IMPLEMENTATIONS OF HIGH PERFORMANCE ARCHITECTURE FOR IEEE 754 COMPLIANT FLOATING-POINT ADDERS

By

BRETT MATHIS

Bachelor of Science in Computer Engineering
Oklahoma State University
Stillwater, Oklahoma
2018

Submitted to the Faculty of the
Graduate College of
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December, 2020

IMPLEMENTATIONS OF HIGH PERFORMANCE ARCHITECTURE FOR IEEE 754 COMPLIANT FLOATING-POINT ADDERS

Thesis Approved:

Dr. James E. Stine

Thesis Adviser

Dr. Keith Teague

Dr. John Hu

ACKNOWLEDGMENTS

I would like to thank my adviser, Dr. James E. Stine, Jr. for both allowing me to join his architecture research group, and for the constant support he has provided in both academic and personal endeavors. I hope to continue working with him for a long time to come.

I would like to thank Dr. Keith Teague and Dr. John Hu for acting as committee members and for their academic advise throughout my college career.

I would like to thank Alex Underwood for helping me navigate some of the bureaucratic pitfalls found in creating a thesis.

Acknowledgments reflect the views of the author and are not endorsed by committee members or Oklahoma State University.

Name: BRETT MATHIS

Date of Degree: DECEMBER, 2020

Title of Study: IMPLEMENTATIONS OF HIGH PERFORMANCE ARCHITECTURE FOR IEEE 754 COMPLIANT FLOATING-POINT ADDERS

Major Field: ELECTRICAL ENGINEERING

Abstract: This thesis presents a direct iteration and implementation on a high performance architecture for IEEE 754 floating-point addition. This thesis improves on the previous architecture's implementation in a variety of sub-operations required for IEEE 754 floating-point addition, which are focused on directly improving critical path delay performance. A key element of this paper is the introduction of a flagged-prefix adder within the main carry-propagation path of an end-around-carry adder. It also provides detailed documentation for the design of IEEE 754 compliant floating-point adders. This is particularly emphasized for uncommon operations and control logic used throughout floating-point addition, including denormalized numbers and multi-precision logic. The full design for this architecture has support for binary16, binary32, and binary64 operations. The full extended range provided by denormalized IEEE 754 values is supported. It also has conversion support between IEEE 754 and two's complement integer values in either binary16, binary32, or binary64 precision. The performance comparisons shown are synthesis results in cmos32soi 32nm GF technology and ARM-based standard cells.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

CHAPTER I

INTRODUCTION

Computing performance has improved dramatically over the last twenty years due to advances in Very-Large Scale Integration (VLSI) technology and integrated circuit processing [4]. This has been partially attributed to Moore's Law which states that the number of transistors on a computer chip nearly doubles approximately every 18 months [4]. Consequently, the demand for smaller, faster, accurate, and more reliable computers makes the design of computer systems more complex. This increase in complexity, along with a myriad of word sizes, rounding modes, and precisions, motivated researchers to develop the IEEE Standard 754 for binary floating-point arithmetic [1, 5].

To make things more challenging, silicon-device fabrication of transistors has changed substantially in the last 20 years [6]. However, this complexity within the manufacturing process has imposed limitations and a set of challenges that researchers will have to overcome in order to design future high-performance systems [7]. These limitations originally dealt with overcoming large amounts of power and energy dissipation for high-speed computer architectures and application-specific integrated circuits. In other words, complex digital designs are getting faster along with subsequently consuming large amounts of energy as designers resort to reducing feature sizes and supply voltages to meet these constraints. Although this has worked in the past, it does not solve issues related to optimizing constraints for both energy and speed [8]. Therefore, there is a need for new designs in IEEE 754 arithmetic that limit size to reduce energy yet still remain fast.

Implementations and modifications to IEEE floating-point addition have been, as a whole, well explored yet not completely documented over the lifespan of the operation. A variety of architecture improvements and implementations have been designed (e.g. [9, 10]) since the original IEEE 754 standard was introduced [2]. Many of these publications, however, do not attempt to maintain full IEEE 754 compliance [10, 11, 12] and/or are seldom documented below the level of abstraction required for microarchitecture operations (e.g. two's complement adders, shifters, leading-zero detection, etc.). In addition, many of these implementations seldom implement or fully verify designs, or they cannot perform this action due to company liability. Most importantly, this work attempts to take advantage of recent advances in the use of late-carry enhanced prefix adders to improve upon the speed and energy of IEEE 754 addition/subtraction [10].

This thesis aims not only to improve on the delay and energy performance of previously published architectures [13, 3], but to improve upon it by using an end-around-carry adder along with a flagged-prefix to optimize the computation of the final result in parallel. The primary adder architecture has received a significant improvement between iterations, in that it now utilizes an end-around-carry adder architecture. This allows it to keep the same delay performance as a parallel adder structure, but at nearly half the area cost with reduced static power consumption. The exponent rounding structure used for denormalized values has also been completely reworked with novel use of a flagged-prefix adder architecture. This integrates an offset value for exponent rounding in denormalized cases, as opposed to applying the rounding offset after the exponent has been calculated, which significantly decreases delay. To further increase the novelty of this design, the exponent subtraction stage of the adder now performs two separate right shifts for normalized and denormalzied exponent differences, while comparison subtractions are performed in parallel. This decreases the fan-out between exponent subtraction and the primary addition

performed, which in turn further decreases delay. This thesis also implements an exact leading-zero anticipator (LZA) for use in post-normalization. Previously, only a leading-zero detector was used once the sum was produced. Changing to an architecture that uses a leading-zero anticipator significantly decreases the delay necessary for post-normalization. Using a leading-zero anticipator is not novel itself, but it is a necessary improvement to make a competitive design. All of these changes will be alluded to as they become relevant in this thesis.

This thesis innovates significantly over other implementations, such as the Z990 processor [10]. This thesis performs IEEE 754 denormal alignment early in the datapath during exponent comparison. In addition, two separate alignment shifters for pre-normalization optimize fan-out, which by proxy optimizes both delay and dynamic power consumption. This paper also accounts for borderline overflow and underflow after the primary addition to handle all extreme edge cases denormalized values may cause. The Z990 does not use either an EAC adder for its primary addition, nor does it use a flagged-prefix adder to decrease delay for exponent rounding [10]. EAC adders have been extensively discussed in [14], but no implementation results have been given.

Some knowledge of the IEEE 754 standard, including terminology for input formatting and exception generation, is required to obtain a full grasp on some of the architectural design decisions made for various applications of floating-point addition. Therefore, it will be briefly covered in Section II. Design decisions for specific microarchitectures used throughout this thesis's design, as well as a brief introductions to the microarchitectures themselves, is covered in Section III. The overall architecture for this design is covered in Section IV along with relevant signals that are difficult to derive for IEEE 754 compliance. A comparison between the architecture presented in this thesis and that of previous iterations [3] is included in Section 4.5. Section 5.1 presents post-layout results for this architecture as well as performance differences

between different components of the current and previous work in cmos32soi 32nm GlobalFoundries (GF) Multi-threshold CMOS (MTCMOS) technology using ARM-based standard cells.

CHAPTER II

IEEE 754 BACKGROUND

IEEE 754 floating-point values consist of three components: a 1-bit sign value, a 5/8/11-bit exponent value, and a 10/23/52-bit mantissa value [15]. These refer to the radix used for half/single/double precision IEEE 754 formats, respectively [1]. An example double-precision IEEE 754 input vector `X[63:0]` might look like the following:

$$\texttt{X[63], X[62:52], X[51:0]} =$$

$$\texttt{1'h0, 11'h3FF, 52'h7\_FFFF\_FFFF\_FFFF} \quad ,$$

where `X[63]` is the input's sign `S`, `X[62:52]` is the input's exponent `E`, and `X[51:0]` is the input's mantissa `M` [1]. Using these components, a decimal output value can be calculated with the following format:

$$\texttt{Out(S,E,M)} = -1^{S} \times 2^{E} \times M \quad .$$

However, some conversion factors must be applied to both the exponent and mantissa values before a direct floating-point output can be calculated. The IEEE 754 standard requires exponents to be represented by an unsigned integer value and a constant offset, which varies between each precision used. Once this offset is applied, it can be used to calculate the correct output value. A reference of offsets for precisions used, as well as conversions between them, is included below:

$$E_{\texttt{F64}} \quad = \quad E_{\texttt{F32}} - 127 + 1023 = E_{\texttt{F32}} + 896 \quad ,$$

[1] To help clarify notation, typical Verilog bit-swizzling usage is employed throughout this work to improve readability.

$$
\begin{aligned}
E_{\texttt{F32}} &= E_{\texttt{F16}} - 15 + 127 = E_{\texttt{F16}} + 112 \ , \\
E_{F64} &= E_{F16} - 15 + 1023 = E_{\texttt{F16}} + 1008 \ , \\
E_{\texttt{F64-offset}} &= E_{\texttt{F64-unsigned}} - 1023 \ , \\
E_{\texttt{F32-offset}} &= E_{\texttt{F32-unsigned}} - 127 \ , \\
E_{\texttt{F16-offset}} &= E_{\texttt{F16-unsigned}} - 15 \ .
\end{aligned}
$$

Converting the mantissa into a value that can be directly used to calculate a decimal floating-point output does not require an additional offset value or arithmetic operation, but the value for the mantissa does have to be mapped onto the domain of $[1, 2)$. Using these methods, the previous input vector `X[63:0]` is converted to its corresponding decimal floating-point representation using the following calculation:

$$
\begin{aligned}
\texttt{Out(S,E,M)} &= -1^{1'h0} \times 2^{\texttt{11'h3FF-11'h1FF}} \times \\
&\qquad (1 + \texttt{52'h}\frac{\texttt{7\_FFFF\_FFFF\_FFFF}}{\texttt{F\_FFFF\_FFFF\_FFFF}}) \ , \\
\texttt{Out(S,E,M)} &= 1.348 \times 10^{308} \ ,
\end{aligned}
$$

where `11'h1FF` is the hexadecimal representation of the IEEE 754 double-precision offset value, and `52'hF_FFFF_FFFF_FFFF` is the maximum possible IEEE 754 double-precision mantissa value, disregarding that it is an exception case. This will allow for a range of values from 2.225×10^{-308} to 1.779×10^{308} without the use of denormalized inputs.

Denormalized inputs, which increase the exponent value range by extending it into the mantissa, further extend the minimum range of values that can be represented from 2.225×10^{-308} to 4.941×10^{-325}. This process in which the low range precision is increased is called gradual underflow [16], [5]. This helps to reduce issues caused by truncation at small values, which becomes especially prevalent when comparing similar input vectors. To demonstrate this, a subtraction example of two similar IEEE 754 inputs is provided. Considering inputs A and B to both be IEEE 754

double precision values set to the following:

$$\texttt{A,B(S, E, M)} = \begin{cases} (\texttt{1'h0}, \texttt{11'h002}, \\ \texttt{52'h0\_0000\_0000\_000F}), \\ (\texttt{1'h0}, \texttt{11'h001}, \\ \texttt{52'hF\_F000\_0000\_0003}) \end{cases} .$$

Without support for denomalized values, the subtraction of A - B results in a value of zero, since the value of the resulting exponent is less than 1. This obviously is not the case, and leaves room for a significant amount of truncation error when similar values are compared. With denormalized value support, significantly different results can occur:

$$\texttt{Result(S, E, M)} = (\texttt{1'h0}, \texttt{11'h000}, \\ \texttt{52'hF\_EFFF\_FFFF\_FFF4}) .$$

This particular example demonstrates the edge case between the normalized and denormalized range for IEEE 754 support, however, denormalized values can have exponents propagate down nearly all of the mantissa's vector size without issue. The need for denormalized implementations in floating-point hardware also goes beyond exclusively floating-point addition [10].

