Op Select	0.08	Multiplier Trees	0.59
Significand Swap	0.09	Small Signif. Align	0.12
Align & Sticky	0.24	Invert	0.02
4:2 CSAs	0.16	LZA	0.37
Significand Compare	0.14	2:1 Select	0.04
Normalization	0.14	Path Select	0.04
Significand Addition	0.33	Sign Logic	0.06
Round Select	0.04	Sticky & Round	0.16
Exponent Adjust	0.22	Post Normalization	0.08

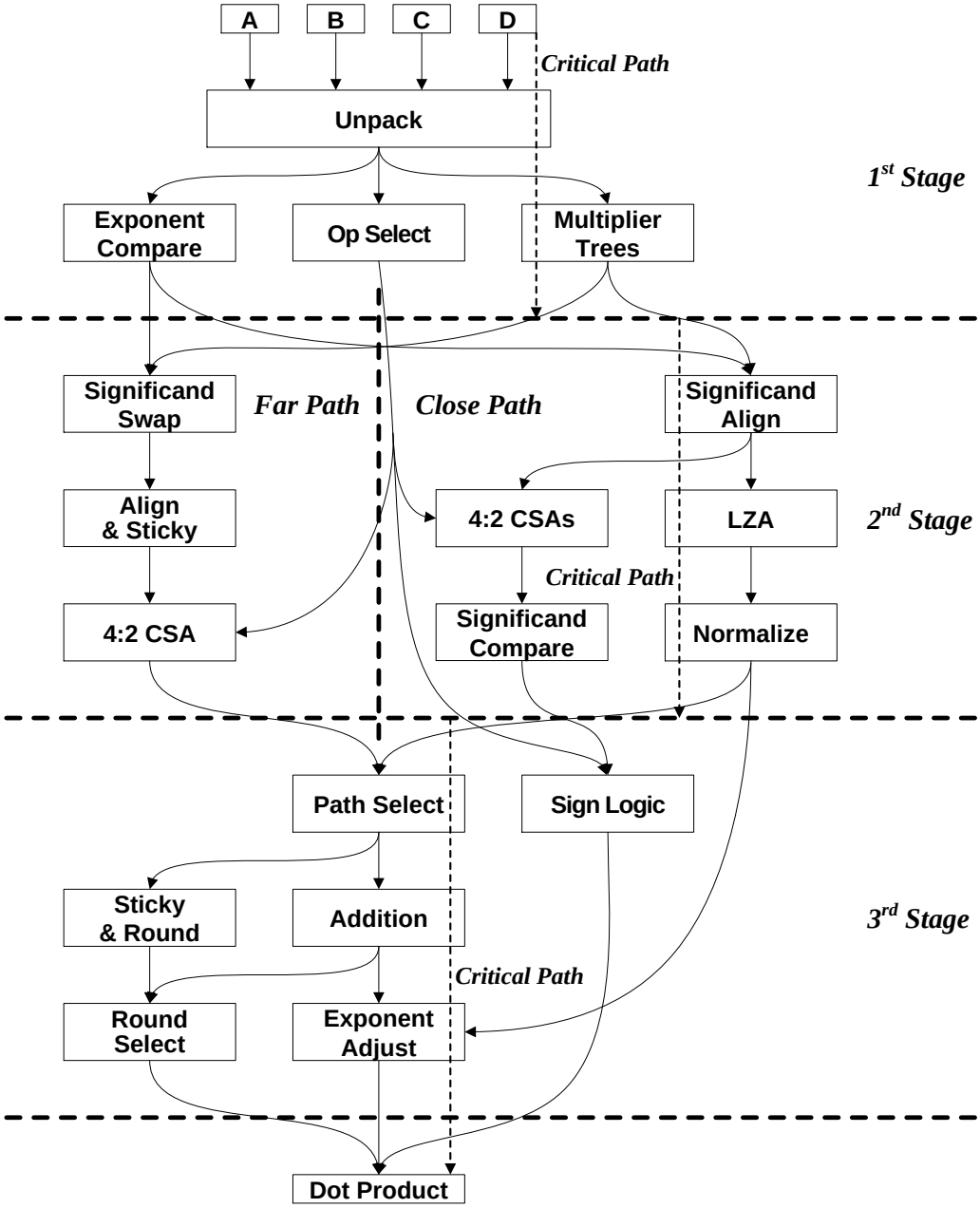


Figure 55. Data Flow of a Pipelined Dual-Path Fused Dot Product Unit [34].

4.3.2 Pipeline Stages of a Dual-Path Fused Floating-Point Dot Product Unit

Based on the data flow analysis, the proposed dual-path fused floating-point two-term dot product unit can be split into three stages. The critical paths of the three pipeline stages are

First stage: Unpack → Multipliers trees

Second stage: Close path significand alignment → LZA → Normalization

Third stage: Path select → Significand addition → Exponent adjust.

4.3.2.1 The First Pipeline Stage

The first pipeline stage contains unpacking, exponent compare logic and multiplier trees. Since the multiplier trees have a long latency, they occupy most of the latency of the first stage. Since the data path including multiplier trees has the largest latency, it becomes the critical path of the first pipeline stage.

4.3.2.2 The Second Pipeline Stage

The second pipeline stage is the dual-path logic which consists of the far path and close path logic as described in Figures 48 and 49. The far path contains the significand swap, significand align, sticky logic and 4:2 reduction tree. The close path logic contains the small significand align, 4:2 reduction tree, significand compare logic, four-input LZA and normalization. Since the close path logic takes a larger latency than the far path logic, it becomes the critical path which determines the latency of the second pipeline stage.

4.3.2.3 The Third Pipeline Stage

The third stage contains the path select, sign logic, significand addition, sticky logic, rounding, and exponent adjust logic. The data path including the path selection, significand addition and exponent adjust logic has the largest latency so that it determines the latency of the third pipeline stage.

In each pipeline stage, several logic components are performed in parallel and the path that takes the largest latency becomes the critical path. Since the second stage takes the largest latency among the three pipeline stages, the latency of the second stage becomes the effective latency which determines the throughput. Due to the latches and control signals between the pipeline stages, the total latency of the pipelined dual-path fused floating-point two-term dot product unit is three times the latency of the second stage. However, the latencies of the three pipeline stages are fairly well balanced so that the throughput is significantly increased compared to the non-pipelined dual-path design.

4.4 Implementation and Results

Previous sections introduced the designs of the three advanced fused floating-point two-term dot product units: 1) Enhanced fused floating-point two-term dot product unit, 2) Dual-path fused floating-point two-term dot product unit, and 3) Pipelined dual-path fused floating-point two-term dot product unit. Each design is implemented for both single and double precision in Verilog-HDL and synthesized with the Nangate 45nm CMOS technology standard cell library. To verify the improvement of the proposed

designs, the logic area, critical path latency, throughput and, power consumption of the three implementations are compared with the discrete design and the traditional fused design as shown in Table 10. All the percentages in the table are ratios compared to the discrete design.

Table 10. Floating-Point Two-Term Dot Product Unit Design Comparison [34].

