ARCHITECTURAL IMPROVEMENTS IN IEEE-COMPLIANT FLOATING-POINT MULTIPLICATION

By
TUAN DANH NGUYEN

Bachelor of Science in Computer Science
Hanoi University of Science and Technology
Hanoi, Vietnam
2008

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
DOCTOR OF PHILOSOPHY
December, 2018

ARCHITECTURAL IMPROVEMENTS IN IEEE-COMPLIANT FLOATING-POINT MULTIPLICATION

Dissertation Approved:

James E. Stine

Dissertation Advisor

Dr. Keith A. Teague

Dr. Carl D. Latino

Dr. Camille DeYong

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor, Dr. James E. Stine, for his continuous and invaluable support of my Ph.D. research. I can not emphasize enough my appreciation for his patience, encouragement, and immense knowledge.

I would like to thank Dr. Keith A. Teague, Dr. Carl D. Latino, and Dr. Camille DeYong for serving my committee and for their insightful comments/questions and encouragement.

I would like to thank my labmate and my friend, Dr. Son V. Bui, for his help not only in my research but also in my life in the U.S.

Last but not least, I would like to thank my family: my parents, my wife, and my daughter, for their love and unconditional support. They always believe in me and encourage me to follow my dreams.

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

Name: TUAN DANH NGUYEN

Date of Degree: DECEMBER, 2018

Title of Study: **ARCHITECTURAL IMPROVEMENTS IN IEEE-COMPLIANT FLOATING-POINT MULTIPLICATION**

Major Field: ELECTRICAL AND COMPUTER ENGINEERING

Abstract: Multiplication has long been an important part of any computer architecture. It has usually been a *common case* for most computer architecture decisions to include in any microarchitecture. However, the difficulty in creating hardware for multiplication because of its inherent shifting of the radix point has been a cogent reason for the need for floating-point hardware in scientific applications. The IEEE 754 floating-point standard was originally ratified in 1985 [1] and later amended in 2008 [2] to make floating-point multiplication easier for users to implement applications. Although floating-point arithmetic creates a mechanism to make things easier for using multiplication, it is complicated both algorithmically and practically for hardware implementations.

This dissertation discusses possible architectural improvements in IEEE-compliant floating-point multiplication for Machine Learning/Deep Learning applications. First, a combined IEEE half and single precision floating-point multipliers is proposed to reduce power dissipation for Deep Learning applications. Second, a novel rounding scheme is proposed that is simpler but comparable with the state-of-the-art rounding schemes. Third, an optimized design is proposed that can handle both denormal and normal numbers. Finally, a hybrid precision design is proposed, aiming to improve the power consumption of Machine Learning/Deep Learning applications. Proposed designs are targeted to Machine Learning/Deep Learning applications-specific processors to improve the latency and power consumption. All designs are implemented in RTL-level Verilog, verified for correctness against open-source TestFloat generated test vectors, and synthesized using an ARM 32nm CMOS library for Global Foundries (GF) cmos32soi technology for estimated power, area and delay analysis.

TABLE OF CONTENTS

Chapter Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

Figure Page

LIST OF NOTATIONS AND ABBREVIATIONS

The following is a list of symbols, variables, and scripts used in this work which may not be defined as they are encountered in the body.

$\wedge$	logic AND
$\vee$	logic OR
$\oplus$	logic XOR
$+$	addition
$\cdot$	multiplication
X[n-1:0]	n-bits binary number X
MSB	Most Significant Bit
LSB	Least Significant Bit
ulp	unit in the last place
FP	Floating-Point
ML	Machine Learning
DL	Deep Learning
CPU	Central Processing Unit
GPU	Graphics Processing Unit

CHAPTER 1

INTRODUCTION

In recent years, deep learning has grown tremendously in its popularity and usefulness [8]. However, deep learning is computationally intensive, power-hungry and often limited by its hardware capability. Basically, deep learning is a deep neural network, which consists in convolutions and matrix multiplications. In [9], authors show that 90% computation of convolutional neural network (CNN), a typical deep neural network, is due to convolution operations. Therefore, multipliers are the most space- and power-hungry arithmetic operators of the digital implementation of CNN. This is also true for other types of machine learning that heavily based on a matrix multiplication.

In addition, multiplication has long been an important part of any computer architecture. It has usually been a *common case* for most computer architecture decisions to include in any microarchitecture. However, the difficulty in creating hardware for multiplication because of its inherent shifting of the radix point has been a cogent reason for the need for floating-point hardware in scientific applications. The IEEE 754 floating-point standard was originally ratified in 1985 [1] and later amended in 2008 [2] to make floating-point multiplication easier for users to implement applications. Although floating-point arithmetic creates a mechanism to make things easier for using multiplication, it is complicated both algorithmically and practically for hardware implementations.

To account for these changes, specifically for rounding, key research was introduced to help alleviate problems related to round-to-nearest-even [4]. Good hardware for rounding in IEEE-compliant floating-point arithmetic is key to expanding algorithms, numerical methods, and applications that exploit techniques to control validation in loss of precision (e.g., through interval arithmetic [10]). Although [4] is paramount in determining good rounding for IEEE 754 arithmetic, it has the deficiency of not supporting all four IEEE rounding modes. Additional work clarifies designs for improving rounding as well as giving optimal hardware [5, 11]. Although these papers give good designs, there are additional questions about verification and possible improvements. Consequently, this dissertation shows modifications that can be done to help optimize this process and efficiently correct for all four rounding modes. This dissertation presents an improved implementation described using the IEEE 754 double-precision floating-point format [1, 2] but it can be easily adapted to all other IEEE-compliant formats with a small modification. Moreover, this dissertation verifies all designs and details whether the designs are specifically IEEE 754 compliant.

In addition, correct rounding of both normal and denormal results further exacerbates the growing complexity of an IEEE 754 multiplier. Due to the importance of high precision in scientific applications [12], the precision must be preserved. Simply truncating denormal results to zero is unacceptable [13], especially with half- and single-precision floating-point numbers. Consequently, having floating-point units that can handle normalized and denormalized numbers is essential, especially for scientific computing [14]. Recently there have been several types of hardware implementations that handle floating-point denormalized IEEE-754 numbers [15, 16, 17]. This dissertation discusses methods of implementing both normalized and denormalized IEEE 754 numbers [2]. In particular, it combines the pre-normalizing and post-normalizing steps in existing methods into a simpler and faster single step.

While most general-purpose CPU/GPU utilize double-precision floating point units, single-precision floating-point is widely used in deep learning as the default format because its advantage in a representable range that makes it suitable for a wide range of applications [2]. Moreover, recent research [18] shows that, in many applications, single-precision floating-point multipliers can be replaced by half-precision floating-point multipliers in training deep neural networks, which have little to no impact on the network accuracy. Smaller multipliers also lead to a lower overall energy footprint even in systems that have multiple IEEE 754 formats implemented in hardware. Therefore, there is a need for a new multipliers that can switch between precision numbers in implementing deep learning.

This dissertation ultimately provides an IEEE 754 compliant floating-point multiplier that can handle half-, single-, and double-precision operations. Previous implementations [19, 20] only demonstrated methods for single- and double-precision operations, however, this design extends the ideas by specifically adapting the architecture for half-precision IEEE 754 multiplication. Half-precision floating-point multipliers are new additions to the 754 standard that are specifically useful for architectures that use machine learning computations [2]. Most importantly, by utilizing smaller amounts of precision, these multipliers can speed up computations for designs that are well suited for neural networks and machine learning applications [21]. Moreover, extensions are added to the multiplier to also handle denormalized IEEE 754 floating-point numbers as well as half- and single-precision floating-point numbers.

The rest of this dissertation is organized as follows: Chapter 2 is the background about IEEE floating-point formats and multiplication. Chapter 3 is a combined IEEE half and single-precision multipliers for deep learning applications. Chapter 4 briefly summarizes and clarify the key contribution in Santoro, Bewick and Horowitz [4] (SBH) method, Quach, Takagi and Flynn [5] (QTF) method, and Even and Seidel [6] (ES) method and then shows our proposed method that can support all rounding modes. Chapter 5 is a novel design that support both normalized and denormalized floating point numbers. Chapter 6 is a hybrid precision design that can switch precision mode easily. Chapter 7 presents how all designs are implemented, verified, and simulated for delay, area, and power. Finally, Chapter 8 is the conclusions and future work.

CHAPTER 2

BACKGROUND

2.1 Floating-Point Formats

The IEEE 754 floating-point standard, originally ratified in 1985 [1] and later ammended in 2008 [2], defines the floating-point format that consists of three parts: sign (S), exponent (E), fraction or mantissa (F) as shown in Figure 2.1. Table 2.1 shows the operand structure for the current IEEE 754-2008 formats including half-precision, single-precision, double-precision and quadruple-precision formats with the size and bias values for each format.

With the IEEE 754 format, the exponent is adjusted so that the sum of the real exponent and a constant (bias) are in a non-negative range. This is designed to avoid using two's complement encoding that can be difficult to handle for multiple fields as well as simplifying implementations for comparison. The bias is given as:

$$bias = 2^{exponent\_size-1} - 1 \tag{2.1}$$

where the value of the exponent is set according to the representation given in the IEEE 754 standard (e.g., double-precision). This value represents the midway case between the minimum and the maximum exponent values.

Normalized (or normal) IEEE floating-point numbers are assigned a mantissa between the range of $[1, 2)$ to create a packed representation of numbers in scientific notation. The encoding assumes that the mantissa has a leading one so that its representation exists between the range of $[1, 2)$. In order to save space, the leading

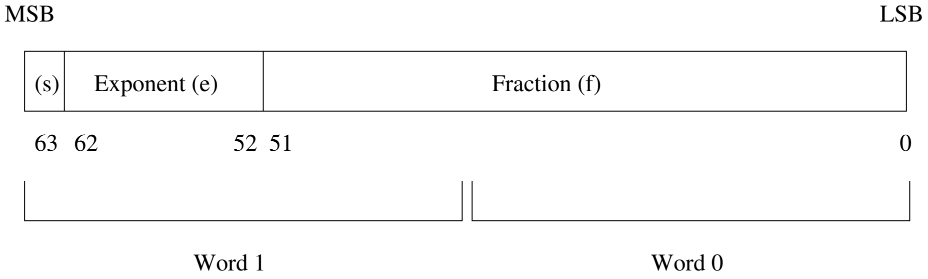


Figure 2.1: Data formats for the floating point

Format	Size	Sign (S)	Exponent (E)	Bias Value (B)	Fraction (F)
Half-Precision (binary16)	16	1	5	15	10
Single-Precision (binary32)	32	1	8	127	23
Double-Precision (binary64)	64	1	11	1023	52
Quad-Precision (binary128)	128	1	15	16383	112

Table 2.1: IEEE 754-2008 floating-point formats

1 is not stored, but hidden. This can be represented by the following equation for a given normalized IEEE 754 floating-point number X:

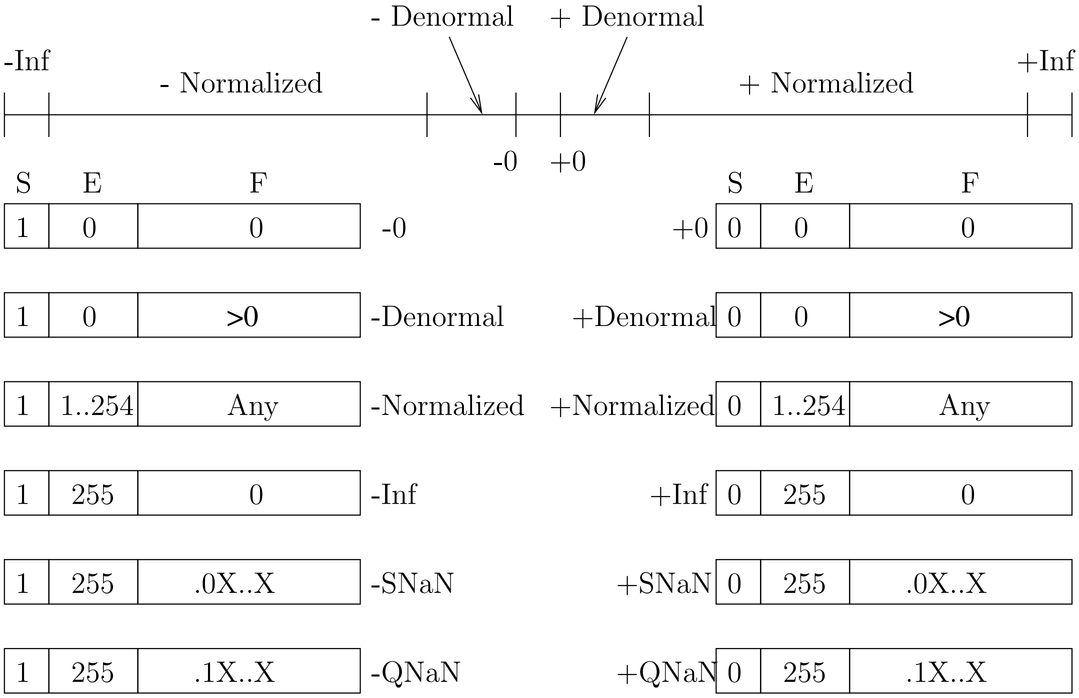
$$X = (-1)^S \cdot 1.F \cdot 2^{E-bias} \tag{2.2}$$

On the other hand, for denormalized (or denormal) numbers, the value of X is given by:

$$X = (-1)^S \cdot 0.F \cdot 2^{1-bias} \tag{2.3}$$

where the leading hidden bit is now 0. Some texts refer to this as subnormal numbers. A zero (0) exponent and non-zero mantissa indicates a denormalized number (i.e., $X_E = 0$ and $X_F \neq 0$). Denormal numbers fill the gap between smallest normal numbers and zero by allowing numbers with reduced precision to exist. Ultimately, denormalized numbers are designed to limit gradual underflow [22, 23].

In addition to normalized and denormalized numbers, IEEE 754 also defines special values including zeros, infinity (Inf), quiet Not-a-Number (qNaN), and signaling Not-A-Number (sNaN). A NaN is utilized to represent numbers not possible for typ-



S	E	F			S	E	F
1	0	0	-0	+0	0	0	0
1	0	>0	-Denormal	+Denormal	0	0	>0
1	1..254	Any	-Normalized	+Normalized	0	1..254	Any
1	255	0	-Inf	+Inf	0	255	0
1	255	.0X..X	-SNaN	+SNaN	0	255	.0X..X
1	255	.1X..X	-QNaN	+QNaN	0	255	.1X..X

Figure 2.2: Normalized, denormalized, and special numbers (binary16)

ical floating-point representations. Figure 2.2 shows the definition for those special values in half-precision (binary16) format.

2.2 Floating-Point Multiplication

Floating-point multiplication complicate accuracy and precision because of the movement of the radix point during a multiplication [12]. That is, the final result must be formatted in the IEEE 754 format before it is written back to a datapath/memory unit as a result [1]. For this reason, floating-point computations can possibly produce unreliable computations due to rounding and exceptions within the IEEE 754 standard [24].

Given two floating point numbers X and Y represented by (S_X, E_X, M_X) and (S_Y, E_Y, M_Y), respectively, the product Z represented by (S_Z, E_Z, M_Z) can be mathematically computed as follows:

$$\begin{aligned} Z &= X \cdot Y \\ &= (-1)^{S_X} \cdot 2^{E_X - B} \cdot M_X \cdot (-1)^{S_Y} \cdot 2^{E_Y - B} \cdot M_Y \\ &= (-1)^{S_X + S_Y} \cdot 2^{(E_X + E_Y - B) - B} \cdot (M_X \cdot M_Y) \end{aligned} \tag{2.4}$$

Since M_X, M_Y are 53 bits number in the $[1, 2)$ domain,the product $M_X \cdot M_Y$ results in a 106 bit number in the $[1, 4)$ range. Therefore, an additional *round_normalize* step is needed to round and then normalize this product to a 53 bit number in the $[1, 2)$ interval. Interestingly, if the product overflows (i.e., great than 2), the exponent should also be updated (increased by 1) to compensate for the rounding and normalization as shown in Equation 2.5.

$$\begin{aligned} S_Z &= S_X + S_Y = S_X \oplus S_Y \\ M_Z &= round\_normalize(M_X \cdot M_Y) \\ E_Z &= \begin{cases} E_X + E_Y - B & \text{if } M_X \cdot M_Y \in [1, 2) \\ E_X + E_Y - B + 1 & \text{if } M_X \cdot M_Y \in [2, 4) \end{cases} \end{aligned} \tag{2.5}$$

Figure 2.3 shows a block diagram detailing the overall generalized IEEE 754 multiplication architecture. The design consists of several stages: unpack (hidden bit and other exception and bit testing), sign, exponent and mantissa logic blocks and final result packing. As per the IEEE 754 standard, five flags are produced Infinite or Divide by 0 (I), Inexact (X), Invalid (V), Overflow (O) and Underflow (U). Some flags, such as Divide by 0, are not appropriate for floating-point multiplication as it is not possible. Regardless if a flag makes sense, correct IEEE 754 functional units produce all five flags.

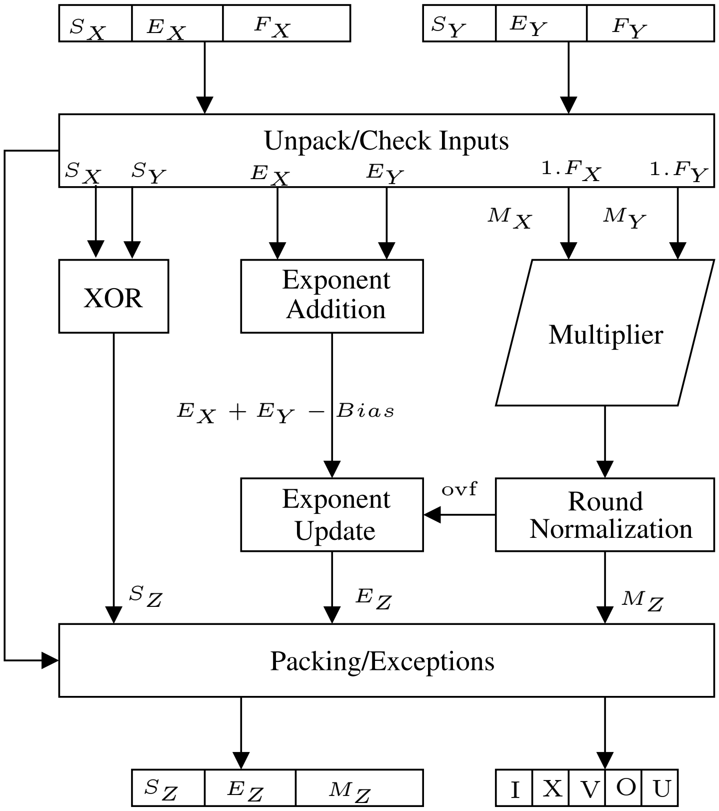


Figure 2.3: Block diagram of multiplier

The first step in floating-point multiplication unpacks the IEEE numbers by finding the hidden bit as well as checking for any special cases (e.g., NaNs). Once this step is completed, the normalized operands are separated into four sub-fields: special values, sign, exponent, and mantissas. To compute the product sign, an XOR gate combines the operand sign bits. Simultaneously, the exponents are added and the bias subtracted to produce an intermediate exponent. Finally the intermediate exponent must be adjusted if the intermediate product in mantissa multiplication overflows into position [53] (since multiplication results in values greater than 2).

The mantissas are multiplied using a binary multiplier (as described in Section 2.3) that includes three steps: partial product generation to generate the partial product bits, partial production reduction to reduce the partial product bits to sum and carry vectors, and a carry propagate addition to add the sum and vectors to produce the product. The output of this stage is then fed into the rounding and normalization step.

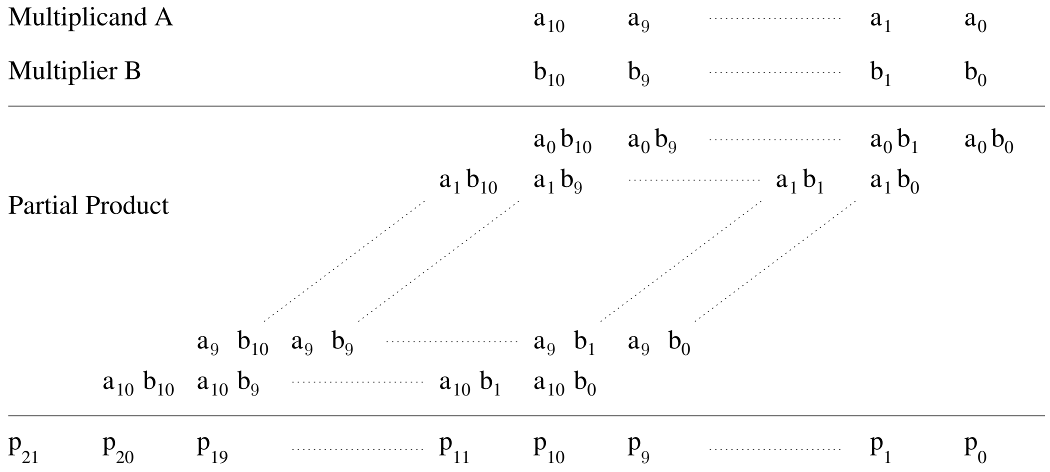


Figure 2.4: Partial products matrix of multiplication

Finally, the result is packed into the IEEE 754 format and exceptions are generated, if necessary. The important and critical stage of rounding is required to correctly round the final result according to its predefined rounding mode. Most general-purpose processing units store the rounding mode in a separate states register [25].

2.3 Mantissa Multiplication

A key step in the FP floating-point multiplication is the mantissa multiplication. Basically, given a m-bit multiplicand A and a n-bit multiplier B, the multiplication produces a $(m+n)$-bit product (i.e.,$P = A \cdot B$. Figure 2.4 illustrates the multiplication of 11-bit half-precision mantissa using a partial product matrix diagram. In many cases when A, B, and P are large, the dot diagram, in which a dot represents a partial product bit, is often used for the sake of simplification [3].

Generally, multiplication includes three separate stages as follows:

1. Partial Product Generation (PPG) - utilizes a collection of gates (e.g. AND gates) to generate the partial product bits $a_i \cdot b_j$.

2. Partial Product Reduction (PPR)- utilizes adders to reduce the partial products to sum and carry vectors.

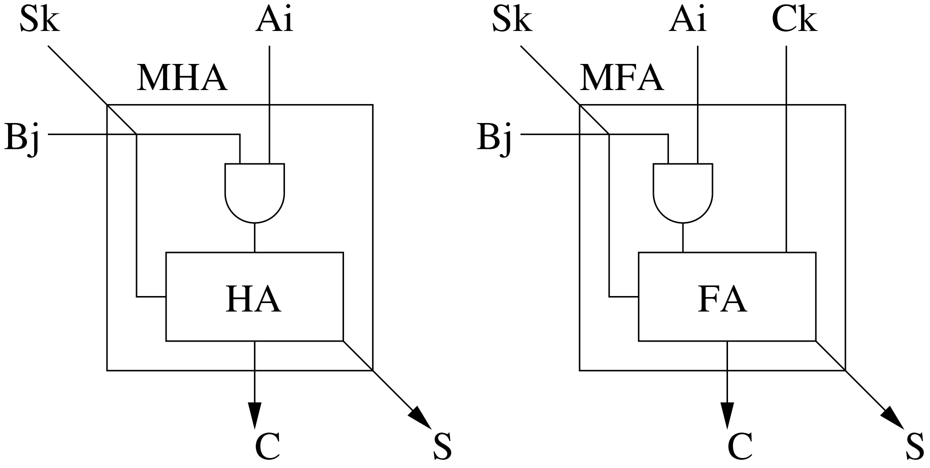


Figure 2.5: Modified half adder and full adder (Adopted from [3])

3. Final Carry Propagate Addition (CPA) - adds the sum and carry vectors to produce the final product.

High speed multipliers are typically classified into parallel multipliers and sequential multipliers. Since the mantissa multiplication is not the focus of this dissertation, only key parallel multipliers will be discussed briefly. The parallel multipliers can be classified further into array multipliers and tree multipliers.

