Approximate Floating Point Arithmetic

A PROJECT REPORT

submitted by

Vikas Chaganti

EE08B053

for the award of the degree

of

BACHELOR OF TECHNOLOGY

and

MASTER OF TECHNOLOGY

under the guidance of

Prof. Shankar balachandran and **Prof. Nitin Chandrachoodan**

DEPARTMENT OF ELECTRICAL ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY MADRAS

CHENNAI-600036

CERTIFICATE

This is to certify that the report titled **"Approximate Floating Point Arithmetic",** submitted by Mr. Vikas Chaganti, to the Indian Institute of Technology Madras, Chennai for the award of the degree of **Master of Technology in Microelectronics and VLSI design** and **Bachelor of Technology in Electrical Engineering**, is bonafide record of research work done by him under my supervision. The contents of the this thesis, in full or parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Dr. Shankar Balachandran
Project Guide
Assistant Professor
Dept. of Computer science
IIT-Madras, Chennai-600036

Place: Chennai
Date: June 2013

ACKNOWLEDGEMENT

I would like to take this opportunity to express my sincere gratitude to my project guide **Dr. Shankar Balachandran,** Department of Computer Science. He put a consider amount of time and effort to explain the concepts related to project and helped me to understand the project clearly. I would like to thank my co-guide **Dr. Nithin Chandrachoodan,** Department of Electrical Engineering, for his for his guidance and suggestions during this project work.

I would not forget to remember my lab mates, Chaitanya, Amit, Sanak, Karthikeyan and Satish for their timely support and guidance till the completion of our project work.

A special thanks to my parents and sister, an endless source of support and encouragement.

Vikas Chaganti

Abstract

Floating Point architectures are area and power demanding structures. In this report I propose several approximations can be used to reduce their area and power requirements. As a result of these approximations, some error is introduced into the computation. Barrel shifter is a big part of Floating Point adder, modification to it's size decreases the area and power requirements of Floating Point Adder. In Floating Point Computations final result is rounded back to fixed length. So use of Truncated multipliers instead of full precision multipliers, result in power and area efficient Floating Point Multiplier and Floating Point Division Unit. Such Floating Point Arithmetic operations are implemented and analized in this report.

Contents

List of Tables

List of Figures

ABBREVIATION

FP	Floating Point
FPU	Floating Point Unit
FPA	Floating Point Addition
FPM	Floating Point Multiplication
FPD	Floating Point Division
ULP	Unit in the Last Place
LSB	Least Significant Bit
MSB	Most Significant Bit
MSP	Most Significant Part
LSP	Least Significant Part
XST	Xilinx Synthesis Tool
XPE	Xilinx Power Analyzer
SEIF	Switching Activity Interchange format
ROM	Read Only Memory

Chapter 1

Introduction

1.1 Approximate Computing

Typically we expect a Arithmetic unit to produce a correct result. But some times the software algorithm does not need such a "strict" correctness, but only limited correctness. If a program is not precise by its nature, such as approximation, video/audio encoding, classification application, it most probably does not require all the computations to be precise, and can exhibit even higher than usual fault tolerance. Whether certain error in computation is acceptable or not depends on the definition of the quality of service (QoS) for this application.

The common programs with high error tolerance in computations are Biological applications, computer vision, machine learning, sensory data analysis and image rendering etc.. Those can be very tolerant to potential error in computation and produce acceptable results even in the presence of significant amount of error. So, it is possible to exploit those characteristics and get performance improvement by simplifying hardware. Moreover, such performance improvements may result in significantly less computation, and, hence, power savings.

1.2 Approximations and Floating Point Arithmetic

Floating point arithmetic is used by applications that require large dynamic range. In the context of approximate computing, many applications with error tolerance, like image rendering, Computer vision and machine learning algorithms use floating point arithmetic. Floating Point computations are lot more complex arithmetic operations and computationally intensive compared to fixed point operations. So some times we may want to reduce the complexity of floating point computations by giving room for some error. In this report several techniques like excluding the computation of redundant LSB bits, while computing fraction of product or quotient, were explained and analyzed.

1.3 Organization of Report

The out line of report is as follows:

Chapter 1 outlines the concept of approximate computing in the context of Floating Point Arithmetic.

Chapter 2 explains an implementation of Floating Point adder, and several optimization techniques to design a fast Floating Point Adder. Some modifications to Barrel Right Shifter are suggested to decrease the area and power requirements.

Chapter 3 describes a method to implement Floating Point Multiplier. Explains about the truncated multiplication and FPM with truncated multiplier in place of full precision multiplier. Area, Power and Error analysis was done for such Floating Point Multipliers.

Chapter 4 explains “Division Through Multiplication” algorithms for fast calculation of division and an algorithm for Floating point division. Use of truncated multiplication for division is discussed and implemented. Area, Power and error analysis is done for Floating Point Multipliers with truncated multipliers.

Chapter 5 is conclusion of the report.

Chapter 2

Floating Point Adder/Subtractor

2.1 Introduction

Floating-point (FP) addition and subtraction are the most frequent FP operations. Both operations use a FP-adder. Floating point addition is the most complicated operation among Floating Point operations. Therefore a lot of effort has been spent on designing FP-adders.

2.2 Optimization Techniques

In this design several optimization techniques have been used to reduce the delay. The optimization techniques that we use include the following techniques:

1. A two path design with a new separation criterion, presented in[2], is used.
2. A simpler design is obtained by using unconditional preshifts for effective subtractions to to unify the range of significands' sum and difference may belong to[2].
3. The sign-magnitude representation of the difference of the exponents and the significands is derived from one's complement representation of the difference.
4. A parallel-prefix adder is used to compute the sum and the incremented sum of the significands [10, 4].

5. Two level radix-8 barrel shifter to reduce FANOUT at each node[1]

2.2.1 Separation FP-Adder into Two Parallel Paths

The FP-adder is separated into two parallel paths that work under different assumptions. The partitioning into two parallel paths enables one to optimize each path separately by simplifying and skipping some steps of the naive addition algorithm. Such a dual path approach for FP-addition was first described by Farmwald [3]. Since Farmwald's dual path FP-addition algorithm, the common criterion for partitioning the computation into two paths has been the exponent difference. The exponent difference criterion is defined as follows: The near path is defined for small exponent differences (i.e., -1, 0,+1), and the far path is defined for the remaining cases.

A different partitioning criterion presented in [2], is used for partitioning the algorithm into two paths: the $N-path$ is define for the computation of all effective subtractions with small significand sums $fsum \in (-1,1)$[1] and small exponent differences $|\delta| \leq 1$, and the R-path for all the remaining cases. We define the path selection signal IS_R as follows:

$$IS\_R \;\Leftrightarrow\; \overline{S.EFF}\ OR\ |\delta| \geq 2\ OR\ fsum \in [1,2) \tag{2.1}$$

The outcome of the R-path is selected for the final result if $IS\_R = 1$, otherwise the outcome of the N-path is selected. This partitioning has the following advantages:

1. In the R-path, the normalization shift is limited to a shift by one position (in Section 2.2.2, we show how the normalization shift may be restricted to one direction). Moreover, the addition or subtraction of the significands in the R-path always results with a positive significand and, therefore, the conversion step can be skipped.

2. In the N-path, the alignment shift is limited to a shift by one position to the right. Under the assumptions of the N-path, the exponent difference is in the range {-1,0,+1}.

[1] $fsum$ is fraction part of sum

Therefore, a 2-bit subtraction suffices for extracting the exponent difference. Moreover, in the N-path, the significand difference can be exactly represented with 53 bits, hence, no rounding is required.