Single Precision					
	Discrete	**Traditional Fused**	**Enhanced Fused**	**Enhanced + Dual-Path**	**Enhanced + Dual-Path + Pipeline**
Area (μm^2)	46,083 (100%)	38,654 (84%)	29,159 (63%)	31,472 (68%)	33,228 (72%)
Latency (ns)	2.58 (100%)	2.54 (98%)	2.14 (83%)	1.87 (72%)	2.01 (78%)
Throughput (1/ns)	0.39 (100%)	0.39 (102%)	0.47 (121%)	0.53 (138%)	1.49 (385%)
Power (mW)	25.40 (100%)	20.77 (82%)	15.17 (60%)	16.16 (64%)	16.94 (67%)
Double Precision					
	Discrete	**Traditional Fused**	**Enhanced Fused**	**Enhanced + Dual-Path**	**Enhanced + Dual-Path + Pipeline**
Area (μm^2)	110,087 (100%)	90,502 (82%)	67,317 (61%)	71,846 (65%)	74,545 (68%)
Latency (ns)	3.24 (100%)	3.18 (98%)	2.56 (79%)	2.21 (68%)	2.35 (73%)
Throughput (1/ns)	0.31 (100%)	0.31 (102%)	0.39 (127%)	0.45 (147%)	1.20 (390%)
Power (mW)	57.22 (100%)	46.33 (81%)	33.17 (58%)	35.09 (61%)	36.58 (64%)

The traditional fused design reduces the area and power consumption by about 20% compared to the discrete design and reduced the latency by 2%, since the fused design shares the logic such as significand addition and rounding. The enhanced fused floating-point dot product unit applies the new alignment scheme to reduce the shift amount. Early normalization is applied to reduce the size of the significand addition and perform the significand addition and rounding in parallel. Also, the four-input LZA reduces the latency by hiding the latency of the inverts and 4:2 reduction trees. As a result, the area and power consumption is reduced by approximately 40%, and the latency

is improved by 20% compared to the discrete design. The dual-path design requires about 5% more area and power consumption than that of the enhanced single path design due to the two path process. However, it eliminates the unnecessary logic in each path so that the latency of the critical path is improved by about 10% compared to the enhanced design.

The double precision implementation requires about twice as much area and power consumption as the single precision implementation due to the larger logic components. However, the tree structures are used for major components such as significand alignment, significand addition, LZA and normalization, which logarithmically increase the latency, the latency for the double precision increases by only 20%. The benefits of the new alignment scheme, early normalization, fast rounding, four-input LZA and dual-path algorithm are shown in both single and double precision.

The proposed pipelined dual-path fused floating-point two-term dot product unit is split into three stages. Table 11 shows the area, latency and power consumption of the three pipeline stages. Each pipeline stage requires latches to maintain the data and control signals between the stages, which increases the area, latency and power consumption. However, the latencies of the three pipeline stages are fairly well balanced so that the throughput is increased to about 2.8 times that of the non-pipelined design.

Table 11. Pipeline Stages for a Dual-Path Fused Two-Term Dot Product Unit [34].

Single Precision			
	Stage 1	**Stage 2**	**Stage 3**
Area (μm^2)	17,484 (53%)	12,143 (36%)	3,601 (11%)
Latency (ns)	0.65 (33%)	0.67 (35%)	0.63 (32%)
Power (mW)	8.96 (53%)	6.41 (38%)	1.57 (9%)
Double Precision			
	Stage 1	**Stage 2**	**Stage 3**
Area (μm^2)	41,293 (56%)	25,658 (34%)	7,503 (10%)
Latency (ns)	0.78 (33%)	0.81 (35%)	0.75 (32%)
Power (mW)	17.91 (56%)	11.42 (36%)	2.65 (8%)

Chapter 5

Improved Architectures for a Fused Floating-Point Three-Term Adder

This chapter presents improved architectures for a fused floating-point three-term adder. The floating-point addition is the most frequently used operation in many algorithms and applications. The floating-point multi-term adder is introduced to handle multiple operands in a single unit to improve the performance as well as the accuracy [9]. There are several issues on the design of the fused floating-point multi-term adder compared to the network of general floating-point two-term adders: 1) Complex exponent procedure, 2) Complement after the significand addition, 3) Large precision significand adder, and 4) Massive cancellation management. Those issues can be covered by investigating a fused floating-point three-term adder. The algorithms and optimizations described in this paper can be also extended to fused floating-point multi-term adders with more than three operands. Therefore, the improved fused floating-point three-term adder will contribute to the next generation floating-point arithmetic unit design.

The proposed fused floating-point three-term adder takes three normalized operands and executes two additions or subtractions as

$$S = A \pm B \pm C.$$

It supports all five rounding modes specified in the IEEE-754 Standard [1]. Several optimization techniques are applied not only to improve the performance but also to reduce the area and power consumption:

1) New exponent compare and significand alignment scheme is proposed. The three exponent differences are computed in parallel by performing the three subtractions. The control logic determines the max exponent and the shift amounts for the three significands. Then, the three significands are shifted by the amount of the corresponding exponent differences. This approach reduces the latency by generating the max exponent and the three shift amounts simultaneously.
2) Two 3:2 reduction trees are used to handle both the inverted and non-inverted significands. Between two significand pairs from the reduction trees, the positive significand pair is selected based on the significand comparison. Since the sum of the positive significand pair becomes positive, the complement after the significand addition can be skipped, which reduces the latency.
3) Early normalization is applied, which was proposed to reduce the latency of the fused floating-point multiply-add unit [4]. By performing the normalization prior to the addition, the length of the significands is reduced by the sticky logic, reducing the significand addition size by half. The sign is also determined prior to the addition so that the addition and rounding can be performed together, which significantly reduces the latency.
4) Since the normalization is performed prior to the addition, the leading zero anticipation (LZA) and normalization shift are on the critical path. In order to reduce the latency, a three-input LZA is proposed, which hides the delay of

the 3:2 reduction trees.

5) In order to increase the throughput, pipelining can be applied. Based on the data flow analysis, the proposed fused floating-point three-term adder is split into three stages. Since the latencies of three stages are fairly well balanced, the throughput is improved.

5.1 Enhanced Fused Floating-Point Three-Term Adder

The traditional fused floating-point three-term adder reduces the area, latency and power consumption compared to the discrete floating-point three-term adder by sharing the common logic [9], [10]. However, it is an initial design so that the optimizations can be applied to improve the performance [35]. Figure 56 shows the modified design for the enhanced fused floating-point three-term adder. In this section, three optimizations for the enhanced fused floating-point three-term adder are proposed: 1) A new exponent compare and significand alignment scheme, 2) Double reduction to avoid the complement after the significand addition, 3) Early normalization and fast rounding scheme and 4) Three-input LZA. Also, the implementation details for the other sub-logic are described.