The IEEE 754 standard has support for a variety of exceptions that occur during floating-point operations, as well as instructions for how to propagate them through into output vectors. Table 2.1 provides a reference for IEEE 754 exceptions and their descriptions. Table 2.2 provides a brief reference of examples for binary vectors that trigger certain exception cases in IEEE 754 floating-point addition, including signaling and quiet Not-a-Numbers (i.e., sNaN and qNaN). The examples provided are formatted as binary32 input vectors.

A total of five different rounding modes are supported by the 2019 IEEE 754 standard [1]. These include: round-to-nearest-even, round-towards-away, round-towards-

Table 2.1: IEEE 754 Exception Description

Exception Type	Description
Invalid	Occurs for non-usable results, such as NaN and $+/-\infty$
Division by Zero	Infinite result is created from non-finite input vectors - Non possible for floating-point addition
Overflow	Result exceeds largest possible finite output - can be negated by rounding
Underflow	Result is small enough and non-zero so that it lies between bound of $+/-2^{exponent-min}$
Inexact	The rounded result differs from calculation with unbounded precision - results can still be used

Table 2.2: IEEE 754 Vector Exception Cases

Exception Case	Examples (binary32 Verilog - 'x' is don't care)
qNaN	Exponent is set to all 1's and mantissa is a non-zero value `1'bx,8'hFF,23'h7xxxxx`
sNaN	All conditions for qNaN exceptions and MSB of mantissa is set to 0 `1'bx,8'hFF,23'h3xxxxx`
$+\infty$	Exponent is set to all 1's, mantissa is set to all zeroes, and a non-subtracting operand is $+\infty$ `1'bx,8'hFF,23'h000000`
$-\infty$	Exponent is set to all 1's, mantissa is set to all zeroes, and a subtracting operand is $+\infty$ `1'bx,8'hFF,23'h000000`
Denormalization	Either input operand is already denormalized, or operation is effective subtraction and the difference between operands is less than $2^{exponent-min}$

Table 2.3: Absolute Error for round-to-nearest-even

(L,R,S)	*Result*	Absolute Error
000	0	+0.00
001	0	+0.25
010	0	+0.50
011	1	-0.25
100	0	+0.00
101	0	+0.25
110	1	-0.50
111	1	-0.25
Avg.	-	+0.00

Table 2.4: Absolute Error for round-towards positive infinity

(L,R,S)	*Result*	Absolute Error
000	0	+0.00
001	1	-0.75
010	1	-0.50
011	1	-0.25
100	0	+0.00
101	1	-0.75
110	1	-0.50
111	1	-0.25
Avg.	-	-0.38

positive-infinity, round-towards-negative-infinity, and round-towards-zero. These separate rounding options are provided for extra utility in use cases where certain operations may want to always round in a particular direction, or where particular rounding scenarios may be more frequent. Round-to-nearest-even has the lowest average error for each rounding case and is utilized as default within the IEEE 754 standard [1]. Tables 2.3 and 2.4 are included to demonstrate the absolute error found in using different rounding modes for each configuration of rounding bits. These rounding bits, *Least* (L), *Round* (R), and *Sticky* (S) are used to determine if a value should be rounded one *ulp* up or down. The specifics of how these bits are calculated is covered in more detail in Section 4.4. The goal of different rounding modes is to provide a minimized error for different input domain intervals. In the case where the distribution across a given input domain is uniform, round-to-nearest-even typically gives the best performance, as is shown in Table 2.3. For domain intervals with more known values skewed with positive rounding intervals than negative, or vice versa, other rounding modes can provide lower average error.

CHAPTER III

MICROARCHITECTURE BACKGROUND

3.1 Prefix Adder Topologies

One of the primary pieces of microarchitecture used throughout this design is a Kogge-Stone prefix adder [17]. It goes to mention that any of the ideas presented throughout this paper can be applied to other prefix adders. A prefix adder is a modified version of a carry-look-ahead adder that instead uses arbitrary operators to compute the generate and propagate signals between each stage of the adder network [15]. These are often called gray or black cells [18], the former of which only produces a generate/carry signal. The outputs of gray or black cells can be easily defined as:

$$\mathtt{g_i} = \mathtt{g_k} + \mathtt{p_k} \cdot \mathtt{g_{k-1}} \ ,$$

$$\mathtt{p_i} = \mathtt{p_k} \cdot \mathtt{p_{k-1}} \ ,$$

where g_i and p_i are the corresponding bitwise generate and propagate signals produced by gray or black cells, and g_k and p_k are the input signals to the gray or black cell. These can be arranged in a variety of configurations to produce high performance adders for various design specifications [15].

To keep delay at an absolute minimum throughout this design, a Kogge-Stone tree is utilized. Kogge-Stone trees have both a minimized critical path delay and the lowest fanout for any current prefix tree. However, this does come at the cost of higher power consumption and large area, since the raw number of gray and black cells used are highest in a Kogge-Stone design. However, since floating-point addition is such a fundamental operation to scientific computation [15], the power cost for using Kogge-

Stone was deemed a necessary trade off. These prefix adders are used throughout this architecture in a variety of bit-widths. Moreover, 12-bit prefix subtracters are used for exponent comparison and exponent rounding, a 64-bit end-around-carry prefix adder is used to perform the primary addition or subtraction operation, and 52-bit prefix adders are used to round the mantissa correctly at the end of this architecture's datapath, as well as detect denormalized underflow and overflow.

Modified versions of prefix adder designs are also used to improve rounding performance, particularly for denormalized cases. Specifically, a flagged prefix adder [18, 19], also implemented with a Kogge-Stone prefix tree design, is used to handle denormalized offsets for modifying exponent values. Flagged prefix adders differ from normal prefix adders by the inclusion of a flag signal, which based on the sum generation for the prefix adder R_k, as well as an unsigned constant offset M_k [18]. Both of these can be used to generate a flag signal F_k to combine with the normal sum generated by the prefix adder, which generates a modified sum with the offset of M_k. A list of equations showing the initial calculations for necessary signals are included below.

Flag prefix architectures take advantage of late-increment operations by modifying cells within the main prefix tree so that group generate (${G_{i-1}}^0$) and group kill ($\overline{{K_{i-1}}^0}$) signals are produced. Consequently, carry-out signals emerging from the prefix tree can be updated to form the appropriate late-carry signals. This is shown as a block diagram in Figure 3.1 where the M_k signal can augment the operation of the addition late. Essentially, a flag-prefix adder is a form of merged arithmetic [20]. The necessary logic for generating both the flagged prefix adder's carry and flag signals is shown here:

$$\begin{aligned} \mathtt{R_k} &= \mathtt{x_k} \oplus \mathtt{y_k} \oplus \mathtt{c_k} \ , \\ \mathtt{c_{k+1}} &= \mathtt{R_k} \cdot \mathtt{M_k} + \mathtt{R_k} \cdot \mathtt{c_k} + \mathtt{M_k} \cdot \mathtt{c_k} \ , \end{aligned}$$

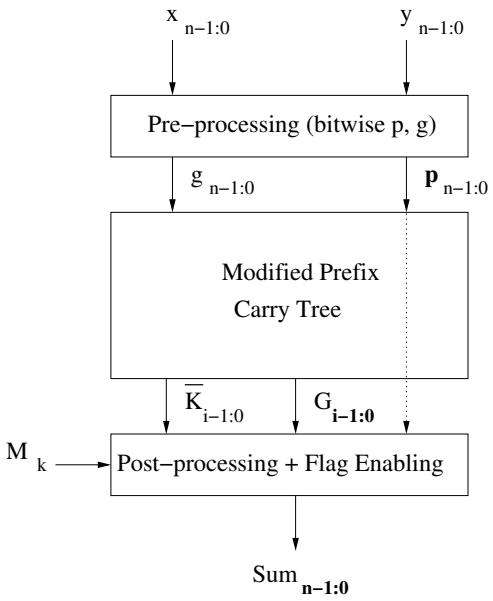

Figure 3.1: Block diagram of a flagged prefix adder

$$\mathtt{F_k} \;=\; \begin{cases} \mathtt{c_k} \;\; if \;\; \mathtt{M_k} = 0 \\ \overline{\mathtt{c}}_\mathtt{k} \;\; if \;\; \mathtt{M_k} = 1 \end{cases} .$$

The implementation of a flag prefix structure using an arbitrary constant M_k can be seen in Figure 3.2. To take advantage of using an unknown constant value, it is necessary to update the late-carry equations from the carry value produced by the prefix tree. This value, along with the flagged signal F_k generated from it, can be combined with the pseudosum of the two addends and M_k. This is used to produce the output flag logic specified in [19], which can be XOR'd with the sum normally generated from the carry prefix adder to produce an output offset by the constant M_k. This is shown in the select logic block of Figure 3.2, where the XOR'd pseudosum and carry R_k and F_k are used to generate the potential combinations of the flag output signal. The constant value M_k is used to choose which value is selected for flag output.

This implementation is used in the proposed design to properly round the exponent value as quickly as possible. By incorporating a constant offset into the exponent rounding adder, it is possible to consider every potential rounding case in a single

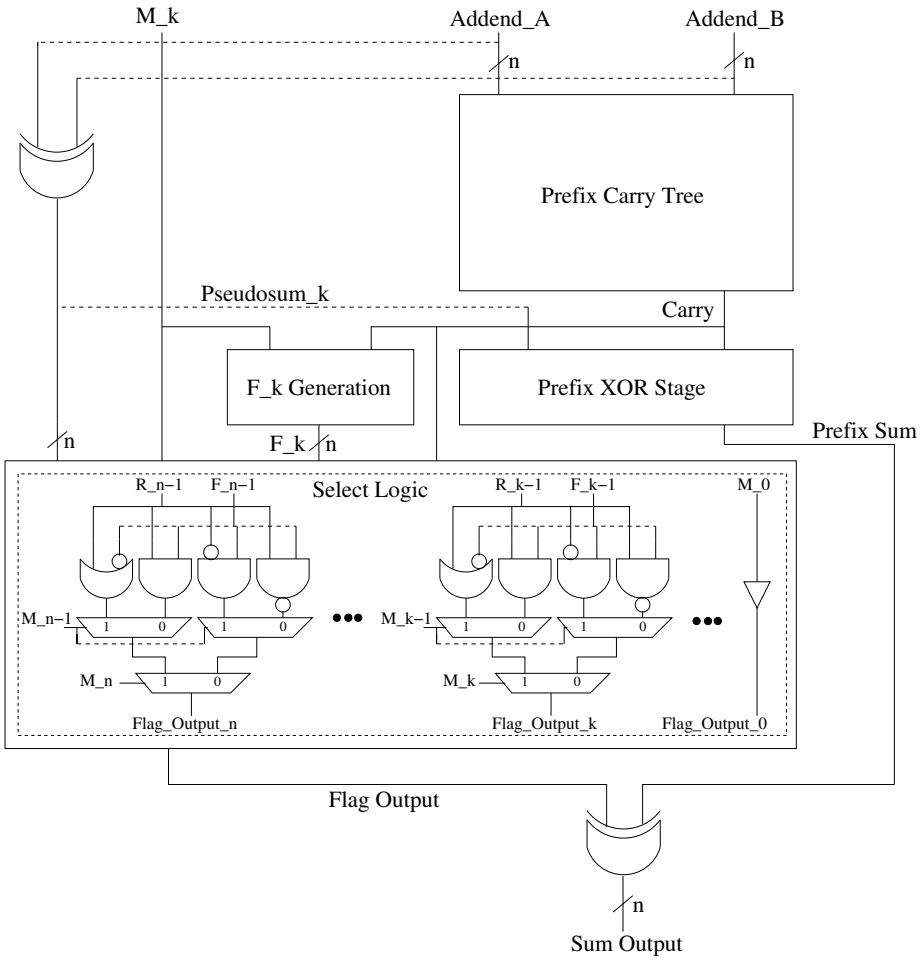

Figure 3.2: Diagram of a flagged prefix adder implementation for adding a constant M

operation. This is done by generating the offset for all the cases in an single value used in place of M_k. The calculation of this is further discussed in Section 4.4. This removes a significant amount of additional delay and power consumption that would otherwise be required by performing this operation with multiple sequential adders, as was done in previous work [3].