Note that the N-path applies only to effective subtractions in which the significand difference $fsum$ is less than 1. Thus, in the N-path it is assumed that $fsum \in (-1,1)$. The advantages of our partitioning criterion compared to the exponent difference criterion stem from the following two observations: (1) A conventional implementation of a far path can be used to implement also the R-path and (2) The N-path is simpler than the near path since no rounding is required and the N-path applies only to effective subtractions. Hence, the N-path is simpler and faster than the near path presented in [3].

2.2.2 Unification of Sginificand Result Ranges

In the R-path, the range of the resulting significand is different in effective addition and effective subtraction. In effective addition, $fl \in [1,2)$[2] and $fsan \in [0,2)$[3]. Therefore, $fsum \in [1,4)$. It follows from the definition of the path selection condition that in effective subtractions $fsum \in (\frac{1}{2},2)$ in the R-path. We unify the ranges of $fsum$ in these two cases to $[1,4)$ by multiplying the significands by 2 in the case of effective subtraction (i.e., preshifting by one position to the left). The unification of the range of the significand sum in effective subtraction and effective addition simplifies the rounding circuitry. To simplify the notation and the implementation of the path selection condition, we also preshift the operands for effective subtractions in the N-path. Note that, in this way, the preshift is computed in the N-path unconditionally, because in the N-path all operations are effective subtractions. In the following, we give a few examples of values that include the conditional preshift (note that an additional "p" is included in the names of the preshifted

[2] fl is fraction part of large floating point number

[3] $fsan$ is small fraction shifted by exponent differences mathematically represented as $(-1)^{S\_EFF} \times 2^{-\delta}$

versions):

$$flp = \begin{cases} 2.fl & if\, S\_EFF \\ fl & otherwise, \end{cases} \tag{2.2}$$

$$fspan = \begin{cases} 2.fsan & if\, S\_EFF \\ fsan & otherwise, \end{cases} \tag{2.3}$$

$$fpsum = \begin{cases} 2.fsum & if\, S\_EFF \\ fsum & otherwise. \end{cases} \tag{2.4}$$

Note that, based on the significand sum $fpsum$, which includes the conditional preshift, the path selection condition (1) can be rewritten as

$$IS\_R \;\Leftrightarrow\; \overline{S\_EFF}\; OR\; |\delta| \geq 2\; OR\; fpsum \in [2,4) \tag{2.5}$$

2.2.3 Sign-Magnitude Computation of a Difference

In this technique, the sign-magnitude computation of a difference is computed using one's complement representation. This technique is applied in two situations:

1. Exponent difference. The sign-magnitude representation of the exponent difference is used for two purposes:

   (a) the sign determines which operand is selected as the "large" operand; and

   (b) the magnitude determines the amount of the alignment shift.

2. Significand difference. In case the exponent difference is zero and an effective subtraction takes place, the significand difference might be negative. The sign of the significand difference is used to update the sign of the result and the magnitude is normalized to become the result's significand.

Let A and B denote binary strings and let $|A|$ denote the value represented by A (i.e., $|A| = \sum_i A[i] .2^i$). The technique is based on the following observation:

$$abs(|A| - |B|) = \begin{cases} |A| + |B| + 1 & if\ |A| - |B| > 0 \\ \overline{|A| + |B|} & if\ |A| - |B| \leq 0 \end{cases} \tag{2.6}$$

The actual computation proceeds as follows: The binary string D is computed such that $|D| = |A| + |\overline{B}|$.We refer to D as the *one's complement lazy difference* of A and B. We consider two cases:

1. If the difference is positive, then $|D|$ is off by an ULP and we need to increment D. However, to save delay, we avoid the increment as follows:
    (a) In the case of the exponent difference that determines the alignment shift amount, the significands are preshifted by one position to compensate for the error.
    (b) In the case of the significand difference, the missing ULP is provided by computing the incremented sum of $|A|$ and $|B|$ using a compound adder.
2. If the exponent difference is negative, then the bits of D are negated to obtain an exact representation of the magnitude of the difference.

2.2.4 Compound Addition

The technique of computing in parallel the sum of the significands as well as the incremented sum is well known. The rounding decision controls which of the sums is selected for the final result, thus enabling the computation of the sum and the rounding decision in parallel.

We used the technique suggested in [10] for implementing a compound adder. This technique is based on a parallel prefix adder in which the carry-generate and carry-propagate strings, denoted by *Gen_C* and *Prop_C*, are computed [4]. Let $Gen\_C[i]$ equal the carry bit that is fed to position *i*. The bits of the sum S of the addends A and B are obtained as usual by:

$$S[i] = XOR(A[i], B[i], Gen\_C[i]). \tag{2.7}$$

The bits of the incremented sum *SI* are obtained by:

$$SI[i] = XOR(A[i], B[i], OR\,(Gen\_C[i], Prop\_C[i])). \tag{2.8}$$

2.2.5 Barrel Shifter Design

Barrel shifter must be capable of performing large shift operations, as large as the number of digits in the significand field. The overall performance of the floating point add/subtract unit is highly dependent on the speed of these two shifters. A barrel shifter can be implemented as single level array where each input bit is directly connected to m output lines. for $m = 53$ the large number of connections make this an undesirable solution.

A two level radix-8 barrel shifter instead of a single level barrel shifter, where the first level shifts from 0-7 bit positions and second level shifts by multiples of 8. Thus, each bit the first level has 8 destinations and 7 in the second level.

2.3 FP Adder Algorithm

The algorithm is partitioned into two parallel paths called the R-path and the N-path. The final result is selected between the outcomes of the two paths based on the signal IS_R (see eqn.(2)). Several block diagrams of our algorithm are depicted in Fig.2.1. We give an overview of the two paths in the following.

2.3.1 R-path

Computations in R-path are specified in this section. Fig.2.2 shows the detailed block diagram of R-path.

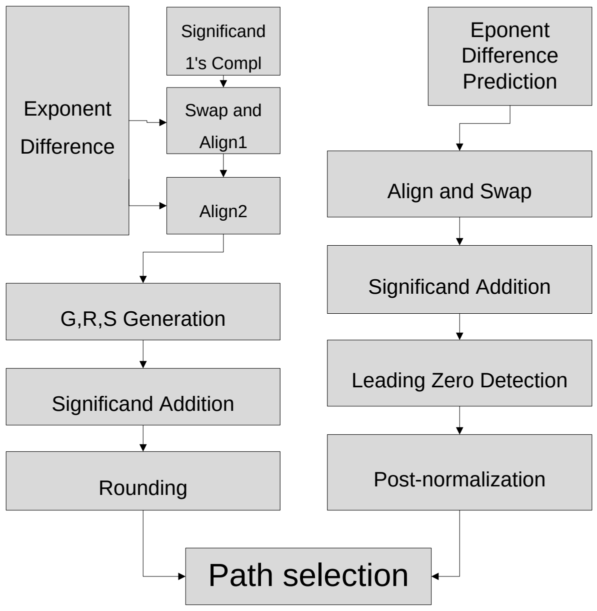

Figure 2.1: Higher-level view of FP-adder implementation

Exponent Difference Calculation

1. The exponent difference is computed for two ranges: The medium exponent difference interval consist of [63, 64], and the big exponent difference intervals consist of [$-\infty$,64] and [65, ∞]. The outputs of the exponent difference box are specified as follows: Loosely speaking, the SIGN_MED and MAG_MED are the sign-magnitude representation of , if is in the medium exponent difference interval. Formally,

$$(-1)^{SIGN\_MED}.(MAG\_MED) = \begin{cases} \delta - 1 & if\ 64 \geq \delta \geq 1 \\ \delta & if\ 0 \geq \delta \geq -63 \\ "don't - care" & otherwise \end{cases} \tag{2.9}$$

The reason for missing δ by 1 in the positive case due to the one's complement subtraction of exponents. This error term is compensated for in the Align1 box(by preshifting).