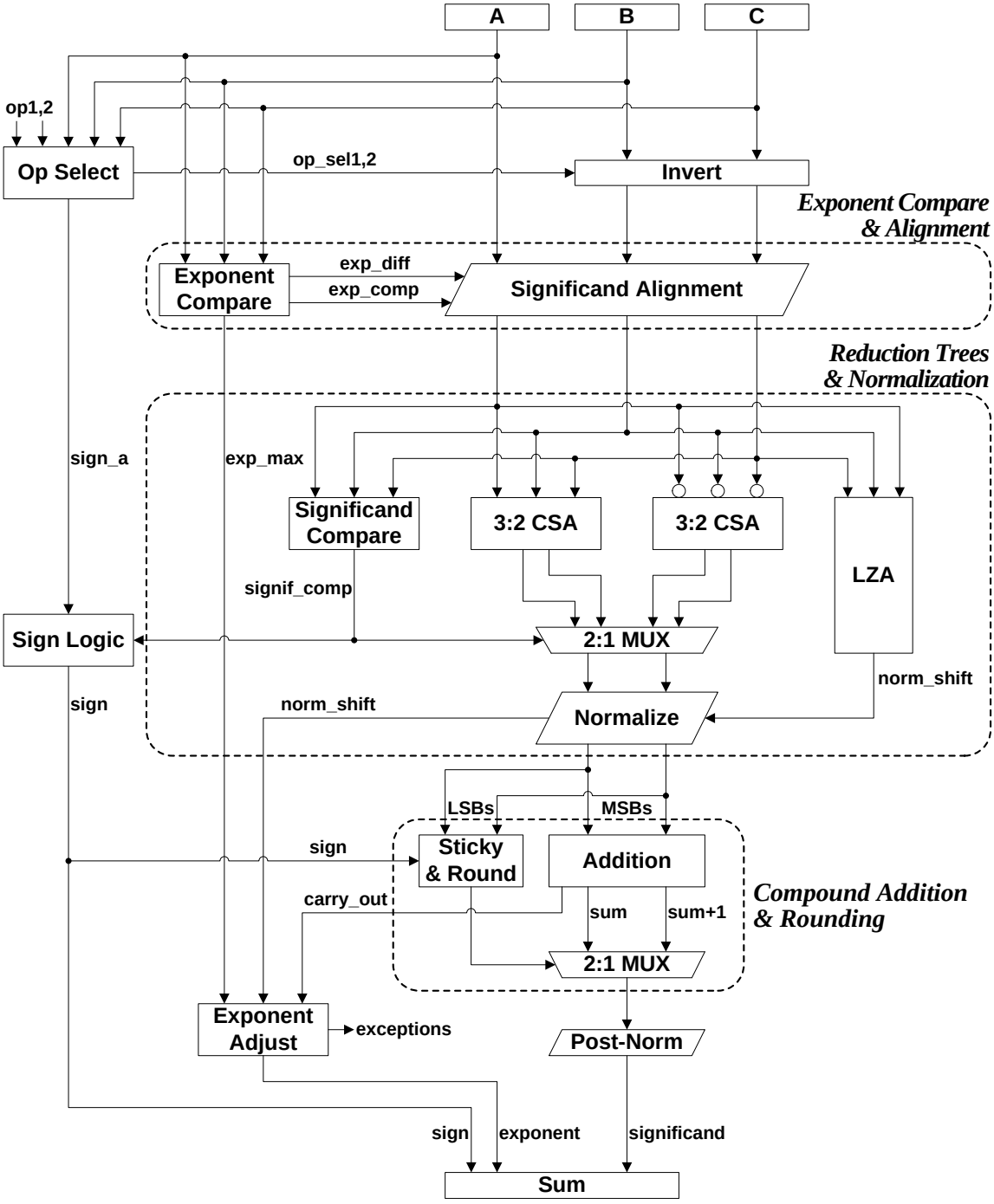


Figure 56. Enhanced Fused Floating-Point Three-Term Adder [35].

5.1.1 New Exponent Compare and Significand Alignment Scheme

To handle the three operands, it is required to determine the max exponent. The traditional fused floating-point three-term adder sorts the exponents and finds the max exponent. The exponent differences are computed by subtracting the each exponent from the max exponent. The other two exponents are subtracted by the max exponent to obtain the shift amount for the significand alignment. Figure 57 shows the traditional exponent compare and alignment logic for a fused floating-point three-term adder.

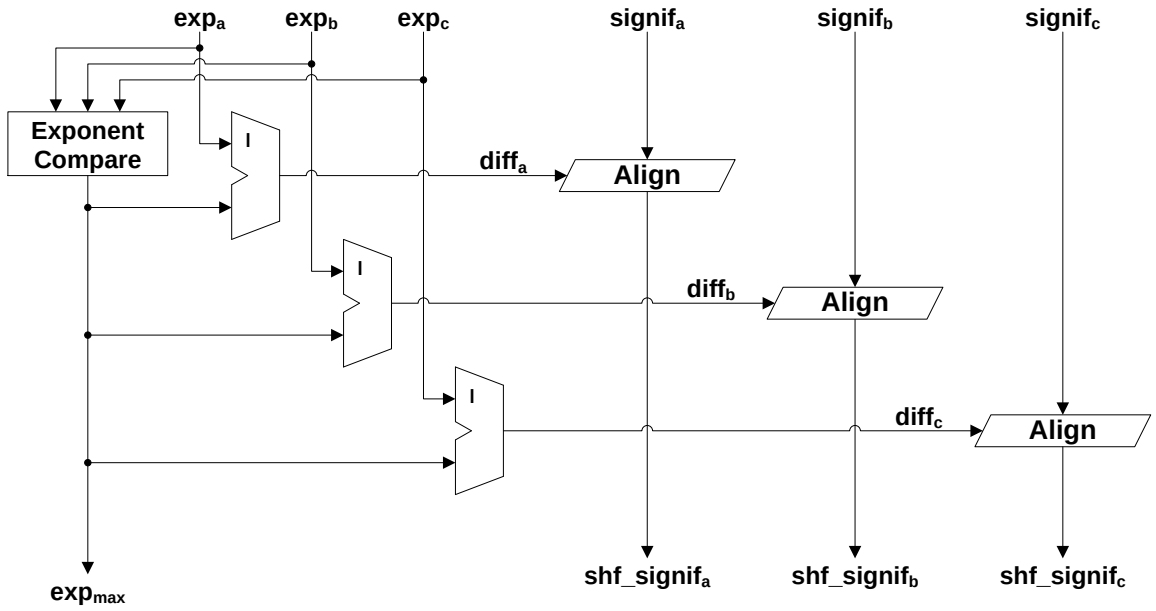


Figure 57. Traditional Exponent Compare for a Fused Three-Term Adder [9].

The traditional exponent compare and significand alignment logic is simple to implement. However, the exponent compare, exponent subtractions and significand alignment are performed sequentially, which takes large delay. In order to reduce the latency, a new exponent compare and significand alignment is proposed as shown in

Figure 58. Three subtractions are performed to compute the exponent differences of the three combinations of exponent pairs ($exp_a - exp_b$, $exp_b - exp_c$ and $exp_c - exp_a$) and the comparison results. The control logic determines the max exponent and the shift amounts of the corresponding significands based on the comparison results. Then, the three significands are aligned by the corresponding shift amount from the control logic. Table 12 shows the control logic that determines the max exponent and shift amounts based on the exponent comparison results. The new exponent compare and significand alignment reduces the latency compared to the traditional method by determining the max exponent and the shift amount at the same time.