3.2 End-Around-Carry and Leading-Zero Anticipation

An end-around-carry (EAC) adder [14] is used for the primary addition and subtraction operations. This adder structure does not offer any significant increases in terms of delay performance, but it significantly reduces the area footprint of what would otherwise require two parallel 64-bit prefix adders for addition and subtraction. In many implementations [3], two parallel prefix adders are used to minimize delay. If there is a case where the result of effective subtraction would be negative (i.e. `A - B` *if* `B > A`) then the two's complement of the difference must be taken in order to achieve the correct result. Two parallel prefix adders are used to simultaneously compute both results, with a significant decrease in delay at the expense of area and power consumption. This can be combined together to form something called a compound adder. Overall, the idea is to integrate the carry within the carry chain so that it does not propoagate twice the length of the adder [14].

An EAC adder provides nearly the same delay results in the tree structure of a single adder by combining the carry equations for a two's complement comparator, as well as the normal carry equations for a prefix adder. The end effect is that this maintains the same critical path length through the prefix adder, at the expense of expanding the delay for any other carry bits. This does increase the number of transistors used in an EAC adder over a normal adder, but the reduced fan-out between stages makes the power cost for an EAC adder worth the implementation difficulty. An example of how the carry equations between a single carry bit of a

two's complement comparator and prefix adder can be combined into an EAC adder as shown below. The width of both the comparator and prefix adder are 4 bits for clarity:

$$\begin{aligned} \texttt{C0\_comp} &= \texttt{G0} + (\texttt{P0} \cdot \texttt{G1}) + (\texttt{P0} \cdot \texttt{P1} \cdot \texttt{G2}) + \\ &\quad (\texttt{P0} \cdot \texttt{P1} \cdot \texttt{P2} \cdot \texttt{G3}) + \\ &\quad (\texttt{P0} \cdot \texttt{P1} \cdot \texttt{P2} \cdot \texttt{P3}) \ , \\ \texttt{C2\_prefix} &= \texttt{G2} + (\texttt{P2} \cdot \texttt{G3}) + (\texttt{P2} \cdot \texttt{P3} \cdot \texttt{Cin}) \ , \end{aligned}$$

where `Cout_comp` is the carry out from a two's complement comparator, and `C2_prefix` is the third carry bit in a normal prefix adder. When these Boolean equations are integrated together, the overall length of the carry chain for the third carry bit becomes the same as the worst-case scenario carry chain for the normal prefix adder:

$$\begin{aligned} \texttt{C2\_ECA} &= \texttt{G2} + (\texttt{P2} \cdot \texttt{G3}) + (\texttt{P2} \cdot \texttt{P3} \cdot \texttt{G0}) + \\ &\quad (\texttt{P2} \cdot \texttt{P3} \cdot \texttt{P0} \cdot \texttt{G1}) + \\ &\quad (\texttt{P0} \cdot \texttt{P1} \cdot \texttt{P2} \cdot \texttt{P3}) \ , \\ \texttt{C0\_prefix} &= \texttt{G0} + (\texttt{P0} \cdot \texttt{G1}) + (\texttt{P0} \cdot \texttt{P1} \cdot \texttt{G2}) \\ &\quad + \ (\texttt{P0} \cdot \texttt{P1} \cdot \texttt{P2} \cdot \texttt{G3}) + \\ &\quad (\texttt{P0} \cdot \texttt{P1} \cdot \texttt{P2} \cdot \texttt{P3} \cdot \texttt{Cin}) \ , \end{aligned}$$

where `C2_ECA` is the third carry bit for an EAC prefix adder, and `C0_prefix` is the first carry bit for a normal prefix adder.

To further decrease the power consumption of an EAC adder, at the cost of delay, the carry out of the adder can instead be selectively integrated into the sum instead of back into the carry chain. This eliminates the need for the additional EAC logic present in the carry chain, as all the additional logic can be considered after the carry chain has been generated. The cost of this is an additional AND and XOR delay on top of the delay for the carry out of the adder. This methodology works regardless

of the carry chain generation system used, and is as follows:

$$\texttt{B\_EAC[n]} \;=\; (\texttt{Cout} \cdot \texttt{SUB\_OP}) \oplus \texttt{B[n]},$$

Where `B[n]` correspond to each of n bits of the second addend B, `Cout` is the carry-out of the carry chain, `SUB_OP` refers to whether a subtraction operation is occurring, and `B_EAC[n]` corresponds to each of n bits of the pseudosum to be XOR'd with the first addend and carry chain, which produces the final sum. This requires a significantly smaller amount of logic to implement, especially at higher radicies, making this an excellent choice to save power. This architecture uses this EAC methodology in its implementation.

Leading-zero detectors and anticipators (LZD's and LZA's) are also extremely important to floating-point addition. Leading-zero detectors, as their name describes, are able to accurately detect a count of the number of continuous zeroes on a binary input string, starting at the MSB [21, 22, 23]. These are used in floating-point addition to detect the proper shift amount needed for pre-normalization and post-normalization of the mantissa. For the post-normalization stage, since the correctly formatted inputs are already known prior, a leading-zero anticipator is used instead. This is used in parallel with the primary addition and subtraction operations, which reduces the critical path delay by removing a LZD directly after the primary addition in this architecture's datapath.

LZD's of any width are typically composed of smaller binary trees of LZD's, recurring down to the smallest input bit pattern that can be considered (i.e. 2 bits). For each LZD, both a signal for the validity of the LZD input pattern and a signal detecting the desired bit pattern need to be produced. The implementation of this for a 2-bit LZD is:

$$\begin{aligned} \texttt{V} &= \texttt{A}_0 + \texttt{A}_1 \;, \\ \texttt{P} &= \texttt{A}_0 \cdot \overline{\texttt{A}_1} \;, \end{aligned}$$

where A_0 and A_1 are the input signal to the LZD. The output for an LZD is invalid if all inputs throughout the LZD tree are zero. By using an OR operator for each node of the LZD tree, it is only necessary to simply invert the signal for the final output and achieve a correct valid signal. In this case, since it is necessary to detect a string of leading zeroes, a one in the LSB of the 2-bit LZD is used. This allows a leading-zero bit pattern on the odd numbered bits for an input string to be detected. Even numbered positions are considered during subsequent stages in the tree. The bit pattern can be manually detected for even strings, but it was found to be more efficient in this implementation to use the valid signals already generated from 2-bit stages. The implementation for this is as follows:

$$\begin{aligned} \mathtt{V4} &= \mathtt{V_0} + \mathtt{V_1} \ , \\ \mathtt{P4_0} &= \mathtt{V_0} \ ? \ \mathtt{P_0} \ : \ \mathtt{P_1} \ , \\ \mathtt{P4_1} &= \overline{\mathtt{V_0}} \ , \end{aligned}$$

where $P4$ is the pattern detection output for a 4-bit LZD, $V4$ detects the validity of the 4 bits it is considering, and both V and P are the outputs from a 2-bit LZD defined above. In this case, $P4_1$ is set to one if the two bit binary string from the least significant LZD is all zeroes, which is the same case for an invalid string when only considering the same LZD. An invalid signal for bit detection, instead of looking for specific bit patterns on even numbered bits in the binary input string for the LZD, is used. This concept can be used throughout the tree to make a LZD with minimal extraneous logic.

The overall goal presented in utilizing leading-zero antcipators is to move the normalizer prior to the adder. Moving the normalizer prior to the adder is not new. It has been done many times for fused-multiply and add (FMA) designs [24]. The reason FMA designs do this is the rounding is faster. However, the need for using LZAs combined with an EAC optimizes efficiency and speed while reducing energy,

especially for IEEE 754 addition and subtraction.

LZA's are, in basic terms, modified LZD's that instead take two inputs and produce a binary string with the correct number of leading zeroes to send into a standard LZD. There are two different classes of LZA's, exact and inexact [22]. Both classes of LZA's, before error correction, can produce a binary string that will allow a LZD to predict the correct number of leading zeroes within two bits. Two bits, however, is not good enough accuracy for the purposes of this design, so the correction logic from an exact or inexact LZA must also be implemented. Inexact LZA's typically have lower delay than exact LZA's, but they relay on prediction logic that is not always correct (i.e. has an error rate). This makes the use of inexact LZA correct logic a non-viable solution for this architecture. The binary correction tree used in [22] proved useful in the implementation of an exact LZA for this design, since it maintains an heavy bias towards reducing delay, albeit at the expense of power consumption.

CHAPTER IV

ARCHITECTURE DESIGN

4.1 Design Methodology and Input Handling

The philosophy of this article's architecture differs significantly in design from that of previous implementations [13, 3]. Overall, the goal is to provide the fastest single-cycle floating-point adder/subtracter possible, whereas, before design trade-offs were made to keep a balance between delay performance and power consumption. This is not to say that power consumption is not considered throughout this design, hence the use of EAC adders, but rather it takes a lower priority to delay performance in most all scenarios. On the other hand, the reduced area content of the EAC adder contributes to an overall lower energy footprint. This section will attempt to document the architecture design process as clearly as possible, making particular note to manually demonstrate some of the more complex Boolean equations required throughout the design. A top-level reference to the architecture can be found in Figure 4.1. Although this architecture can also be pipelined, the design is not pipelined so that it may be better compared against in the future.