2.3.1 Array Multipliers

Carry-Save Array Multipliers (CSAM) are a simple multiplier in which partial product bits are added in the array topology, similar to the paper-and-pencil multiplication process. The CSAM first generates partial product bits by utilizing AND gates and then uses an array of Carry-Save Adders (CSAs) to perform partial product reduction [3].

To perform Carry-Save Array Multiplication, adders are modified so that they can perform both partial product generation and addition. Two kind of modified adders are being used and called the modified half adder (MHA) and the modified full adder (MFA) as in Figure 2.5. The MHA consists of an AND gate to generate the partial product bit, followed by one half adder (HA) to add this partial product bit

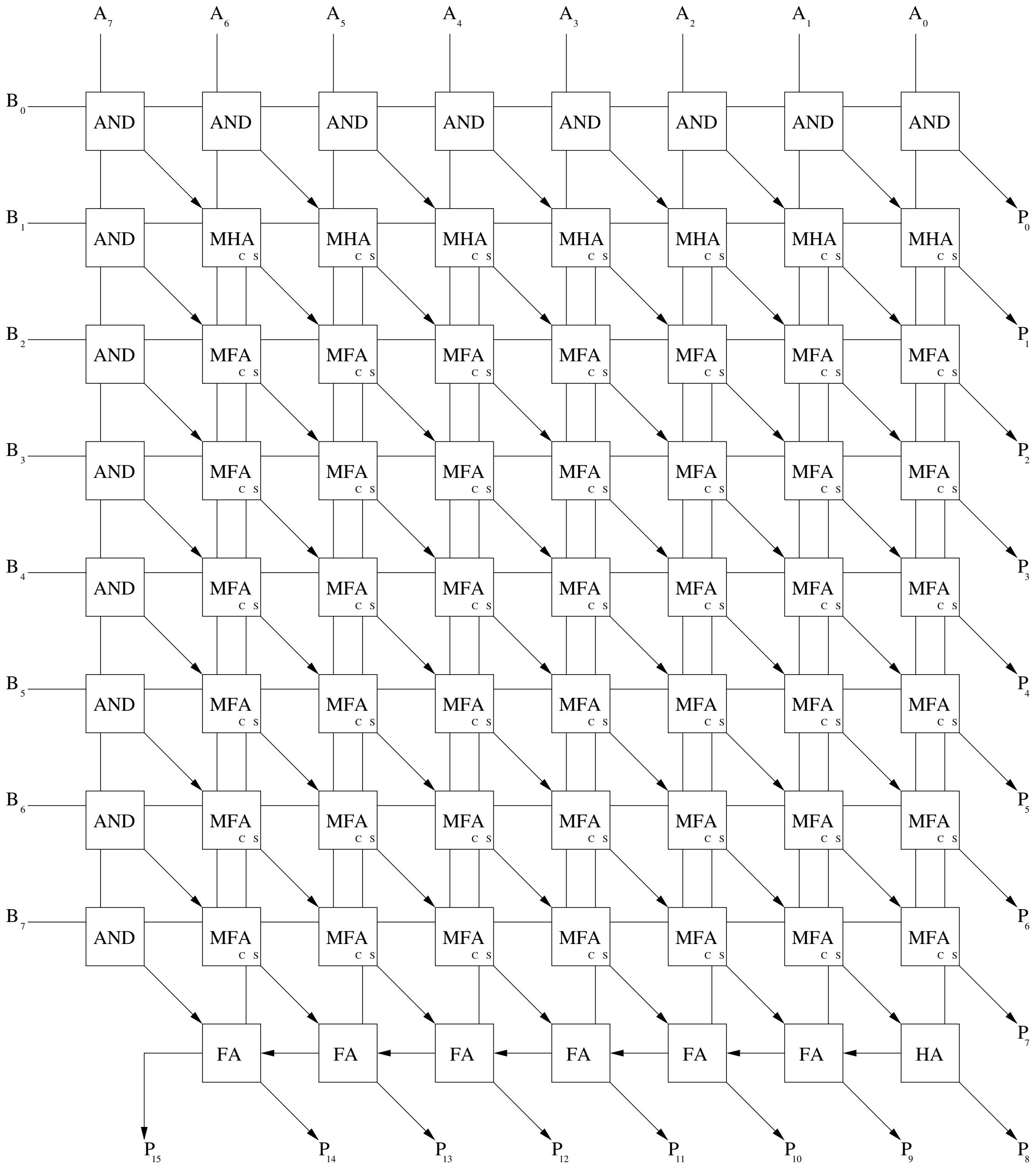


Figure 2.6: Example of 8 × 8-bit carry-save array multiplier [3]

and another one from the previous row. Similarly, the MFA consists of an AND gate for generating partial product bit, followed by a full adder (FA) to add this partial product bit with the sum and carry from previous row. Given these modified adders, the CSAM can be implemented as in Figure 2.6.

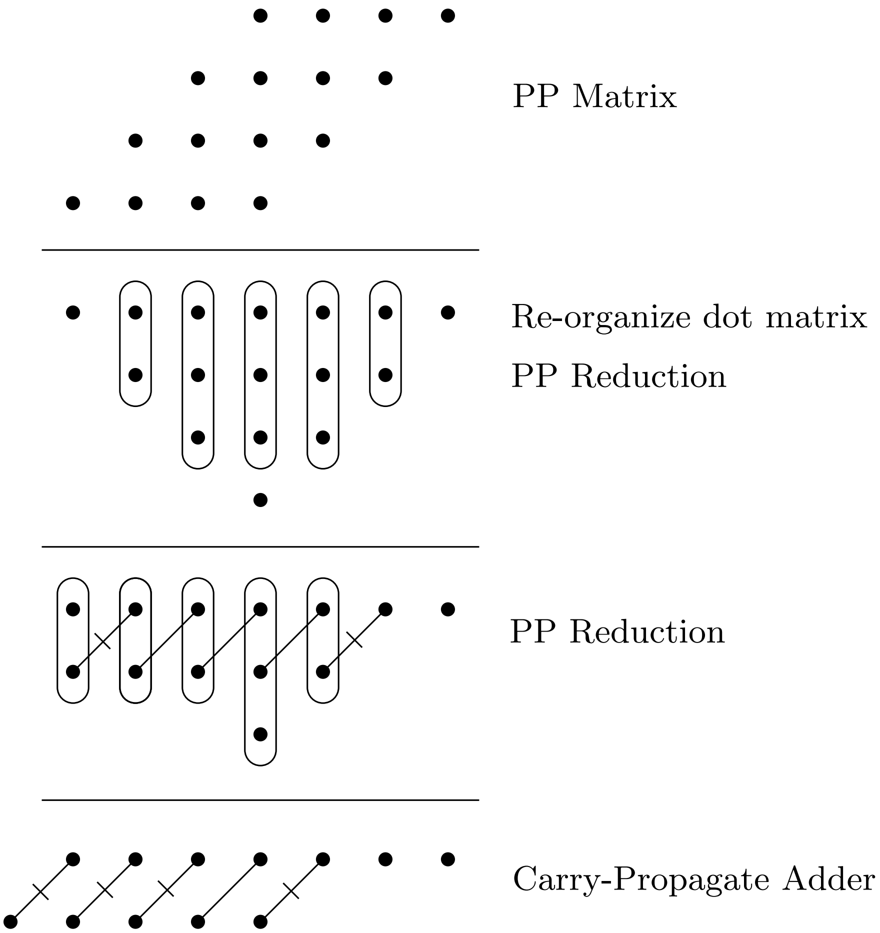


Figure 2.7: Example of 4 × 4-bit Wallace tree multiplier (Adopted from [3])

2.3.2 Tree Multipliers

Tree multipliers use the tree topology for the partial product reduction stage. In particular, tree multipliers reduce the partial products down until their height is equal to 2 then apply a high-speed CPA to get the final result.

The first type of tree multiplier is called a Wallace tree multiplier that was introduced in 1964 [26]. Dadda multipliers [27] were later introduced one year after Wallace method to enhance Wallace method by reducing the number of reduction stages. Another tree multiplier to optimize the placement of adders called a column-compression multiplier has a smaller area and delay compared with both Wallace and Dadda methods [28]. Since the mantissa multiplication is not the focus of this dissertation, only Wallace multiplier will be discussed to understand the process in tree multipliers as follows.

Basically, the Wallace method groups rows of partial product bits into sets of three [26]. Within each three row set, full adders reduce columns with three bits and half adders reduce columns with two bits. When used in multiplier trees, full adders and half adders are often referred to as $(3, 2)$ and $(2, 2)$ counters, respectively [3]. Rows that are not part of a three row set are transferred to the next reduction stage for subsequent reduction.

To visualize Wallace trees, a dot diagram of the multiplication matrix can be be used as in Figure 2.7 [3]. A dot represents a partial product bit. In addition, an uncrossed diagonal line represents the outputs of a FA (or $(3, 2)$ counter) and a crossed diagonal line represents the outputs of a HA (or $(2, 2)$ counter). An oval is also utilized to show the transition from reduction stages. As shown in Figure 2.7, this multiplier takes 2 reduction stages with matrix heights of 3 and 2. In the final stage, a fast CPA is utilized by adding these two arrays together to get final product. The total delay is proportional to the logarithm of the operand word length, therefore, tree multipliers are normally faster than array multipliers. However, they produce more congestion than CSAMs.

2.4 Rounding for IEEE Floating Point Multiplication

The challenging part within IEEE 754 floating-point multiplication is the ability to round the result after the multiplier prodces its output. Naive versions of rounding are easily implemented in hardware; however, the resulting hardware results in a long critical path due to the large carry chain lengths to produce an answer (i.e. approximately two full carry-propagate additions). Even though it is not presented in this dissertation, the IEEE 754-2008 standard has 128-bit (quad) support [2] that demands efficient hardware designs, especially for rounding within IEEE 754 floating-point multiplication. Consequently, the designs presented in this dissertation are

```
round_to_nearest ($x.sig$, $x.rem$) //RN mode
    if ($x.rem > 0.5$)
        return ($x.sig + 1$)
    else if ($x.rem < 0.5$)
        return ($x.sig$)
    else if ($x.rem == 0.5$)
        if ($x.sig$ is even) return $x.sig$
        else if ($x.sig$ is odd) return ($x.sig + 1$)
round_toward_zero($x.sig$, $x.rem$) //RZ mode
    return $x.sig$
round_toward_positive($x.sig$, $x.rem$) //RP mode
    if ($x.rem > 0$ and $x$ positive)
        return ($x.sig + 1$)
    else return $x.sig$
round_toward_negative($x.sig$, $x.rem$) //RM mode
    if ($x.rem > 0$ and $x$ negative)
        return ($x.sig + 1$)
    else return $x.sig$
round_toward_infinity($x.sig$, $x.rem$) //RI mode
    if ($x.rem > 0$)
        return ($x.sig + 1$)
    else return $x.sig$
```

Figure 2.8: Rounding modes definition

important in reducing the critical path delay and optimizing the hardware.

The IEEE 754 2008 standard requires four rounding modes: `roundTiesToEven`(RN), `roundTowardZero` (RZ), `roundTowardPositive` (RP) and `roundTowardNegative` (RM) [1]. Figure 2.8 shows the pseudo-code for each rounding mode, given $x.sig$ and $x.rem$ are the significant part and remaining part, respectively. However, these four rounding modes can be reduced to three modes: RN, RZ and RI modes, in which RI is the `roundTowardInfinity` mode [6]. In particular, the RP mode can be implemented as the RI mode for positive numbers and the RZ mode for negative numbers. Similarly, the RM mode can be implemented as the RZ mode for positive numbers and as the RI mode for negative numbers. Although the RN mode is default for the IEEE 754 standard, the other three rounding modes are important for methods for reliable computing [10, 24].

For binary implementations, three rounding modes can be implemented by using three bits named the least-significant bit `l`, guard bit `g` and sticky bit `t` [12]. Subsequently, the intermediate product (P) is defined as the product of M_x and M_Y, where P is split into two sections, the mantissa (M_P) and remainder (REM_P). The last bit `l` is defined as the least-significant bit of M_P, while the guard bit `g` is defined as the most significant bit of REM_P. The guard bit is utilized to avoid loss of precision usually a result of two closely representable numbers. Finally the sticky bit `t` is defined as the logical OR of all bits after `g` [12]. It is important to note that the dividing line between M_P and REM_P depends on the range of P. If P overflows the bits are all shifted to the left by 1 position. Together the three bits `l`, `g` and `t` can be combined to check all comparisons required by all four rounding modes. Therefore, all rounding modes can be combined by a single rounding decision bit `r` to be added to the least significant bit of M_P utilizing the following simplified logic:

$$r = \begin{cases} 0 & \text{if } RZ \\ g \wedge (l \vee t) & \text{if } RN \\ g \vee t & \text{if } RI \end{cases} \tag{2.6}$$

Based on this simple scheme, all IEEE rounding modes can be implemented as seen in Figure 2.9. Assuming M_X`[52:0]` and M_Y`[52:0]` are two 53-bit input-operand mantissas with hidden leading ones included ($M_X[52] = M_Y[52] = 1$), a simple rounding

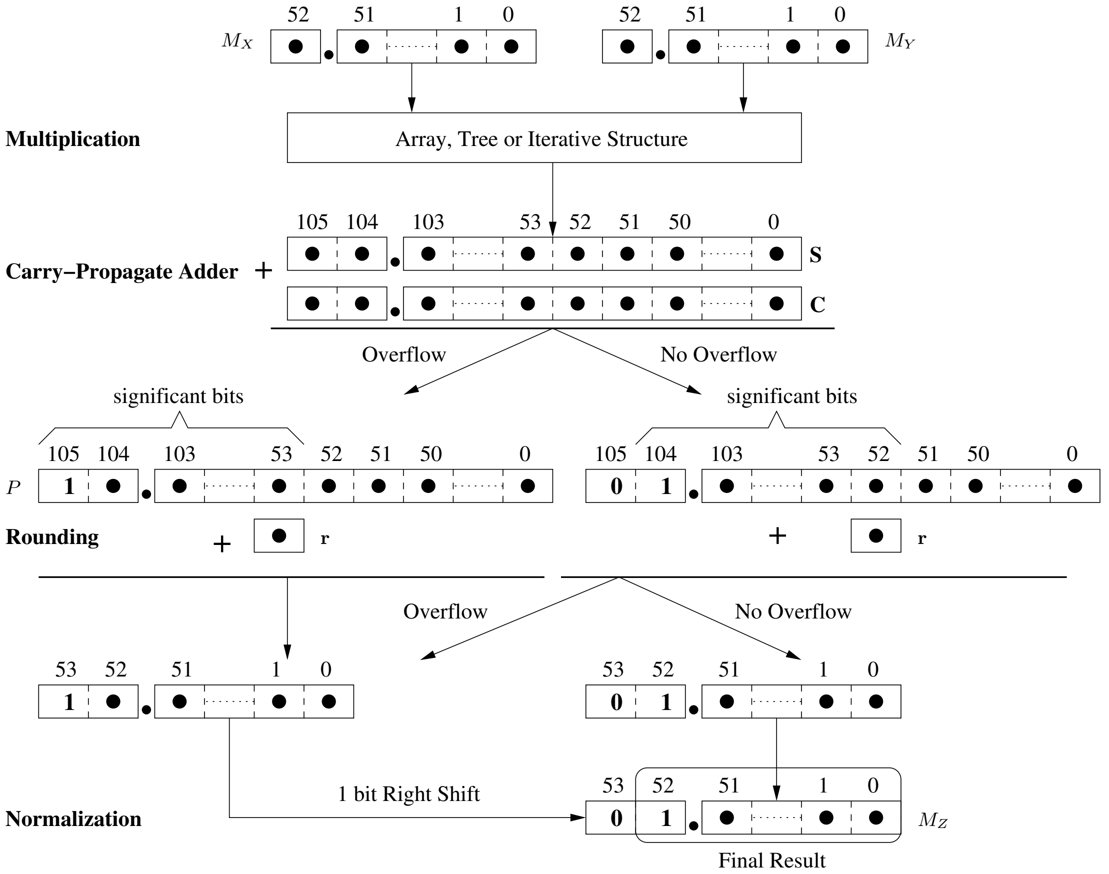


Figure 2.9: An implementation of four IEEE rounding modes (Adopted from [4])

scheme includes four steps is described as follows:

- **Multiplication**: The mantissa multiplication generates the partial products and then reduces to carry-save form that includes a 106-bit carry `C[105:0]` and a 106-bit sum `S[105:0]` vectors.

- **Carry-Propagate Adder (CPA)**: The sum and carry vectors are added using a 106-bit carry-propagate adder (CPA) to generate the 106-bit exact product `P[105:0]`.

- **Rounding**: P must then be rounded to 53 significant bits. First, the rounding decision bit r is computed based on (`l`, `g`, `t`) bits and rounding modes as in Equation 2.6. Assuming $v = P[105]$. If $v = 0$ (no overflow), $l = P[52]$, $g = P[51]$ and `t` is the logical OR of `P[50:0]`. if $v = 1$ (overflow), $l = P[53]$, $g = P[52]$

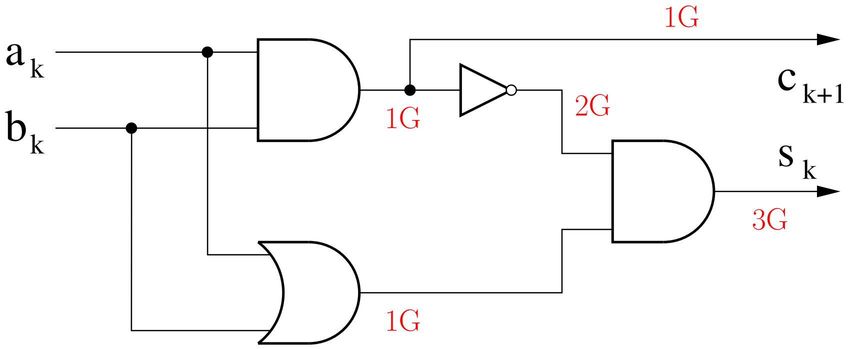


Figure 2.10: An example of delay analysis for half adder

and `t` is the logical OR of `P[51:0]`. The rounding bit `r` is then added to the LSB (which is `P[52]` if there is no overflow and is `P[53]` if overflow) by a 54-bit carry-propagate adder (CPA).

- **Normalization**: Finally, the rounded product needs to be normalized (divided by 2) for the mantissa domain $[1, 2)$ by a right shift if it is equal to or larger than 2. If normalization is needed, then the exponent must be updated to not alter the final result.

This scheme is simple and straightforward. However, it requires one 106-bit CPA, another 53-bit CPA and normalization shifters that are performed in series. As a result, it is not optimal for high-performance designs.

2.5 Linear Delay Analysis

In this dissertation, several algorithms, including state-of-the-art algorithms and proposed algorithms, are discussed. Therefore, an important point to the presented algorithms is, how to characterize the algorithmic aspects of each implementation. Since the actual delay report from synthesis tool is technology-dependent and not always fair, in this dissertation, the linear delay analysis is also used to initially evaluate the delay of an algorithm.

To make sure that each implementation is compared fairly, a given design will be measured based on the gate delay unit [3]. One gate delay (1G) is the delay of AND, OR, and NOT gates. Other gates, such as XOR or MUX, will be derived from three basic gates. Although one could argue that the proposed delay numbers are not practical, these numbers give each implementation a specific cost that can allow a good algorithmic comparison. Moreover, these delay number can also be altered so that they are more practical and address circuit-level constraints.

Figure 2.10 shows an example of linear delay analysis for half adder. The delay is annotated along the way from the inputs (a_k, b_k) to the outputs (c_k, s_k). Based on the Figure 2.10, the half adder has the following critical paths:

$$\begin{aligned} a_k, b_k \rightarrow c_k &= 1G \\ a_k, b_k \rightarrow s_k &= 3G \end{aligned}$$

2.6 Chapter Summary

This chapter first introduces about the IEEE 754-2008 floating-point formats, rounding modes, and an overview about floating-point multiplication. In summary, compared to fixed-point multiplication, the IEEE 754 floating-point multiplication has a much-better range and radix point handling. However, it is also much more complicated and is the leading cause of round-off issues. In floating-point multiplication, rounding is arguably the most complicated component. Although multiplication in general and rounding in particular have been researched for many years, there are still room for architectural improvements that are discussed in following remaining chapters of this dissertation.

CHAPTER 3

A COMBINED IEEE HALF- AND SINGLE-PRECISION FP MULTIPLIERS FOR DEEP LEARNING

3.1 Floating-Point Multipliers for Deep Learning

In recent years, deep learning has grown tremendously in its popularity and usefulness [8]. However, deep learning is computationally intensive, power-hungry and often limited by its hardware capability. Basically, deep learning is a deep neural network, which consists in convolutions and matrix multiplications. A typical example of deep learning is the convolutional neural network (CNN) that includes many convolution and max-pooling layers connected in series to extract features from input images/videos. In [9], authors show that 90% computation of CNN is due to convolution operations. Therefore, multipliers are the most space and power-hungry arithmetic operators of the digital implementation of a CNN. This is also true for other types of deep learning that heavily based on matrix multiplication.

Deep learning applications often requires computations with real numbers and therefore, requires fixed-point or floating-point representations. In comparison with fixed-point numbers, floating-point numbers have an advantage in representable range, which makes it suitable for a wider range of application [2]. Often, existing general-purpose CPUs/GPUs, and software implementations use single-precision floating-point (binary32) as the default format for these types of applications [29, 30, 31, 32].

However, recent research [18] shows that, in many applications, single-precision floating-point multipliers can be replaced by half-precision floating-point multipliers in training deep neural networks, which have little to no impact on the network accuracy. Because half-precision floating-point is only one-half of single-precision floating-point in terms of bit width, using half precision multipliers can save power, area, and delay. Typically, however, single-precision multipliers are still preferred in many cases when high accuracy are required or compatibility matters. Therefore, there is a need for using both single-precision and half-precision multipliers in implementing deep learning.

This chapter proposes a novel combined IEEE half-precision and single-precision multipliers for deep learning [33]. The combined multipliers is basically an IEEE-754 2008 compliant single-precision floating-point multipliers with additional logic to perform IEEE-compliant half precision multiplications. Our design can be easily configured to run in the half-precision mode for power saving or in the single-precision mode for accuracy. Compared to conventional IEEE single-precision multipliers, the combined multipliers require only a small amount of additional area and delay, while offer a significant reduction in power dissipation.

In [34], authors proposed a quadruple-precision floating-point multipliers that can perform one quadruple-precision multiplication, or two double-precision multiplication in parallel. In [35], authors proposed a double-precision floating-point multipliers that can perform one double-precision or two parallel single-precision multipliers. Here we propose a novel combined IEEE single and half-precision floating-point multipliers for deep learning applications.

3.2 A Combined IEEE binary16 and binary32 Multipliers

Generally, the proposed design is an IEEE-754 2008 compliant single-precision floating-point multipliers with additional logic to perform IEEE-compliant half-precision multiplications. Our design can be easily configured to run in the half-precision mode for power saving or in the single-precision mode for accuracy. Using standard IEEE binary32 input operands, our combined multipliers include four inputs:

1. A 32-bit multiplicand A
2. A 32-bit multiplier B
3. A 2-bit rm control signal to select a specific mode among four supported IEEE-compliant rounding modes
4. A 1 bit op control logic is also enabled to switch between binary32 multipliers and binary16 multipliers

The outputs include:

1. A 32-bit product Z
2. A 5-bit IEEE-compliant floating-point flags F

When running in binary32 mode, inputs A and B and output Z are normal binary32 numbers. When running in binary16 mode, the first half the input operands of A, B and Z are binary16 numbers or word-aligned to the input operand. As described in the previous section, the sign computation is exactly the same for both binary32 and binary16 modes. Therefore, only exponent addition and mantissa multipliers need to be modified to adapt to binary16 multipliers.