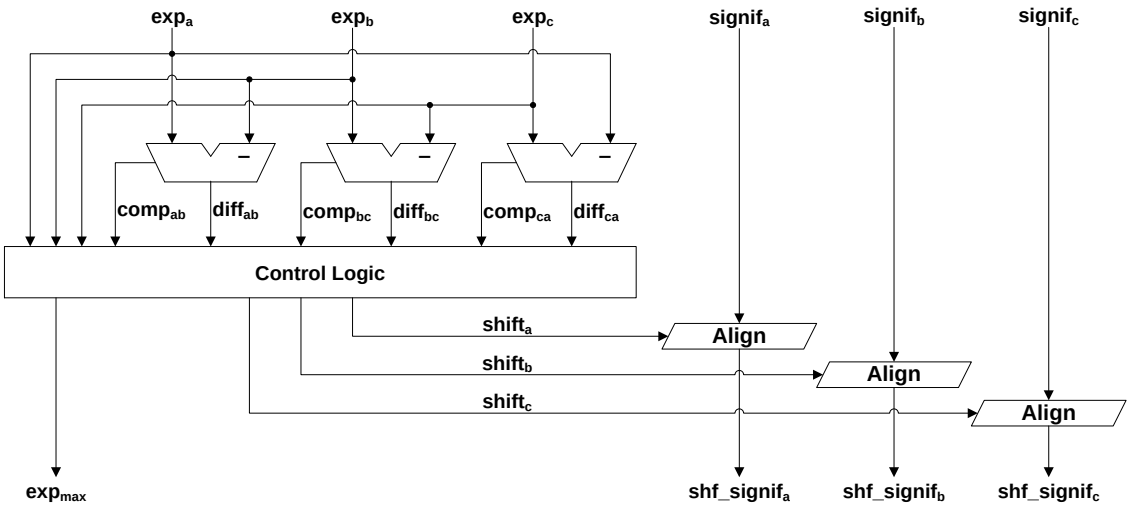


Figure 58. New Exponent Compare for a Fused Three-Term Adder [35].

Table 12. Exponent Compare Control Logic [35].

$comp_{ab}$	$comp_{bc}$	$comp_{ca}$	exp_{max}	$shift_a$	$shift_b$	$shift_c$
0	0	0	N/A	N/A	N/A	N/A
0	0	1	exp_c	$diff_{ca}$	$diff_{bc}$	0
0	1	0	exp_b	$diff_{ab}$	0	$diff_{bc}$
0	1	1	exp_b	$diff_{ab}$	0	$diff_{bc}$
1	0	0	exp_a	0	$diff_{ab}$	$diff_{ca}$
1	0	1	exp_c	$diff_{ca}$	$diff_{bc}$	0
1	1	0	exp_a	0	$diff_{ab}$	$diff_{ca}$
1	1	1	any	0	0	0

5.1.2 Double Reduction and Significand Compare

The aligned significands are passed to two reduction trees. The two reduction trees take both inverted and non-inverted significands and generate two significand pairs. Between two significand pairs, a positive pair is selected based on the significand comparison. In case the exponent differences are small ($diff \leq 2$), full comparison using the tree comparator is required for the significand comparison. However, the delay for the significand compare is hidden by the three-input LZA, which is described with more details in the next section. The significand comparison result is also used for the sign logic. Since the sum of the selected significand pair is positive, the complement after the significand addition is unnecessary. By skipping the complement after the significand addition, the latency of the critical path is reduced.

5.1.3 Early Normalization and Fast Rounding Scheme

One of the general issues on the fused floating-point three-term adder design is the high precision significand addition. The traditional fused floating-point three-term

adder aligns the significands up to 2f + 3 bits, where f is the number of significand bits. The aligned significands are passed to the reduction trees and significand addition as shown in Figure 59, which requires a high precision significand addition. Also, the traditional fused floating-point three-term adder performs the normalization after the significand addition, which requires a large significand adder and compliment followed by the round logic.

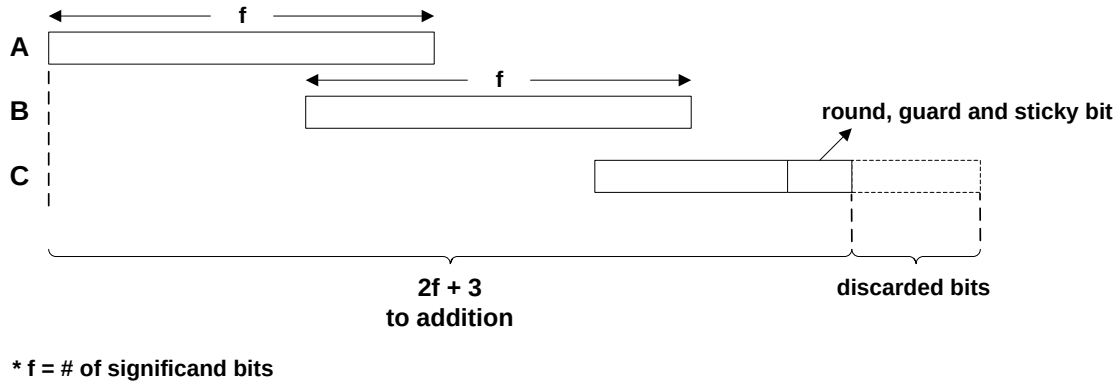
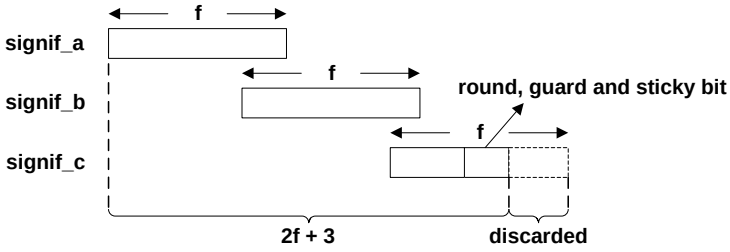


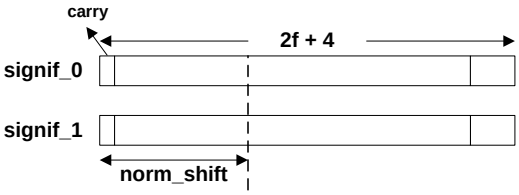
Figure 59. Traditional Alignment for a Fused Three-Term Adder.

To reduce the overhead, early normalization is applied, which is previously proposed for the floating-point multiply-add unit [4]. Figure 60 shows the early normalization procedure and the sticky logic. The significand pair from the reduction is normalized by the shift amount from the LZA. By normalizing the significands prior to the significand addition, f + 1 bits of significand pair are used for the significand addition and the round logic can be performed in parallel, which significantly reduces the critical path latency.

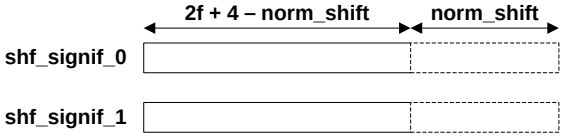
- **Before reduction**



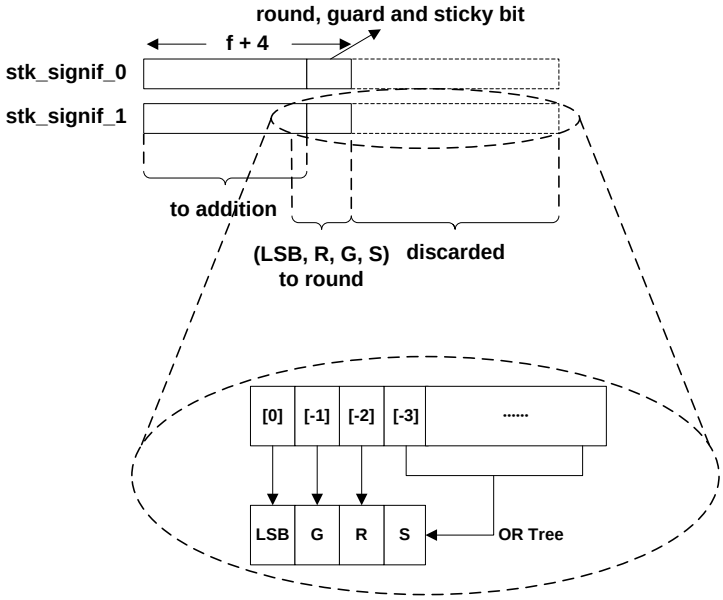
- **After reduction**



- **Normalization**



- **Sticky**



*** f = number of significand bits**

Figure 60. Early Normalization for a Fused Three-Term Adder [35].