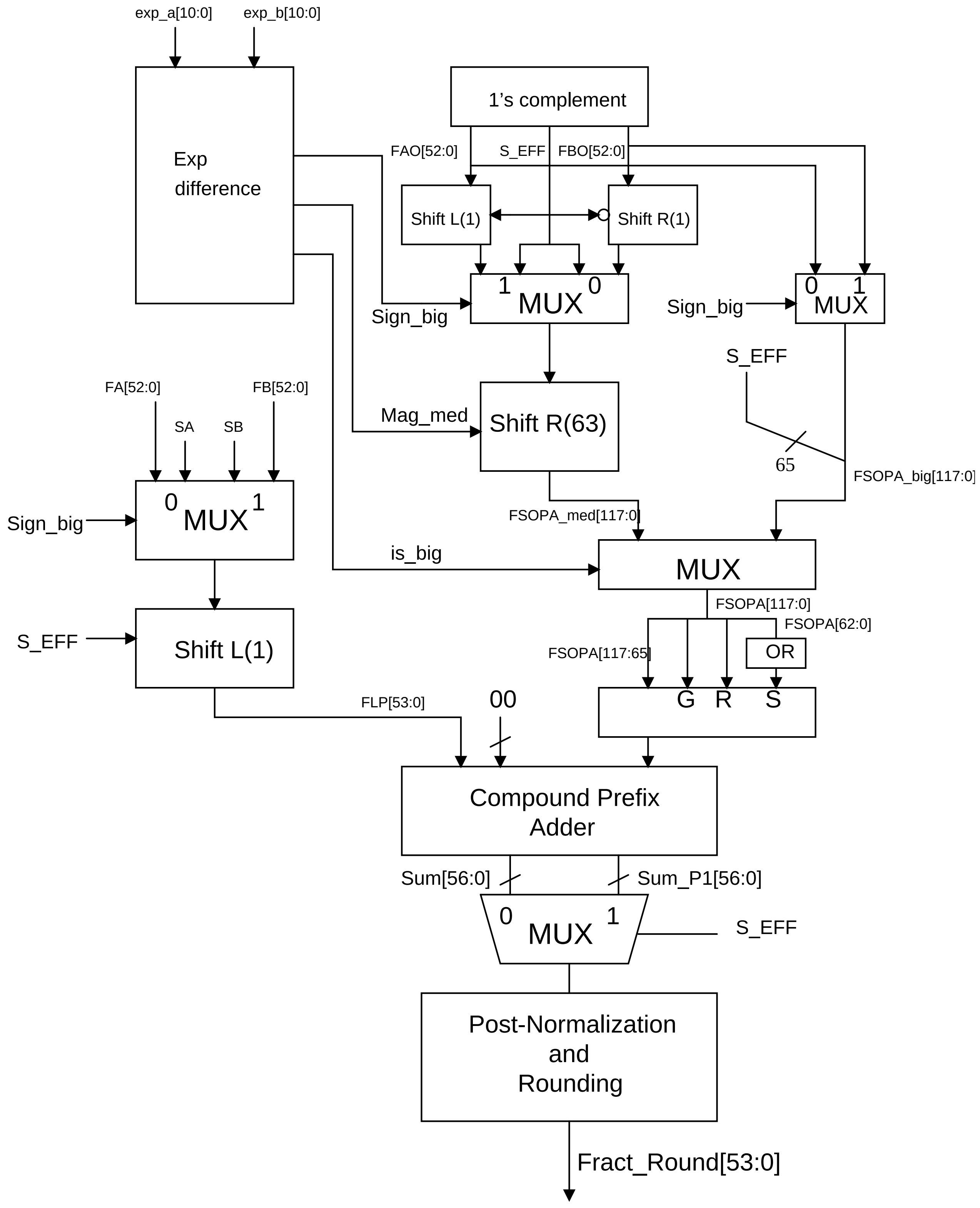

Figure 2.2: Detailed Block Diagram of R-path

$SIGN\_MED$	S_EFF	pre-shift	align-shift	accumulated right shift	$FSOP'[54:0]$
0	0	0	1	1	{00,FBO[52:0]}
	1	1	1	0	{1,FBO[52:0],1}
1	0	0	0	0	{0,FBO[52:0],0}
	1	1	0	-1	{FBO[52:0],11}

Table 2.1: Value of $FSOP'[54:0]$

2. SIGN_BIG is the sign bit of exponent difference . IS_BIG is a flag defined by:

$$IS\_BIG = \begin{cases} 1 & if\ \delta \geq 65\ or\ \delta \leq -64 \\ 0 & otherwise \end{cases} \tag{2.10}$$

Preshift and Align

1. The One's Complement box calculate the signals FAO, FBO, and S_EFF. The FAO and FBO signals are defined by

$$FAO[52:0],\ FBO[52:0] = \begin{cases} FA[52:0],\ FB[52:0] & if\ S\_EFF = 0 \\ not(FA[52:0]),\ not(FB[52:0]) & otherwise. \end{cases} \tag{2.11}$$

   The computations performed in the Preshift and Align 1 region are relevant only if the exponent difference is in the medium exponent difference interval. The significands are pre-shifted if an effective subtraction takes place. After the preshifting, an alignment shift by one position takes place if $SIGN\_MED = 1$. Table 2.1 summarizes the specification of $FSOP'[54:0]$.

2. The small operand is selected for the medium exponent difference (based on $SIGN\_MED$) interval and for the large exponent difference interval (based on $SIGN\_BIG$). The Preshift region deals with preshifting the minuend in case an effective subtraction takes place.

3. In the big exponent difference intervals, the "required" alignment shift is at least 64

positions. Since all alignment shifts of 55 positions or more are equivalent, we may limit the shift amount in this case.So $FSOP'[54:0]$ is right shifted by MAG_MEG if IS_BIG is 1, otherwise right shifted by 64

$$FSOPA[118:0] = \begin{cases} FSOP'[54:0] \ll MAG\_MED & if\ IS\_BIG = 1 \\ FSOP'[54:0] \ll 64 & otherwise. \end{cases} \tag{2.12}$$

Sum Calculation And Rounding

1. In the Swap region, the large operand is selected based on $SIGN\_BIG$. Large fraction is shifted by one position to right if $S\_EFF = 1$

$$FLOP[53:0] = \begin{cases} \{0, FL[52:0]\} & if\ S.EFF = 1 \\ \{FL[52:0], 0\} & otherwise. \end{cases} \tag{2.13}$$

2. GRS Generation block generates Guard(G), Round(R) and Sticky(S) bits

$$G = FSOPA[64],\ R = FSOPA[63]\ and\ S = OR(FSOPA[62:0]). \tag{2.14}$$

The first 53 bits of $FSOPA[117:65]$ appended with G, R, and S bits is given as input to the compound adder.

$$FSOP[55:0] = \{FSOPA[117:65], G, R, S\}. \tag{2.15}$$

3. The fractions $FLOP$ and $FSOP$ are aligned and added using the compound adder. The result is choosen from sum or sum plus one based on the effective value of S_EFF

$$FSUM[56:0] = \begin{cases} FLOP[55:0] + FSOP[55:0] & if\ S\_EFF = 1 \\ FLOP[55:0] + FSOP[55:0] + 1 & otherwise. \end{cases} \tag{2.16}$$