Before the datapath flow is discussed in more detail, it is good to the components used in this architecture and what performance benefits they offer. The exponent comparison operation performed uses LZD's as a necessity to account for denormalized values, and the comparison between the normalized and denormalized exponent operands are run in parallel with four 12-bit Kogge-Stone prefix adders. These prefix adders offer the lowest fan-out and therefore smallest delay of nearly any prefix adder

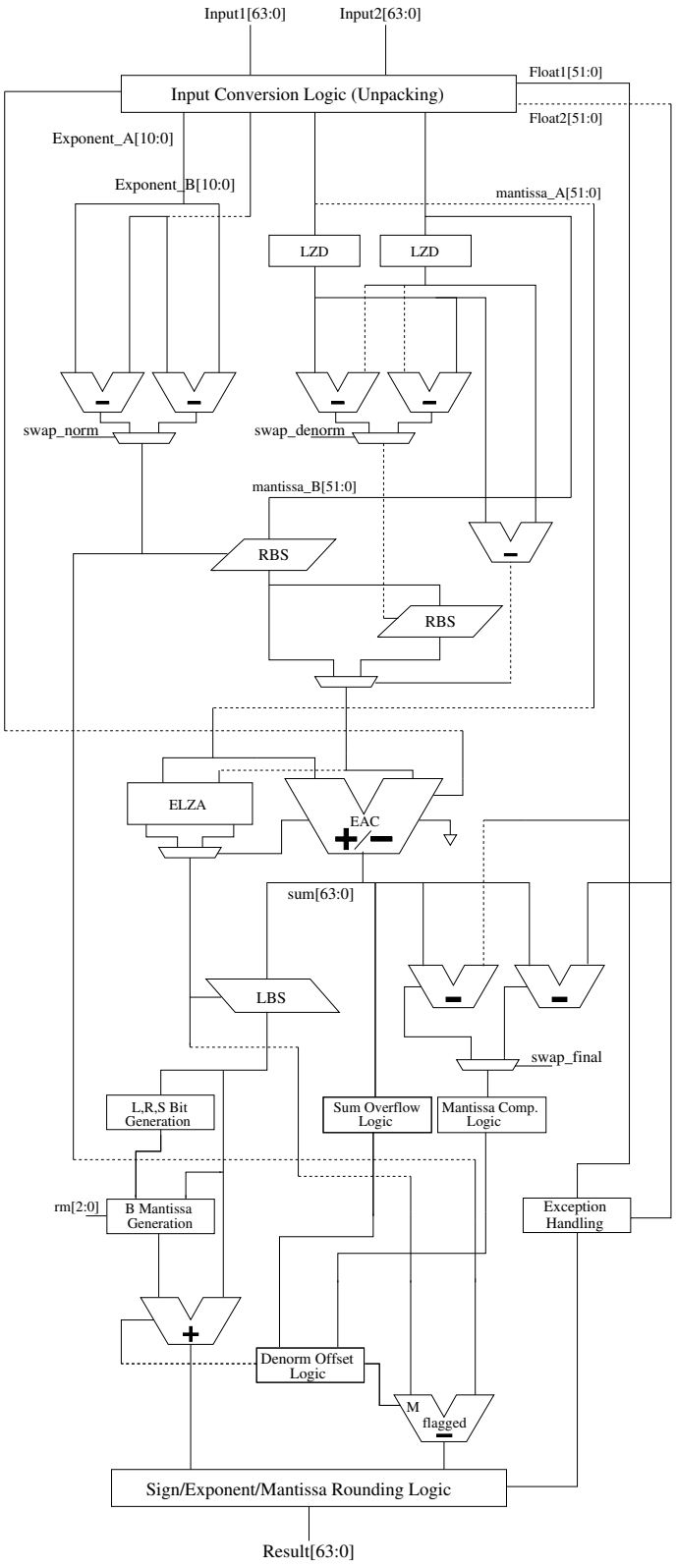

Figure 4.1: Top-level Design of IEEE 754 Floating-point Adder/Subtracter Architecture

architecture. Two parallel shifters for pre-normalization are used to reduce the fan-out between the exponent comparison and pre-normalization shift operations, further decreasing delay. An EAC adder with a Kogge-Stone carry prefix tree is used to maintain a very small delay and keep power-consumption levels within a reasonable domain (i.e. within 20% of leading designs). An ELZA is used to compute the post-normalization shift value in parallel with the primary addition operation. An ELZA tree is large, but necessary to not rely on inexact predictions of the post-normalization shift value. The power consumption is mostly offset by the EAC adder. Two comparators used for denorm edge cases and the adder used for mantissa rounding are also Kogge-Stone prefix adders. This is again to minimize critical path delay. The exponent rounding architecture uses a flagged Kogge-Stone prefix adder to keep delay small and account for all rounding cases in a single addition.

IEEE 754 compliant floating-point addition requires certain operands be performed regardless of the implementation specifics. Following the datapath from input to output [15]: a pre-normalization stage has to correctly align the mantissa values for both operands, the primary addition or subtraction operation occurs on the normalized mantissas, and the result of said addition or subtraction then has to be post-normalized to account for any leading zeroes in the result's mantissa, which finally has to be appropriately rounded. The structure of this section will mirror the flow of the datapath wherever possible.

Beginning to detail the datapath, this architecture takes binary64 values as inputs, both as a two's complement integers and as any supported IEEE 754 precision input. Although this design can easily be configured, as is done later, for only IEEE 754 addition or subtraction, the architecture works for both designs in an efficient implementation. In the event that an IEEE 754 input is given that uses any precision lower than IEEE 754 double-precision, all of the least-significant bits that are not used by that precision are simply set to zero. For example, if an IEEE 754 single-precision

input is given, the exponent bits 0 through 2 are zeroed, as IEEE 754 single-precision exponents uses 8 exponent bits, instead of 11. These input values are initially fed into an input-conversion module. This module decodes both the input operands themselves as well as the opcodes provided to the architecture. It checks for whether the operation is a precision conversion, type conversion, and easily recognizable patterns for some exception cases. There are two opcodes provided to the input conversion module, and as necessary for specific cases throughout the rest of the architecture. These are `P[1:0]` and `op_type[3:0]`. The former of which controls the precision to be used for each piece of microarchitecture, and the latter controls the operation to be completed, be it an arithmetic or conversion operation. A full list of instructions that can be performed by both opcodes can be found in Table 4.1. The logic used for determining when conversion operations are active is shown below, where `conv_SP` determines if a single-precision conversion is occuring, and `conv_HP` determines the same for half-precision:

$$\begin{aligned}
\texttt{conv_SP} &= \overline{\texttt{op_type[3]}} \oplus \overline{\texttt{P[1]}} \cdot \\
&\quad \texttt{P[0]} \cdot (\overline{\texttt{op_type[2]}} + \overline{\texttt{op_type[1]}}) \ , \\
\texttt{conv_HP} &= \overline{\texttt{op_type[3]}} \oplus \overline{\texttt{P[0]}} \cdot \texttt{P[1]} \cdot \\
&\quad (\overline{\texttt{op_type[2]}} + \overline{\texttt{op_type[1]}} + \\
&\quad \texttt{op_type[0]}) \ .
\end{aligned}$$

A few other useful aspects of the operations supported, including negation, are also computed and shown here:

$$\begin{aligned}
\texttt{negate} &= \texttt{op_type[3]} \cdot \texttt{op_type[2]} \cdot \\
&\quad \overline{\texttt{op_type[1]}} \cdot \texttt{op_type[0]} \ , \\
\texttt{abs_val} &= \texttt{op_type[3]} \cdot \texttt{op_type[2]} \cdot \\
&\quad \overline{\texttt{op_type[1]}} \cdot \overline{\texttt{op_type[0]}} \ ,
\end{aligned}$$

$$\texttt{Float1[63]} = (\texttt{op1[63]} \oplus \texttt{negate}) \cdot \overline{\texttt{abs\_val}} \ ,$$

$$\texttt{Float2[63]} = \texttt{op2[63]} \ .$$

The effective signs of the operands are also calculated in `Float1[63]` and `Float2[63]`.

In parallel to the input conversion module, a dedicated exception module is used to detect any input vector considered invalid or denormalized, as was previously described in Table 2.1. This allows a few different number formats to be checked, namely: infinity, signaling (`sNaN`) or quiet NaN's (`qNaN`), and zero. Operations are declared invalid if either input is a `sNaN`, or if both inputs are infinite and effective subtraction takes place. Operations are considered denormalized if operand A is itself denormalized and conversion is not occurring, or if operand B is denormalized and either an addition or subtraction operation is occurring. Operations can be determined to be positive infinity if operand A is positive or if operand B is negative infinity and a subtraction operation is occurring, and the same can be determined for negative infinity by inverting the signs of both operands. This, however, requires that neither operand is a `sNaN` or `qNaN`. All of this is covered by the equations included below, using the operand nomenclature from Table 4.1:

$$\begin{aligned}
\texttt{add_sub} &= \overline{\texttt{op_type[2]}} \cdot \overline{\texttt{op_type[1]}} \ , \\
\texttt{Invalid} &= \texttt{ASNaN} + \texttt{BSNaN} + \\
&\quad (\texttt{add_sub} \cdot \texttt{AInf} \cdot \texttt{BInf} \cdot \\
&\quad (\texttt{A[63]} \oplus \texttt{B[63]} \oplus \texttt{op_type[0]})) \ , \\
\texttt{Denorm} &= \texttt{ADenorm} \cdot (\texttt{op_type[2]} + \\
&\quad \overline{\texttt{op_type[1]}}) + \texttt{BDenorm} \cdot \texttt{add_sub} \\
\texttt{ZQNaN} &= \texttt{Invalid} + \texttt{ANaN} + \\
&\quad (\texttt{BNaN} \cdot \texttt{add_sub}) \ , \\
\texttt{ZPInf} &= (\texttt{AInf} \cdot \texttt{A[63]} +
\end{aligned}$$

$$
\begin{aligned}
& \texttt{add\_sub} \cdot \texttt{BInf} \cdot (\overline{\texttt{B[63]}} \\
& \oplus \texttt{op\_type[0]}) \cdot \overline{\texttt{ZQNaN}} \ , \\
\texttt{ZNInf} \ = \ & (\texttt{AInf} \cdot \overline{\texttt{A[63]}} + \\
& \texttt{add\_sub} \cdot \texttt{BInf} \cdot (\texttt{B[63]} \\
& \oplus \texttt{op\_type[0]}) \cdot \overline{\texttt{ZQNaN}} \ .
\end{aligned}
$$

For ease of use outside the exception module, all of the exception signals are output into a single vector format `sel_inv[3:0]`, or 'select invalid'. The encoding for this output vector is shown in Table 4.2, which is used in necessary scenarios throughout the rest of the architecture.