The MSBs of the normalized significands are passed to the significand addition and the LSBs are passed to the sticky logic. The sticky logic is performed to generate round, guard and sticky bits. The first and second bits under the LSB become the guard and round bits and the sticky bit is set if at least one bit of the rest of the LSBs is 1, which can be implemented with OR trees. The four bits including the LSB, guard, round and sticky bits are used for the round logic and the rest of the LSBs are discarded.

5.1.4 Three-Input LZA

Since the normalization is performed prior to the significand addition, the LZA and normalization is on the critical path. To use the traditional two-input LZA, the three significands are required to be reduced to two by performing a 3:2 reduction, which increases the delay. The three-input LZA reduces the overhead of the reduction by encoding the three inputs at once.

The three-input LZA can be implemented by extending the traditional two-input LZA [19]. Since the significands are inverted based on the effective operation, the W vector, which is generated with bitwise operations is always positive as

$$W = A + B + C$$

$$w_i = a_i + b_i + c_i, \qquad w_i \in \{0, 1, 2, 3\},$$

where a_i, b_i, c_i are the i^{th} bits from the MSB of the three significands. The W vector can be represented by one of the four elements, 0_i, 1_i, 2_i and 3_i, indicating w_i equals to 0, 1, 2 and 3, respectively. The W vector is pre-encoded into three symbols, z_i, t_i and g_i as

$$z_i = 1 \; if \; w_i = 0$$

$$t_i = 1 \; if \; w_i = 1$$

$$g_i = 1 \; if \; w_i = 2.$$

To handle the case if w_i is equal to 3, two consecutive bits are involved for pre-encoding. For example, bit pattern 0_i3_{i-1} is considered as 1_i1_{i-1}. Thus, the three symbols are represented as

$$z_i = 0_i(0_{i-1} + 1_{i-1}) + 3_i(2_{i-1} + 3_{i-1})$$

$$t_i = 0_i(2_{i-1} + 3_{i-1}) + 1_i(0_{i-1} + 1_{i-1}) + 2_i(2_{i-1} + 3_{i-1}) + 3_i(0_{i-1} + 1_{i-1})$$

$$g_i = 1_i(2_{i+1} + 3_{i+1}) + 2_i(0_{i-1} + 1_{i-1}).$$

The number of leading zeros is computed with the three symbols as

$$f_i = t_{i+1}(g_i\overline{z}_{i-1} + z_i\overline{g}_{i-1}) + \bar{t}_{i+1}(z_i\bar{z}_{i-1} + g_i\overline{g}_{i-1}).$$

The F vector is passed to the leading zero detector (LZD). The LZA produces the leading zeros, which becomes the shift amount of the normalization. For fast normalization, the MSBs of the shift amount are generated so that the LZD logic and the normalization shifter are overlapped [4].

Most of the two-input LZAs are inexact due to the possible 1 bit error. Similarly, the proposed three-input LZA also requires correction logic. For fast error detection and correction, concurrent error correction logic can be used, which was previously proposed [19] – [21][7]. More details on the correction logic are described in Section 3.2.2.5.

[7] The error correction logic in [19] is modified by [20] and [21] to improve the accuracy and eliminate the redundancy, respectively.

5.1.5 The Other Sub-Logic

The significand addition and round logic can be implemented as that of the fused floating-point two-term dot product unit, which is described in the previous chapter. In this section, the rest of sub-logic designs for the enhanced fused floating-point three-term adder are presented: 1) Operation select logic, 2) Significand compare logic, 3) Sign logic and 4) Exponent adjust logic.

5.1.5.1 Operation Select Logic

The operation select logic generates the two effective operations, *op_sel*1 and *op_sel*2. The two operation bits determine if the second and third significands are inverted for the significand subtractions, respectively. Using the three sign bits and two op codes, the effective operations are selected as

$$op\_sel1 = sign_a \oplus (sign_b \oplus op1)$$

$$op\_sel2 = sign_a \oplus (sign_c \oplus op2),$$

where *op*1 and *op*2 are the first and second op codes, respectively.

5.1.5.2 Sign Logic

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 generates the sign bit of the sum using the sign of the first operand and the significand comparison result as

$$sign_{sum} = sign_a \oplus comp_{signif}.$$

The sign bit is passed to the final result as well as the round logic.

5.1.5.3 Exponent Adjust Logic

The max exponent which is determined by the exponent compare logic is adjusted by subtracting the shift amount from the LZA and adding the carry-out of the significand addition as shown in Figure 61. Since the three significands generate a carry-out of up to 2, two carry-out bits are used for the adjustment. The normalization shift amount is subtracted in case of the massive cancellation. Using the selection bits and the carry-outs from the addition and subtractions, the exceptions are detected. The three exception cases specified in IEEE-754 Standard [1] are detected as

$$overflow = \begin{cases} 1 & if\ exp \geq max\_exp \\ 0 & otherwise \end{cases}$$

$$underflow = \begin{cases} 1 & if\ exp \leq 0 \\ 0 & otherwise \end{cases}$$

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

where *round_up* is the rounding decision of the significand result.

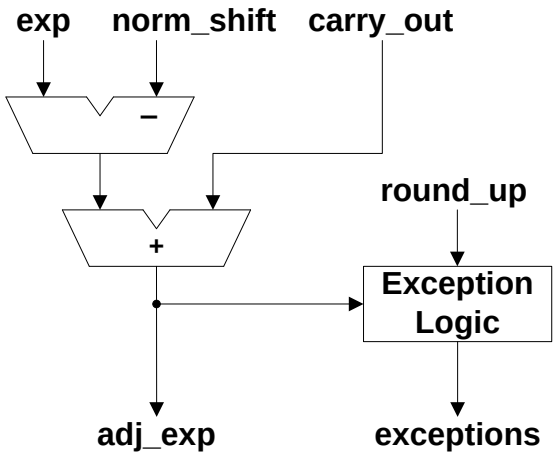


Figure 61. Exponent Adjust for an Enhanced Fused Three-Term Adder [35].

5.2 Pipelined Fused Floating-Point Three-Term Adder

Pipelining is applied to the fused floating-point three term adder to improve the throughput. Based on the data flow analysis, the proposed enhanced fused floating-point three-term adder can be split into three pipeline stages so that the results are produced every cycle. In the pipelined logic, the slowest pipeline stage determines the effective latency of the entire logic. Therefore, it is important to properly arrange the logic components so that the longest latency is as short as possible. In this section, the data flow analysis to arrange the logic components of the proposed enhanced fused floating-point three-term adder and the composition of pipeline stages is presented.