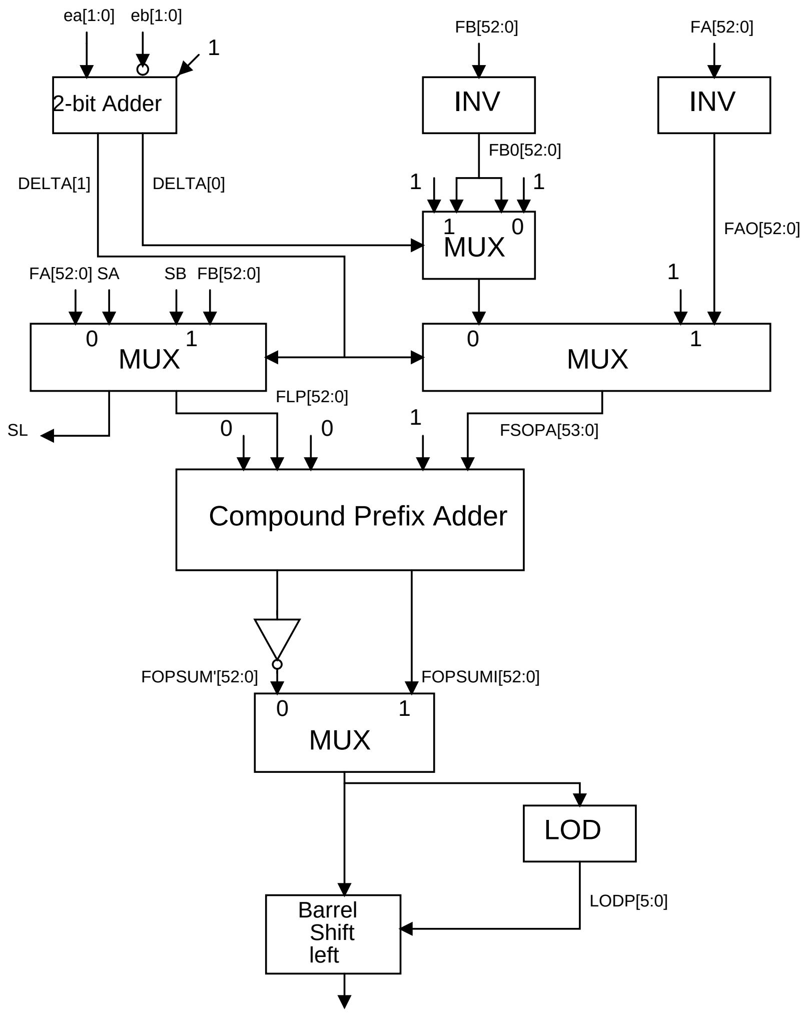

Figure 2.3: Detailed Block Diagram of N-Path

4. The rounding block will finish normalization and implements four IEEE standard rounding schemes(Round-to-zero, Round-to-nearest-even, Round-to-plus-infinity and round-to-minus-infinity). The exponent is adjusted accordingly.

2.3.2 N-path

The N-path works under the assumption that an effective subtraction takes place, the significand difference (after the swapping of the addends and preshifting) is less than 2, and the absolute value of the exponent difference $|\delta|$ is less than 2. The computations in N-path are explained below. A detailed block diagram of the N-path and the central signals are depicted in . 2.3.

Exponent Difference Prediction

1. The Small Exponent Difference box outputs DELTA [1 : 0] representing $ea[1:0]-eb[1:0]$.

Align and Swap

1. The input to the Small Significands: Select, Align, and Preshift box consists of the inverted significand strings FAO and FBO. The selection means that if the exponent difference equals 1, then the subtrahend corresponds to FA, otherwise it corresponds to FB. The preshifting means that the significands are preshifted by one position to the left (i.e. multiplied by 2). The alignment means that if the absolute value of the exponent difference equals 1, then the subtrahend needs to be shifted to the right by one position (i.e., divided by 2). The output signal FSOPA is therefore specified by

$$FSOPA[53:0] = \begin{cases} \{1, FAO[52:0]\} & if\ ea-eb=-1 \\ \{FBO[52:0], 1\} & if\ ea-eb=0 \\ \{1, FBO[52:0]\} & if\ ea-eb=1. \end{cases} \tag{2.17}$$

2. The Large Significands: Select and Preshift box outputs the minuend $FLP[53:0]$ and the sign-bit of the addend it corresponds to. The selection means that if the exponent difference equals 1, then the minuend corresponds to FB, otherwise it corresponds to FA. The preshifting means that the significands are preshifted by one position to the left (i.e., multiplied by 2). The output signal $FLP[53:0]$ is therefore specified by

$$FLP[53:0], SL = \begin{cases} \{FB[52:0], 0\}, SB & if\ ea-eb=-1 \\ \{FA[52:0], 0\}, SA & if\ ea-eb\geq 0. \end{cases} \tag{2.18}$$

Significand Addition

1. In N-path only effective subtraction happens, so sum is calculated using *Lazy one's*

complement subtraction. When predicted exponent difference is zero the sum can be eaither positive or negative. A compound adder is used to calculate both *sum* and $sum+1$. If the sign bit of sum is 1 then the result is complement of *sum*, otherwise $sum+1$

$$abs\_FPSUM[53:0] = \begin{cases} \overline{FOPSUM[53:0]} & if\ FOPSUM[54] = 1, \\ FOPSUMI[53:0] & otherwise. \end{cases} \tag{2.19}$$

2. A priority encoder with priority to MSB is used to find *leading one* in $abs\_FPSUM[53:0]$. In normalization step $abs\_FPSUM[53:0]$ is shifted left by number of leading zeros. A two level radix-8 barrel shifter is implemented to do left shift. The exponent is adjusted accordingly.

2.3.3 Path Selection

R-path is selected if $IS\_R = 1$, otherwise N-path is selected. Eqn.1 explains when IS_R will become 1. A signal IS_R1 is defined such that it will be 1, when $|\delta| \geq 2$:

$$IS\_R1 = IS\_BIG \vee (OR(MAG\_MED[5:1])) \vee (MAG\_MED \wedge \overline{SIGN\_BIG}). \tag{2.20}$$

IS_R2 is equivalent to $fsum \in [1,2)$ and is computed in N-path

$$IS\_R2 = abs\_FPSUM[53] \tag{2.21}$$

and IS_R is calculated using:

$$IS\_R = IS\_R1 \vee IS\_R2 \vee \overline{S\_EFF} \tag{2.22}$$

	Floating Point Adder
Area(in LUTs)	1947
Estimated Power (mW)	16.62
Delay (ns)	28.9
Area of Barrel Right Shifter (in LUTs)	439

Table 2.2: Area, Power and Delay details of Floating Point Adder

2.4 Testing and Implementation

2.4.1 Testing

To verify the functional correctness of the implemented design, a functionally equivalent MATLAB program has been written and used to verify the functional correctness of the verilog design. For rounding mode, all special cases along with 8000 random generated numbers are given as inputs to MATLAB program and verilog design. Post-synthesis simulation is done using *Isim* and results are written into file. This result is compared with the results generated using MATLAB. Ideally, these two results should match, otherwise the design is functionally wrong.

2.4.2 Implementation

This Floating Point Adder is implemented on Vertex-5 LX100T FPGA board. Power of the implemented design is estimated using Xilinx Power Analyzer (XPE). For more accurate estimation power a SAIF(Switching Activity Interchange format) file is generated through post-route simulation, and it as input to the XPE tool. SAIF file contains the activity information of the nodes in design during the simulation. Area, power and delay details are given in table 2.2

Module Name	Power Estimation (mW)
Floating Point Adder	16.62
N-path	6.26
R-path	6.89
Barrel right shifter	3.14

Table 2.3: Power Distribution in Floating Point Adder sub modules

2.5 Approximate Barrel Shifter Design

The number of LUTs occupied by design and, power estimation are given in table 2.2. Out of 1947 LUTs used by floating point adder, 439 LUTs used by right shifting Barrel Shifter. Barrel Right shifter consumes significant (22.4%) amount of total floating point adder area. Out of 6.89 mW power consumed by total R-path, 3.14 mW power is used by only barrel shifter. Barrel shifter's estimated power is almost half of the total power consumed by entire R-path.