3.2.1 Exponent Addition

For this design, $E_A = A$[30:23] and $E_B = B$[30:23] are 8-bit input exponents. In binary32 mode, the 8-bit output exponent E_Z is basically the sum of E_A and E_B subtracted by the bias value *127*. However, if there is an overflow in mantissa multiplication, the output exponent should be updated by adding 1 ulp (unit at the last place). This can be implemented effectively by using a 8-bit carry select adder (CSA) that compute the sum of biased exponent E_A and E_B with both carry in 0 and 1 in parallel. The correct output will be selected by the signal that indicate the normalization shift in the mantissa multiplication.

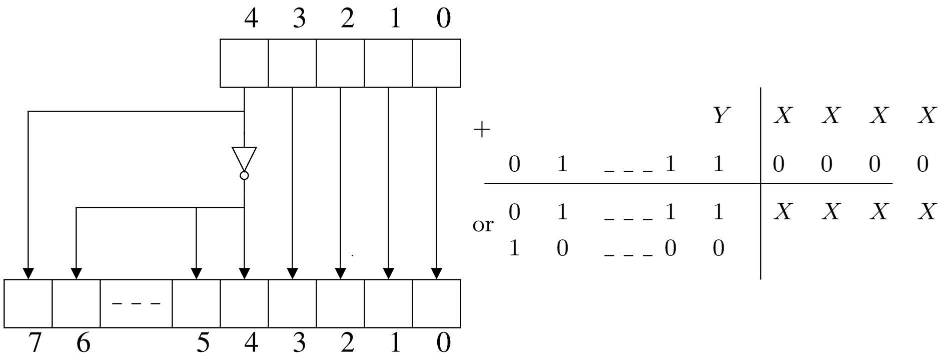


Figure 3.1: Converting binary16 exponent (5-bits) to binary32 exponent (8-bits)

In binary16 mode, however, the exponents are only 5-bit width and the bias value is 15. Clearly, a minimum adjustment to the 8-bit CSA that can perform 5-bit exponent addition is desired. Therefore, a smart transformation is proposed to convert forward and backward between a 5-bit binary16 exponent and 8-bit binary32 exponent.

It is assumed E_{F16} and E_{F32} are the representation of the exponent E_F in the half-precision and single-precision formats respectively. Recall from the previous section that:

$$E_F = E_{F32} - 127 = E_{F16} - 15$$

As a result, the half-precision exponent can be converted to the single-precision exponent by figuring out the difference in their bias as following:

$$E_{F32} = E_{F16} - 15 + 127 = E_{F16} + 112$$

To implement this equation, a simple 8-bit adder can be used to add 112 to the half-precision exponent. However, there is a better solution. The difference of 112 (0111_0000_2 in binary representation) can be adjusted in logic as in Figure 6.1. It can be seen that the last 4 significant bits of E_{F32} are the same as the last 4 significant bits of E_{F16}. The most 4 significant bits of E_{F32} can be either 0111 or 1000 when the most significant bit Y of E_{F16} is 0 or 1 correspondingly. This is implemented by

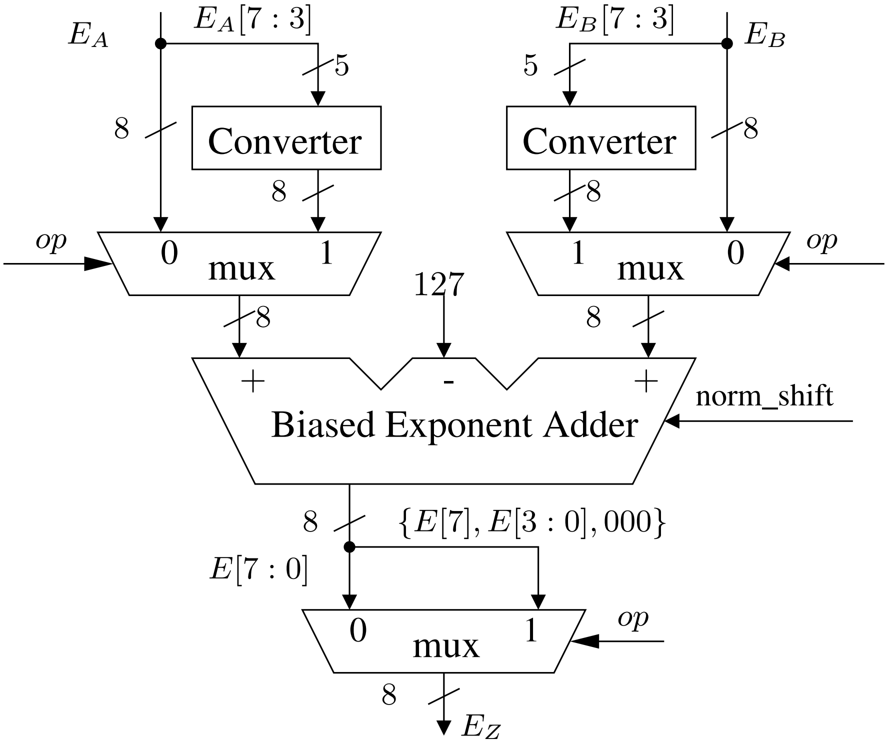


Figure 3.2: A combined IEEE half and single-precision exponent addition

simply using an inverter gate as seen in the Figure 3.1.

Based on this conversion method, the combined half and single precision exponent addition design is shown in Figure 3.2. In the binary32 mode ($op = 0$), this design utilizes a standard binary32 exponent addition. The biased exponent adder is the 8-bit Carry-Select Adder that computes $E_A + E_B - 127$ and $E_A + E_B - 127 + 1$ in parallel. If a normalization shift is required in the mantissa multiplication ($norm\_shift = 1$), the output exponent E_Z is $E_A + E_B - 127 + 1$. Otherwise, the output exponent E_Z is $E_A + E_B - 127$.

In binary16 mode ($op = 1$), the 5-bit half-precision exponents `E_A[7:3]` and `E_B[7:3]` are first converted to 8-bit single precision exponents using the converters as in Figure 6.1. The 8-bit output exponent $E[7:0]$ is finally converted back to half-precision exponent by taking only the most significant bit and the last 4 significant bits. Clearly, by using this scheme, minimal overhead is required to perform both single and half-precision exponent addition.

3.2.2 Mantissa Multipliers

It is assumed that $M_A[23{:}0]$ and $M_B[23{:}0]$ are the two 24-bit input mantissas with hidden leading one included ($M_A[23] = M_B[23] = 1$). In binary32 mode, the mantissa multiplication basically includes a 24×24 multipliers that compute the product P of two input mantissas M_A and M_B, followed by a rounding and normalization logic to round and normalize P to the 24-bit output mantissa M_Z.

Recall that the IEEE 754 standard defines four rounding modes to be supported: round-to-nearest (RN), round-toward-zero (RZ), round-toward $+\infty$ (RP) and round-toward $-\infty$ (RM). Although RN is default for the IEEE 754 standard and most implementations, the other three rounding modes are important for methods in reliable computing [10]. In binary implementation, this can be done by using four bits named sign bit S, last bit L, guard bit G and sticky bit T [12]. Using (S, L, G, T) bits, four rounding modes can be combined by a single rounding decision bit RV to be added to the least significant bit:

$$RV = \begin{cases} 0 & \text{if } RZ \\ G \wedge (L \vee T) & \text{if } RN \\ \overline{S} \wedge (G \vee T) & \text{if } RP \\ S \wedge (G \vee T) & \text{if } RM \end{cases} \tag{3.1}$$

Using the rounding scheme above, the mantissa multiplication for binary32 can be implemented as in Figure 3.3. This implementation includes four steps as follows:

1. Compute 48-bit exact product P using a 24×24 multipliers
2. Normalize P to domain $[1, 2)$ if needed. Since $1 \leq M_A, M_B < 2$, product P is in $[1, 4)$ range. If $P \geq 2$ ($P[47]{=}1$), overflow occurs. A normalization by shifting one position to the right is needed to produce normalized product $NP[47{:}0] \in [1, 2)$.

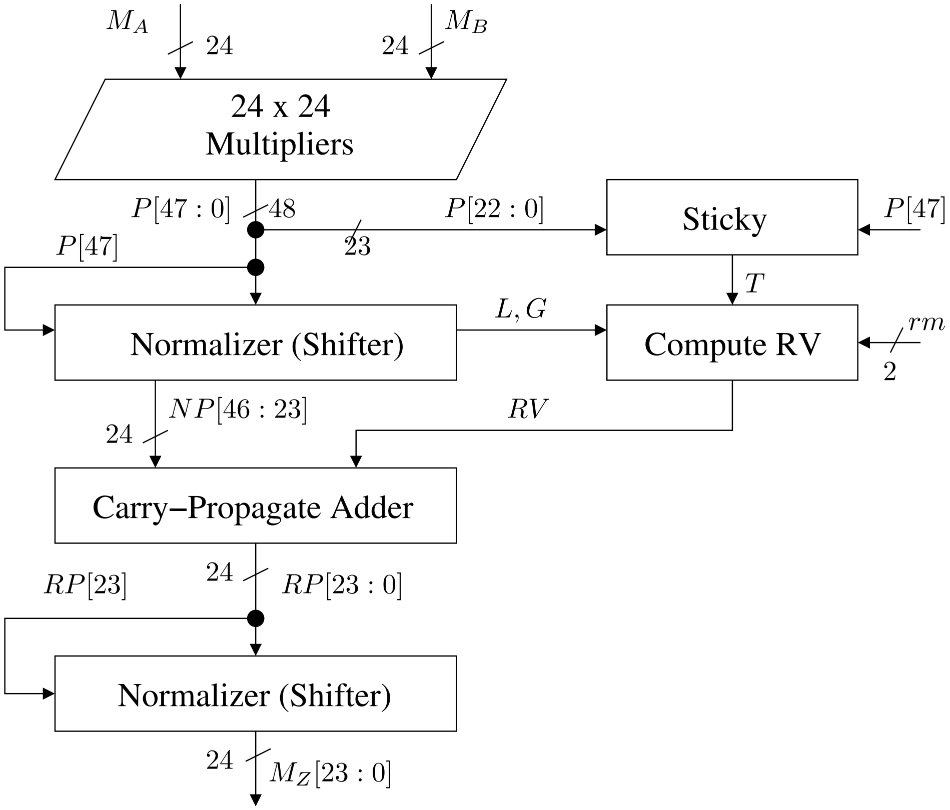


Figure 3.3: Mantissa multipliers for single-precision

3. Round the normalized product (NP) to a rounded product (RP). The last bit L and guard bit G are taken directly from NP as follows:

$$L = NP[23]; G = NP[22]$$

The sticky bit T is computed in parallel with normalization as follows:

$$T \;=\; \begin{cases} OR(P[21:0]) & \text{if no overflow(P[47]=0)} \\ OR(P[22:0]) & \text{if overflow(P[47]=1)} \end{cases}$$

The rounding decision (RV) is then computed based on rounding modes as in Equation 3.1. The sign S in equation is the sign S_Z of the output Z. A 24-bit Carry-Propagate Adder (CPA) is then used to add RV to normalized product NP to produce 24 -bit rounded product RP.

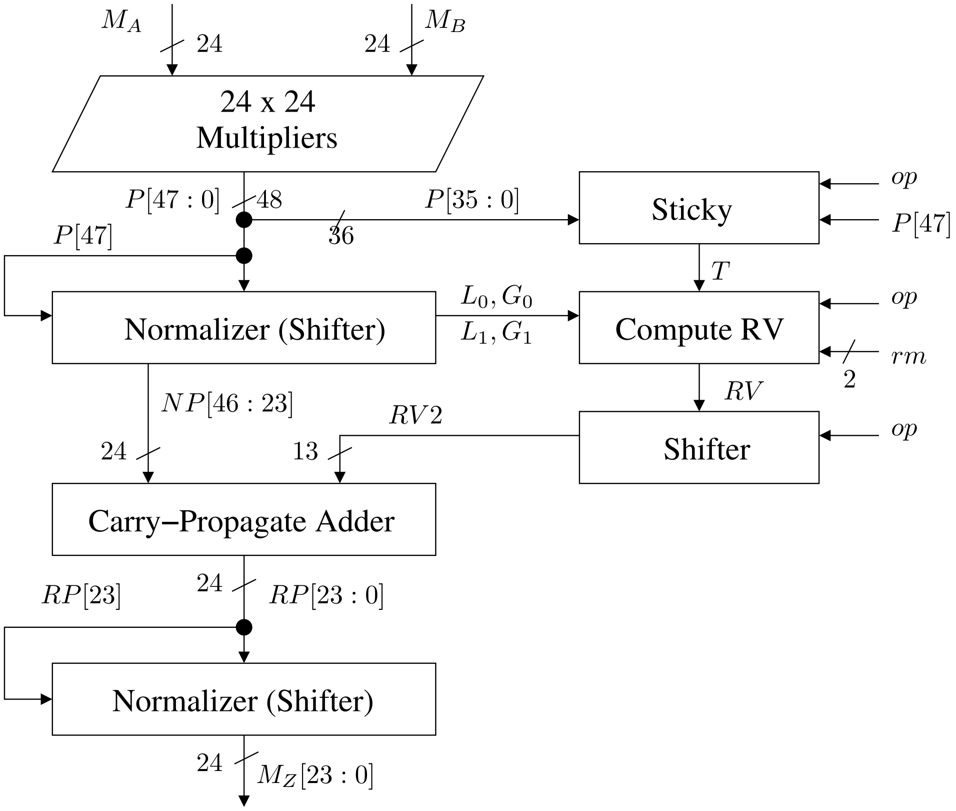


Figure 3.4: Modifed mantissa multipliers

4. Finally, the rounded product (RP) needs to be normalized if overflow occurs ($RP[23] = 1$) due to rounding. Hence, another conditional shifter is used to right shift RP one bit to produce the final result M_Z[23:0].

This implementation can be adjusted to perform both binary32 and binary16 mantissa multiplication, as in Figure 3.4. It is worth to notice that in binary16 mode, the 11-bit mantissa is left-aligned in the 24-bit mantissas. This is commonly done with most floating-point representations in most general-purpose processors [36]. In comparison with the binary32 multipliers, the combined multipliers changes only in the rounding step (step 3 in binary32 multipliers) as follows:

1. **Sticky**: The Sticky logic is modified to compute stick bit T for both modes. In

binary16 mode ($op = 1$):

$$T = \begin{cases} OR(P[34:25]) & \text{if no overflow(P[47]=0)} \\ OR(P[35:25]) & \text{if overflow(P[47]=1)} \end{cases}$$

2. **Compute RV**: Use the last bit L_0 and the guard bit G_0 when $op = 0$ and L_1 and G_1 when $op = 1$. L_0 and G_0 are the same as L and G in binary32 multipliers while L_1 and G_1 are computed as follows:

$$L_1 = NP[36]; G_1 = NP[35]$$

3. **Shifter**: Once the rounding value RV available, it should be added to the least significant bit that is $NP[23]$ in binary32 mode and $NP[36]$ in binary16 mode. Therefore, a 13-bit left shifter is used to shift RV to the right position for each mode.

3.3 Experimental Results

The proposed multipliers are implemented in RTL- compliant Verilog and then synthesized in an ARM 32nm CMOS library in IBM/GF cmos32soi technology. 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® Design Compiler™ (DC) in topographical mode and loaded by ARM-based flip-flops using a PVT process

Table 3.1: Post-synthesis results for the proposed design in cmos32soi 32nm IBM/GF technology

Multipliers	Power [uW]			Area [um^2]	Delay [ps]
	Dynamic	Static	Total		
DW_fp_mult (binary32)	778.582	1,029.175	1,808.757	5,162.742	484.179
Combined Mulipliers (binary32)	558.654	680.775	1,239.429	3,309.228	454.004
Combined Mulipliers (binary16)	151.501	666.142	817.643	3,309.228	454.004

at 25° C using TT corners. The average power estimation was achieved by running the simulation on 50,000 random test vectors generated by TestFloat-3b [37]. Because there are no existing similar designs, this chapter compares the results versus a Synopsys® DesignWare™ (DW) binary32 multiplier implementation. The Synopsys DW unit only contains a IEEE- compliant single-precision unit and this implementation has advantages in providing both operations configurable by input control bits. The area, delay, and power dissipation of each of these multipliers are found in Table 3.1. As shown in the Table3.1, the binary16 mode can save 44% power compared to binary32 mode and save 55% power compared to Synopsys DesignWare multipliers. Technically, the RTL-level implementations also employ DesignWare elements as they are coded to take ad- vantage of DesignWare, but at a lower level of hierarchy and an with some additional ancillary logic discussed in this chapter.

3.4 Chapter Summary

In summary, a combined IEEE binary32 and binary16 multipliers is presented for many deep learning implementations in which binary16 can be safely used to train and run a network. With a configurable control signal, our proposed multipliers can be easily configured to switch between single and half-precision IEEE floating-point modes. This design is completely verified with Hauser's SoftFloat scheme for the correctness. Compared to the IEEE standard binary32 multipliers, the proposed multipliers has a small overhead while provide a significant savings of 44% in power dissipation when running in binary16 mode.

CHAPTER 4

A NOVEL ROUNDING SCHEME FOR IEEE-COMPLIANT FP MULTIPLICATION

This chapter first introduces the state-of-the-art in rounding for IEEE 754 Floating Point Multiplication. The previously-used methods described in Section 4.1 all optimize on the most delay-intensive portion of the computation: rounding. Consequently, most of the methods for optimizing here are optimizing rounding for the output of the multiplier unit, usually produced in carry-save notation (i.e., carry/sum). Any of these method can be utilized with different multiplier algorithms and consequently impact other areas related to use of floating-point multiplication such as energy and power dissipation. However, the major contribution of this dissertation is improving the rounding unit for delay optimization [19, 38].

4.1 Previous Work

4.1.1 Santoro, Bewick and Horowitz (SBH) Method

An early optimization for floating-point multiplication architectures, Santoro, Bewick and Horowitz [4] (SBH) demonstrate a method that reduces the hardware requirements to a single compound adder (CA) and small amount of support logic. The most significant contribution of their work demonstrates how to perform most of the rounding and post-normalization in parallel, thereby, removing a carry propagate adder from the critical path. Compound adders take advantage of utilizing redundant hardware and its use is critical in optimizing hardware for any implementation [39]. Normally, compound adders use the same hardware except for critical components,

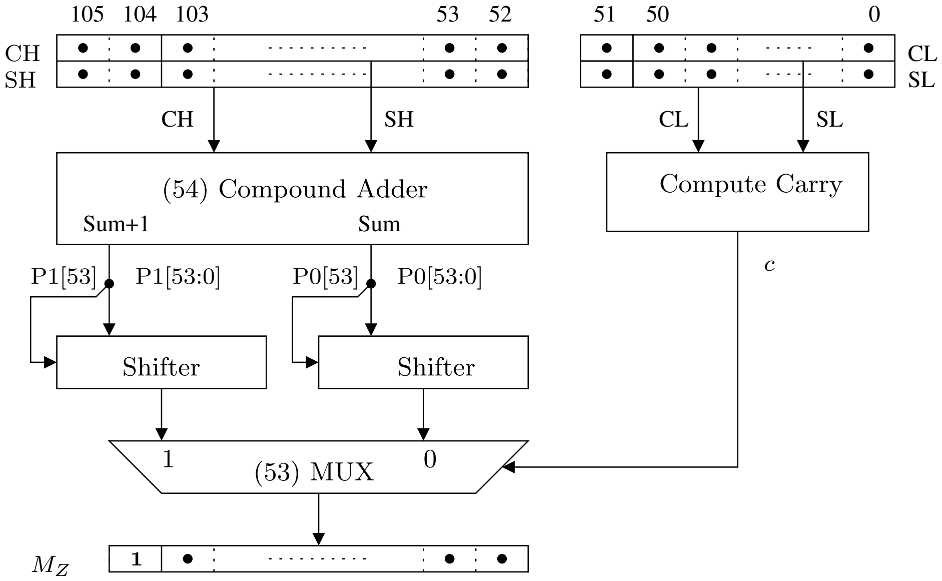


Figure 4.1: SBH [4] implementation of RZ mode

such as the carry-chain logic [4].

As with most floating-point units, designs can optimize the critical path by computing the carry and save portions of the multiplier [12]. Consequently, SBH's method utilizes the upper 54 most significant bits for SH, CH ($PH = SH + CH$) and the the 52 least significant bits for SL, CL ($PL = SL + CL$), respectively. Additional bits are utilized to help optimize the final result, such as the MSB (v), LSB (l), guard bit (g) and the sticky bit (t) of the exact product P.

4.1.1.1 RZ mode

For RZ mode, since the rounding bit $r = 0$, the rounded product is simply the exact product truncated to the 52 least-significant bits. To speed up performance, it is desirable to perform the computation of significant part SH, CH and remaining part SL, CL in parallel. Since c is only available after the addition of SL and CL, a compound adder (CA) can be used to pre-compute both PH and PH+1 in parallel as shown in Figure 4.1. Subsequently, two right shifters are used to normalize PH and PH+1, respectively, if they overflow. The carry-in c can then be used to select the correct output from the two normalized products.

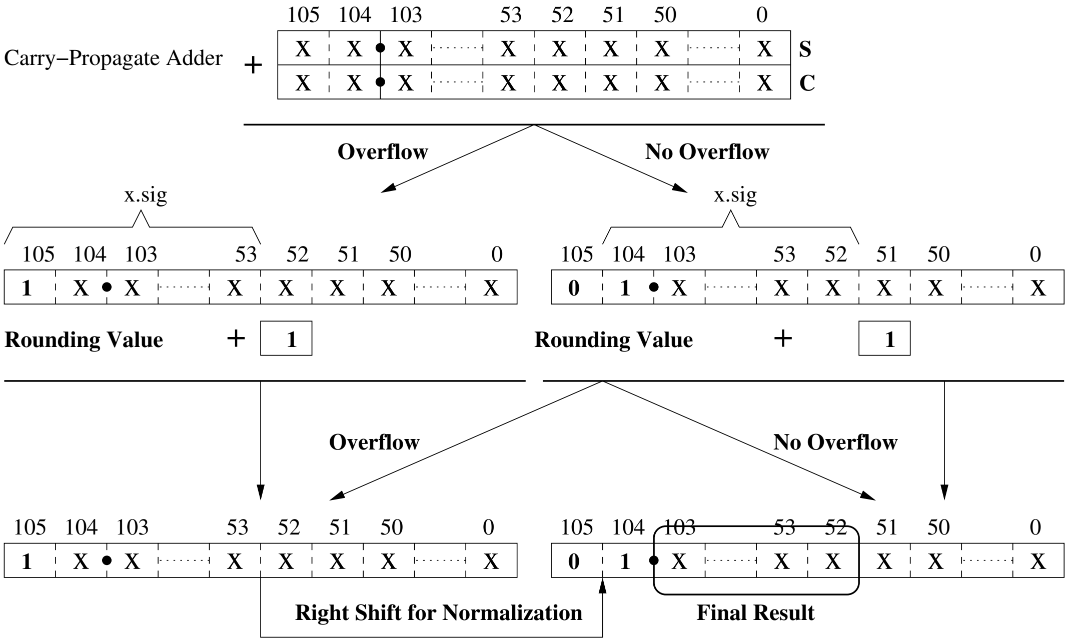


Figure 4.2: A simple implementation of RNU mode

4.1.1.2 RN mode

The RN mode is arguably the most complicated mode compared to RZ and RI modes. The method within SBH smartly designs for round-to-nearest/up (RNU) mode (`roundTiesToAway` mode in IEEE 754-2008 standard) and then modifies the design to produce RN mode. The RNU mode is RN mode except in case of tie ($x.rem = 0.5$) where RNU mode always rounds up. In terms of implementation, RNU can be implemented by simply adding 1 to the guard bit position. The implementation in Figure 2.9 can be modified to perform RNU mode as shown in Figure 4.2.