The conversion to double from single or half precision occurs inside the logic of the input converter, while the conversion from double precision to either single or half precision occurs during rounding. This is necessary since the primary adder for this architecture is 64 bits wide, and thus all operations that occur within the architecture have to be double precision. The conversion between precision types occurs in a number of steps. The sign of the original operand is kept the same and directly transferred to the new value. The exponent must be converted between precisions as well, and unfortunately is the most complex part of precision conversion. The MSB is kept the same between exponents, no matter the specific precision conversion. When increasing precision (e.g. converting from half to single precision), three bits of the opposite value of the MSB are buffered between the MSB and the bit immediately preceding it. For decreasing precision, these three bits are truncated. Otherwise, bits are directly transferred between precisions for converting exponents. This effectively adds or subtracts the difference between conversion factors for exponent values, using techniques discussed in [1]. For example, when converting between single and double precision, the three addition bits buffered in the exponent are equivalent to adding 896_{10} to the exponent. This is the difference between conversion factors for

Table 4.1: Operations for Adder/Subtracter

Operation	op_type	P	Description
add.d	0000	00	Add two 754 double precision numbers
add.s	0000	01	Add two 754 single precision numbers
add.h	0000	10	Add two 754 half precision numbers
sub.d	0001	00	Subtract two 754 double precision numbers
sub.s	0001	01	Subtract two 754 single precision numbers
sub.h	0001	10	Subtract two 754 half precision numbers
cvt.w	0010	00	Convert a 64-bit two's complement integer to a 754 double precision number
cvt.w	0010	01	Convert a 64-bit two's complement integer to a 754 single precision number
cvt.w	0010	10	Convert a 64-bit two's complement integer to a 754 single precision number
cvt.b	0011	00	Convert a 32-bit two's complement integer to a 754 double precision number
cvt.b	0011	01	Convert a 32-bit two's complement integer to a 754 single precision number
cvt.b	0011	10	Convert a 32-bit two's complement integer to a 754 half precision number
cvt.h	0110	00	Convert a 16-bit two's complement integer to a 754 double precision number
cvt.h	0110	01	Convert a 16-bit two's complement integer to a 754 single precision number
cvt.h	0110	10	Convert a 16-bit two's complement integer to a 754 half precision number
abs.d	0100	00	Absolute value of a 754 double precision number
abs.s	0100	01	Absolute value of a 754 single precision number
abs.h	0100	10	Absolute value of a 754 half precision number
neg.d	0101	00	Negate a 754 double precision number
neg.s	0101	01	Negate a 754 single precision number
neg.h	0101	10	Negate a 754 half precision number
cvt.s.d	0111	00	Convert from a single precision number to a 754 double-precision number
cvt.d.s	0111	01	Convert from a double precision number to a 754 single-precision number
cvt.h.d	0111	10	Convert from a half precision number to a 754 double-precision number
cvt.d.h	0111	11	Convert from a double precision number to a 754 half-precision number
cvt.s.h	1111	10	Convert from a single precision number to a 754 half-precision number
cvt.h.s	1111	11	Convert from a half precision number to a 754 single-precision number

Table 4.2: IEEE 754 exception detection

sel_inv[3:0]	Output State
0000	Normal
0001	Quiet NaN
0010	Negative Infinity
0011	Positive Infinity
0100	+Bzero and +Azero (and vice-versa)
0101	+Bzero and -Azero (and vice-versa)
1000	Convert SP to DP or HP

double precision (1023_{10}) and single precision (127_{10}). A graphical version of this can be found in Figure 4.2. When converting between floating-point formats, mantissa conversion is easy to do. The mantissa is either truncated to the proper size when decreasing precision, or the LSB is buffered with zeroes when increasing in precision.

In order to make sure the architecture comparisons are as accurate as possible, the required hardware for these conversion instructions is removed. The results without this hardware is presented in Section 5.1.

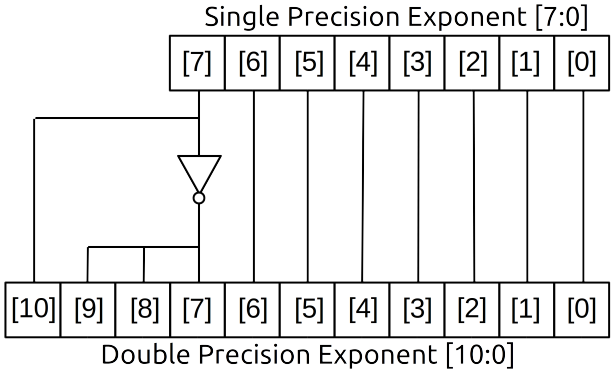

Figure 4.2: IEEE 754 Single-Precision to Double Precision Conversion

4.2 Exponent Comparison and Pre-normalization

For non-conversion operations, the exponent and mantissa values for both operands are immediately compared to determine the correct shift amount necessary for the pre-normalization stage of floating-point addition. The exponent values themselves are sent through a pair of parallel 12-bit carry-prefix subtracters using the Kogge-Stone tree structure covered earlier. The exponent inputs are swapped between adders, so that comparisons of both `exp1 - exp2` and `exp2 - exp1` are done at the same time. However, this neglects how to handle the case of denormalized exponents. Since denormalized input operands have an exponent value of zero, the leading zeroes in the mantissa have to be used for exponent comparison instead. To implement this, a pair of leading zero detectors are used on both mantissa values. The results of which are then immediately send to another pair of Kogge-Stone prefix adders, in parallel with those used for normal exponent values. Both of these sets of prefix adders provide separate shift amounts for both the normalized and denormalized range of exponent values. The calculation for the potential denormalized shift values are:

$$\begin{aligned} \texttt{lz\_diff1} &= \texttt{ZP\_exp1} - \texttt{ZP\_exp2} \ , \\ \texttt{lz\_diff2} &= \texttt{ZP\_exp2} - \texttt{ZP\_exp1} \ , \end{aligned}$$

where `ZP_exp1` and `ZP_exp2` are the leading zeroes from both mantissas. In order to determine which shift values are used for pre-normalization, the differences of both sets of prefix adders are each sent through a MUX. The select signals for both are defined as:

$$\begin{aligned} \texttt{zeroB} &= \texttt{op\_type[2]} + \texttt{op\_type[1]} \ , \\ \texttt{swap\_norm} &= \texttt{expdiff\_12[11]} \cdot \overline{\texttt{zeroB}} \ , \\ \texttt{swap\_denorm} &= \texttt{lz\_diff\_12[11]} \cdot \overline{\texttt{zeroB}} \ . \end{aligned}$$

The output `swap_denorm` is used to determine which exponent difference should be used for denormalized cases, and `swap_norm` does the same for normalized cases. The MSB for the first of both sets of prefix subtracters is used to determine which exponent is the appropriate shift value for pre-normalization. This is also used to determine which input operand is smaller, and therefore which mantissa needs to be right shifted.

Once both shift amounts are provided from the exponent comparison stage of the architecture, the pre-normalization of both input vectors can begin. This is done by taking the exponent from the smaller mantissa and right shifting it by the differences between exponents. A right shift is performed for both the normalized exponent difference and denormalized exponent difference. Two 57-bit barrel shifters are used to perform the right shift operations, which is referred to as RBS in Figures 4.1 and 4.3. Mantissas that are shifted by only normalized exponent comparison and mantissas that are shifted by both normalized and denormalized exponent comparison are calculated. `swap_denorm`, in combination with the carry out from the first denormalized exponent, is used to determine which normalized shift value is used for the rest of the datapath. A block diagram detailing exponent comparison and pre-normalization operations can be referenced in Figure 4.3. Mantissa integer conversion also happens at this stage in the architecture, but it is not on the critical path. Integer conversion for all supported precisions is relatively simple, as the mantissa only needs to be sign extended to the corresponding width needed:

```
 IntValue[15:0]  =  op1[15:0] ,
IntValue[31:16]  =  P[1] ? {16{op1[15]}} :
                    op1[31:16] ,
IntValue[63:32]  =  P[1] ? {32{op1[15]}} :
                    (P[0] ? {32{op1[31]}} :
```

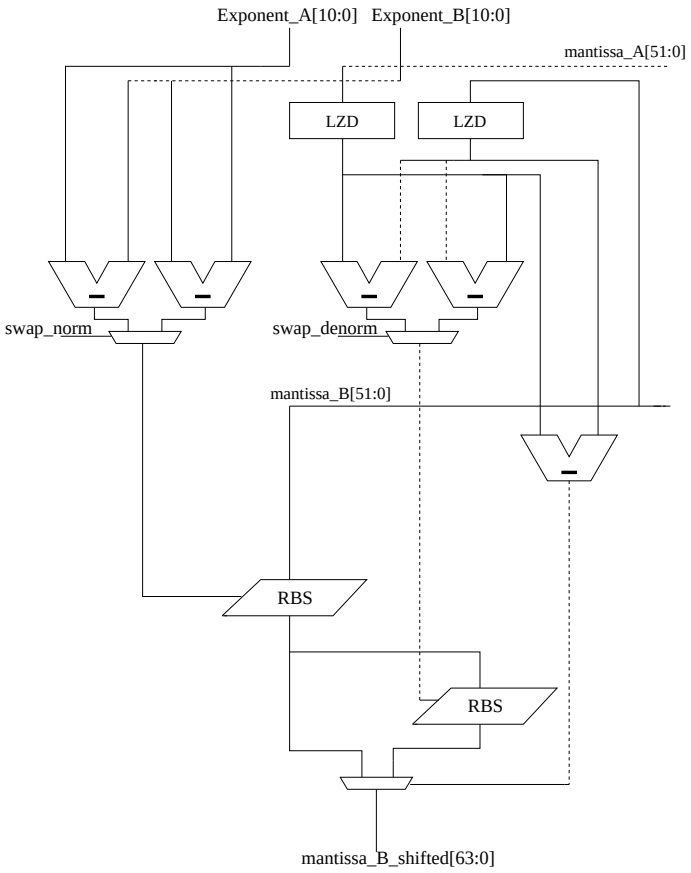

Figure 4.3: Exponent Subtraction and Mantissa Formatting Subsections

`op1[63:32]`) .

For ease of implementation, the operand that needs to be converted is always set to the first operand.

Previous iterations of this architecture [3] had a critical path which involves two sequential comparison subtractions and one right shift. Through further implementation and testing, it has been discovered that a critical path involving one comparison subtraction and two right shifts has a lower critical path delay when combined with larger pieces of microarchitecture. This is most likely due to the decreased fanout of two separate barrel shifters, which unfortunately comes at the cost of increased static power consumption due to the larger number of MUX's used in synthesis.

4.3 Primary Addition/Subtraction and Post-normalization

Once both mantissas have been normalized with each other, they can be sent to the primary adder/subtracter for the architecture. A particularly useful component of this architecture is the introduction of an adder/subtracter structure that uses a 64-bit end-around-carry prefix adder (EAC) to simultaneously compare the sums of the two input values and compute the corresponding necessary output to keep the result between the range of $[1, 2)$. As mentioned in the previous section, an EAC prefix adder can take the place of a prefix adder-subtracter pair. This provides an obvious decrease in device area and power consumption, but it also provides some delay performance benefits as well. For one, only having one large adder to drive in the datapath significantly decreases the necessary fan-out of other microarchitectures. The output sign is also unnecessary to calculate, since the proper mantissa is always selected from the EAC adder, reducing the logical path delay by a MUX.

To prepare the sum of the EAC prefix adder for rounding, the two mantissa inputs sent to the EAC prefix adder are also fed into a pair of exact leading-zero anticipators (ELZA). In the case where the EAC's result is based on the two's complement of the smaller mantissa, the same mantissa value is inverted before it reaches the second ELZA. This way, no matter whether the two's complement result is used, the appropriate post-normalization shift value for the sum of the EAC can be computed. This will detect the number of leading zeroes needed to post-normalize the sum produced by the EAC prefix adder. This shift value, along with the sum produced from the EAC, is immediately moved into a 64-bit left barrel shifter, referenced as LBS in Figures 4.1 and 4.4. The extra bit width is to ensure none of the data produced from the EAC adder is lost until the mantissa is rounded. This produces a mantissa within the required fixed domain values for IEEE 754 support, and the shift amount required to reach this domain is kept to later adjust the final exponent value during rounding.

Figure 4.4 shows the datapath flow for this, as well as some of the exponent underflow logic. The reason this underflow logic is necessary is to account for exponent rounding due to denormalized edge cases. In other words, it needs to be determined if the sum from the EAC is small enough to be designated as denormalized, or conversely, if the sum from the EAC is large enough to result in a normalized value from two denormalized inputs. If the criteria for either of these cases are met, the `norm_ovflow` or `norm_unflow` signal will be set high.

For either case, the rounded exponent may need to be offset by one if `'norm_ovflow'` or `'norm_unflow'` occurs, as the value for this exponent would translate from zero to one in the case of `'norm_ovflow'` or one to zero for `'norm_unflow'`. The value for `'norm_ovflow'` is easy to determine. It can be seen without any extra operations by using the 52nd bit of the pre-normalized sum, where the LSB for a double-precision exponent would change to one, thus making the result normalized. This is only used in the event of denormalized operands increasing to the normalized range, so other scenarios where this might occur do not need to be considered.