The total power consumed and area requirements of barrel shifter can be reduced by allowing some approximation into computation. In the IEEE complaint floating point adder designed in previous section have barrel shifter width of 118 bits. Out of these 118 bits 63 LSB bits are used for computation of sticky bit, 1 bit for round bit. Sticky bit and round bit are used to correctly round the result after addition. If we exclude the computation of sticky and round bits, maximum value of error introduced into the computation of fraction is one *LSB*. By excluding the computation of sticky and round bits, we can reduce the width of the Barrel Shifter to 54 bits, which reduces the LUTs required to implement the barrel right shifter by half. The Barrel shifters power also approximately reduced by half.

Resources required by barrel shifter can be further reduced by decreasing the no of control bits in barrel shifter. The reduction in control bits, limits the maximum shift that occurs introducing error into fraction computation, when the exponent difference s higher than maximum allowable shift. Table 3 gives maximum allowable shift for corresponding no of control bits. The above approximations we can not use for Left barrel shift in N-path

	Floating Point Adder (Barrel right shifter width=118)	Floating Point Adder (Barrel right shifter width=54)
rounding mode	RZ, RN, RPI, RNI	Only RZ
LUTs	1947	1637
Estimated Power	16.62 mW	13.89 mW
Critical path delay	28.9ns	29.2ns
Error in other rounding modes	0	*one LSB*

Table 2.4: Area and Power Comparison among FPAs

No of Control Bits (n)	Maximum Allowed Shift Value
6	64
5	32
4	16
3	8
2	4
1	2

Table 2.5: Maximum allowable shift

for denormalization as it introduces error in MSB side.

The result section have quantitative results for all above scenarios.

2.6 Results

Due to decrement in number of control bits the design of barrel shifter become more simple, less area and power consuming. Fig.2.4 and Fig.2.5 shows the decrement in the area and power requirements.

But decrement in control bits result in decrement maximum allowable shift. Error introduced into fraction when the exponent difference between two inputs is grater than maximum allowable right shift. Fig.2.6 shows that error increases very rapidly compared to area and power savings. So only when error tolerance is very high we do such approximations.

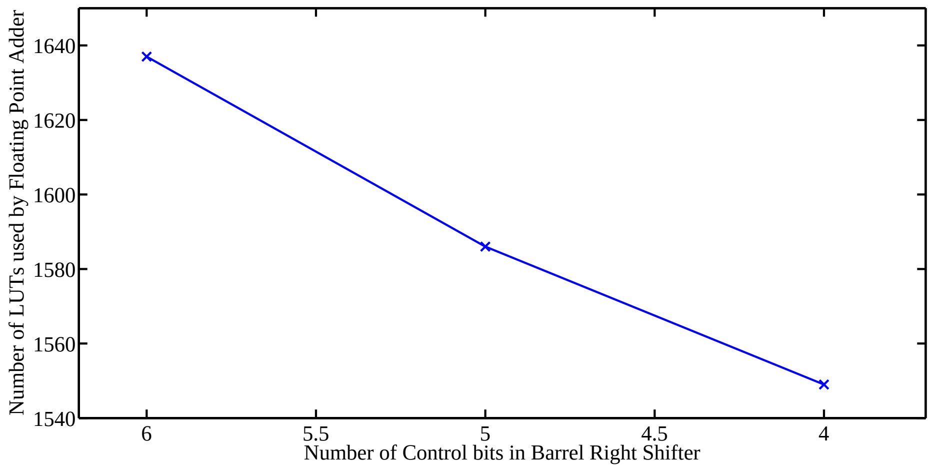

Figure 2.4: Reduction in Area by reducing control bits in Barrel Shifter

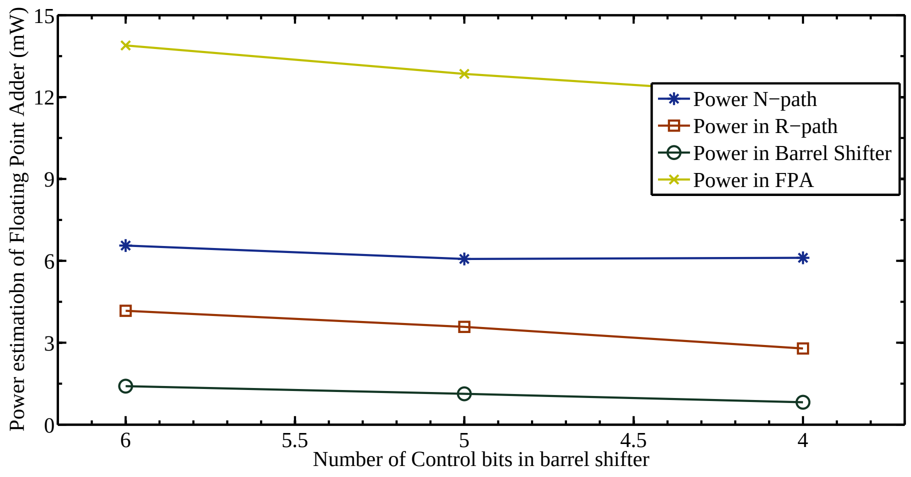

Figure 2.5: Reduction in Power by Reducing number of control bits in Barrel Shifter

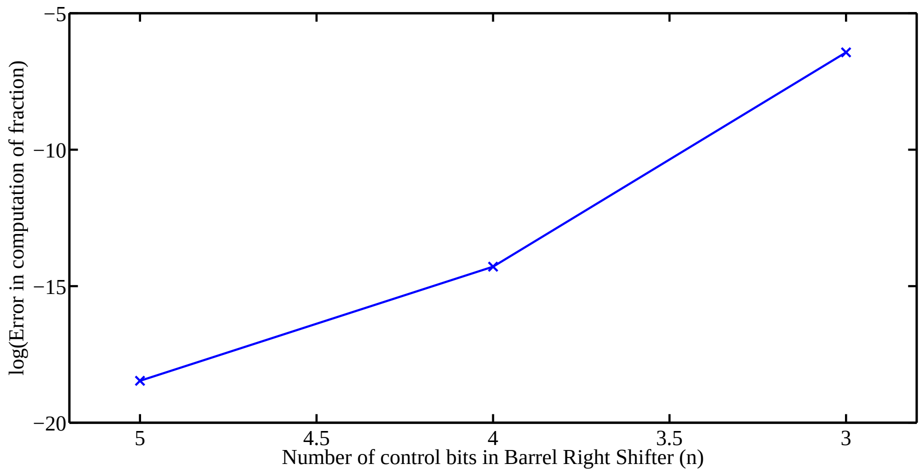

Figure 2.6: Error due to Decrement in number of Control bits in Barrel Shifter

Chapter 3

Floating Point Multiplier

3.1 Introduction

Floating point multiplication is one of the most frequent arithmetic operations. The design of fast, and energy and area efficient implementation of Floating point Multiplier is very important. The most computationally intensive part in Floating Point Multiplier is Fixed point multiplication between two fractions of multiplicands. So performance Floating point multipliers highly depends on the implementation of fraction multiplier.

This chapter compiles of an implementation of double precision Floating point multiplication for normalized numbers and analyzes how truncated multiplication in place of full precision multiplication, results in area and power reduction for very little error introduced.