Although simpler than RN, RNU mode is still far more complicated than RZ mode. To compute `SH`, `CH` in parallel with `SL`, `CL`, the total value INC is required to add to the ulp of `SH`, `CH`. This `INC` is the sum of carry `c` from `SL`, `CL` and the rounding value to be added to the exact product as illustrated in Claim 1.

Claim 1. In RNU mode,

$$INC \;=\; \begin{cases} c+g & \text{if no overflow } v=0 \\ c+1 & \text{if overflow } v=1 \end{cases} \tag{4.1}$$

Proof. If there is no overflow, RNU rounds up (i.e., adds 1 to the LSB of `SH`, `CH`) if and only if $g = 1$. Therefore, $INC = c + g$. If overflow occurs, rounding then adds 1 to the guard bit position, which is now the LSB of `SH` and `CH`. Therefore, $INC = c + 1$. ■

From Equation 4.1, INC can be 0, 1, or 2. The naive solution pre-computes all three possible outputs `PH`, `PH+1` and `PH+2` using `INC` to select the correct rounded product. However, based on Claim 2 below, the standard CA can be used as in RZ mode.

Claim 2. Assuming `rs`, `rc` is the MSB of `SL`, `CL`, respectively, the prediction bit $p = rs \vee rc$ produces:

$$INC \;\in\; \begin{cases} \{0,1\} & \text{if } p=0 \\ \{1,2\} & \text{if } p=1 \end{cases} \tag{4.2}$$

Proof. If $p = 0$, then $rs = rc = 0$. As a result, the sum of `SL` and `CL` produces no carry out. In other words, $c = 0$. Therefore, INC is $INC = g$ or $INC = 1$ and in both cases, $INC \in 0, 1$. If $p = 1$, it is obvious for the overflow case to compute $INC \in 1, 2$. In the no overflow case, because $rs + rc \geq 1$, c and g cannot be both zeros. As a result, $INC \in 1, 2$. ■

Based on this observation, SBH's method uses the 2 outputs of a CA as shown in Figure 4.3. A row of 54 half-adders (HAs) is used to make room for a prediction bit p before the CA. If $p = 0$, the outputs `P0`, `P1` of the CA are `(PH, PH+1)`. If $p = 1$,

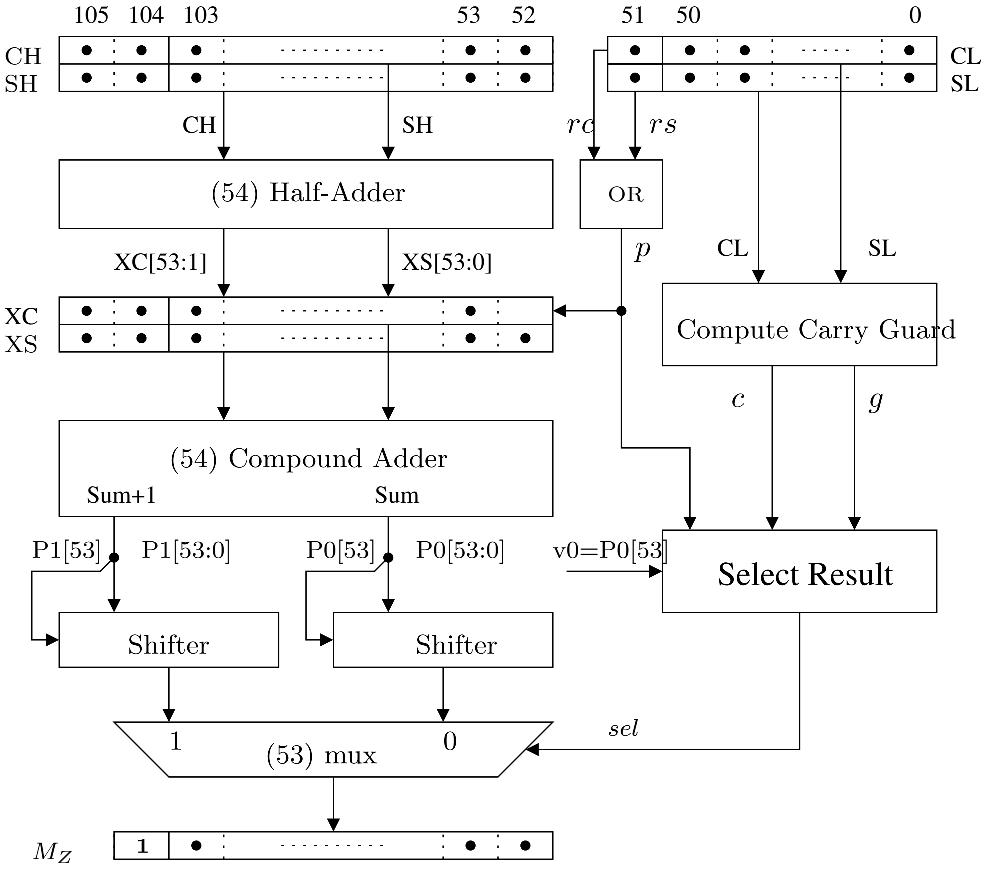


Figure 4.3: SBH [4] method for RNU mode

the outputs of CA are effectively (PH+1, PH+2).

Once all possible rounded products P0 and P1 have been computed, the correct result can be selected. As explained in [4], the correct rounded product depends on INC, which in turn depends on c, g and the overflow bit v, as in Equation 4.1. At this point, both g, c bits should be available from the **Compute Carry Guard** module. However, a designer still needs to identify the correct overflow bit v, which is the MSB of the exact product P. The original SBH method requires additional logic to select the correct overflow bit v between two MSBs v_0, v_1 of P0 and P1. However, based on the Claim 3 below, this logic can be removed safely for optimization.

Claim 3. To select the correct output, $v_0 = P0[53]$ can be treated as v in Equation 4.1.

Proof. Recall that v is the MSB of the exact product P $(PH + c)$ while v0 is the MSB

Table 4.1: Compound adder output selection for RNU modes

p	INC	Correct Output	sel
0	0	PH	0
0	1	$PH+1$	1
1	1	$PH+1$	0
1	2	$PH+2$	1

of $PH+p$. Therefore, the claim is obvious if $p=c$ and p and c need to be examined whether they are different. This happens only when $p=1$ and $c=0$. However, in this case, $g=1$ because $rs+rc=1$. Therefore, according to Equation 4.1, $INC=1$ regardless of the value of v. ■

Based on this observation, Table 4.1 shows the value of `sel` to select the CA output correctly. Interestingly, if $p=0$ then `sel` equals `INC`, whereas, if $p=1$, because a prediction bit 1 is pre-added, the sel should be equal to $INC-1$. This can be implemented by using an XOR gate. Moreover, post-normalization can be performed before the final selection to speed up the performance.

4.1.1.3 RI mode

For designs based on RNU mode instead of the original RN mode, SBH's method requires only a single CA that can produce 2 rounded products (`PH`, `PH+1`) or (`PH+1`, `PH+2`) in parallel. However, this design does not work for RI mode.

Claim 4. In RI mode:

$$INC = \begin{cases} c+(g\vee t) & \text{if no overflow } v=0 \\ c+2\cdot(l\vee g\vee t) & \text{if overflow } v=1 \end{cases} \tag{4.3}$$

Proof. If no overflow occurs, from Equation 2.6, a rounding decision is added $r=(g\vee t)$ to last bit position 52. Therefore, in this case, $INC=c+(g\vee t)$. If overflow occurs, because of a normalization shift, the last bit position now is 53. Therefore,

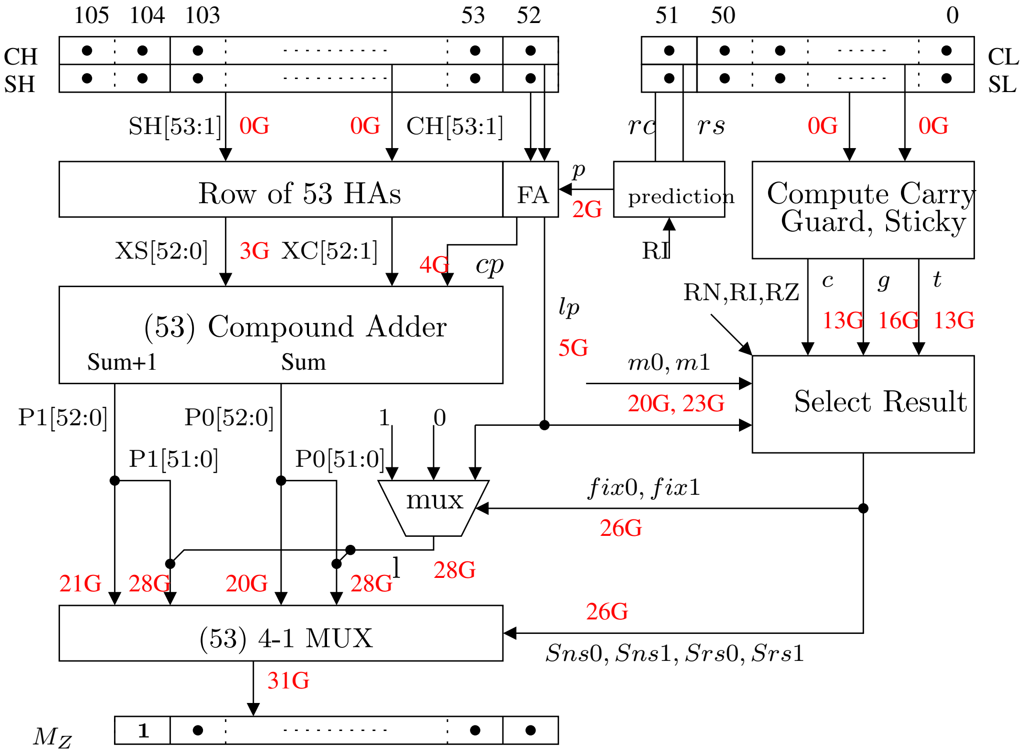


Figure 4.4: QTF [5] method implementation for all IEEE modes

the rounding decision $r = (l \vee g \vee t)$ is added to bit 53. This is equivalent to adding $2 \cdot (l \vee g \vee t)$ to INC. ■

From Equation 4.3, it can be seen that the total carry INC can be 0, 1, 2 or 3. As a result, the significant part logic has to be able to pre-compute all 4 outputs from PH to PH+3 to implement the RI mode. However, as described in the previous section, SBH's method only determines 3 outputs from PH to PH+2 and, therefore, it does not work for RI mode.

4.1.2 Quach, Takagi and Flynn (QTF) Method

The limitation of SBH's method in implementing RI mode comes from the limitation of the original 2 outputs from the CA (Sum, Sum+1). To solve this problem, QTF's method proposes a special CA that is able to pre-compute 3 outputs (Sum, Sum+1, Sum+2) in parallel [5]. In general, this special compound adder computes only (Sum,

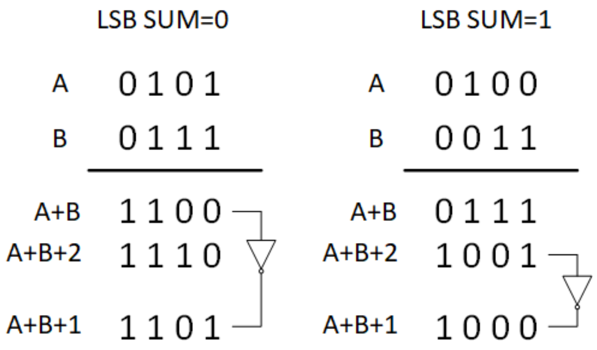


Figure 4.5: Two examles illustrating how third sum is obtained with only two carry chains

Sum+2) in parallel and includes a small amount of additional logic to generate (Sum+1) from Sum and Sum+2. Given l is the LSB of output Sum, it is easy to see that l is also the LSB of output Sum+2. When $l = 0$, Sum+1 can be generated by selecting Sum that changes the last bit from 0 to 1. When $l = 1$,Sum+1 can be generated by selecting Sum+2 that changes the last bit from 1 to 0. This trick is useful in that it can be utilized instead of using another adder (e.g., carry-select/increment adder) by inverting the least-significant bit of the calculated sums. It also has the advantage of consuming a small amount of additional logic (i.e., an inverter) and no impact upon the critical path. This is shown in Figure 4.5 where the LSB is used to determine if the LSB is inverted or not depending on whether the LSB is even or odd.

The method by Quach, Takagi and Flynn (QTF) replaces the LSB of the half adder row in the SBH method by a full adder (FA) [5]. The inputs to this FA are CH[0], SH[0] and the prediction bit p. The carry out cp of the FA feeds into the LSB of the CA while the sum of the FA serves as the lp bit. The 54-bit CA in the SBH method is subsequently reduced to a 53-bit compound adder (CA). The input to this CA comes from a 53-bit half adder row. This arrangement allows the CA to effectively compute PH and PH+2 directly and generate PH+1 from PH and PH+2. A 3:1 multiplexor (mux) is used to select the correct LSB by keeping the lp or setting it to 0 or 1. The fix0, fix1 signals determine when lp should be not changed or set to 0 and 1, respectively.

4.1.2.1 Errors in Quach, Takagi and Flynn Hardware

To produce the correct result, a final-selection `4:1` mux selects from the high order 53 bits of `P0` and `P1` (if overflow occurs) and the lower order 53 bits of `P0` and `P1` (if no overflow occurs). It is worth noticing that the LSB of the lower order values are from the `3:1` mux as shown in Figure 4.4. Four control signals `Sns0`, `Sns1`, `Srs0`, `Srs1` are used to select the correct output. For $Sns0 = 1$ and $Sns1 = 0$, the lower order bits of `P0` and `P1` are selected if there is no overflow. Similarly, $Srs0 = 1$ and $Srs1 = 1$ select the higher order bits of `P0`, `P1`, respectively, if an overflow occurs. To examine the no overflow/overflow condition, QTF's method uses two bits `m0`, `m1` that are selected as the MSB of `P0` and `P1`, respectively. Two bits `n0`, `n1` are the second-to-least significant bits of `P0` and `P1` and serve as the last bit in an overflow case. However, QTF's assertion is false with $m0 = 0$ and $m1 = 1$ as shown as shown in Claim 5 below:

Claim 5. The QTF method utilizes control signals in RN mode as follows (also, given in [5] and verified/corrected as discussed later):

$$\begin{aligned}
p &= 0 \\
l &= lp \\
Sns0 &= \overline{m0} \wedge \overline{c} \wedge (\overline{g} \vee \overline{l}) \vee \overline{m1} \wedge c \wedge \overline{g} \wedge \overline{l} \\
Sns1 &= \overline{m0} \wedge \overline{c} \wedge g \wedge l \vee \overline{m1} \wedge c \wedge (g \vee l) \\
Srs0 &= m0 \wedge \overline{c} \wedge (\overline{l} \vee \overline{g} \wedge \overline{n0} \wedge \overline{t}) \\
& \quad \vee m1 \wedge c \wedge \overline{g} \wedge \overline{n1} \wedge \overline{l} \wedge \overline{t} \\
Srs1 &= m0 \wedge l \wedge \overline{c} \wedge (n0 \vee t \vee g) \vee \\
& \quad m1 \wedge c \wedge (n1 \vee l \vee t \vee g) \\
fix0 &= \overline{m0} \wedge l \wedge \overline{c} \wedge g \vee \overline{m1} \wedge l \wedge c \wedge (\overline{g} \vee \overline{t}) \\
fix1 &= \overline{m0} \wedge \overline{l} \wedge \overline{c} \wedge g \wedge t \vee \overline{m1} \wedge \overline{l} \wedge c \wedge \overline{g} \qquad (4.4)
\end{aligned}$$

Unfortunately, QTF's method produces false assertions when $t = 1$, $g = 0$, $lp = 0$, $c = 1$, $m0 = 0$ and $m1 = 1$.

Proof. When $t = 1$, $g = 0$, $lp = 0$, $c = 1$, $m0 = 0$ and $m1 = 1$, the correct output should be `PH+1` because $c = 1$ and $g = 0$. Since $lp = 0$, `PH+1` should be selected from `PH` (then assert $lp = 1$). In other words, $Sns0 = 1$ and $fix1 = 1$. However, according to the equations above, when the control signals $Sns0 = 0$, $fix1 = 0$ $Srs1 = 1$, results in selecting the wrong output `PH+2`. ■

The main reason for this is that with the special CA, the MSB of `P1` can not be selected as `m1` (i.e., defined as the MSB of `PH+1`). Recall that with a special CA, the output `PH+1` can be selected from either `PH` or `PH+2`. Therefore, if $lp = 0$ (which means `m1` should be the MSB of `P0` instead), the equation will take the wrong `m1` and then the wrong output. To fix this issue, additional logic is required to generate the true `m1` from the MSB of `P0` and `P1` as follows:

$$m1 = (lp \wedge P1[53]) \vee (\overline{lp} \wedge P0[53]) \tag{4.5}$$

This can introduce some delay and complicate the Boolean equation for this control signal but only minimally.

4.1.2.2 Linear Delay Analysis

Since the actual delay is technology-dependent and not always fair and intuitive, in this section, we perform the linear delay analysis that based on the gate delay unit. One gate delay (1G) is the delay of AND, OR and NOT gates. The Carry and Sum vector bits are assumed available at the same time $0G$

- MUX and XOR gates take 3 gate delay (3G)
- The 53-bit Compound Adder is implemented using parallel prefix adder. The output Sum is valid after $4 + 2 \cdot \log_2(53) = 16$ gate delay. The output $Sum + 1$ is valid after $5 + 2 \cdot \log_2(53) = 17$ gate delay.
- Similarly, Carry c and Guard g are also generated using 52 -bit parallel prefix adder. The Guard g is valid after $4 + 2 \cdot \log_2(52) = 16$ gate delay. The Carry c is valid after $1 + 2 \cdot \log_2(52) = 13$ gate delay.
- Sticky t is generated using the trick as in [40]. The delay is $7 + \log_2(52) = 13$ gate delay.

Based on these assumptions, the timing estimation of QTF method is annotated in Figure 4.4. Since prediction bit `p` is valid after 2 gate delay and `CH[0], SH[0]` is valid after 0 gate delay, the FA carry out `cp` is valid after 4 gate delay make `XC` vector valid after 4 gate delay. The Compound Adder will provide `P0` and `P1` after 16 and

17 gate delay respectively. As a result, m0 is valid after 20 gate delay and m1 is valid after 23 gate delay based on Equation 4.5.

Carry c, Guard g and Sticky t are ready after $13, 16, 13$ gate delay respectively. Because fix0, fix1 are computed using $m0, m1$, which adds 2 more gate delay, they are valid after 26 logic levels. Similarly, Sns0, Sns1, Srs0, Srs1 are valid after 26 logic levels.

fix0 and fix1 are used to select correct l that is only valid after 28 gate delay. Finally, the 4:1 mux will output M_Z after 31 gate delay.

4.1.3 Even and Seidel (ES) Method

An efficient method called the Even/Seidel rounding algorithm (ES) works by reducing all rounding modes to a single truncation operation [41, 6]. The ES method adds an injection constant, depending only on the rounding mode, to the intermediate product such that all modes reduce to truncation at the rounding bit position. The injection constant (INJ) is defined as:

$$INJ = \begin{cases} 2^{-53} & \text{if RN} \\ 2^{-52} - 2^{-104} & \text{if RI} \\ 0 & \text{if RZ} \end{cases} \tag{4.6}$$

The ES algorithm provides little advice about implementing injection bits as well as computing the sticky t, guard g and carry c bits. For convenience, a block diagram of the injection hardware has been included in Figure 4.6. The vector INJ[51:0], whose value is assigned depending on each rounding mode, as in Equation 4.6, is injected into the lower part of the vector through a Carry-Save Adder. The carry out c_{inj} of this adder then feeds into a FA together with two LSBs of CH and SH, similar to the QTF method. The row of 53 HAs makes room for injecting the carry out of

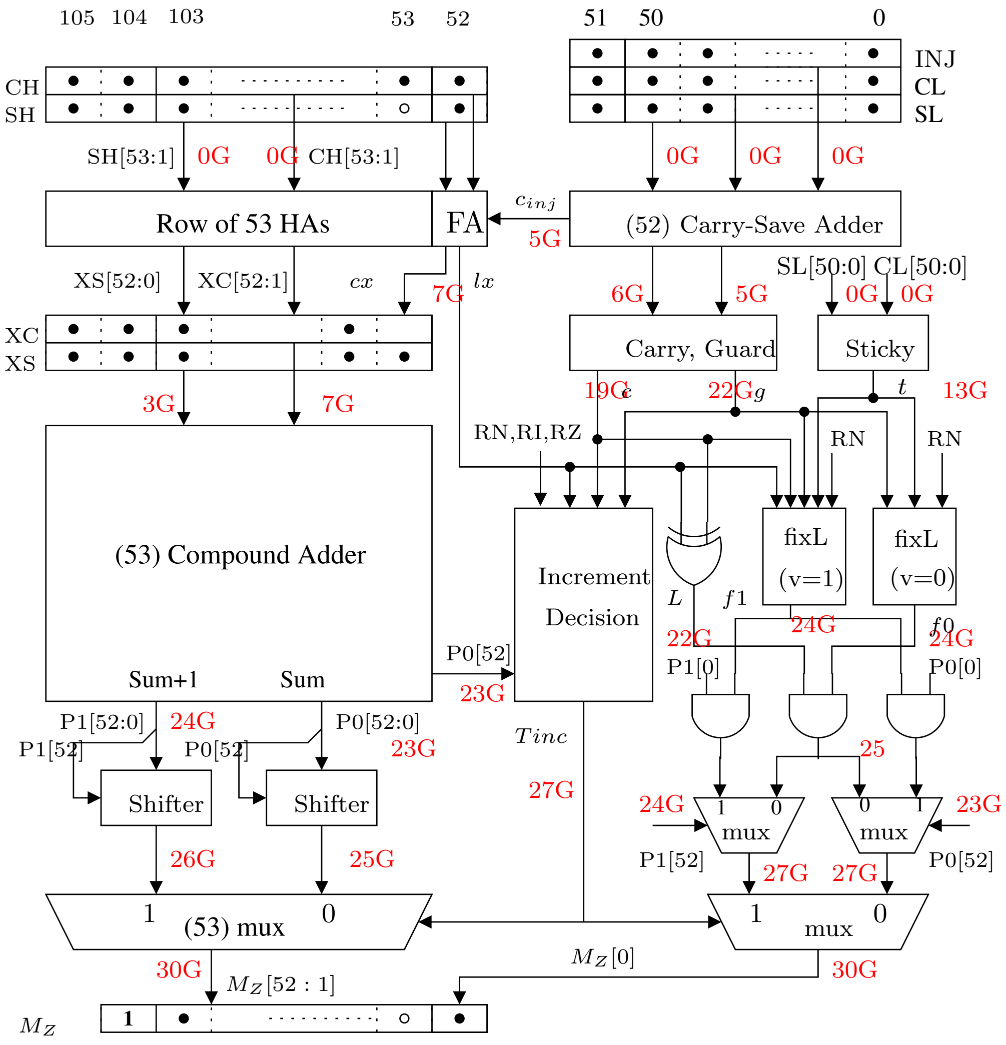


Figure 4.6: ES [6] Method Implementation for Rounding

this FA. Finding `g` and `c` requires computing the carry propagate chain through all bits in the lower path (`51:0`), however, only the sum bit needs be generated at bit 51. Using the technique from [40], the sticky bit can be computed directly from its carry-save format as shown in Figure 4.6.