The case for the `'norm_unflow'` signal is unfortunately difficult to compute, since it only occurs during edge cases where the difference between two values results in a denormalized number. This edge case only happens during effective subtraction, and there has to be a magnitude decrease between the largest original operand and the sum the EAC adder produces. To check all cases where this occurs, two 52-bit subtraction comparators have to be used to fall under the critical path delay used during mantissa rounding. The logical equation for this is shown below:

$$\begin{aligned} \texttt{norm\_unflow} \;=\; & ((\texttt{opA\_Norm} + \texttt{opB\_Norm}) \cdot \\ & (\overline{\texttt{Float1[63]}} \oplus \overline{\texttt{Float2[63]}})) \\ & ?\ \texttt{mantissa\_comp} : \texttt{1'b0}\ , \end{aligned}$$

where `mantissa_comp` is the Boolean result of the comparison between the EAC's

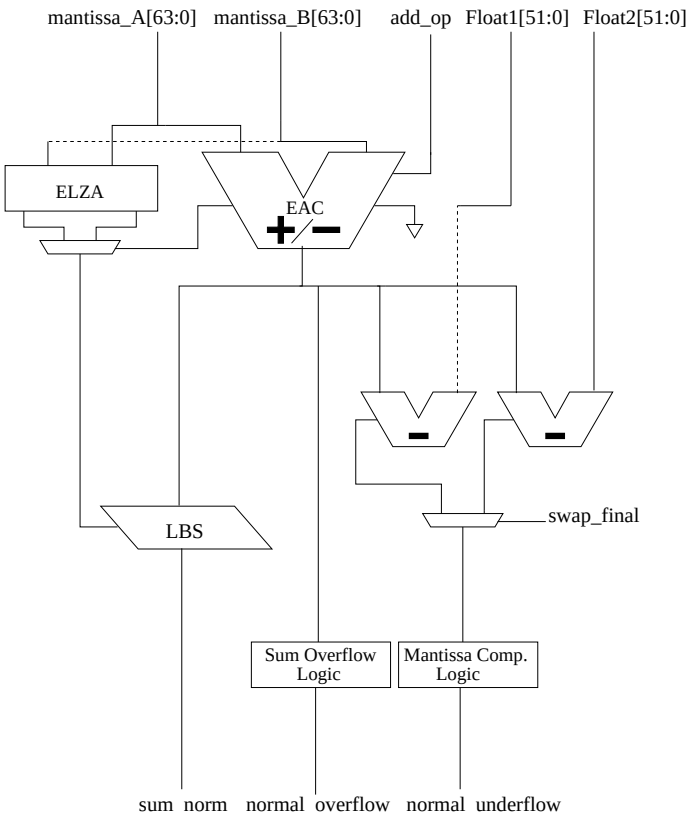

Figure 4.4: EAC Prefix Adder and Post-normalization Datapath

sum and the largest input mantissa value. To select between comparisons, the `swap_denorm` signal is used. Just like with `'norm_ovflow'`, this value is only necessary for calculations on denormalized edge cases, so whether normalized values will set this signal high does not have to be considered. This will allow exponent values to be correctly rounded, no matter the edge case. If `'norm_unflow'` is detected, the exponent value is decreased by one, and if `'norm_ovflow'` is detected, the exponent value is increased by one. This is all taken into account during the computation of the denomalized offset, which is used for the final exponent value before rounding.

4.4 Rounding

In order to be fully IEEE 754 compliant, the rounding process must consider all exception cases and invalid results supported and produce rounding flags in addition

Table 4.3: IEEE 754 rounding mode bits

`rm[2:0]`	IEEE 754 Rounding Mode
000	round-to-nearest-even
001	round-towards-zero
010	round-towards positive infinity
011	round-towards minus infinity
100	round-towards away

to a final result [2]. The encoding used for each rounding mode is shown in Table 4.3, using `rm[2:0]` as the signal name. Rounding must be able to normalize the mantissa if it exceeds its maximum value (i.e., ≥ 2.0), typically called post-normalization [1]. In Figure 4.5, rounding is effectively divided into two datapaths. One datapath handles mantissa rounding while the other handles rounding for the exponent. The rounding for the exponent is dependent on the carry out of the adder used for mantissa rounding, forcing both adds to be part of the critical path.

Mantissa rounding for IEEE 754 compliance is handled by using a least-significant bit (L), round digit (R), and a sticky bit (S). The bits are set accordingly by the following where S_{HP}, S_{SP}, and S_{DP} represent the appropriate sticky bits for half, single and double precision, respectively [25]:

$$\texttt{(L, R, S)} = \begin{cases} \texttt{A[53]}, \texttt{A[52]}, S_{HP} \;\; \textit{if } P = 10 \\ \texttt{A[40]}, \texttt{A[39]}, S_{SP} \;\; \textit{if } P = 01 \\ \texttt{A[11]}, \texttt{A[10]}, S_{DP} \;\; \textit{if } P = 00 \end{cases} .$$

The S_{HP}, S_{SP}, and S_{DP} sticky bit values represent the logical OR of all bits preceding the round digit for each precision. All of these values are computed in the L, R, S-bit *Generation* block of Figure 4.5, and the correct version is chosen based on the precision needed by the operation. A signal `HP_output` is set high if the rounding module's output needs to be half precision, based on the `P[1:0]` vector. All of these rounding bits are used to determine whether the value of one needs to be added to the

post-normalized mantissa in order to correctly round up or down. A one is required to add to the result if ((the rounding mode is round-to-nearest) and (R is one) and (S or L is one)) or ((the rounding mode is towards plus or minus infinity (`rm[1] = 1`)) and (the sign and `rm[0]` are the same) and (R or S is one)) or (R is one) and (rounding mode is towards away). This can be written by the following Boolean logic:

$$\begin{aligned} \texttt{add\_one} \;=\; & (\overline{rm[1]} \cdot \overline{rm[0]} \cdot R \cdot (L + S)) + \\ & (rm[1] \cdot (Asign \oplus \overline{rm[0]}) \cdot (R + S)) \\ & +\; (\texttt{rm[2]} \cdot R)\ . \end{aligned}$$

This value is the output of the *B Mantissa Generation* block found in Figure 4.5. This is used in combination with necessary precision logic to generate a vector `B[63:0]` to add to `A[63:0]`. The one is added where the LSB of the mantissa is for each precision, normalized to a 64-bit vector:

$$\begin{aligned} \texttt{B[63:0]} \;=\; & \texttt{\{\{10\{1'b0\}\},} \\ & \texttt{add\_one} \cdot \texttt{HP\_output}, \texttt{\{12\{1'b0\}\},} \\ & \texttt{add\_one} \cdot \texttt{P[0]}, \texttt{\{28\{1'b0\}\},} \\ & \texttt{add\_one} \cdot \overline{\texttt{P[0] + HP\_output}\}}. \end{aligned}$$

After this value is known, the carry-out from this sum can be utilized in combination with values from underflow logic, referenced in Figure 4.4. This can be used to adjust the value of the exponent, along with the number of bits that were required to post-normalize the sum produced from the EAC (i.e. `norm_shift`) per the following equation:

$$\begin{aligned} \texttt{Texp} \;=\; & \texttt{\{1'b0, Aexp\}} - \\ & \texttt{\{\{6\{1'b0\}\}, norm\_shift\}} \\ & +\; \texttt{denorm\_OS}\ , \end{aligned}$$

```
denorm_OS = normal_underflow ?
            (cout_mantB ?
            ({{9{VSS}},VSS,VDD,VSS} :
            {{9{VSS}},VSS,VSS,VDD}) :
            (cout_mantB ?
             {{9{VSS}},VDD,VSS,VSS} :
            ({{9{VSS}},VSS,VDD,VDD}) :
            {{10{VSS}},VDD,cout_mantB} .
```

The `cout_mantB` signal is what comes from the final carry out of the adder next to the $BMantissaGeneration$ block. `normal_underflow` selects between offset values of one through four, depending on `cout_mantB`. This offset value is used as the arbitrary value of M for a flagged prefix subtracter. This allows all denormalized exponent rounding scenarios to be accounted for within the delay of a single carry prefix adder. This can be seen in the subtracter structure found in Figure 4.5. This is significantly faster than using underflow or overflow in boundary solutions, as is shown in [10].

A few more rounding considerations must be taken into account before the computation of the final rounded exponent. The normalized exponent `Texp` is set to all ones during NaN and Infinite exception cases, and all zeros during zero and some denormalized value cases. To implement an overflow trap on the normalized exponent, the two MSB's of the exponent are inverted during an operation where overflow occurs. The bits that are actually inverted will vary based on precision. For example, bits 7 and 8 are inverted during a single precision operation.

Fortunately, rounding for the mantissa is much easier to account for than either the sign or exponent value. The mantissa is set to all ones during exception cases where the result is equal to the largest floating point value representable, or during NaN's. The mantissa also has to be set to all zeros during either zero or Infinity

exception cases.

By far, the more difficult part of an IEEE 754 value to correctly round is the sign. The final sign value output depends on the rounding mode and any overflow or underflow cases. Specifically, the sign of the final result is one if the result is not zero and the sign of A is one, or if the result is zero and the rounding mode is round-to-minus infinity. The final result must also be considered zero if `exp_valid` is zero (i.e., the exponent is not a valid exponent). If underflow occurs into the denormal range, the original sum and unmodified exponent values (i.e., the exponent values immediately preceding exponent comparison) are used to determine the resulting sign. During an addition operation, if the original sum has a MSB of one, any one of the input operands were originally normalized, and the original exponents have different MSB's, then the sign of A is set to zero. For subtraction operations occurring during underflow, if exclusively either original operand is normalized, and the signs of original exponents are the same, then the sign of A is set to zero. Otherwise, A is left as it would be regardless of underflow. This is summarized in Table 4.4, where descriptions of the vectors included are as follows: `A_Norm` and `B_Norm` indicate if the original operands were normalized. `exp_A_unmod` and `exp_B_unmod` refer to the original input operand's exponents. `Asign` is the sign of A determined after the primary addition/subtraction operation occurs, and `sum` is the mantissa output immediately before post-normalization occurs.