3.2 FP Multiplication Implementation

Fig.3.1 shows the computations involved in Floating Point Multiplication of two double precision normalized floating point numbers. The product of two floating point numbers can be represented by the following equation

$$(-1)^{sign_{product}} 1.f_{product}.2^{e_{product}} = (-1)^{sign_1 \oplus sign_2}(1.f_1 \times 1.f_2).2^{(e_1+e_2-bias)}. \tag{3.1}$$

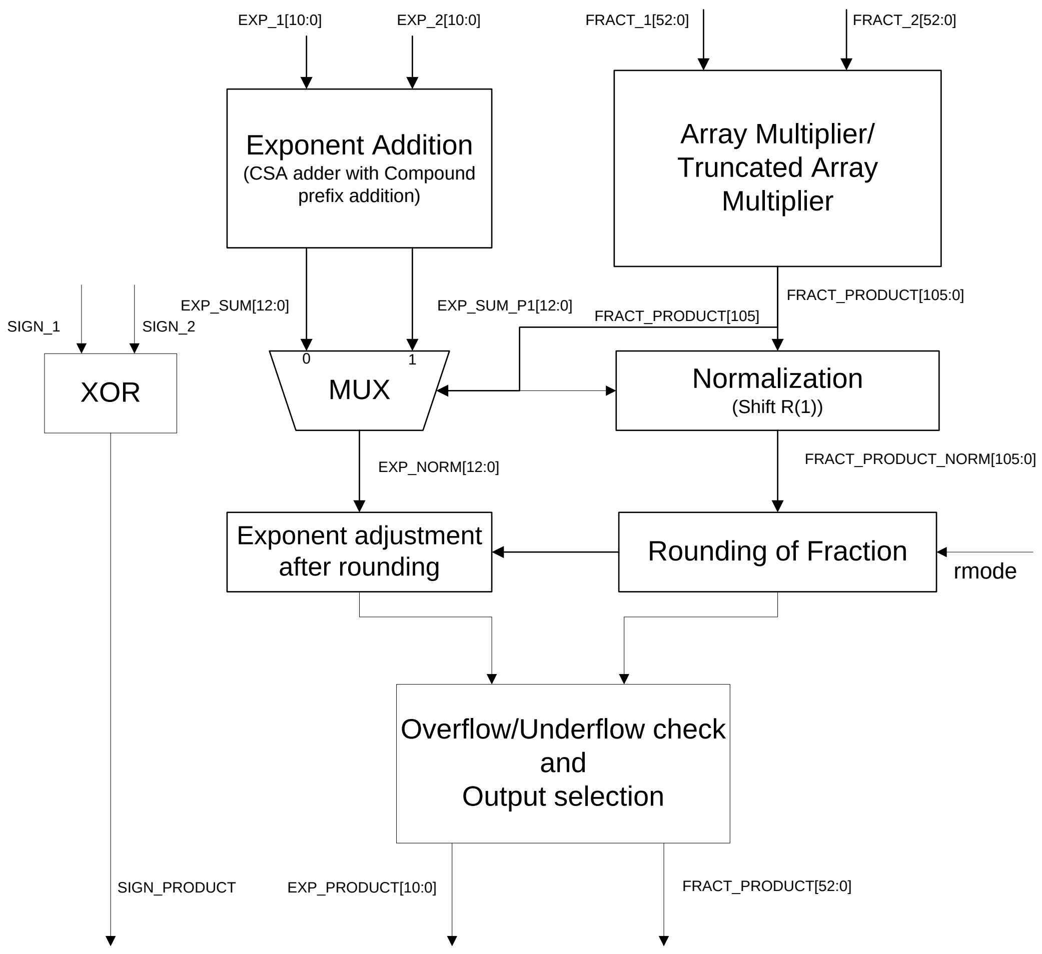

Figure 3.1: Double Precision Floating point multiplier

Sign of the product is positive if both the multiplicands are positive or both negative, negative otherwise. The algorithm and implementation of double precision Floating Point Multiplier is explained below.

Exponent Addition

1. As shown in the eqn.(3) we need to add both the exponents and subtract the *bias*(1023 for 64 bit floating point multiplier) value from the sum. This can be expressed in the following equation

$$EXP\_SUM[12] = EXP\_1[10:0] + EXP\_2[10:0] - 10'd\,1023 \tag{3.2}$$

2. A CS(Carry Save Adder) used to calculate the sum of EXP_1,EXP_2 and two's complement of 1023. Compressed the three input adder into two input adder using 3:2 compressor(Full Adder), and added them with the compound prefix adder to calculate EXP_SUM and $EX\_SUM\_P1(EXP\_SUM+1)$ concurrently. This way we can reduce one addition operation during normalization to adjust the exponent.

Fraction Multiplication

1. A full precision Array multiplier is used to compute the product of two fractions. I have used array multiplier as it have very less latency compared to sequential multiplier and less design time compared to tree multipliers while designing truncated multipliers of different size later. The Fig.3.2 shows the architecture for 4-bit array multiplier.

Normalization and Rounding

1. Both input fractions are normalized, hence $f_1 \in [1,2)$, $f_2 \in [1,2)$ and $f_{product} = f_1 \times f_2 \ \in [1,4)$. A normalization step(shift right) is needed to bring the fraction $f_{product}$ into [1,2). exponent is adjusted by selecting the EXP_SUM_P1 if right shift is needed to normalize, EXP_SUM otherwise.

2. The normalized fraction is rounded with standard IEEE rounding modes Round-to-Zero, Round-to-Nearest-Even, Round-to-plus-Infinite and Round-to-minus-Infinite.

3.3 Binary Multiplication

In this section we briefly describe different methods or algorithms to find product of to fixed point binary numbers. We focus mainly on unsigned multipliers as only unsigned multiplication is used in Floating Point Multiplication.

The multiplication can be divided into tw0 stages, 1) Partial product generation, 2)Summation of partial products.

3.3.1 Partial Product Generation

In order to achieve the final product value P , partial products are generated by multiplying the multiplicand by each of the multiplier bits, and assigning to the resulting output bits the weight obtained by combining the originating bits weight.

The partial product terms that build the partial product matrix for unsigned fixed-point multipliers can be obtained by adding a row of zeros whenever the corresponding multiplicand bit is zero, and adding a shifted version of the multiplicand whenever it is a zero. The process can be described as per the following equation

$$P = XY = \sum_{i=0}^{N-1}\sum_{j=0}^{N-1} x_i y_j 2^{i+j} = \sum_{i=0}^{2N-1} p_i 2^i. \tag{3.3}$$

The partial product terms are generated by combining the corresponding multiplicand and multiplier bits by making use of AND gates, and then aligned to their corresponding final weight.

3.3.1.1 Booth Encoding

The size of partial product matrices, derived from the large amount of digits required to represent numbers in binary formats, puts a big effort in the subsequent combination of the partial product terms.

The original algorithm presented by Andrew Booth in 1951, relied on a re-codification of the multiplier in a redundant number system. This results in fewer partial products to be combined, thereby improving the multiplier speed. Instead of generating the partial products using independent bits from the multiplicand, row-generating bits get partitioned in overlapping groups of three, each of them being decoded into a partial product as per the indicated selection table .

In general radix-4 Booth implementations result into (n + 2)/2 partial products, where n is the operand length. They can be obtained by shifting and two's complement negation (negate bits and add 1).

Multiplier bits	Selection
000	+0
001	+Multiplicand
010	+Multiplicand
011	+2×Multiplicand
100	-2×Multiplicand
101	-Multiplicand
110	-Multiplicand
111	-0

Table 3.1: Radix-4 Booth Encoding Table

Although improvements in area, power and timing are derived from the reduction in partial product terms and the subsequent combinatorial circuitry required to obtain the product value; area, timing and power overheads are added by the complex data recoding.

3.3.2 partial product combination

After generation of the partial product array, the partial product terms have to be added or combined together to form the final product value. Sequential multipliers are implemented as small architectures that recursively calculate and different groups of partial product terms. They offer the advantage of smaller area, thus reducing silicon requirements and static power consumption but require several clock cycles for performing a multiplication. The term parallel multipliers is used to refer to any multiplier that employs more than a single adder in the addition section, ideally resulting in a reduction of cycles required to combine the partial product terms equal to the implemented number of adders.