4.1.3.1 Optimization of Even-Seidel Rounding

To account for `c` carrying into the upper datapath and the overflow, the ES paper utilizes a special increment decision, $Tinc$ (`INC` in [6]). The equation definition in the ES paper is correct; however, there is a small change in the implementation logic that is needed within [6]. `Tinc` should increment when the majority of `g`, `c`, and `lx` are high, and the mantissa has overflowed $P0[52] = 1$, and the rounding mode is RN. In the case of $lx = 1$, $g = 1$, and $c = 0$ the logic incorrectly sets $Tinc = 0$. The following equation correctly implements $Tinc$:

$$Tinc = \begin{cases} (c \wedge g \wedge RN) \vee (lx \wedge g \wedge RN) & \\ \vee\ (c \wedge l \wedge (RZ \vee RN)) & \\ \vee\ ((lx \vee c) \wedge RI) & \text{if P0[52]} = 0 \\ lx \wedge c & \text{if P0[52]} = 1 \end{cases} \tag{4.7}$$

In the case when the rounding mode is RN, the LSB needs to be corrected in a tie case ($x.rem = 0.5$ or $g = 1, t = 0$) and the LSB of exact product $l = 1$. This LSB is pulled down to 0 (even) by AND-ing it with a bit that equals to 0 only for a tie-case of RN. Two blocks **fixL(v=1)** and **fixL(v=0)** generate `f1` and `f0` to fix the LSB in case of overflow and no overflow, respectively.

$$\begin{aligned} f0 &= \overline{g} \vee t \vee \overline{RN} \\ f1 &= \overline{lx \oplus c} \vee \overline{g} \vee t \vee \overline{RN} \end{aligned} \tag{4.8}$$

4.1.3.2 Linear Delay Analysis

Based on the same delay assumptions as in 4.1, the timing estimation of ES rounding is annotated in Figure 4.6. The Carry-Save Adder is used to add the injection `c_inj` first and account for 5 gate delay. On the left side, the FA adds 2 more gate delay before Compound Adder. Therefore, the overflow bit `v0=P0[52]` and `v1=P1[52]` are valid after 23 gate delay and 24 gate delay respectively. T_{inc} in Equation 4.7 is valid after 27 gate delay. As a result, the $M_Z[52:1]$ is ready after 30 logic levels since MUX will add 3 gate delay.

On the right part, the Carry `c`, Guard `g` and Sticky `t` will be valid after $19, 22 and 19$ gate delay levels in that order. The control logic $f1$ and $f0$ are both valid after 24 gate delay. The row of 3 AND gates and two MUXes levels will add 6 gate delay, making the final Z[0] valid after 30 gate delay. Combined with the left part delay, ES method is 30 gate delay.

4.2 Proposed Method

Although the SBH method is smart, intuitive and simple to implement, it does not work optimally for RI mode. As an improved solution to SBH's method, QTF's method introduces a special compound adder (CA) that can produce 3 outputs in parallel. Using this special CA, QTF method works for all rounding modes. However, to perform rounded product selection, QTF's method utilizes a rounding table which includes 6 control signals in total that can be simplified. This architecture utlimately utilizes the best parts of [6] and combines with the advancements in [5].

Therefore, it is desirable for a method that is as simple and elegant as SBH's method but also works for all rounding modes as QTF and ES methods. In this section, an improved design is introduced that also based on the special CA as QTF's method but uses a simple formula to build selection logic as SBH's method without using injection bits.

4.2.1 A Simplified Special Compound Adder

First, a `3:1` mux in QTF's special CA is replaced by a smaller `2:1` mux as in Figure 4.7. Given m-bit inputs `A` and `B`, the CA first computes in parallel $Y0 = A + B$ and $Y2 = A + B + 2$. Because the increment is 2, `Y0` and `Y2` are both odd or both even (i.e., $Y0[0] = Y1[0]$). Therefore, the $m - 1$ most-significant bits of both `Y0` and `Y2` are passed through a m-bit mux controlled by `sel1`. The LSB of `Y0` (named `L`) and its inverted value are passed through another 1-bit mux controlled by `sel0`.

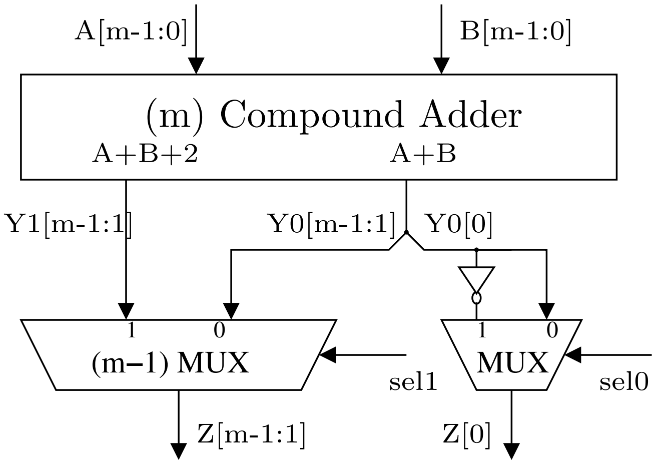


Figure 4.7: The simplified special compound adder (CA)

Table 4.2: Decoder for special $(Sum, Sum + 1, Sum + 2)$ CA

Z	$L = Y0[0]$	`sel1`	`sel0`
Sum	Don't Care	0	0
Sum+2	Don't Care	1	0
Sum+1	0	0	1
Sum+1	1	1	1

The value of `sel1` and `sel0` to select the expected Z is shown in Table 4.2. To select `(Sum+1)`, `Sum` can be increased (when $Y0[0] = 0$) or decreased `(Sum+2)` (when $Y0[0] = 1$). In terms of implementation, in both case, `L` is inverted and selects the corresponding `Sum` or `Sum+2`.

4.2.2 RI Mode

Based on the proposed special CA, a novel hybrid design is introduced that works for RI mode as well as a design that can be extended for RN and RZ modes as shown here:

$$INC = \begin{cases} c + (g \vee t) & \text{if no overflow} \\ c + 2 \cdot (l \vee g \vee t) & \text{if overflow} \end{cases} \tag{4.9}$$

Based solely on this equation, the INC can be 0, 1, 2 or 3. As a result, the implementation should be able to compute four outputs `PH`, `PH+1`, `PH+2` and `PH+3` while the special CA is able to compute only three outputs `Sum`, `Sum+1` and `Sum+2`. However, there is a simple way to solve this problem by using the same prediction bit `p` as with RN mode for SBH's method as illustrated within Claim 7.

Claim 7. In the RI mode, given $p = rs \vee rc$:

$$INC \quad \in \quad \begin{cases} \{0,1,2\} & \text{if p=0} \\ \{1,2,3\} & \text{if p=1} \end{cases} \tag{4.10}$$

Proof. If $p = 0$, means both rs and rc are zeros. Therefore, $c = 0$ and `INC` will be $g \vee t$ or $2 \cdot (l \vee g \vee t)$. In both cases, $INC \in \{0,1,2\}$. If $p = 1$, because `c` and `g` cannot be both 0 so $INC > 0$ in this case. ■

Based on this observation and our proposed special CA, an improved design that works for RI mode is shown in Figure 4.8. Similar to QTF's method, a row of 53 HAs is utilized to add `SH` and `CH` (except the LSBs) and one FA to add the prediction bit `p` and two LSBs of `SH`, `CH`. The sum bit `lp` is used to compute the correct LSB of final product on the right while the carry bit `cp` is injected into the LSB of carry vector `XC` on the left. A 53 bit compound adder is then used to pre-compute two possible outputs `P0`, `P1`. Both `P0` and `P1` are normalized before the final selection logic. On the right side of Figure 4.8, the carry `c`, guard `g`, and sticky `t` bits are computed based on `SL` and `CL` bits. Based on the last bit of `lp` and `c`, `g`, `t` bits and the overflow bit $v_0 = P0[52]$, the **Select Result** module generates `sel1` and `sel0` signals to select the correct output from the CA based on the correct value of `INC` only as shown within Claim 8.

Claim 8. In RI mode, $v_0 = P0[52]$ and `lp` can be used as `v` and `l` in Equation 4.3

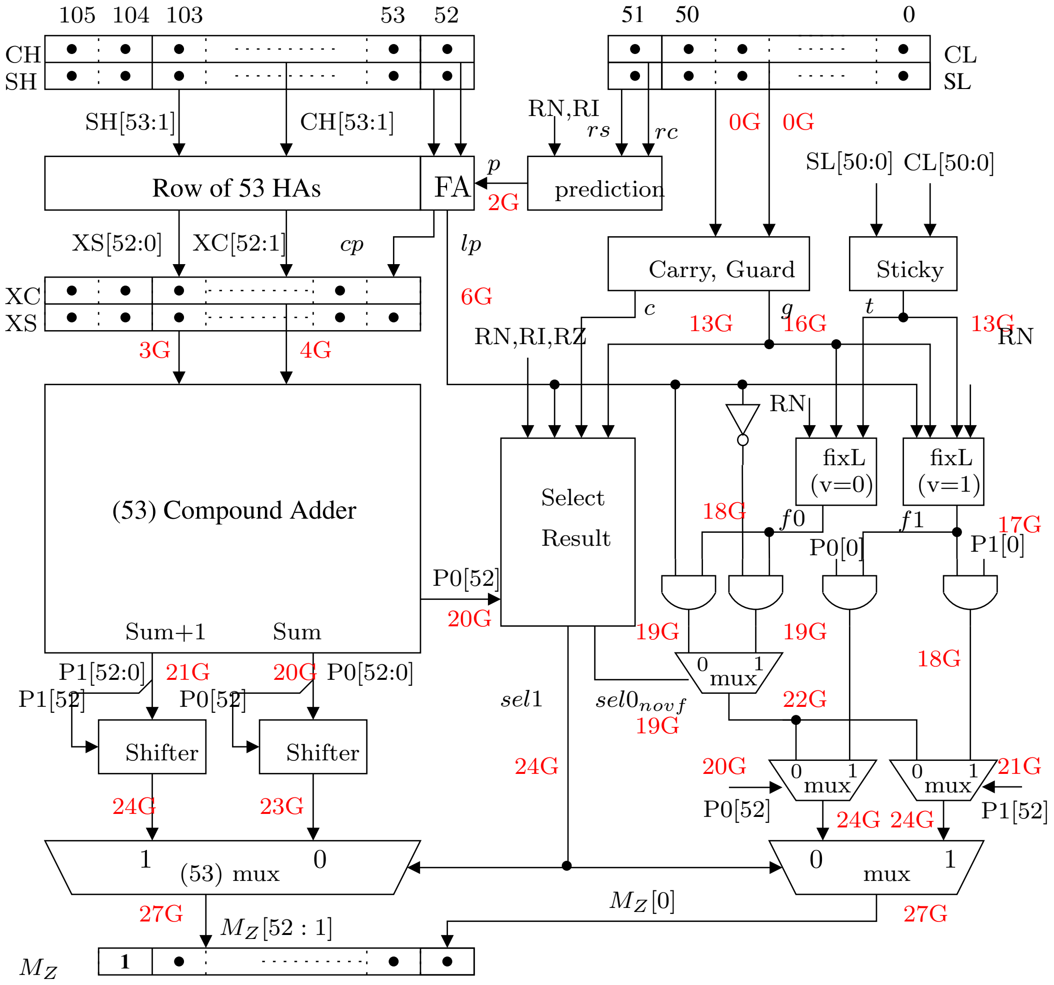


Figure 4.8: Proposed method for all IEEE rounding modes

to select the correct output:

$$INC = \begin{cases} c + (g \vee t) & \text{if } v_0 = 0 \\ c + 2 \cdot (lp \vee g \vee t) & \text{if } v_0 = 1 \end{cases} \qquad (4.11)$$

Proof. Since v_0 (`lp`) is the MSB (LSB) of $PH + p$ while `v` (`l`) is the MSB (LSB) of `PH+c`, `p=c` needs to be checked. Therefore, only $p = 1$ and $c = 0$ should be checked. In this case, $g = 1$ and, therefore, $(lp \vee g \vee t)$ will be the same as $(l \vee g \vee t)$ (both 1). For an overflow, the correct v is the MSB of PH (since $c = 0$) but we falsely selected MSB of `PH+1`. If they are different, this means that the MSB of `PH` is 0 and the MSB of `PH+1` is 1, and `PH` must be all 1 leading by 0. Specifically, it is 01.11...11 with 52

Table 4.3: Generate $sel1$, $sel0$ from INC, p, lp

p	INC	Z	$sel1$	$sel0$
0	0	Sum	0	0
0	1	$Sum+1$	lp	1
0	2	$Sum+2$	1	0
1	1	Sum	0	0
1	2	$Sum+1$	lp	1
1	3	$Sum+2$	1	0

bits of 1 after the radix point. Because $p = 1$ and $c = 0$, the guard bit is $g = 1$, which means the correct product is 10.00..00 and after normalization, the final product is 1.00..00. In this design, `v` is selected as the MSB of (`PH+1`), which is 1. Therefore, the `INC` signal is 2, which means the output selected is `PH+2`. Clearly, with `PH` is 01.11..11, $PH + 2$ is 10.00..01 and after normalization, the final product is 1.00..00, which is exactly the same as the correct value. ■

Given `p,INC, lp` the value of control signals `sel1, sel0` can be found in Table 4.3. The Boolean equations for `sel1,sel0` can be found in Claim 9. Notice that because $v0$ will arrive late, the $sel1$ and $sel0$ are computed for both cases (overflow and no overflow) in parallel and once $v0$ is ready, it will select the correct $sel1$ and $sel0$.

Claim 9. The Boolean equations for `sel1,sel0` are:

$$\begin{aligned} sel0 &= \begin{cases} (g \vee t) \wedge (\overline{p} \vee c) & \text{if } v_0 = 0 \\ \overline{c} \wedge p & \text{if } v_0 = 1 \end{cases} \\ sel1 &= \begin{cases} lp \wedge (g \vee t) \wedge (\overline{p} \vee c) & \text{if } v_0 = 0 \\ lp \vee (g \vee t) \wedge (\overline{p} \vee c) & \text{if } v_0 = 1 \end{cases} \end{aligned} \tag{4.12}$$

Proof. If $v_0 = 0$ (no overflow), then $INC = c + (g \vee t)$:

- From Table 4.3, $sel0 = 1$ when $INC - p = 1$. If $p = 0$, then $c = 0$, so $sel0 = (g \vee t)$. If $p = 1, c = 0$, then $g = 1$, so $sel0 = 0$. Otherwise, if $p = 1, c = 1$, then $sel0 = g \vee t$. In summary, $sel0 = (g \vee t) \wedge (\overline{p} \vee c)$

- From Table 4.3, $sel1 = lp$ when $sel0 = 1$, and $sel1 = 1$ when $INC - p = 2$. However, $INC - p = c + (g \vee t) - p \leq 1$, therefore $sel1 = lp \wedge sel0$.

If $v_0 = 1$ (overflow), then $INC = c + 2 \cdot (lp \vee g \vee t)$:

- Similarly, $sel0 = 1$ iff $INC - p = 1$. This happens only if $c = 0, p = 1$. Therefore, $sel0 = \overline{c} \wedge p$.

- If $lp = 1$ then $INC = c + 2 \geq 2$. Hence, from Table 4.3, $sel1 = 1$. Otherwise, if $lp = 0$, then $INC = c + 2 \cdot (g \vee t)$ and $sel1 = 1$ iff $INC - p = 2$. This happens only if $g \vee t = 1$ and $p = c$. Since if $p = 0$ then $c = 0$, condition $p = c$ is reduced to $\overline{p} \vee c$.

■

Since the LSB has to be fixed only in RN mode, the two signals $f1$ and $f0$ are both 1 for RI mode. In addition, $sel0_{novf}$ is used to select between lp and its invert logic, instead of $sel0$, to save one MUX. This is because if overflow occurs ($v0 = 1$), the later MUX will select P0[0] or P1[0], regardless value of the output of this MUX. All Boolean equations for the control signals in RI mode can be found in Table 4.4.

4.2.3 RN Mode

Since the special CA can output all `Sum`, `Sum+1`, `Sum+2` in parallel, the prediction bit is not necessary in RN mode as for the QTF method. However, to avoid complicated logic in selecting the overflow bit v, as seen in QTF method, we propose that prediction bit $p = rs \vee rc$ should be used for the RN mode similar to the RI mode. With the prediction bit `p`, the correct overflow bit `v` can be safely selected from $v_0 =$`P0[52]`

Table 4.4: Boolean equations for RI mode

Mode	Equations
RI	$p = rs \vee rc$ $sel0_{novf} = (g \vee t) \wedge (\overline{p} \vee c)$ $sel0_{ovf} = \overline{c} \wedge p$ $sel1_{novf} = lp \wedge (g \vee t) \wedge (\overline{p} \vee c)$ $sel1_{ovf} = lp \vee (g \vee t) \wedge (\overline{p} \vee c)$ $f1 = 1$ $f0 = 1$

as proven in previous section. The Boolean equation for RN mode is shown in Claim 10 as follows:

Claim 10. The Boolean equations for `sel1,sel0` are:

$$sel0 \quad = \quad \begin{cases} g \wedge (\overline{p} \vee c) & \text{if } v_0 = 0 \\ \overline{p} \vee c & \text{if } v_0 = 1 \end{cases}$$

$$sel1 \quad = \quad \begin{cases} lp \wedge g \wedge (\overline{p} \vee c) & \text{if } v_0 = 0 \\ lp \wedge (\overline{p} \vee c) & \text{if } v_0 = 1 \end{cases} \tag{4.13}$$

Table 4.5: Boolean equations for RN mode

Mode	Equations
RN	$p = rs \vee rc$ $sel0_{novf} = g \wedge (\overline{p} \vee c)$ $sel0_{ovf} = \overline{p} \vee c$ $sel1_{novf} = lp \wedge g \wedge (\overline{p} \vee c)$ $sel1_{ovf} = lp \wedge (\overline{p} \vee c)$ $f1 = \overline{lp} \vee g \vee t$ $f0 = \overline{g} \vee t$

Proof. If $v_0 = 0$ (no overflow), then $INC = c + g$:

- From Table 4.3, $sel0 = 1$ when $INC - p = 1$. If $p = 0$, then $c = 0$, so $sel0 = g$. If $p = 1, c = 0$, then $g = 1$, so $sel0 = 0$. Otherwise, if $p = 1, c = 1$, then $sel0 = g$. In summary, $sel0 = g \wedge (\overline{p} \vee c)$
- From Table 4.3, $sel1 = lp$ when $sel0 = 1$, and $sel1 = 1$ when $INC - p = 2$. However, $INC - p = c + g - p \leq 1$, therefore $sel1 = lp \wedge sel0$.

If $v_0 = 1$ (overflow), then $INC = c + 1$:

- Similarly, $sel0 = 1$ when $INC - p = 1$. This happens only if $p = c$ (i.e. $\overline{p} \vee c = 1$)
- Since $INC - p = c + 1 - p \leq 1$, $sel1 = 1$ when $sel0 = 1, lp = 1$. Therefore, $sel1 = lp \wedge sel0$ (i.e. $sel1 = lp \wedge (\overline{p} \vee c)$.

■

Similar to ES's method, additional logic is utilized to fix the LSB in the tie-case of RN mode. However, since no injection bits are added, the equations for $f1$ and $f0$ are simpler than ES's method.

$$\begin{aligned} f0 &= \overline{g} \vee t \\ f1 &= \overline{lp} \vee g \vee t \end{aligned} \tag{4.14}$$

Interestingly, `lp` is utilized instead of `l` in the formula for $f0$. Similar to Claim 8, if $p = c$ then $lp = l$, however, if $p = 1$, $c = 0$, `lp` and `l` are different. Fortunately, since $g = 1$ the result will not change. All Boolean equations for control signals in RN mode can be found in Table 4.5.

4.2.4 RZ Mode

In RZ mode, because no rounding bits are added, $INC = c$. Since no prediction bit is pre-added, the `INC` correctly select the CA output. Given `INC` and `lp`, two select bits `sel1`, `sel0` can be easily computed similarly to RI and RN modes. In addition, similar to RI mode, in RZ mode, `f1` and `f0` are all asserted (i.e., 1). The Boolean equations for RZ mode is shown in Table 4.6.

4.2.5 Linear Delay Analysis

The delay annotation is shown in Figure 4.8. In the left side, the prediction logic will add 2 gate delay to the row of HA and the FA will add 2 more gate delay before the compound adder. Therefore, the Sum and Sum+1 outputs of compound adder will be available after 20 and 21 gate delay respectively. Control `sel1` is valid after 24 gate delay. As a result, the final $M_Z[52:1]$ is valid after 27 gate delay.

In the right side, since the Carry `c`, Guard `g` and Sticky `t` are valid after $13, 16, 13$ gate delay respectively, two control logic `f0, f1` are valid after 18 and 17 gate delay respectively. The row of AND gates adds 1 more gate delay. Because $sel0_{novf}$ is valid after 19 gate delay, `Z[0]` is valid after 27 gate delay. Combined with the left side, the critical path of proposed rounding design is 26 logic levels, given the input Carry/Sum. As seen in Table 4.7, comparing to ES and QTF methods, the proposed method is 3 and 4 gate delay faster respectively.

Table 4.6: Boolean equations for RZ mode

Mode	Equations
RZ	$p = 0$ $sel0 = c$ $sel1 = lp \wedge c$ $f1 = 1$ $f0 = 1$

Table 4.7: Theoretical delay (logic levels) comparison

Method	Figure	Delay
QTF	4.4	31
ES	4.6	30
Proposed	4.8	27

4.3 Chapter Summary

In this chapter, we propose a clarification and optimization on rounding for IEEE 754-compliant floating-point multiplication. SBH's method is first clarified and provied that it is not applicable for RI mode. Then, QTF's method is summarized and clarified on how it solved the limitations of SBH's method using a special CA. However, QTF's method is based on a rounding table and its design is not optimized and produces inaccurate results. It uses 6 control signals and a 3:1 mux for the LSB and a 4:1 mux for the final result selection. Based on a formula similar to the ES method, a hybrid QTF/ES design is presented that uses only 2 control signals and a standard 2:1mux. The experimental results illustrate an efficient, fast and low-power implementation using a 32nm library.

CHAPTER 5

AN OPTIMIZED IEEE MULTIPLIER SUPPORTING DENORMAL NUMBERS

Denormal numbers fill the gap between the smallest normal and zero by allowing numbers with reduced precision. As the value decreases so does the precision in what is called gradual underflow [22, 42]. In several applications graudal underflow can preserve expected program behavior, where as truncation maybe erratic. No situation better demonstrates the value of denormals than the difference of two small numbers A and B. If A and B are nearly equal but different, $A - B$ may result in a number smaller than the minimum normal. Without gradual underflow $A - B$ may yield 0 which misleads applications (for example $if((A - B) == 0)\{...\}$). The same applies to other floating point operations. Consequently, there is a need for hardware which handles both normal and denormal numbers [17].