Finally, in order to be IEEE 754 compliant, the architecture must output the correct five IEEE 754 flags. These flags are Inexact, Underflow, Overflow, Divide by 0, and Invalid. Since a Division by 0 cannot occur, this is always deasserted. Overflow only occurs if the exponents produce its maximum value. Similarly, underflow occurs if the exponent produces a binary value of 0 or below. The overflow and underflow flags should not be set if the input was infinite or NaN, or if the output of the adder is zero. The final result is Inexact if any rounding occurs ((i.e., R or S is one), or (if the

Table 4.4: Rsign Expression Table

Result Case	Subcases	
Denormalized	Subcase	Boolean Expression
	Addition	$sum[63] \cdot (ANorm{+}BNorm) \cdot \overline{(expAunmod[11]\oplus \overline{expBunmod[11]})}$
	Subtraction	$sum[63] \cdot (ANorm{+}BNorm) \cdot (expAunmod[11]\oplus expBunmod[11])$
	Norm Unflow	$\sim Asign$
Normalized	Zero Exception & rm -∞	1'b1
	Invalid Exception & rm -∞	1'b1
	Conversion	$Asign$
	All Others	$Asign \cdot exp\_valid$

Table 4.5: Invalid/Valid IEEE 754 [1, 2] Operations

Operation (+/-)	sNaN	qNaN	Normalized Number	Infinity	Zero
sNaN	qNaN	qNaN	qNaN	qNaN	qNaN
qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
Normalized Number	qNaN	qNaN	IEEE 754 [1]	Infinity	IEEE 754
Infinity	qNaN	qNaN	Infinity	qNaN/Infinity	qNaN
Zero	qNaN	qNaN	IEEE 754	Infinity	Zero

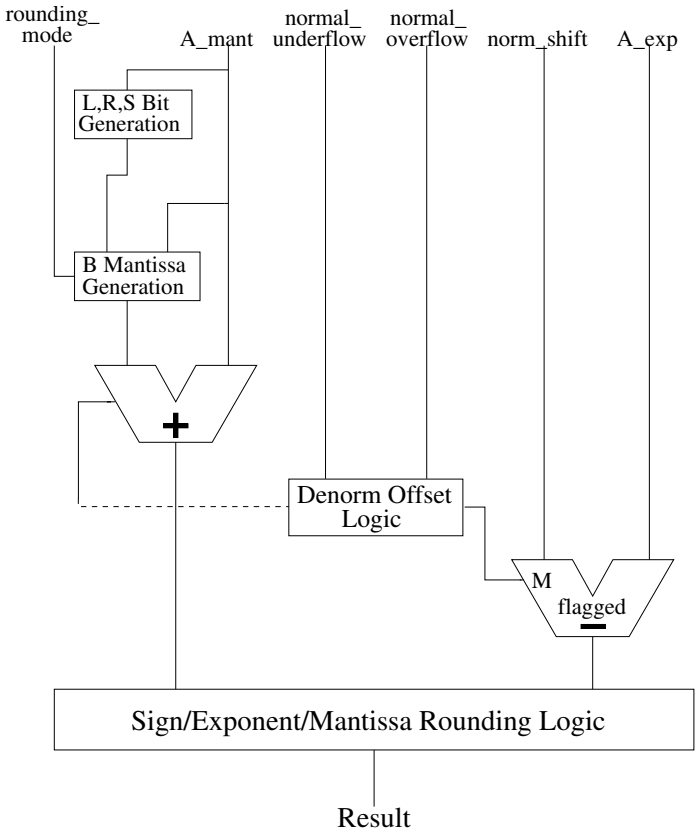

Figure 4.5: Simplified Block Diagram of Rounding Module

result overflows) or (if the result underflows and the underflow trap is not enabled)) and (the value of the result was not previously set by an exception case). A summary of Invalid vs. Valid operations is shown in Table 4.5.

4.5 Comparison to Previous Work

The architecture implementation presented in this thesis has been directly iterated upon from previous publications, namely [3]. Most of the architecture's datapath has been changed, but I will highlight the most significant and impactful changes to the architecture that have been made, following the same datapath order used in Section IV. This includes a subsection on the novel exponent rounding structure introduced for this thesis only.

In previous work, the structure of the subtractors used for exponent comparison

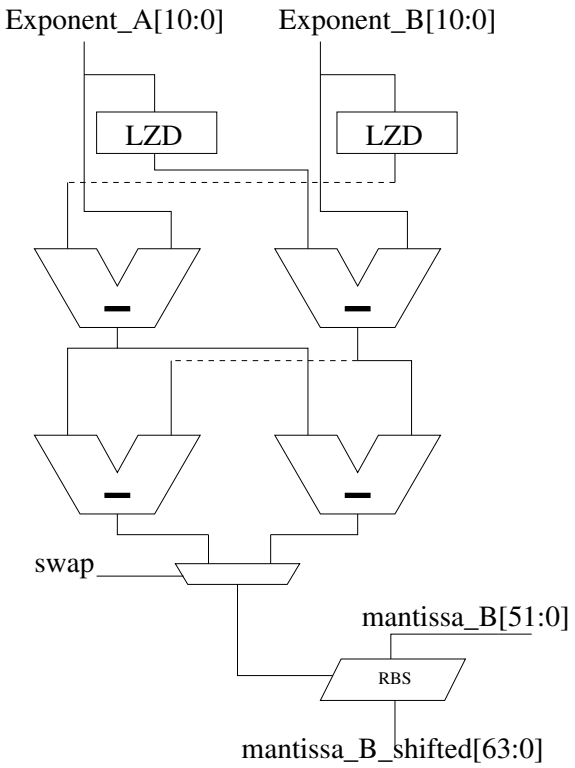

Figure 4.6: Exponent Comparison Architecture from [3]

were less parallelized. An initial set of subtractors were used to account for exponent differences that would occur in denormalized values. The results produced from this were moved into a second set of subtractors, which would compare the denormalized differences with the values the exponent operands provide directly. The current architecture takes this subtractor structure and performs both the denormalized and normalized exponent comparison in parallel. The cost for performing this operation in parallel is that the pre-normalization shift has to occur sequentially. Since the normalized shift value can be computed much faster than the denormalized shift value, the initial normalization shift can occur much earlier. This means the critical path for the exponent comparison stage follows through the LZD's, denormalized subtractors, and finally the second of the sequential pre-normalization shifters. This effectively saves the delay of an entire subtractor for the critical path. Figure 4.6 shows the architecture used for exponent comparison to previous work, while Figure 4.3 shows

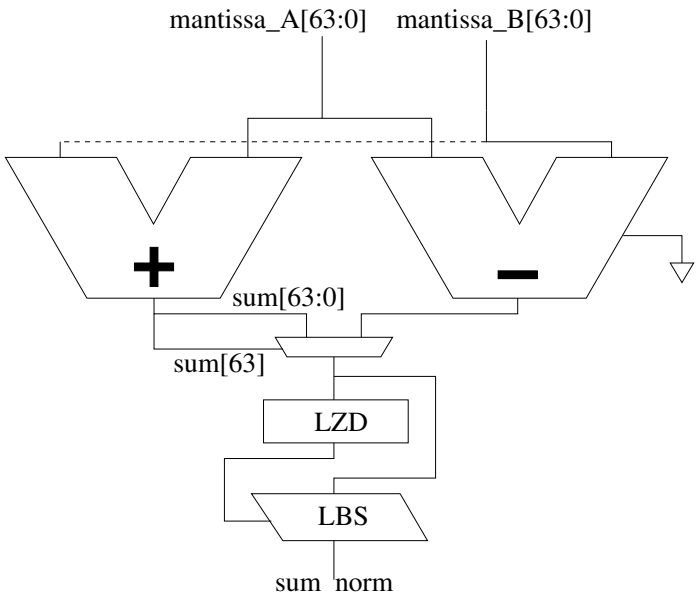

Figure 4.7: Primary Addition Architecture from [3]

the architecture used in this thesis.

There are also significant differences in the structure for which the primary additional operation occurs. Beforehand, parallel adders were used in order to produce the two's complement value for each addition or subtraction. Then, based on the sign of the addition, the correct value would be chosen. Following that, it is necessary to find the number of leading zeros in the mantissa and perform a post-normalization shift after that. This process has been heavily parallelized, in that now the post-normalization shift value is computed in parallel with the primary addition by use of an ELZA, and the primary addition itself is made much more power efficient by the use of an EAC adder architecture. The critical path now only follows the EAC carry chain and the post-normalization shift, as opposed to a normal prefix adder, a LZD, and then a post-normalization shift. Figure 4.7 shows the architecture used for the primary addition in previous work, while Figure 4.4 shows the architecture used in this thesis.

A novel iteration has been made in terms of the exponent rounding structure, which has led to significant performance benefits over the previous architecture used.

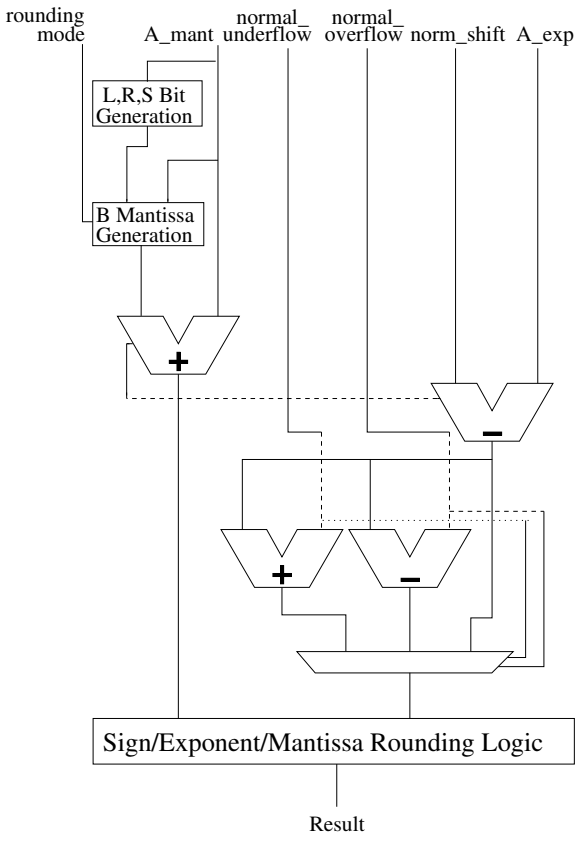

Figure 4.8: Rounding Architecture from [3]

Initially, two sequential adders were used to first compute the rounded exponent, and then either round up or down depending on the presence of `norm_unflow` or `norm_ovflow`. In the current architecture, these are used in combination with the carry out from the adder used to round the mantissa to produce a single constant to use as input to the flagged adder. This allows exponent rounding to be performed in a single addition operation, as opposed to two. The exponent rounding critical path follows through the least, round, and sticky bit generation, which is used to produce a constant B to add to the mantissa. The carry out is combined with `norm_ovflow` and `norm_unflow` to produce a constant in the flagged prefix adder, the sum of which is finally rounded. Figure 4.8 details the previously used architecture and Figure 4.5 shows the architecture used in this thesis.

CHAPTER V

RESULTS

5.1 Results and Conclusion

The proposed design is implemented in RTL-compliant Verilog and designs are then synthesized using an ARM 32nm CMOS library for Global Foundries (GF) cmos32soi technology optimizing on delay. To verify the correctness, all implementations are tested against random test vectors generated by TestFloat [26] and passed completely. Additional random denormalized vectors via a Java program were also generated to give completeness and coverage. The ARM standard-cell library utilizes multiple values of V_T to aid in synthesis (i.e., MTCMOS). Synthesis was optimized for delay utilizing Synopsys® (SNPS) Design Compiler™ (DC) in topographical mode using a PVT process at 25° C using TT corners. Topographical synthesis, provided by Synopsys® DC™ (DC) ensures synthesis that accurately predicts timing, area and power by including information from the standard-cell layouts and underlying interconnect.