Full Parallel multipliers represent the case where the maximum possible level of parallelism is applied, implementing all the required partial products in parallel and performing their combination in bigger multi-operand addition blocks in a single cycle. Parallel multipliers are typically implemented using smaller carry save adders, in either array or tree structures.

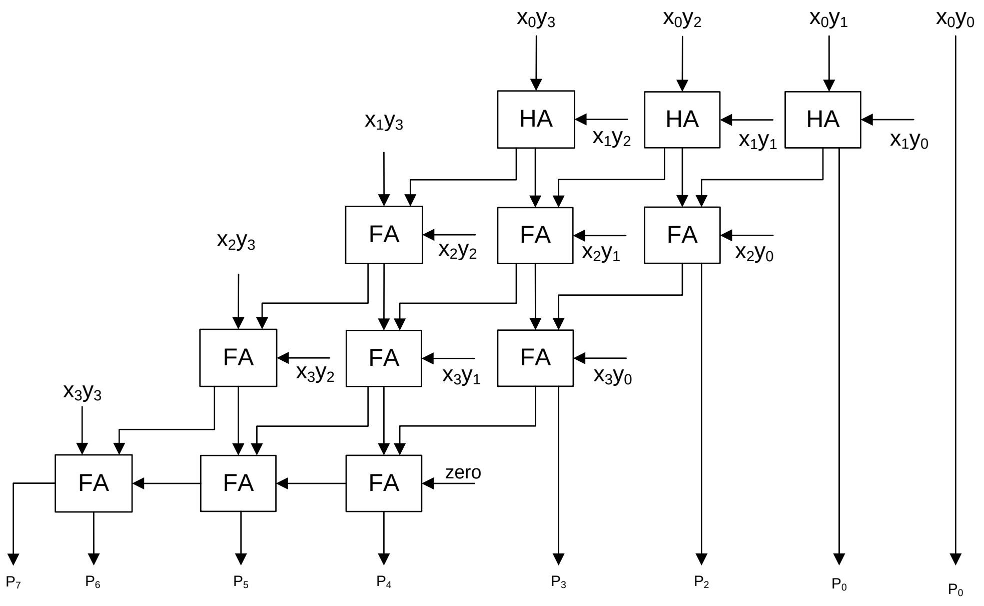

Figure 3.2: 4-bit Array Multiplier Implementation

3.3.3 Array Multipliers

Array multipliers have been extensively used in industrial applications as they represent a compromise between design effort and optimization, usually resulting in compact structures with a regular and simple layout . A classic array multiplier implementation consists of rows of FA cells where the outputs of each row is connected into the input ports of the following one. The delay of the array to produce the final carry is proportional to $2N - 2$, where N is the bit-width of the data.

Benefits of the array implementations include short wiring and tidy layouts while their major drawback is the linear increase on the structure delay. Improvements on the partial product combination can be achieved by applying tree structures that recombine their partial products according to different strategies at the expense of wiring complexity.

Even though tree multipliers have less combinatorial delay I chose array multiplier in our Floating Point Multiplier, because later in chapter 5 we need to generate truncated multiplier of different width(no. of output bits) for comparative study and design time for Array Multiplier is less compared to Tree Multiplier(Wallace Multiplier).

$$
\begin{array}{cccccccc|ccc|ccccc}
 & & & & & & & x_0y_0 & x_0y_0 & x_0y_0 & x_0y_0 & x_0y_0 & x_0y_0 & x_0y_0 & x_0y_0 & x_0y_0 \\
 & & & & & & x_0y_0 & x_0y_0 & x_0y_0 & x_0y_0 & x_0y_0 & x_0y_0 & x_0y_0 & x_0y_0 & x_0y_0 & \\
 & & & & & x_0y_0 & x_0y_0 & x_0y_0 & x_0y_0 & x_0y_0 & x_0y_0 & x_0y_0 & x_0y_0 & & & \\
 & & & & x_3y_0 & x_3y_0 & x_3y_0 & x_3y_0 & x_3y_0 & x_3y_0 & x_3y_0 & x_3y_0 & & & & \\
 & & & x_4y_0 & x_4y_0 & x_4y_0 & x_4y_0 & x_4y_0 & x_4y_0 & x_4y_0 & x_4y_0 & & & & & \\
 & & x_5y_0 & x_5y_0 & x_5y_0 & x_5y_0 & x_5y_0 & x_5y_0 & x_5y_0 & x_5y_0 & & & & & & \\
 & x_6y_0 & x_6y_0 & x_6y_0 & x_6y_0 & x_6y_0 & x_6y_0 & x_6y_0 & x_6y_0 & & & & & & & \\
x_7y_0 & x_7y_0 & x_7y_0 & x_7y_0 & x_7y_0 & x_7y_0 & x_7y_0 & x_7y_0 & & & & & & & & \\
\hline
p_{15} & p_{14} & p_{13} & p_{12} & p_{11} & p_{10} & p_9 & p_8 & p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0 \\
\hline
 & & & \multicolumn{2}{c}{\underbrace{\quad N \quad}} & & & & & \underbrace{N-h} & & & & \underbrace{\quad h \quad} & &
\end{array}
$$

Figure 3.3: 8-bit Truncated Multiplier

3.4 Floating Point Multiplier with Truncated Multiplier

In Floating point multiplication it is not necessary to compute the exact least significant part of the product, truncated multipliers allow power, area and timing improvements by skipping the implementation of sections of the least significant part of the partial product matrix. Instead of computing the full-precision output, the output results from the sum of the first $N+h$ columns (where $0 \leq h \leq N-1$) plus an estimation of the erased bits.

Fig.3.3 displays a generic partial product matrix, where the partial product matrix is split into two main regions, the least significant part (LSP), which contains the N least significant columns of the partial product matrix, and the most significant part (MSP), that includes the most significant columns of partial product terms. LAP can be also be split in two regions, $LSPmajor$ being the $N-h$ most significant column, and $LSPminor$ being the least significant columns of the partial product matrix.

The product resulting from a full-width multiplier, where the partial product is fully implemented thus resulting in an exact 2N bit result can be described as:

$$P_{full} = S_{MSP} + S_{LSP} \tag{3.4}$$

where S_{MSP} is the sum of the partial product bits belonging to MSP and S_{LSP} is the sum of those belonging to the LSP. In floating point multiplication, product values generated by fixed width $N \times N$ bit multipliers are truncated or rounded back to N bits.

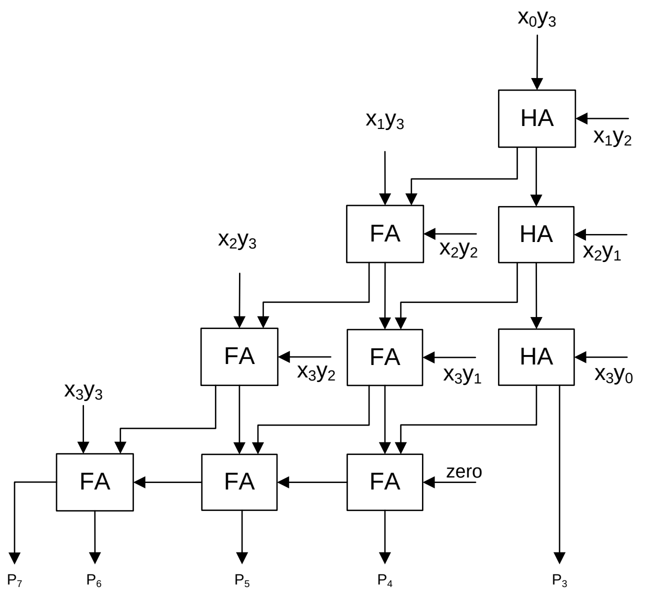

Figure 3.4: 4-bit Truncated Array Multiplier

Truncation allows a way of reducing the complexity of the multiplier unit by discarding the lower parts of the partial product matrix. This results in most of the error being generated in the lower weighted bits of the output that are discarded when converting the output back to the original bit width. By doing so, significant savings in power and complexity can be achieved, at the expense of error introduced into result.