5.1 A Simple Design for Denormal Numbers

To begin, this dissertation builds a straightforward adaptation of the presented architectures as a means to explain denormal support. Then a second design dramatically improves performance by coalescing several shifters [38]. With the first approach, support of denormals is achieved by converting denormals into normals in a process known as unpacking. Unpacking explicitly defines the value of the hidden bit and normalizes the mantissa prior to multiplication. This simplifies the rounding logic by ensuring the rounding position is located in one or two digits. While the normalization adds considerably to the critical path, the second design optimizes the

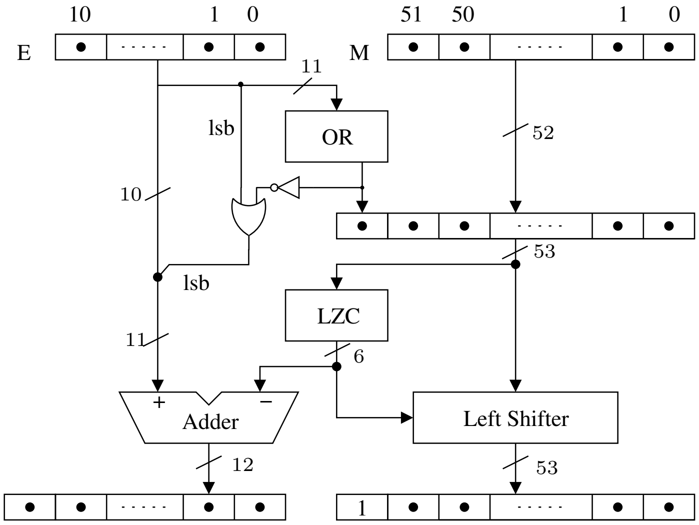


Figure 5.1: Converts an IEEE 754 double precision number into the normalized representation.

delay.

Figure 5.1 shows the hardware needed to generate the hidden bit and normalize the mantissa. Computing the hidden bit depends entirely on the value of the exponent. If the exponent is 0 then the number is denormal and the hidden bit is 0. Otherwise the hidden bit is 1. This can be implemented by OR-ing all the exponent bits and then inserting the result bit to the bit 52 of mantissa. In addition, the exponent must be adjusted when it is a denormal as the numerical value of the exponent is 1 rather than zero. This is implemented by a simple AND to set the LSB of the exponent when the output of the OR logic is 0 (denormal).

After unpacking, a leading-zero counter (LZC) finds the number of prefix zeros in the mantissa. This number then adjusts the exponent with an extended range to produce 12-bit exponents. The exponent is now in the range −53 to 2046 with an bias of 1023. In parallel, a left shifter normalizes the mantissa. The LZC uses the design from [43], while the shifter is a standard left barrel shifter. Converting denormal to normal format introduces a large delay as both the LZC and barrel shifter are in the critical path and both take around 6 levels of logic each.

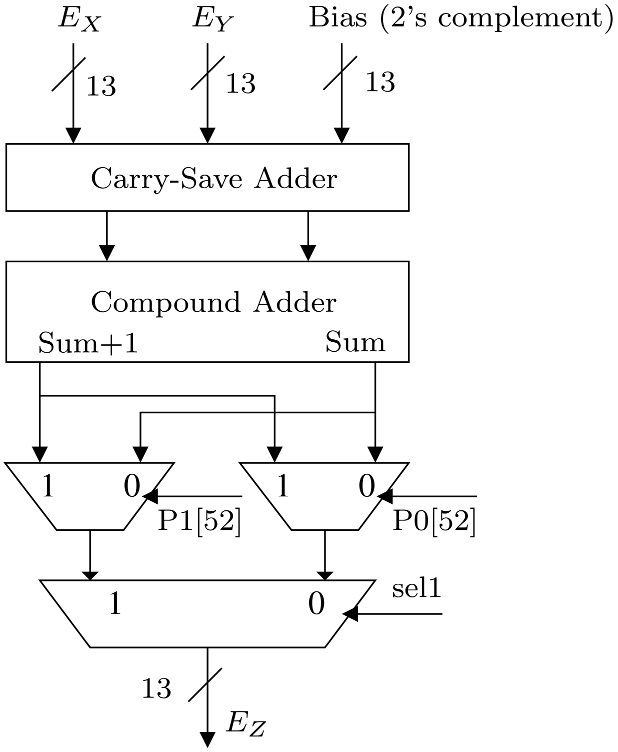


Figure 5.2: Exponent addition and adjustment.

5.1.1 Exponent addition and adjustment

Figure 5.2 shows an optimized implementation of bias exponent and adjustment. To compute the output exponent a bias (1023) is subtracted from the sum of the two exponents, E_X and E_Y. However, if overflow occurs in mantissa multiplication, the result exponent is increased by 1. An efficient solution uses a compound adder to compute in parallel $E_X + E_Y - bias$ and $E_X + E_Y - bias + 1$. Two muxes are then used to select the right output. The first mux will select the right overflow bit from $P0[52]$ and $P1[52]$ using $sel1$ control signal (see Figure 4.8). This overflow bit will ultimately indicate if the exponent should be increased by 1.

It is important to note that the format of the exponent values as this dictates further computations of the mantissa. As explained previously, the exponent is in the range of $[-53, 2046]$, and when two are added and the bias of 1023 subtracted the range increases to $[-1129, 3069]$. Therefore, two input exponents and bias are all sign extended to a 13 bit two's complement number.

5.1.2 Rounding and packing

Given the normalized mantissas after unpacking, all three rounding schemes can be used for the mantissa multiplication. However, it is worth noticing that the result exponent E_Z can be less than $e_{min} = 1 - bias$ (0 in biased representation), even in the case both inputs are normal numbers. In this case, the output is no longer a valid IEEE 754 number.

To correct this problem, the mantissa is right shifted so that the exponent is increased to e_{min} . The shift amount needs to computed by finding how far the exponent is below the minimum value e_{min}. This is accomplished by computing $shift = min(max(0, -E_Z + 1), 54)$. The max function finds the exact amount below e_{min}, which is $(1 - bias) - (E_Z - bias) = -E_Z + 1$. However, it is possible to produce values larger than the number of bits in the mantissa (53). The min function limits the shift to no more than 54-bit.

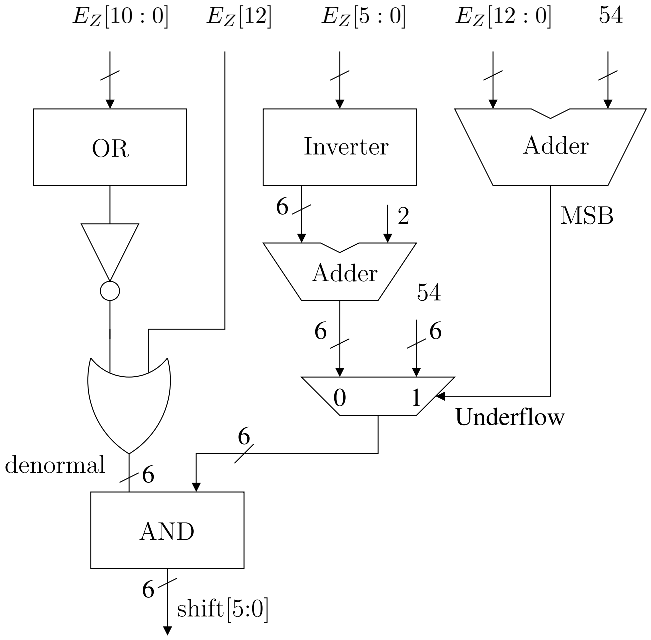


Figure 5.3: Compute shift amount to denormalize the mantissa if exponent < emin

Figure 5.3 shows the hardware used to generate the shift amount along with adjusting the exponent. The denormal logic detects when the exponent is below e_{min} by asserting a high value if the MSB are 1 or every bit (i.e., `[10:0]`) is zero. The lower 6-bit of the exponent are then inverted and incremented by 2, effectively computing the two's complement operation and increasing it by 1. This value is the inverted shift amount when the resulting number is denormal. Finally the bottom path computes the extreme underflow $E_Z < -53$; when this value is high the multiplexor limits the shift to 54.

Given the shift amount, a barrel shifter is used to right shift the mantissa. Interestingly, an additional rounding step must be added to round the shifted mantissa to a 52-bit standard mantissa [20]. This ultimately requires an additional sticky bit logic and an extra +1 adder.

5.2 Proposed Multipliers

It can be seen that the simple design for denormal numbers using multistep gradual rounding as described above is not optimal. It costs a high price in terms of delay and area as two shifting stages are required (left shifting in unpacking and right shifting in packing), the sticky bit must be computed twice and an extra +1 adder is needed after rounding.

The first optimization can be made by moving the right shifter from the packing circuit and inserting it between the partial production reduction and rounding logic [20, 38]. Two right shifters are needed, one for the carry vector and the other for sum vector. Shifting the carry/sum vectors effectively denormalizes the mantissa in the event of an exponent below e_{min} and aligns the carry/sum vectors to the correct rounding position. This will remove the need for an additional re-rounding step as in previous method.

Because the shift now occurs prior to overflow detection (i.e., the mantissa can be

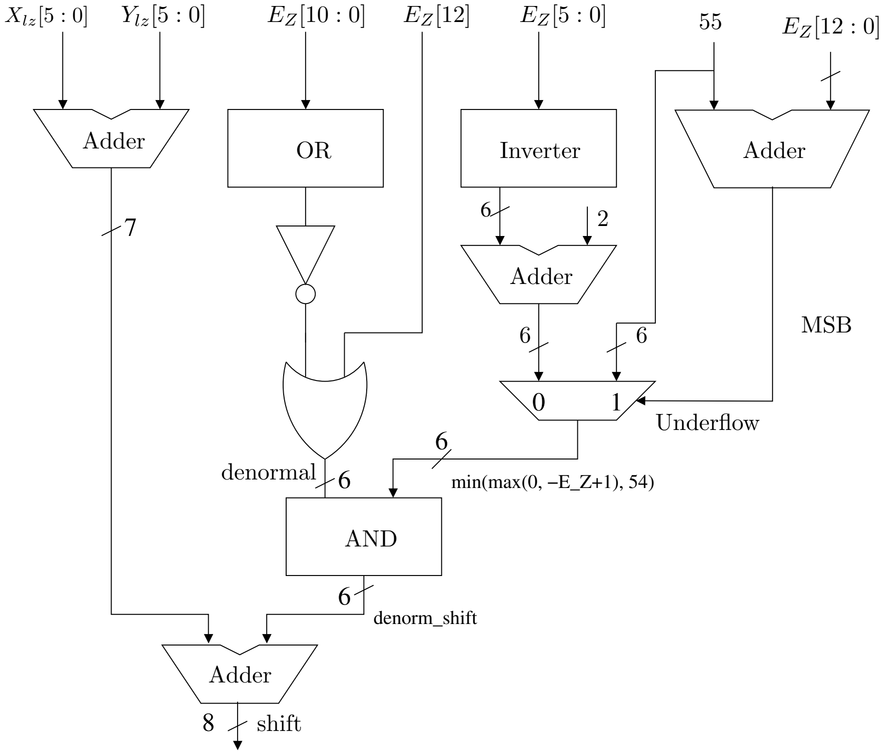


Figure 5.4: Shift is computed by first constraining the exponent, E_Z to 0 to -55 and inverting the result

in the $[2, 4)$ interval) the shifter will need a maximum shift of 55 instead of 54 as in the simple design. Therefore, the shift amount needs to be modified to handle the increased range (i.e., $shift = min(max(0, -E_Z + 1), 55))$). It is important to note that the trailing bits shifted right must be preserved to compute the sticky bit, which necessitates increasing the shifter width from 106 bit to 161 bit (+55).

An additional optimization can be made to remove two left shifters in the unpacking logic [38]. In the simple design, the mantissas need to be normalized (left shifted) before rounding to ensure the rounding position was predictable. However, these two left shifters can be removed if the carry-save right shifters are modified to shift in both directions. Doing so dramatically reduces the unpacking delay by eliminating a shifter and removing the leading-zero detection from the critical path.

Computing the shift amount must now account for the sum of the leading zeros detected in both operand mantissas. This should be subtracted from the shift amount such that

$$\begin{aligned} shift\texttt{[7:0]} &= \min(\max(-E_Z + 1, 0), 55) - \\ &\quad (X_{lz} + Y_{lz}) \ , \end{aligned} \tag{5.1}$$

where X_{lz} and Y_{lz} are leading-zero counts for the unpacking logic of X and Y, respectively. Figure 5.4 shows the required changes to the shift amount computation.

Figure 5.5 shows the mantissa portion of the block diagram. It is worth to note that, although the mantissa outputs from unpacking logic are no longer a normalized mantissa, no modification to the rounding algorithm is required as the left/shift shifters will ensure the carry-save intermediate products are right aligned in both normal and denormal cases. It can be seen that with the optimized design, only a small change in the result selection logic (in bit-width) is needed. This is also the reason the proposed method is preferred to the injection based rounding since the injection rounding requires a wide additional carry-save adder to add the injection amount.

5.3 Linear Delay Analysis

Since the mantissa computation is always on the critical path, its estimated linear delay is provided in this section. For the sake of simplicity, the `Sum` and `Carry` vectors are assumed arriving at 0 gate delay. In addition, since `shift[7:0]` should arrive sooner than `Sum/Carry`, it can be assumed valid at 0 gate delay, too.

The delay estimation of the proposed design is shown in Figure 5.5. The left/right shifter is simply a right barrel shifter with 6 logic levels (each logic level is a 2-1 mux) with 2 more mux level for left shift. As a result, the shifted `CH,SH` vectors are valid after 24 gate delay. Given `SH, CH`, the rounding logic is the same as the proposed

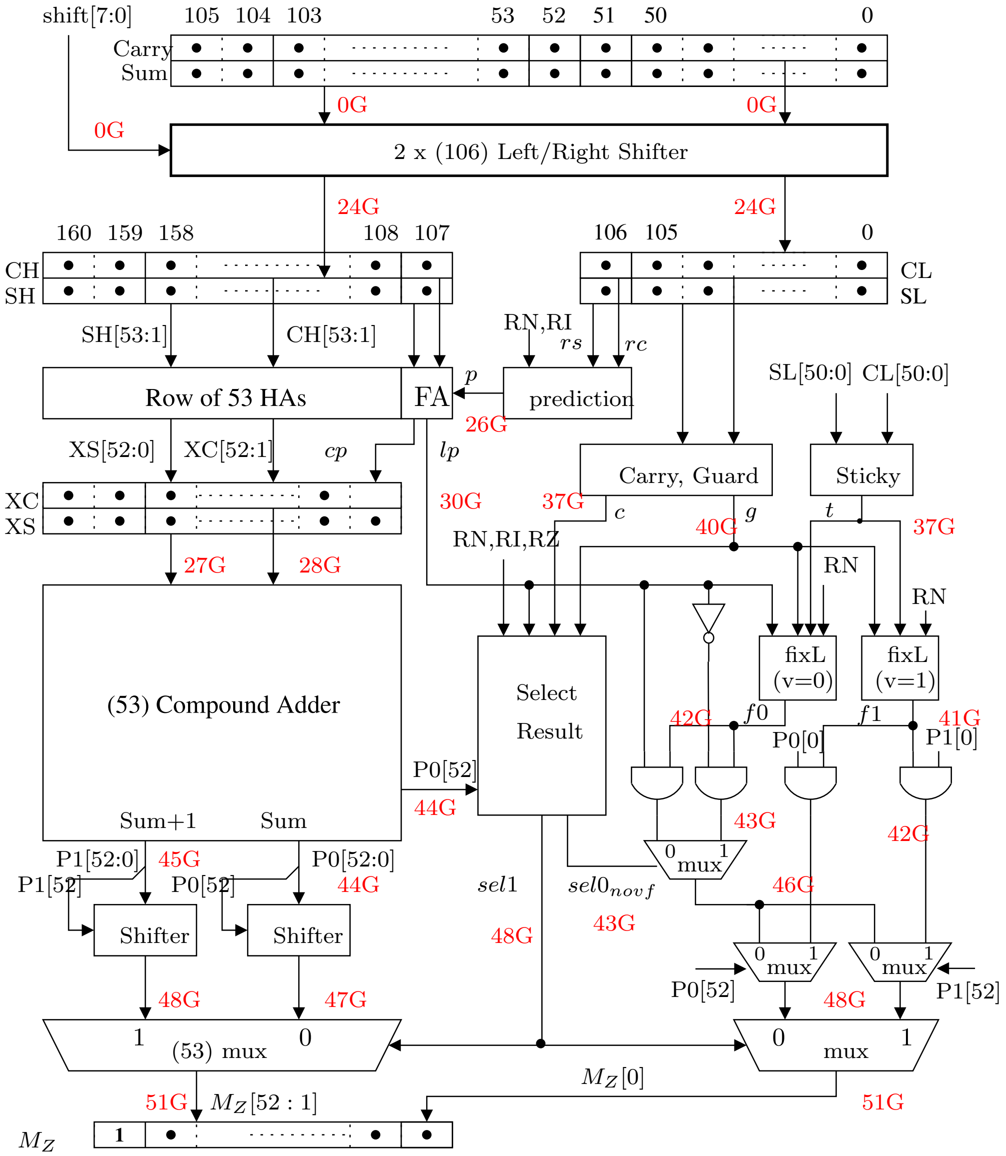


Figure 5.5: Block diagram of the mantissa path when shifting the carry save vectors before rounding

rounding scheme in previous section that takes 27 gate delay. Therefore, the total delay of the proposed design is 51 gate delay.

5.4 Chapter Summary

In summary, correct rounding of both normal and denormal results further exacerbates the growing complexity of an IEEE 754 multiplier. Due to the importance of high precision in scientific applications [12], the precision must be preserved. Simply truncating denormal results to zero is unacceptable [13]. Consequently, having floating-point units that can handle normalized and denormalized numbers is essential, especially for scientific computing [14]. Recently there have been several types of hardware implementations that handle floating-point denormalized IEEE-754 numbers [15, 16, 17]. This chapter discusses methods of implementing both normalized and denormalized IEEE 754 numbers [2]. In particular, it combines the pre-normalizing and post-normalizing steps in existing methods into a simpler and faster single step.

CHAPTER 6

A HYBRID IEEE PRECISION MULTIPLIERS SUPPORTING DENORMAL NUMBERS

In this chapter, we extend the multiplier to handle half-word operations within the IEEE-754 2008 standard with small overhead logic to perform the IEEE-compliant half and single precision multiplications [38]. Single precision floating-point is the default format for many deep learning frameworks while half-precision floating-point multipliers are new additions to the 754 standard [2] that are specifically useful for architectures that use machine learning computations. By utilizing smaller amounts of precision, these multipliers can speed up computations for designs that are well suited for neural networks and machine-learning applications [21]. Moreover, extensions are added to the multiplier to also handle denormalized IEEE 754 floating-point numbers as specified in previous chapter. Finally, the multiplier architecture is also updated to handle rounding within the IEEE 754-2008 standard [2].

The proposed hybrid precision multiplier is designed to be easily configured to run in different precision modes using an additional control signal `fm[2:0]`. When running in binary64 mode, inputs A and B and output Z are normal binary64 numbers. When running in binary32 or binary16 modes, the first half the input operands of A, B and Z are binary32/binary16 numbers or word-aligned to the input operand. As described in previous section, the sign computation is exactly the same for all operations. Therefore, only exponent addition and mantissa multipliers need to be modified to adapt to binary32 and binary16 formats.

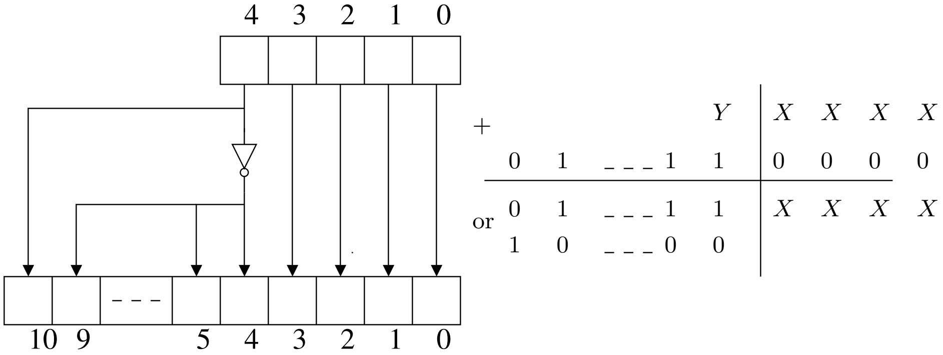


Figure 6.1: Converting binary16 exponent (5-bits) to binary64 exponent (11-bits)

6.1 Exponent addition

In binary64 mode, the exponent is 11-bit and the bias value is 1023. However, in binary32 mode, the exponents are 8-bit width and the bias value is 127 and in binary 16 mode, the exponents are 5-bit and the bias value is 15. Using a simple conversion technique, a small modification can be applied to the 11-bit exponent addition to easily perform 8 and 5-bit exponent addition [38].

Assuming E_{F16}, E_{F32} and E_{F64} be the representation of the exponent E_F in the half precision, single precision and double precision formats respectively.

$$\begin{aligned} E_F &= E_{64} - (2^{(10-1)} - 1) \\ &= E_{F32} - (2^{(8-1)} - 1) = E_{F16} - (2^{(5-1)} - 1) \ . \end{aligned} \tag{6.1}$$

As a result, the half-precision or single-precision exponents can be easily converted to double precision exponent by figuring out the difference in their bias as following:

$$\begin{aligned} E_{F64} &= E_{F16} - 15 + 1023 = E_{F16} + 1008 \\ E_{F64} &= E_{F32} - 127 + 1023 = E_{F32} + 896 \ . \end{aligned} \tag{6.2}$$

To implement these equation, a simple 11-bit adder can be used to add 1008 to the half precision exponent. However, there is a better solution. The difference of

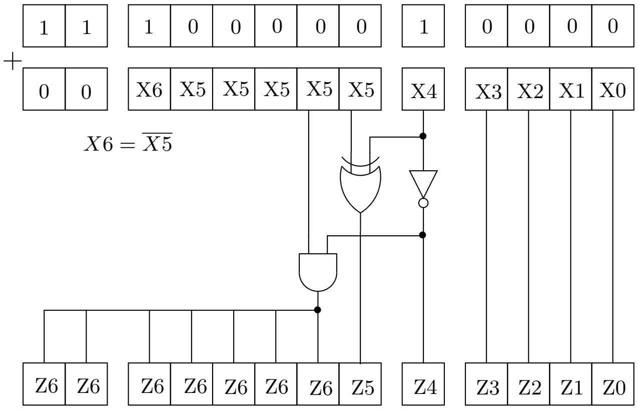


Figure 6.2: Converting sign-extended binary64 exponent to sign-extended binary16 exponent

1008 ($011\_1111\_0000_2$ in binary representation) can be adjusted in logic as in Figure 6.1. It can be seen that the last 4 significant bits of E_{F64} are the same as the last 4 significant bits of E_{F16}. The most 7 significant bits of E_{F64} can be either 011_1111 or 100_0000 when the most significant bit Y of E_{F16} is 0 or 1 correspondingly. This is implemented by simply using an inverter gate as seen in the Figure 6.1 and previously presented in Chapter 3.

The same technique can be used for binary32 exponent numbers, except the difference is 896 ($011\_1000\_0000_2$ in binary) instead of 1008. It is easy to see the last 7 significant bits of E_{F64} are the same as last 7 significant bits of E_{F32}. The most 4 significant bits of E_{F64} can be either 0111 or 1000 when the most significant bit Y of E_{F32} is 0 or 1 correspondingly. It can be implemented by simply using an inverter gate to invert the MSB of E_{F32} to 3-bit `[9:7]` of E_{F63}`[10:0]`.