Table 5.1 shows the post-synthesis results for cmos32soi GF 32nm technology using the Synopsys® DC™ and Synopsys® Power Compiler™ synthesis software. The average fanout-of-4 (FO4) delay measured with SPICE is 5.95ps for 32nm technology. Table 5.2 shows additional synthesis results for the proposed architecture in cmos32soi GF 32nm used in combination with architecture from the previous implementation [3]. In these additional results, the architecture for the operations of exponent comparison, the primary addition, and rounding have been replaced by its

Table 5.1: Post-synthesis Results for the Proposed IEEE 754 compliant Architecture in cmos32soi 32nm GF technology

IEEE 754 Adder	# Cells	Area [um^2]	Delay [ps/FO4]	Power [mW]			
				Internal	Switching	Leakage	Total
Proposed IEEE 754 Denormized/ Normalized FP Adder (LVT)	5,908	7,533.7	469.12/78.84	9.060	11.482	4.125	24.667
IEEE 754 Denormalized/ Normalized FP Adder (LVT)	8,475	9,585.9	619.19/104.07	10.536	13.698	5.147	29.382
SNPS DW (LVT)	5,269	6,840.1	605.03/101.69	11.035	14.527	3.711	29.274
Proposed IEEE 754 Denormized/ Normalized FP Adder (RVT/LVT)	6,238	7,681.3	481.26/80.88	8.829	11.418	3.835	24.082
IEEE 754 Denormalized/ Normalized FP Adder (RVT/LVT)	7,168	8,874.1	623.46/104.78	13.420	16.638	4.467	34.525
SNPS DW (RVT/LVT)	5,151	6,932.4	595.80/100.13	10.652	13.890	3.411	27.953

Table 5.2: Post-synthesis Results with/without Enhancements for the Proposed IEEE 754 compliant Architecture in cmos32soi 32nm GF technology

IEEE 754 Adder	# Cells	Area [um^2]	Delay [ps]	Power [mW]			
				Internal	Switching	Leakage	Total
Proposed Design (LVT)	5,908	7,533.7	469.12	9.060	11.482	4.125	24.667
Proposed Design w/o Exponent (LVT)	7,937	10,218.7	574.49	10.910	13.970	5.542	30.421
Proposed Design w/o EAC (LVT)	6,401	6,746.7	532.96	5.455	7.387	3.546	16.388
Proposed Design w/o Rounding (LVT)	7,895	9,956.5	576.41	13.660	17.705	5.610	36.975
Proposed Design (RVT/LVT)	6,238	7,681.3	481.26	8.829	11.418	3.835	24.082
Proposed Design w/o Exponent (RVT/LVT)	8,207	10,210.5	585.38	10.323	13.674	4.999	28.996
Proposed Design w/o EAC (RVT/LVT)	6,118	6.655.0	544.52	5.632	7.324	2.930	15.886
Proposed Design w/o Rounding (RVT/LVT)	7,051	8.857.8	583.10	11.744	15.140	4.471	31.356

corresponding implementation in [3]. This shows a direct comparison on the performance improvement between designs. It is also informative to show the performance differences between using exclusively low-threshold MTCMOS cells and cells with a combination of voltage thresholds, as would normally be seen in practice. These are referred to as *LVT* for low-voltage threshold and *RVT/LVT* for regular-voltage and low-voltage threshold, respectively. Since *LVT* generally has better performance for delay , all subsequent discussion comparing designs shall refer to the *LVT* results of either table.

The design is compared against intellectual property generated by Synopsys' DesignWare™ (DW) floating-point adder /subtracter, `DW_fp_addsub` design. The DW design is also IEEE 754 compliant, however, it only computes results using IEEE 754 double-precision arithmetic. To make the comparison between designs as direct as possible, the operational subset during synthesis has been limited to be the same as DW, i.e., the architecture is limited to exclusively performing double-precision arithmetic. Taking this into account, all of the results, including those from previous work, can be directly compared to DW. For delay, this architecture with exclusively double-precision arithmetic support is 28.97% faster than DW, with a critical delay time of 469.12 ps. For comparisons between other technologies, a unitless delay can be calculated based on the FO4 delay. This can be done by dividing the FO4 delay result from the technology used [4].

The resulting power consumption between the proposed architecture and DW is, due to the use of EAC adders [13] and fanout optimization, smaller. The proposed architecture reduces the power to 84.26% of the DW reference design. This can be observed in the large difference in internal power consumption between architectures. This design does have a larger leakage power consumption than the DW reference, due to the increased number of cells.

The increased parallelization the proposed design has over DW does require more

hardware to implement, although at a comparable level. Looking at the DW reference the proposed architecture uses 110.14% of area consumed by DW.

Comparing this architecture to the previous work it is based on also shows significant improvement. In terms of delay performance, the changes that have been made to each operation's architecture have yielded improvements. Changing the exponent structure decreased the delay by 22.46%, swapping the adder structure and adding a LZA improved it by 13.61%, and putting a flagged-prefix adder in the rounding structure also improved performance by 22.87%. Power performance is also improved when the exponent comparison and rounding structures are replaced, by 23.33% for exponent comparison and 49.90% for rounding. Due to the lack of an exact LZA tree, the power consumption is worse for the architecture in comparison to previous work, by 66.23%. The area used in between replacing components of the previous work also shows improvement, excluding when the primary adder structure is replaced. This is mainly due to the necessary hardware overlap required when splicing together architecture components, and again, the lack of an exact LZA tree reduces area when addition structure are swapped. Area improvements of 35.64% are found for exponent comparison and 32.16% for rounding, while a reduction in area by 11.66% is shown when comparing addition structures.

The key to this implementation is parallelizing all of the adder and subtracter structures possible in the design and optimizing the load each floating-point operation has to drive, which allows synthesis to better optimize the critical path through this architecture. Similar designs use the same idea for three-operand addition [9]. Although earlier articles suggest two parallel computation paths [12], this thesis uses it to compute other important conversion utilities that may be useful for common general-purpose and application-specific architectures. This floating-point adder can still potentially be improved on its delay timing, although not significantly. This could be done by implementing inexact LZA's in place of the exact LZA's used for

the post-normalization of the EAC adder's sum. As mentioned previously, this design can be easily pipelined for additional performance. This design can be made into a three stage pipeline by placing registers after the exponent subtraction stage as well as before the post-normalization process in the datapath.

The design of this IEEE 754 compliant floating-point adder shows extremely high levels of performance while maintaining a substantial level of utility. This architecture is useful for any floating-point designs that requires high precision and unparalleled delay and energy performance. Another strong emphasis is that the results can be further improved by implementing in a complete custom-cell VLSI design. However, the strong results show that the optimization in a standard-cell design significantly outperforms IP-based designs as well as previous results.

REFERENCES

[1] "IEEE standard for floating-point arithmetic," *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pp. 1–84, 2019.

[2] "IEEE standard for binary floating-point arithmetic," *ANSI/IEEE Std 754-1985*, pp. 1–14, 1985.

[3] B. Mathis and J. E. Stine, "A well-equipped implementation: Normal/denormalized half/single/double precision IEEE 754 floating-point adder/subtracter," in *2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, vol. 2160-052X, pp. 227–234, July 2019.

[4] N. Weste and D. Harris, *CMOS VLSI Design: A Circuits and Systems Perspective*. USA: Addison-Wesley Publishing Company, 4th ed., 2010.

[5] W. Kahan, "Lecture Notes on the Status of IEEE Standard 754 for Binary Floating-Point Arithmetic," tech. rep., University of California, Berkeley, 1996. Available at http://www.cs.berkeley.edu/~wkahan.

[6] K. J. Kuhn, "CMOS scaling beyond 32nm: Challenges and opportunities," in *2009 46th ACM/IEEE Design Automation Conference*, pp. 310–313, 2009.

[7] M. Horowitz, D. Stark, and E. Alon, "Digital circuit design trends," *IEEE Journal of Solid-State Circuits*, vol. 43, no. 4, pp. 757–761, 2008.

[8] M. B. Taylor, "A landscape of the new dark silicon design regime," *IEEE Micro*, vol. 33, no. 5, pp. 8–19, 2013.

[9] J. Sohn and E. E. Swartzlander, Jr., "A fused floating-point three-term adder," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 61, pp. 2842–2850, Oct 2014.

[10] E. M. Schwarz, M. Schmookler, and S. D. Trong, "FPU implementations with denormalized numbers," *IEEE Transactions on Computers*, vol. 54, pp. 825–836, July 2005.

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

[12] P. . Seidel and G. Even, "Delay-optimized implementation of IEEE floating-point addition," *IEEE Transactions on Computers*, vol. 53, pp. 97–113, Feb 2004.

[13] B. Mathis and J. E. Stine, "A novel single/double precision normalized IEEE 754 floating-point adder/subtracter," in *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pp. 278–283, July 2019.

[14] E. M. Schwarz, *Binary Floating-Point Unit Design*, pp. 189–208. Boston, MA: Springer US, 2006.

[15] M. D. Ercegovac and T. Lang, *Digital Arithmetic*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1st ed., 2003.

[16] J. T. Coonen, "Underflow and the denormalized numbers," *Computer*, vol. 14, pp. 75–87, March 1981.

[17] 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, no. 8, pp. 786–793, 1973.

[18] N. Burgess, "The flagged prefix adder and its applications in integer arithmetic," *Journal of VLSI signal processing systems for signal, image and video technology*, vol. 31, pp. 263–271, Jul 2002.

[19] J. E. Stine, C. R. Babb, and V. B. Dave, "Constant addition utilizing flagged prefix structures," in *2005 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 668–671 Vol. 1, May 2005.

[20] E. E. Swartzlander, Jr., "Merged arithmetic," *IEEE Transactions on Computers*, vol. C-29, no. 10, pp. 946–950, 1980.

[21] V. G. Oklobdzija, "An algorithmic and novel design of a leading zero detector circuit: comparison with logic synthesis," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 2, no. 1, pp. 124–128, 1994.

[22] J. D. Bruguera and T. Lang, "Leading-one prediction with concurrent position correction," *IEEE Transactions on Computers*, vol. 48, no. 10, pp. 1083–1097, 1999.

[23] 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, no. 12, pp. 1726–1727, 2009.

[24] T. Lang and J. D. Bruguera, "Floating-point multiply-add-fused with reduced latency," *IEEE Transactions on Computers*, vol. 53, no. 8, pp. 988–1003, 2004.

[25] I. Koren, *Computer Arithmetic Algorithms*. Natick, MA, USA: A. K. Peters, Ltd., 2nd ed., 2001.

[26] J. Hauser, "The SoftFloat and TestFloat Validation Suite for Binary Floating-Point Arithmetic," tech. rep., University of California, Berkeley, 2018. Available at http://www.jhauser.us/arithmetic/TestFloat.html.

[27] N. Weste and D. Harris, *CMOS VLSI Design: A Circuits and Systems Perspective*. USA: Addison-Wesley Publishing Company, 4th ed., 2010.

[28] "Ieee standard for floating-point arithmetic," *IEEE Std 754-2008*, pp. 1–70, 2008.

[29] J. E. Stine and M. J. Schulte, "A combined two's complement and floating-point comparator," in *2005 IEEE International Symposium on Circuits and Systems*, pp. 89–92 Vol. 1, May 2005.

[30] J.-M. Muller, *Elementary Functions: Algorithms and Implementation*. Secaucus, NJ, USA: Birkhauser Boston, Inc., 1997.

VITA

Brett Mathis

Candidate for the Degree of

Master of Science

Thesis: IMPLEMENTATIONS OF HIGH PERFORMANCE ARCHITECTURE FOR IEEE 754 COMPLIANT FLOATING-POINT ADDERS

Major Field: Electrical Engineering

Biographical:

Education:

Completed the requirements for the Master of Science in Electrical Engineering at Oklahoma State University, Stillwater, Oklahoma in December, 2020.

Completed the requirements for the Bachelor of Science in Computer Engineering at Oklahoma State University, Stillwater, Oklahoma in 2018.

Experience:

Graduate Research Assistant - VLSI Computer Architecture Research Group OSU
June 2018 - December 2020