5.2.1 Data Flow Analysis

In order to achieve the proper pipelining for the fused floating-point three-term adder, the arrangement of the components is investigated. Each component is implemented in Verilog-HDL and synthesized with the Nangate 45nm CMOS technology standard-cell library. The latencies of the various elements of the single precision enhanced fused floating-point three-term adder are listed in Table 13.

Figure 62 shows the data flow and critical path of the enhanced fused floating-point three-term adder. 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 enhanced fused floating-point three-term adder is split into three pipeline stages. Each pipeline stage is executed every cycle so that the largest latency determines the throughput of the design.

Table 13. Component Latencies in an Enhanced Fused Three-Term Adder [35].

Components	Latency (ns)	Components	Latency (ns)
Unpack	0.02	Exponent Compare	0.27
Op Select	0.08	Significand Align	0.18
Invert	0.02	LZA	0.35
Significand Compare	0.22	3:2 CSAs	0.12
Sign Logic	0.06	2:1 Select	0.04
Normalization	0.14	Sticky & Round	0.16
Significand Addition	0.28	Round Select	0.04
Exponent Adjust	0.18	Post Normalization	0.08

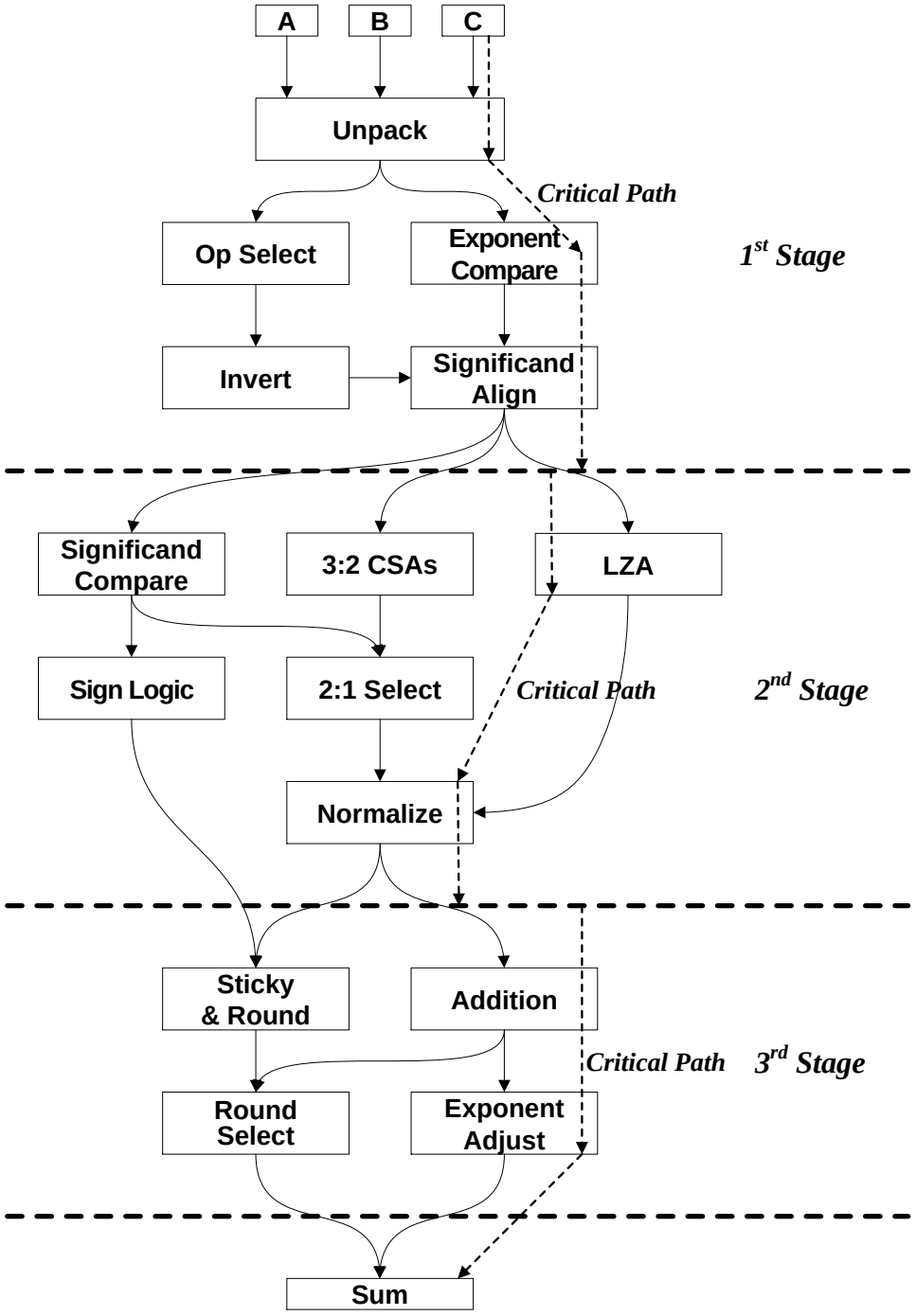


Figure 62. Data Flow of a Pipelined Enhanced Fused Three-Term Adder [35].

5.2.2 Pipeline Stages of a Dual-Path Fused Floating-Point Dot Product Unit

Based on the data flow analysis, the proposed enhanced fused floating-three-term adder can be split into three stages. The critical paths of the three pipeline stages are

First stage: Unpack → Exponent compare → Significand alignment

Second stage: LZA → Normalization

Third stage: Significand addition → Exponent adjust.

5.2.2.1 The First Pipeline Stage

The first pipeline stage contains unpacking, operation select, invert, exponent compare logic and significand alignment. The operation select and exponent compare logic can be performed in parallel. Since the data path including the exponent compare logic has the largest latency, it becomes the critical path of the first pipeline stage.

5.2.2.2 The Second Pipeline Stage

The second pipeline stage consists of three data paths. The first path contains the significand reduction trees and 2:1 selection. The second path contains the significand compare and sign logic. The third path contains the LZA and normalization. Since the data path including the LZA and normalization takes the largest latency, it becomes the critical path which determines the latency of the second pipeline stage.

5.2.2.3 The Third Pipeline Stage

The third stage contains the significand addition, sticky logic, rounding, and exponent adjust logic. The data path including the significand addition and exponent

adjust logic has the largest latency so that it determines the latency of the third pipeline stage.

In each pipeline stage, several logic components are performed in parallel and the path that takes the largest latency becomes the critical path. Since the third stage takes the largest latency among the three pipeline stages, the latency of the third stage becomes the effective latency which determines the throughput. Due to the latches and control signals between the pipeline stages, the area and power consumption are increased compared to the non-pipelined fused floating-point three-term adder. Also, the total latency of the pipelined fused floating-point three-term adder is three times the largest latency among the three pipeline stages. However, the latencies of the three pipeline stages are fairly well balanced so that the throughput is significantly increased compared to the non-pipelined fused floating-point three-term adder.

5.3 Implementation and Results

Previous sections introduced the designs of the two advanced fused floating-point three-term adders: 1) Enhanced fused floating-point three-term adder and 2) Pipelined fused floating-point three-term adder. Each design is implemented for both single and double precision in Verilog-HDL and synthesized with the Nangate 45nm CMOS technology standard cell library. To verify the improvement of the proposed designs, the logic area, critical path latency, throughput and, power consumption of the two implementations are compared with the discrete design and the traditional fused design as

shown in Table 14. All the percentages in the table are ratios compared to the discrete design.