Subsequently, the 13-bit output exponent $E64_Z$ has to be converted back to its right format E_Z. Because the result exponent is sign extended to 13-bit and can be negative, a simply conversion that take the MSB of $E64_Z$ and the LSBs as in the forward-converters does not work. The naive implementation use an adder to add the result exponent with the 13-bit 2's complement of the difference in exponent (1008 in binary16 and 896 in binary32). However, here we propose a better way as shown in Figure 6.2. Given a 5-bits binary16 exponent, when converting it to 11-bits binary64

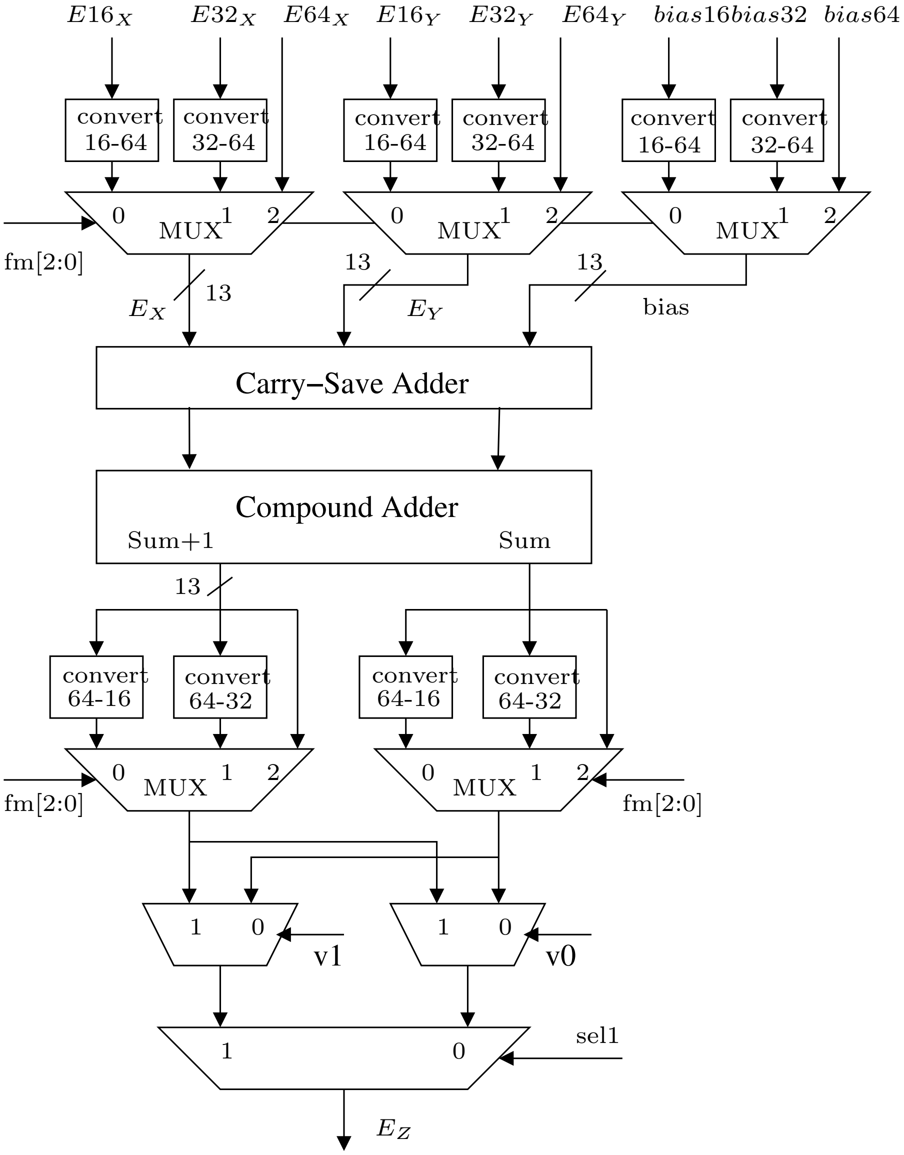


Figure 6.3: A combined IEEE precision exponent addition

format, its range is from 011_1110_0000 to 100_0001_1111. In other word, its format will be:

$$X_6X_5X_5X_5X_5X_5X_4X_3X_2X_1X_0 \text{ with } X_6 = \overline{X_5}.$$

Adding negative two's complement of the difference in exponent (e.g. 1008 in binary16) can be implemented by using one inverter, one XOR, and one AND gate as in Figure 6.2. Using the converters as described, the original binary64 exponent addition can be slightly modified to adapt for binary16 and binary32 exponent as

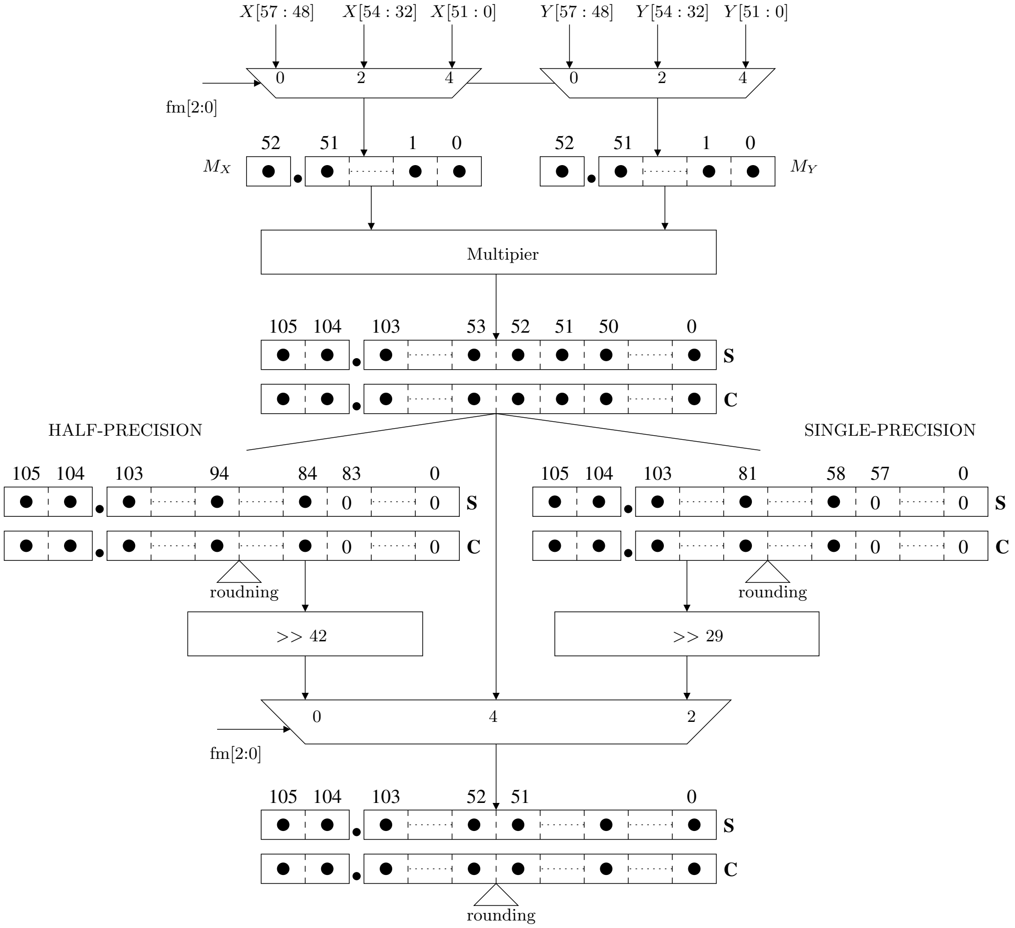


Figure 6.4: Aligning rounding position for half precision and single precision modes

in Figure 6.3. $E16_{\texttt{X[5:0]}}$, $E32_{\texttt{X[7:0]}}$ and $E64_{\texttt{X[10:0]}}$ are unpacked exponents of binary16, binary32 and binary64 numbers, respectively. First, $E16_X$ and $E32_X$ are converted to binary64 exponents using techniques above. The control signal `fm` will then select the right exponent format to feed into the biased addition. Similar logic is applied to $E16_Y, E32_Y$ and $E64_Y$ exponents as well as the bias selection. In order to handle the possibility of overflow within the exponent, the bias numbers utilize a 13-bit two's complement representation.

6.2 Mantissa multiplication

Since the mantissa is left-aligned, the only difference between precision modes are the rounding position. In particular, given the `Sum[105:0]` and `Carry[105:0]` shown in Figure 5.5, the rounding position for binary64 is at position 52, position 81 for binary32 and 94 for binary16. Based on this observation, to minimize the overhead, the binary16 and binary32 mantissas should be right shifted such that they are aligned with the binary64 mantissa at the same rounding position (52 position). The shift amount for binary16 mantissa is 42 while the shift amount for binary32 mantissa is 29 [38].

In the original binary64 design, the `shift[7:0]` value is used to left/right the mantissas to the correct rounding position. Therefore, it is optimal to adjust this value to the binary16 and binary32 modes as in Figure 6.5. In comparison with the original design in Figure 5.4, the proposed design introduces a mux to select the correct rounding difference with binary64 (42 for binary16 and 29 for binary32) and adds this value to the final shift amount. In addition, another mux is also inserted to account for the difference in the maximum length to be considered underflow (55 for binary64, 26 for binary32 and 13 for binary16).

Given the same rounding position, the compound adders and select result logic are exactly the same for all modes. However, because the mantissa is righted shifted to align with binary64 mantissa, the overflow bits is also moved along. The overflow bit for binary16 now is $v0 = P0[10]$ and $v1 = P1[10]$ while the overflow bit for binary32 is $v0 = P0[23]$ and $v1 = P1[23]$. Similarly, a simple mux can be used to select the right overflow bit position for each mode.

Finally, the final result must be left-aligned to pack into the right format. If binary16 mode, the mantissa needs to be shifted 42-bit to the left while in binary32 mode, the shift amount is 29. The control signal fm will select the final left-aligned mantissa to pack.

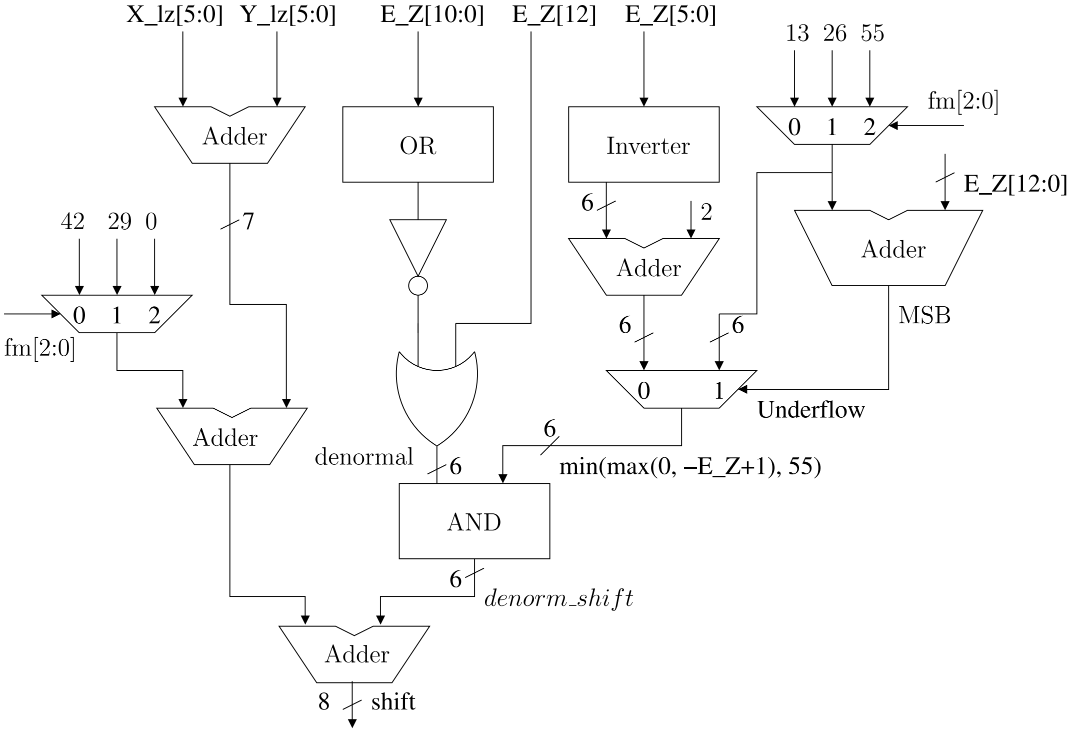


Figure 6.5: A computing shift amount for half/single/double-precision design, taking into account the difference between rounding positions

6.3 Linear Delay Analysis

Since the mantissa computation is always on the critical path, its estimated linear delay is provided in this section. For the sake of simplicity, the `Sum` and `Carry` vectors are assumed arriving at $0G$. In addition, since `shift[7:0]` should arrive sooner than `Sum/Carry`, it can be assumed valid at $0G$,too.

The left/right shifter is simply a barrel shifter similar to the denormal design that takes 24 gate delay. Since the overflow bit `v0,v1` depending on the control signal `fm`, they are valid after 47 and 48 respectively. Finally, the result should be left shifted, which is nothing but a mux, if in binary32/binary16 modes. Therefore, the total delay of the proposed design is 54 gate delay as shown in Figure 6.6.

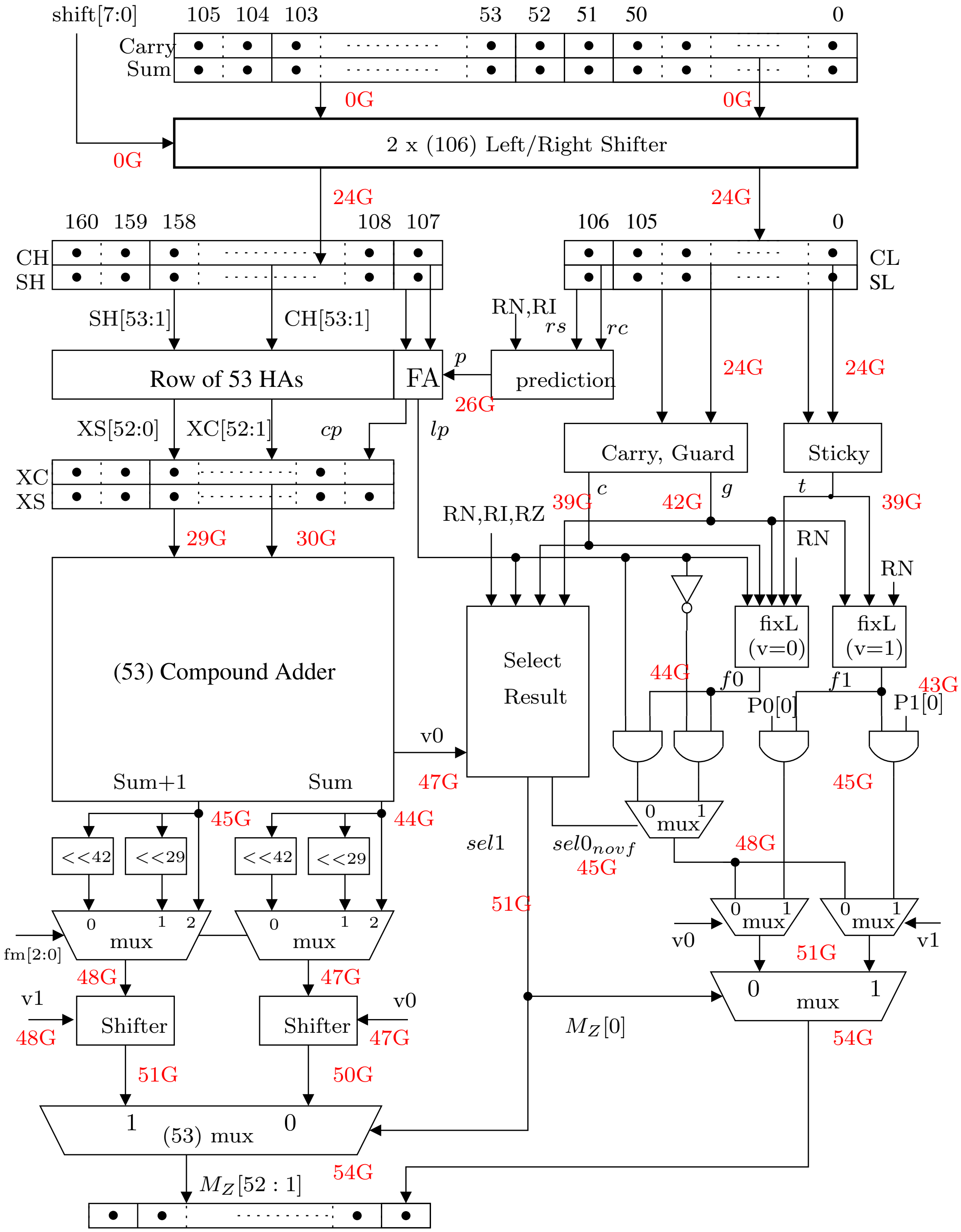


Figure 6.6: A combined IEEE precision mantissa multiplication

6.4 Chapter Summary

In summary, this chapter is an extension to [19] by demonstrating an IEEE 754 compliant floating-point multiplier that can handle half, single, and double precision operations. The previous implementations [20, 19] only demonstrated new methods for single and double precision operations, however, this design extends the ideas by specifically adapting the architecture for half-precision IEEE 754 multiplication. Half-precision floating-point multipliers are new additions to the 754 standard [2] that are specifically useful for architectures that use machine learning computations. By utilizing smaller amounts of precision, these multipliers can speed up computations for designs that are well suited for neural networks and machine-learning applications [21]. Moreover, extensions are added to the multiplier to also handle denormalized IEEE 754 floating-point numbers as well as half and single-precision floating-point numbers [2].

CHAPTER 7

EXPERIMENTAL RESULTS

This chapter first introduces briefly about the design methodology including Application Specific Integrated Circuits (ASIC) design flow, Register Transfer Level (RTL) coding, verification, and topographical synthesis in Section 7.1. The experimental results for each proposed design are discussed in details and compared with existing designs in Section 7.2.

7.1 Methodology

All designs in this dissertation are implemented using the general design flow shown in Figure 7.1 [7]. Design starts with the product requirement that will be translated into behavioral/functional specification, then proceeds to the structural level (gates and registers). This step is called Register Transfer Level (RTL) synthesis since the designs are captured at the memory and logic level in an Hardware Description Language (HDL). The HDL is then transformed to a physical description that are ready for the chip fabrication (physical synthesis or layout generation). Generally, the synthesis steps (both logic and physical synthesis) are automated.

In Figure 7.1, design steps has been divided into the front end stage (behavioral level) and back end stage (structural and physical levels). This partition is used to build Application Specific Integrated Circuits (ASICs). In an ASIC flow, the design can be developed at the RTL-level and then passed to a different team that completes the rest of the flow to build an actual chip. In other words, only a behavioral HDL needs to be designed and simulated at the behavioral level. In this dissertation, all

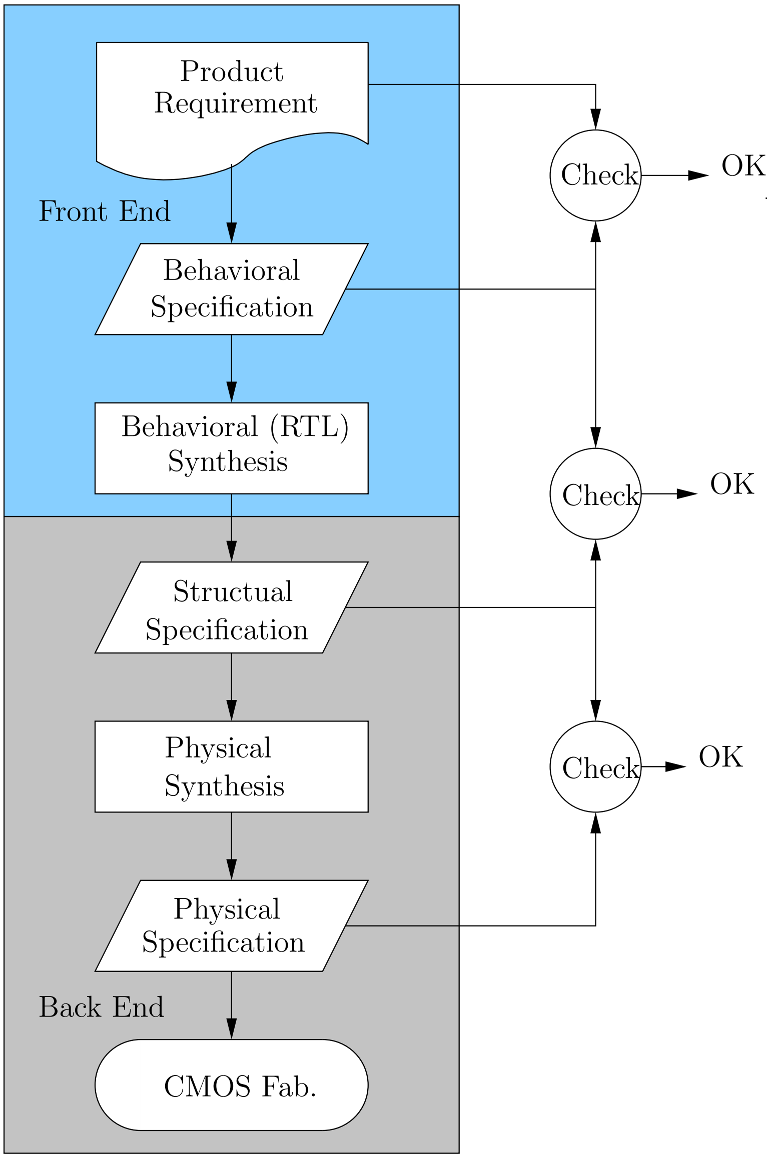


Figure 7.1: Generalized design flow (Adopted from [7])

designs are implemented and simulated using an ASIC design flow.

7.1.1 ASIC Design Flow

The behavioral synthesis normally transform a behavioral RTL description to a structural gate-level netlist. A typical behavioral synthesis design flow for an ASIC (shortly ASIC design flow) is shown in Figure 7.2 [7]. RTL-level descriptions are typically better for synthesis in that they are easier for flows to translate their final netlist into realizable hardware.

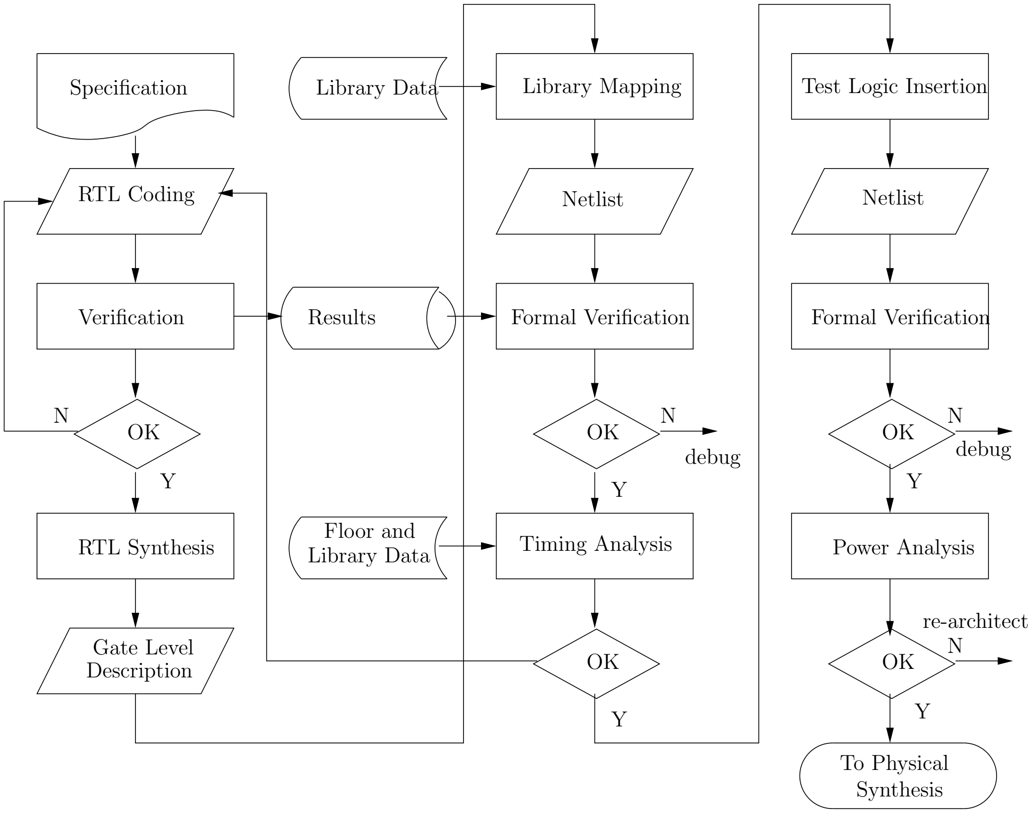


Figure 7.2: Typical ASIC design flow (Adopted from [7])

7.1.2 Logic Design and Hardware Description Languages

The flow starts with a specification, in this dissertation the IEEE 754-2008 floating-point standard and multiplication algorithm descriptions, which is described in details in previous chapters. The next step is to translate the algorithms into a circuit. Basically, translating algorithms into circuit schematics is a time-consuming and prone-to-error task, while designers normally require quick feedback on whether a logic design is reasonable. HDLs provide an effective way to specify or describe the design at a high level of abstraction to increase the productivity and accuracy. Verilog, SystemVerilog, and VHDL are the most popular form of HDLs in use today.

In this dissertation, all proposed designs are implemented in RTL-compliant Verilog. In addition, although designs are described in double precision format, all implementations in this dissertation are developed using parameters to work on all IEEE 754-2008 formats.

7.1.3 Design Verification

Before a design can be synthesized, it has to be verified for correctness. Verification is an very important task of design flow and requires designing good test cases using theories related to Boolean simplification and Design for Testability [3].

Floating-point has long-been hampered by verification as it is difficult to verify completely. Exhausted test (or brute force test) is really a mission impossible for most of the IEEE 754-2008 floating-point formats. For example, one 32-bit single precision floating-point number (binary32) requires 2^{32} or 4 billion test vectors. For multiplication, since there are two operands, it requires 2^{64} test vectors to cover all possible inputs, which is practically impossible to implement. As a result, several methods have been proposed along with Bill Kahan's popular 1980s paranoia test. John Hauser, one of Bill Kahan's student, made a great new version including all floating-point formats called SoftFloat/TestFloat [37].

In this dissertation, to verify the correctness of proposed designs, we first generate random test vectors using a modified form of Hauser's TestFloat program [37]. A set of SystemVerilog testbenches are then developed to simulate designs (in Verilog code) using ModelSim to verify the correctness based on comparing the actual outputs of designs and the outputs of TestFloat-3c given the same input test vectors.

7.1.4 Topographical Synthesis

The modern integrated circuits is characterized by small feature sizes and complicated layout. In old design flows without topographical synthesis, the wire loads are used to characterize delays between gates. Unfortunately, wire loads model (WLM) are simplistic and only use simple first-order approximation techniques [44, 45]. However, with sub-micron technology, wires delay becomes dominant and cannot be ignored, especially below 180nm feature sizes. Therefore, to get a more accurate delay, area, and power, this work incorporate underlying information from layout through Milkyway™ database generation and topographical synthesis through the Design Compiler™ (DC) programs from Synopsys®(SNPS).

In particular, the proposed designs are implemented in RTL-compliant Verilog and then synthesized in an ARM 32nm CMOS library in Global Foundries (GF) cmos32soi technology optimizing on delay. 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® 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.

7.1.5 Power Analysis

Since power dissipation is a key factor in the circuit design, power analysis is critical in evaluating any implementations. Power dissipation (P_{total}) in CMOS circuits comes from two components [7]:

- Dynamic dissipation ($P_{dynamic}$) due to:
  - Charging and discharging load capacitance as gates switch (P_{switch})
  - "short-circuit" current while both pMOS and nMOS stacks are partially ON ($P_{short\_circuit}$)
- Static dissipation (P_{static}) due to:
  - leakage (subthreshold leakage, gate leakage, and junction leakage) ($P_{leakage}$)
  - contention current in ratioed circuits ($P_{contention}$)

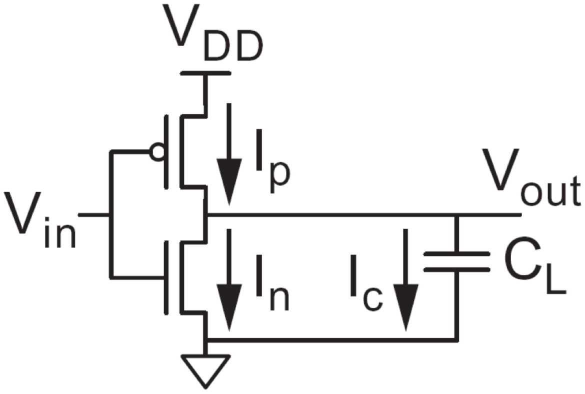


Figure 7.3: CMOS Inverter (Adopted from [7])

Putting this together gives the total power dissipation of a circuit:

$$\begin{aligned} P_{total} &= P_{dynamic} + P_{static} \\ &= P_{switching} + P_{short\_circuit} + P_{leakage} + P_{contention} \end{aligned}$$

For example, Figure 7.3 shows a CMOS inverter driving a load capacitance. When the input switches from 1 to 0, the pMOS transistor turns ON and charges the load to VDD. When the input switches from 0 back to 1, the pMOS transistor turns OFF and the nMOS transistor turns ON, discharging the capacitor. Suppose that the gate switches at some average frequency f_{sw}. Over some interval T, the load will be charged and discharged $T \cdot f_{sw}$ times. As a result, the average power dissipation $P_{switching}$ is

$$P_{switching} = C \cdot V_{DD}^2 \cdot f_{sw}$$

Because most gates do not switch every clock cycle, it is often more convenient to express switching frequency f_{sw} as an activity factor α times the clock frequency f. Now, the dynamic power dissipation may be rewritten as

$$P_{switching} = \alpha \cdot C \cdot V_{DD}^2 \cdot f$$

For a long time, switching power was long utilized as a justification for smaller feature sizes as the power reduces quadratically with voltage scaling [7]. That is, Denard scaling typically reduces the voltage as a result of smaller features sizes. The activity factor is the probability that the circuit node transitions from 0 to 1, because that is the time the circuit consumes power. Dynamic power also includes a short-circuit power component ($P_{short\_circuit}$) caused by power rushing from VDD to GND when both the pullup and pulldown networks are partially ON while a transistor switches [7].

Static power is consumed even when a chip is not switching. Prior to the 90nm process, leakage power was of concern primarily during sleep mode because it was small compared to dynamic power. However, in nanometer processes with low threshold voltages and thin gate-oxides, leakage can account for as much as 1/3 of total active power [7, 46].

In this dissertation, the average power estimation is achieved by running the simulation with over $50,000$ random test vectors utilizing an annotated Value Change Dump (VCD) and subsequently converted to a Switching Active Interchange Format (SAIF) for analysis through DC topographical. The dynamic, static (leakage), and total power of each implementation are extracted from Design Compiler reports.

7.2 Delay, Area, and Power Analysis

7.2.1 A Novel Rounding Scheme for IEEE 754-2008 FP Multiplication

For comparison, the proposed multipliers, ES method and QTF method were implemented and synthesized. The ES, QTF, and proposed methods do not have exceptions due to time constraints and since it does not fall on the critical path would only impact area. The Synopsys DesignWare implementation in shown as comparison, however, this unit has exception logic [38].

As seen on the Table 7.1, our method is 2% faster than ES method, 3% faster

Table 7.1: Post-synthesis results for the proposed design (without denormalized numbers support) in cmos32soi 32nm GF technology at 10 GHz

Methods	Delay	# Cells	Area	Power [mW]		
	[ps]		[um^2]	Dynamic	Static	Total
DW_mult (normalized)	410	12,438	17,515	26.02	11.34	37.35
QTF (normalized)	372	13,178	18,226	19.25	11.90	31.12
ES (normalized)	370	12,943	17,477	18.92	11.30	30.20
Proposed (normalized)	361	12,925	17,757	19.07	11.53	30.59

than QTF method, and 12% faster than DesignWare implementation. The proposed method is 2% larger than ES method but 3% smaller than QTF method. In term of power dissipation, our method is about the same as ES method but still 2% better than QTF method, and 18% better than DesignWare. An important element is here the QTF method is not accurate when testing. A small numer of errors (< 10) occured because of the way the QTF method computes `m1` that was described above making it non IEEE compliant. These errors were **not intentional** and due to the manner which the MSB is computed in the logic, it can be corrected to achieve IEEE compliance.

Table 7.2: Post-synthesis results for the proposed design (supporting denormalized numbers) in cmos32soi 32nm GF technology at 10 GHz

Methods	Delay	# Cells	Area	Power [mW]		
	[ps]		[um^2]	Dynamic	Static	Total
DW_fp_mult (denormalized)	526	17,869	24,476	41.10	15.22	56.32
QTF (denormalized)	590	14,858	18,603	25.52	10.06	35.58
ES (denormalized)	583	14,517	19,317	24.60	10.44	35.04
Proposed (denormalized)	567	13,820	17,838	22.60	9.54	32.14

Table 7.3: Post-synthesis results for the proposed hybrid design (supporting denormalized numbers) in cmos32soi 32nm GF technology at 10 GHz

Methods	Delay	# Cells	Area	Power [mW]		
	[ps]		[um^2]	Dynamic	Static	Total
Hybrid (double-precision)	628	17,070	21,448	29.37	11.81	38.18
Hybrid (single-precision)	628	17,070	21,448	8.00	11.57	19.57
Hybrid (half-precision)	628	17,070	21,448	2.91	11.43	14.34

7.2.2 An Optimized IEEE FP Multiplier Supporting Denormalized Numbers

For comparison, the proposed multiplier supporting denormalized numbers is implemented with three rounding methods: our proposed method, QTF method, and ES method. Similar to previous experiments, the Synopsys DesignWare implementation is also shown as comparison [38].

As seen on the Table 7.2, the proposed design using our rounding method is 4% faster than using QTF rounding and 3% faster than using ES rounding but 8% slower than DesignWare multiplier. Moreover, our proposed design also uses 7% less number of cells than QTF rounding, 5% less number of cells than ES rounding, and 22% less number of cells than DesignWare multiplier. In addition, about power dissipation, our proposed design consumes 10% less power than both QTF and ES methods and 43% less power than DesignWare multiplier. It is worth to notice that DesignWare implements exception processing (NaN, Inf, Zero) while our design does not. However, it is clear that our proposed design is both faster, smaller, and more power-saving than QTF and ES rounding methods in designs that support denormalized numbers.

7.2.3 A Hybrid Precision IEEE FP Multiplier Supporting Denormalized Numbers

As described in previous chapter, the hybrid precision design aims to improve the power dissipation of deep learning applications that can be trained and run with low precision numbers. Therefore, in this experiment, the power dissipation of proposed design is measured in three precision modes: double precision (default), single precision, and half precision [38]. As seen on Table 7.3, switching to the half and single precision mode can save 62% and 49% power dissipation comparing to the double precision mode respectively. In addition, comparing to the double precision design in Table 7.2, the area overhead is fairly small (20%) given the power saving.

7.3 Chapter Summary

This chapter briefly introduces about the ASIC design flow that is used to implement designs described in previous chapter. In particular, proposed and state-of-the-art designs are implemented in RTL-compliant Verilog, verified against TestFloat-3c software, and then synthesized ARM 32nm CMOS library technology using Synopsys Design Compiler. Based on experimental results, our proposed rounding method is both faster and smaller than QTF and ES method in both normalized and denormalized design implementations. In addition, the combined precision and hybrid precision designs can reduce significantly power dissipation for deep learning applications.

CHAPTER 8

CONCLUSIONS AND FUTURE WORK

8.1 Conclusions

In conclusion, in this dissertation, we have presented the general knowledge of IEEE floating point multiplication and architectural improvements that can be applied to increase the performance and reduce the power dissipation.

We first proposed a combined IEEE half and single precision multipliers for many deep learning implementations in which half precision can be safely used to train and run a network. With a configurable control signal, our proposed multipliers can be easily configured to switch between single and half precision IEEE floating-point modes. Our design was completely verified with Hauser's SoftFloat scheme for the correctness. Compared to IEEE standard binary32 multipliers, the proposed multipliers has a small overhead while provide a significant savings of 44% in power dissipation when running in binary16 mode.

We then proposed a clarification and optimization on rounding for IEEE 754-compliant floating-point multiplication. SBH's method is first clarified and provided that it is not applicable for RI mode. Then, QTF's method is summarized and clarified on how it solved the limitations of SBH's method using a special CA. However, QTF's method is based on a rounding table and its design is not optimized and produces inaccurate results. It uses 6 control signals and a 3:1 mux for the LSB and a 4:1 mux for the final result selection. Based on a formula similar to the ES method, a hybrid QTF/ES design is presented that uses only 2 control signals and a standard 2:1mux. The experimental results illustrate an efficient, fast and low-power

Table 8.1: Normalized experimental results for proposed design (without denormalized numbers support)

Methods	Delay [ps]	# Cells	Area [um^2]	Power [mW] Dynamic	Static	Total
DW_mult (normalized)	1.14	0.96	0.99	1.36	0.98	1.22
QTF (normalized)	1.03	1.02	1.03	1.01	1.03	1.02
ES (normalized)	1.02	1.00	0.98	0.99	0.98	0.99
Proposed (normalized)	1.00	1.00	1.00	1.00	1.00	1.00

implementation using a 32nm library. Table 8.1 shows the experimental results of DesignWare FP multiplier (without denormalized numbers support), ES method, and QTF method that are normalized respect to the results of the proposed design for an easy comparison.

Correct rounding of both normal and denormal results further exacerbates the growing complexity of an IEEE 754 multiplier. Due to the importance of high precision in scientific applications [12], the precision must be preserved. Simply truncating denormal results to zero is unacceptable [13]. Consequently, having floating-point units that can handle normalized and denormalized numbers is essential, especially for scientific computing [14]. Recently there have been several types of hardware implementations that handle floating-point denormalized IEEE-754 numbers [15, 16, 17]. This dissertation discusses methods of implementing both normalized and denormalized IEEE 754 numbers [2]. In particular, it combines the pre-normalizing and

Table 8.2: Normalized experimental results for proposed design (supporting denormalized numbers)

Methods	Delay [ps]	# Cells	Area [um^2]	Power [mW] Dynamic	Static	Total
DW_fp_mult (denormalized)	0.93	1.29	1.37	1.82	1.60	1.75
QTF (denormalized)	1.04	1.08	1.04	1.13	1.05	1.11
ES (denormalized)	1.03	1.05	1.08	1.09	1.09	1.09
Proposed (denormalized)	1.00	1.00	1.00	1.00	1.00	1.00

post-normalizing steps in existing methods into a simpler and faster single step. Table 8.2 shows the experimental results of DesignWare FP multiplier (supporting denormalized numbers), ES method, and QTF method that are normalized respect to the results of the proposed design for an easy comparison.

Finally, we proposed an IEEE 754 compliant floating-point multiplier that can handle half, single, and double-precision operations. The previous implementations [20, 19] only demonstrated new methods for single and double-precision operations, however, this design extends the ideas by specifically adapting the architecture for half-precision IEEE 754 multiplication. Half-precision floating-point multipliers are new additions to the 754 standard [2] that are specifically useful for architectures that use machine learning computations. By utilizing smaller amounts of precision, these multipliers can speed up computations for designs that are well suited for neural networks and machine learning applications [21]. Moreover, extensions are added to the multiplier to also handle denormalized IEEE 754 floating-point numbers as well as half and single-precision floating-point numbers [2].

8.2 Future Work

For the future work, our combined IEEE half, single and double precision multipliers can be modified to handle four binary16 and two binary32 operations at the same time with a relatively small modification. A combined Fused Multiply-Add (FMA) that perform $A \times B + C \times D$, in which A, B, C, D can be binary32 or binary16 numbers, is also promising if utilizing our design.

In addition, since the Compound Adder is a key component in any rounding designs. However, existing designs all use Carry-Select Adder and duplicate carry-chain to implement this Compound Adder, which is not optimal in terms of area, power and delay. Therefore, an improvement in Compound Adder can help speed up the overall performance and reduce the area/power of the Multipliers.

REFERENCES

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

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

[3] J. E. Stine, *Digital computer arithmetic datapath design using verilog HDL*. Springer Science & Business Media, 2012.

[4] M. R. Santoro, G. Bewick, and M. A. Horowitz, "Rounding algorithms for IEEE multipliers," in *Proceedings of 9th Symposium on Computer Arithmetic*, pp. 176–183, Sep 1989.

[5] N. T. Quach, N. Takagi, and M. J. Flynn, "Systematic IEEE rounding method for high-speed floating-point multipliers," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 12, no. 5, pp. 511–521, 2004.

[6] G. Even and P.-M. Seidel, "A comparison of three rounding algorithms for IEEE floating-point multiplication," *IEEE Transactions on Computers*, vol. 49, no. 7, pp. 638–650, 2000.

[7] N. H. Weste and D. Harris, *CMOS VLSI design: a circuits and systems perspective*. Pearson Education India, 2015.

[8] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

[9] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pp. 367–379, IEEE, 2016.

[10] "IEEE standard for interval arithmetic," *IEEE Std 1788-2015*, pp. 1–97, June 2015.

[11] N. Burgess, "Prenormalization rounding in IEEE floating-point operations using a flagged prefix adder," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 13, pp. 266–277, Feb 2005.

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

[13] N. J. Higham, *Accuracy and Stability of Numerical Algorithms.* Society for Industrial and Applied Mathematics, second ed., 2002.

[14] G. Gerwig, H. Wetter, E. Schwarz, J. Haess, C. Krygowski, B. Fleischer, and M. Kroener, "The IBM eserver z990 floating-point unit," *IBM Journal of Research and Development*, vol. 48, pp. 311–322, May 2004.

[15] P.-M. Seidel, "How to half the latency of IEEE compliant floating-point multiplication," in *Proceedings of the 24th Euromicro Conference*, vol. 1, pp. 329–332 vol.1, Aug 1998.

[16] X. Hong and J. Jingping, "Research and optimization on rounding algorithms for floating-point multiplier," in *International Conference on Computer Science and Electronics Engineering, (ICCSEE)*, vol. 1, pp. 137–142, March 2012.

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

[18] M. Courbariaux, Y. Bengio, and J. David, "Low precision arithmetic for deep learning," *CoRR*, vol. abs/1412.7024, 2014.

[19] T. D. Nguyen, S. Bui, and J. E. Stine, "Clarifications and optimizations on rounding for IEEE-compliant floating-point multiplication," in *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pp. 1–8, July 2018.

[20] R. Thompson and J. E. Stine, "An IEEE 754 double-precision floating-point multiplier for denormalized and normalized floating-point numbers," in *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pp. 62–63, July 2015.

[21] J. Dean, D. Patterson, and C. Young, "A new golden age in computer architecture: Empowering the machine-learning revolution," *IEEE Micro*, vol. 38, pp. 21–29, Mar 2018.

[22] 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.

[23] W. Kahan, "A brief tutorial on gradual underflow," *Available as a PDF file at http://www. cs. berkeley. edu/˜ wkahan/ARITH_17U. pdf*, 2005.

[24] "IEEE standard for interval arithmetic (simplified)," *IEEE Std 1788.1-2017*, pp. 1–38, Jan 2018.

[25] C. SPARC International, Inc., *The SPARC Architecture Manual: Version 8*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1992.

[26] C. S. Wallace, "A suggestion for a fast multiplier," *Computers, IEEE Transactions on*, vol. EC-13, pp. 14–17, 1964.

[27] L. Dadda, "Some schemes for parallel multipliers," *Alta Frequenza 34*, pp. 349–365, 1965.

[28] K. A. C. Bickerstaff, M. J. Schulte, and E. E. Swartzlander, Jr., "Reduced area multipliers," in *Application-Specific Array Processors, 1993. Proceedings., International Conference on*, pp. 478 –489, Oct. 1993.

[29] J. Bergstra *et al.*, "Theano: a CPU and GPU math expression compiler," in *Proceedings of the Python for Scientific Computing Conference (SciPy)*, June 2010. Oral Presentation.

[30] R. Collobert, K. Kavukcuoglu, and C. Farabet, "Torch7: A matlab-like environment for machine learning," in *BigLearn, NIPS Workshop*, 2011.

[31] M. Abadi *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pp. 265–283, 2016.

[32] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *arXiv preprint arXiv:1408.5093*, 2014.

[33] T. D. Nguyen and J. E. Stine, "A combined IEEE half and single precision floating point multipliers for deep learning," in *2017 51st Asilomar Conference on Signals, Systems, and Computers*, pp. 1038–1042, Oct 2017.

[34] A. Akkas and M. J. Schulte, "A quadruple precision and dual double precision floating-point multiplier," in *Digital System Design, 2003. Proceedings. Euromicro Symposium on*, pp. 76–81, IEEE, 2003.

[35] M. K. Jaiswal and H. K.-H. So, “Dual-mode double precision/two-parallel single precision floating point multiplier architecture,” in *Very Large Scale Integration (VLSI-SoC), 2015 IFIP/IEEE International Conference on*, pp. 213–218, IEEE, 2015.

[36] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[37] 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.

[38] T. D. Nguyen, S. R. Thompson, and J. E. Stine, “Architectural improvements in IEEE-compliant floating-point multiplication,” *IEEE Transactions on Computers*, 2018. Submitted.

[39] E. E. Swartzlander, Jr., “Merged arithmetic,” *IEEE Transactions on Computers*, vol. C-29, pp. 946–950, Oct 1980.

[40] R. Yu and G. Zyner, “167 MHz radix-4 floating point multiplier,” in *Proceedings of the 12th Symposium on Computer Arithmetic*, pp. 149–154, Jul 1995.

[41] P. M. Seidel, *On the Design of IEEE Compliant Floating-Point Units and Their Quantitative Analysis.* PhD thesis, Saarland University, 1999. Available at http://scidok.sulb.uni-saarland.de.

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

[43] G. Dimitrakopoulos, K. Galanopoulos, C. Mavrokefalidis, and D. Nikolos, “Low-power leading-zero counting and anticipation logic for high-speed floating point units,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, pp. 837–850, July 2008.

[44] S. Golson, “Resistance is futile! building better wireload models,” in *Proc. of SNUG*, 1999.

[45] K. D. Boese, A. B. Kahng, and S. Mantik, “On the relevance of wire load models,” in *Proceedings of the 2001 international workshop on System-level interconnect prediction*, pp. 91–98, ACM, 2001.

[46] J. E. Stine and J. Grad, “Low power and high speed addition strategies for VLSI,” in *2006 8th International Conference on Solid-State and Integrated Circuit Technology Proceedings*, pp. 1610–1613, Oct 2006.

VITA

Tuan Danh Nguyen

Candidate for the Degree of

Doctor of Philosophy

Dissertation: **ARCHITECTURAL IMPROVEMENTS IN IEEE-COMPLIANT FLOATING-POINT MULTIPLICATION**

Major Field: Electrical and Computer Engineering

Biographical:

Education:

Completed the requirements for the Doctor of Philosophy in Electrical and Computer Engineering at Oklahoma State University, Stillwater, Oklahoma in December, 2018.

Completed the requirements for the Bachelor of Science in Computer Science at Hanoi University of Science and Technology, Hanoi, Vietnam in 2008.

Experience:

Research Assistant at VLSI Computer Architecture Research Group, Oklahoma State University, Stillwater, OK, USA (01/14-12/18)

Graduate Assistant at Technology Development Center, Oklahoma State University, Stillwater, OK, USA (06/16-12/18)

Systems Engineering Intern at WalmartLabs, Sunnyvale, CA, USA (05/18-08/18)

Data Analysis Intern at American Institutes for Research, Washington DC, USA (05/15-08/15)

Technical Leader at Viettel Software Center, Viettel Telecom, Hanoi, Vietnam (08/08-06/2011)