The traditional fused design reduces the area, power consumption and latency by about 30%, since the fused design shares the logic such as significand addition and rounding. The enhanced fused floating-point three-term adder applies the new exponent compare and alignment scheme to reduce the latency by performing exponent comparison and shift amount generation in parallel. Early normalization and fast rounding enables reducing the significand addition size and performing the significand addition and rounding in parallel, which significantly improves the performance. Also, the three-input LZA reduces the latency by hiding the latency of the 3:2 reduction and significand comparison. As a result, the area, power consumption and latency are reduced by approximately 45% compared to the discrete design.

Table 14. Floating-Point Three-Term Adder Design Comparison [35].

Single Precision				
	Discrete	**Traditional Fused**	**Enhanced Fused**	**Enhanced + Pipeline**
Area (μm^2)	15,403 (100%)	11,381 (74%)	8,921 (58%)	9,503 (62%)
Latency (ns)	2.64 (100%)	1.91 (72%)	1.46 (55%)	1.62 (61%)
Throughput (1/ns)	0.38 (100%)	0.52 (138%)	0.68 (181%)	1.85 (489%)
Power (mW)	7.77 (100%)	5.58 (72%)	4.13 (53%)	4.65 (60%)
Double Precision				
	Discrete	**Traditional Fused**	**Enhanced Fused**	**Enhanced + Pipeline**
Area (μm^2)	34,606 (100%)	24,622 (71%)	19,293 (56%)	20,882 (60%)
Latency (ns)	3.32 (100%)	2.33 (70%)	1.76 (53%)	1.95 (59%)
Throughput (1/ns)	0.30 (100%)	0.43 (143%)	0.57 (189%)	1.54 (511%)
Power (mW)	15.46 (100%)	10.70 (69%)	8.02 (52%)	8.98 (58%)

The double precision implementation requires about twice as much area and power consumption as the single precision implementation due to the larger logic components. However, since tree structures are used for major components such as significand alignment, significand addition, LZA and normalization, which logarithmically increase the latency, the latency for the double precision increases by only 20%. The benefits of the new alignment scheme, early normalization, fast rounding and three-input LZA are shown in both single and double precision.

The proposed pipelined enhanced fused floating-point three-term adder is split into three stages. Table 15 shows the area, latency and power consumption of the three pipeline stages. Each pipeline stage requires latches to maintain the data and control signals between the stages, which increases the area, latency and power consumption. However, the latencies of the three pipeline stages are fairly well balanced so that the throughput is increased to about 2.7 times that of the non-pipelined design.

Table 15. Pipeline Stages for an Enhanced Fused Three-Term Adder [35].

Single Precision			
	Stage 1	**Stage 2**	**Stage 3**
Area (μm^2)	2,223 (23%)	3,989 (42%)	3,291 (35%)
Latency (ns)	0.51 (32%)	0.53 (34%)	0.54 (34%)
Power (mW)	1.09 (23%)	2.05 (44%)	1.51 (33%)
Double Precision			
	Stage 1	**Stage 2**	**Stage 3**
Area (μm^2)	4,620 (22%)	8,639 (42%)	7,423 (36%)
Latency (ns)	0.60 (32%)	0.63 (34%)	0.65 (34%)
Power (mW)	1.93 (22%)	3.99 (44%)	3.06 (34%)

Chapter 6

Conclusion and Future Work

This chapter presents the conclusion on the improved architectures for fused floating-point arithmetic units and summarizes the implementation results and trade-offs. Finally, the chapter finishes the dissertation with suggestions of future work for the design and implementation of fused floating-point arithmetic units.

6.1 Conclusion

In this dissertation, improved architectures for three fused floating-point arithmetic units are presented: 1) Fused floating-point add-subtract unit, 2) Fused floating-point two-term dot product unit, and 3) Fused floating-point three-term adder. Most general purpose processors (GPP) and application specific processors (ASP) can be benefit from the improved fused floating-point arithmetic units. The fused floating-point add-subtract unit is useful for the digital signal processing (DSP) applications such as fast Fourier transform (FFT) and discrete cosine transform (DCT) butterfly operations. To improve the performance of the fused floating-point add-subtract unit, a new alignment scheme, fast rounding, dual-path algorithms and pipelining are applied. The enhanced fused floating-point add-subtract unit applying the new alignment scheme and fast rounding saves about 45% of the area and power consumption and reduces 8% of the latency compared to the discrete floating-point add-subtract unit. Also, two dual-path

algorithms are proposed. The low power dual-path fused floating-point add-subtract unit reduces the critical path latency by about 20% compared to the discrete floating-point add-subtract unit with a small increase in the area and power consumption. The high-speed dual-path fused floating-point add-subtract unit is more optimized to improve the performance so that it reduces latency by 30% compared to the discrete floating-point add-subtract unit. Additionally, a pipelining to increase the throughput of the dual-path fused floating-point add-subtract unit is applied. It uses two pipeline stages and the latencies are well balanced so that the throughput is increased up to 1.8 times that of the non-pipelined dual-path floating-point add-subtract unit.

The fused floating-point two-term dot product unit is useful for many DSP applications such as matrix multiplication, complex multiplication, FFT and DCT butterfly operations. To improve the performance of the fused floating-point two-term dot product unit, several optimizations are applied: a new alignment scheme, early normalization and fast rounding, four-input leading zero anticipation (LZA), dual-path algorithm and pipelining. The enhanced fused floating-point two-term dot product unit applying the new alignment scheme, early normalization and fast rounding and four-input LZA reduces the area and power consumption by 40% and improves the performance by 20% compared to the discrete floating-point two-term dot product unit. Further improvement is achieved by use of the dual-path algorithm. The dual-path fused floating-point two-term dot product unit consists of a far path and a close path and one path is selected based on the exponent difference. Since the dual-path design eliminates unnecessary logic in each path so that the latency is reduced by about 30% compared to

the discrete fused floating-point two-term dot product unit. Pipelining can be applied to improve the throughput. Based on the data flow analysis, the dual-path fused floating-point two-term dot product unit can be split into three stages. Since the latencies of the three stages are fairly well balanced, the throughput is 2.8 times that of the non-pipelined dual-path fused floating-point two-term dot product unit.

The fused floating-point three-term adder is useful for many algorithms and applications which uses multiple additions in serial. To improve the performance of the fused floating-point three-term adder, a new exponent compare and significand alignment scheme, double reduction, early normalization and fast rounding, three-input LZA and pipelining are applied. The enhanced fused floating-point three-term adder applying the new exponent compare and alignment scheme, early normalization and fast rounding and three-input LZA reduces the area, power consumption and latency by 45%. Pipelining can be applied to the enhanced fused floating-point three-term adder to improve the throughput. The pipelined fused floating-point three-term adder consists of three pipeline stages. The three pipeline stages take about the same level of latencies so that the throughput of the entire logic is increased to 2.7 times that of the non-pipelined fused floating-point three-term adder.

Table 16 shows the proposed three fused floating-point units and the applied optimizations in each floating-point unit. The proposed fused floating-point add-subtract unit applied a new alignment scheme, dual-path algorithm and pipelining to improve the performance. The improved fused floating-point two-term dot product unit applied all the optimizations, a new alignment scheme, four-input LZA, dual-path algorithm and

pipelining. The proposed fused floating-point three-term adder applies a new alignment scheme, three-input LZA and pipelining to improve the performance.

Table 16. Proposed Fused Floating-Point Units and Applied Optimizations.

FPUs	Optimizations			
	New Alignment	Multi-Input LZA	Dual-Path	Pipelining
FAS	X		X	X
FDP2	X	X	X	X
FADD3	X	X		X

The optimizations applied for the proposed three fused floating-point units have trade-offs in terms of the evaluation categories: area, latency, throughput and power consumption. Table 17 shows the trade-offs of the optimizations for the evaluation categories. A new alignment scheme and multi-input LZA have benefits for all the categories by reducing significand adder size and skipping reductions, respectively. The dual-path algorithms take advantages of latency and throughput by skipping unnecessary logic in each path with relatively small increase of area and power consumption. Pipelining increases area, latency and power consumption due to the latches in between the pipeline stages. However, the pipelining significantly improves the throughput in case the latencies of the pipeline stages are well-balanced. Modern DSP applications require various specifications depending on the purpose. Thus, the trade-off analysis described above is useful to decide how to design and implement the floating-point units.

Table 17. Trade-offs of the Optimizations for the Evaluation Categories.

Category	Optimizations			
	New Alignment	Multi-Input LZA	Dual-Path	Pipelining
Area	+	+	–	–
Latency	+	+	++	–
Throughput	+	+	++	+++
Power	+	+	–	–

6.2 Future Work

The proposed fused floating-point arithmetic units achieve low area, low power and high performance. The improved fused floating-point units can be used for the next generation DSP application development such as FFT, DCT, matrix multiplication and complex multiplication. The design and implementation for those improved application specific processors (ASP) will be an interesting research topic. It also involves the investigation to the trade-offs of various optimization techniques for the specific applications. Also, the improved fused floating-point units will contribute to the next generation floating-point unit development for the general purpose processors (GPP) by extending the current instruction set architectures (ISA) to apply the proposed fused floating-point operations.

The floating-point unit architectures introduced in this dissertation can be extended to the other floating-point units. The multi-term dot product unit and multi-term

adder can be designed and implemented by extending the proposed two-term dot product unit and three-term adder, respectively. Another interesting topic is to design and implement the fused floating-point unit to compute the square root of the sum of the squares which is used for the calculation of the magnitude of complex numbers. Finally, those improved fused floating-point units will contribute the application specific processors (ASP) development and next generation floating-point unit development for the general purpose processors (GPP).

Bibliography

[1] *IEEE Standard for Floating-Point Arithmetic*, ANSI/IEEE Standard 754-2008, New York: IEEE, Inc., 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] J. D. Bruguera and T. Lang "Floating-point fused multiply-add: reduced latency for floating-point addition," *Proceedings of the 19th IEEE Symposium on Computer Arithmetic*, pp. 42 – 51, 2005.

[6] 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*, pp. 519 – 522, 2008.

[7] E. E. Swartzlander, Jr. and H. H. Saleh, "FFT Implementation with Fused Floating-Point Operations," *IEEE Transactions on Computers*, Vol. 61, pp. 284 – 288, 2012.

[8] 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.

[9] A. Tenca, "Multi-Operand Floating-Point Addition," *Proceedings of the 19th IEEE Symposium on Computer Arithmetic*, pp. 161 – 168, 2009.

[10] Y. Tao, G. Deyuan, F. Xiaoya, R. Xianglong, "Three-Operand Floating-Point Adder," *Proceedings of the 12th IEEE International Conference on Computer and Information Technology*, pp. 192–196, 2012.

[11] 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*, pp. 767 – 776, 2008.

[12] E. E. Swartzlander, Jr. and H. H. Saleh, "Floating-Point Implementation of Complex Multiplication," *Proceedings of the 43rd Asilomar Conference on Signals, Systems and Computers*, pp. 926 – 929, 2009.

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

[14] S. F. Oberman, H. Al-Twaijry and M. J. Flynn, "The SNAP Project: Design of Floating Point Arithmetic Units," *Proceedings of the 14th IEEE Symposium on Computer Arithmetic*, pp. 156 – 165, 1997.

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

[16] 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.

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

[18] 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.

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

[20] 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.

[21] P. Kornerup, "Correcting the Normalization Shift of Redundant Binary Representations," *IEEE Transactions on Computers*, Vol. 58, pp. 1435 – 1439, 2009.

[22] 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.

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

[24] A. Naini, A. Dhablania, W. James, and D. Das Sarma, "1-GHz HAL SPARC64 Dual Floating Point Unit with RAS Features," *Proceedings of 15th IEEE*

Symposium on Computer Arithmetic, pp. 173 – 183, 2001.

[25] L. Dadda, "Some Schemes for Parallel Multipliers," *Alta Frequenza*, vol. 34, pp. 349 – 356, 1965.

[26] Y. Watanabe, N. Homma, T. Aoki, and T. Higuchi, "Arithmetic Module Generator with Algorithm Optimization Capability," *Proceedings of the IEEE International Symposium on Circuits and Systems*, pp. 1796 – 1799, 2008.

[27] E. Quinnell, E. E. Swartzlander, Jr., and C. Lemonds, "Floating-Point Fused Multiply-Add Architectures," *Proceedings of the 41st Asilomar Conference on Signals, Systems and Computers*, pp. 331 – 337, 2007.

[28] E. Quinnell, *Floating-Point Fused Multiply-Add Architectures*, Ph.D. dissertation, Electrical and Computer Engineering, The University of Texas at Austin, 2007.

[29] J. Sohn and E. E. Swartzlander, Jr., "Improved Architectures for a Fused Floating-Point Add-Subtract Unit," *IEEE Transactions on Circuits and Systems–I*, Vol. 59, pp. 2285 – 2291, 2012.

[30] J. Min, J. Sohn and E. E. Swartzlander, Jr., "A Low-Power Dual-Path Floating-Point Fused Add-Subtract Unit," *Proceedings of the 46st Asilomar Conference on Signals, Systems and Computers*, pp. 998 – 1002, 2012.

[31] 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, 1994.

[32] 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.

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

[34] J. Sohn and E. E. Swartzlander, Jr., "Improved Architectures for a Floating-Point Fused Dot Product Unit," *Proceedings of the 21th IEEE Symposium on Computer Arithmetic*, pp. 41 – 48, 2013.

[35] J. Sohn and E. E. Swartzlander, Jr., "Improved Architectures for a Fused Floating-Point Three-Term Adder," *IEEE Transactions on VLSI Systems*, in process.

Vita

Jongwook Sohn was born in Seoul, Korea on December, 4th 1982. He received a Bachelor of Science degree in Electrical Engineering from Korea University, Seoul, Republic of Korea in 2009. After completing his undergraduate school, he entered the Graduate School of the University of Texas at Austin and received a Master of Science degree in Electrical and Computer Engineering from the University of Texas at Austin in 2011. He has been 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 has been working as an engineer in Atom and SoC Development Group at Intel Corporation, Austin, Texas since 2011.

Permanent E-mail address: jongwook.sohn@utexas.edu

This dissertation was typed by the author.