The simplest scheme to obtain a truncated multiplier consists of removing the lower h columns of the partial product matrix that form the *LSPminor*, and use the bits in $LSPmajor$ for rounding. The maximum value of h(columns that can be removed) is $N-1$. It results in the multiplier requirements being almost halved in both area and power, at the expense of a big error with a strong negative bias being introduced in the multiplier output. Fig.3.4 shows a 4-bit truncated array multiplier.

The product of a Truncated multiplier is formed by the addition of the bits in the *MSP* and $LSPmajor$:

$$P_{D-Truncated} = S_{MSP} + S_{LSPmajor} \tag{3.5}$$

53-bit truncated array multipliers with number of truncated bits, "h" ranging from

46 to 52 are implemented and used in floating point multiplier. When $h \leq 46$, the maximum error introduced due to truncation is one LSB. This error is introduced during rounding from incorrect calculation of *sticky bit* or *round bit.* IF two 53-bit number are integers with out rounding, maximum error introduced into first $N+1(54)$ bits due to truncation of h bits given by eqn.6

$$Error_{trnc_{max}} = \left[\sum_{n=1}^{h} \frac{n.2^{n-1}}{2^{N-1}} \right] \tag{3.6}$$

A comparison between two Floating Point Multipliers, one with $h = 0$, and other with $h = 46$ is given in Table2.1.For $h = 47$, the maximum error $Error_{trnc_{max}} = 1$. Further error for random inputs is give in section1.6.

3.5 Testing

To verify the functional correctness of the implemented design, a functionally equivalent MATLAB program has been written and used to verify the functional correctness of the verilog design. For rounding mode, all special cases along with 8000 random generated numbers are given as inputs to MATLAB program and verilog design. Post-synthesis simulation is done using *Isim* and results are written into file. This result is compared with the results generated using MATLAB. Ideally, these two results should match, otherwise the design is functionally wrong.

3.6 Results

The use of Truncated multiplier for fraction calculation in Floating Point Multiplication results in less area on chip and less power. But as a result we get error in product. In this section several quantitative results have been produced to give an estimate of error introduced due to truncation of bits and area and power savings.

Area and power estimations are obtained from Xilinx Synthesis Tool(XST) and Xilinx Power Analyzer(XPE). Table 2 have the area, power and delay estimations of IEEE

	Floating Point Multiplier (Full precision Array Multiplier)	Floating Point Multiplier (Truncated Array Multiplier, n = 46)
Area (in LUTs)	4411	2802
Power (mW)	87.21	57.78
Critical Path Delay(ns)	68.480	71.122
Maximum Error	zero	*one LSB*

Table 3.2: Area, Power and Delay estimates of IEEE compliant FPM vs FPM with Array Multiplier.

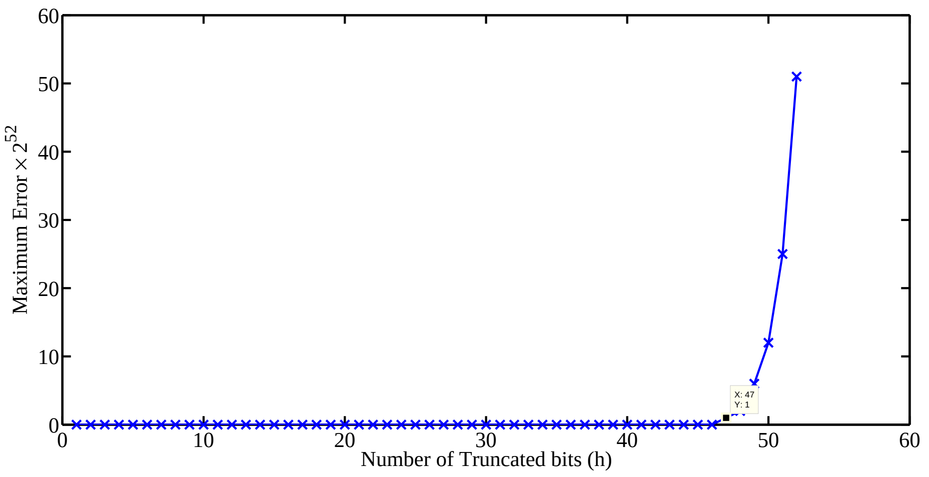

Figure 3.5: Maximum Error in Product Fraction in FPM with truncated multiplier

complaint Floating point multiplier (full precision Array multiplier and all four rounding modes).

For further analysis ans simulations we use only one rounding mode, i.e. Round-to-Nearest, a more accurate of rounding of four rounding modes.

The error estimated using MATLAB models of the Floating Point multiplier. By using truncated multipliers in place of full precision multipliers no error is introduced into exponent calculation. Hence Error we estimating is only error in fraction calculation. Fig.3.5 shows the maximum error introduced in first N+1 bits of product due to truncation of h LSB bits, where $0 \leq h \leq 52$.

If probability that a bit is 0 or 1 is equal to $\frac{1}{2}$, then probability that a product term equal to *zero* is $\frac{3}{4}$ and $\frac{1}{4}$ for it to be *one*. Expected error is calculated using the following

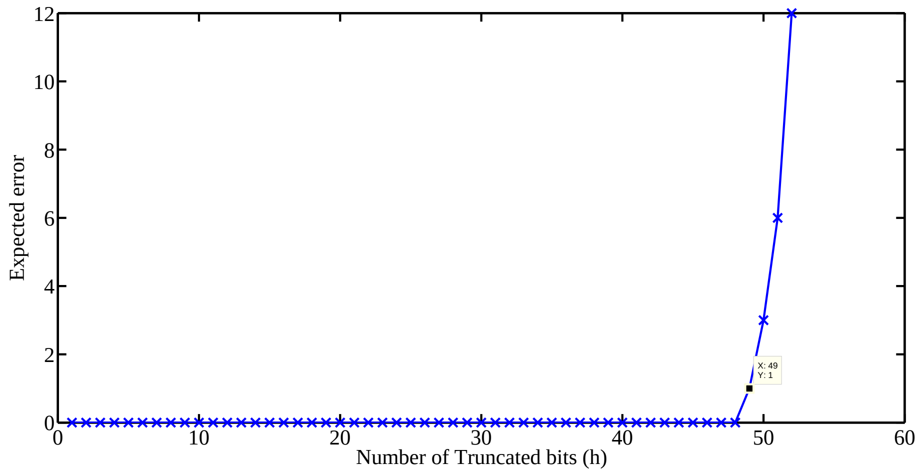

Figure 3.6: Expected Error in Product Fraction in FPM with truncated multiplier

probabilities

$$P(prod\_term = 1) = 1/4$$

$$P(prod\_term = 0) = 3/4.$$

Hence the expected error is :

$$Expected\ Error = \frac{Error_{trnc_{max}}}{4}.$$

Expected error is plotted against h(Number of Truncated bits) in Fig.3.6

Other than these estimates error is is measured for 2000 random inputs and average error is plotted is shown in Fig.3.7. Comparing Fig.3.6 and Fig.3.7 average error and estimated error are almost matching. From this we ma say that if we take sufficiently large random inputs, then the average error match with the estimate error.

Area and power analysis is done only for Floating Point Multipliers, with $h = 0$ or $47 \leq h \leq 52$. For $h \leq 46$ maximum error in fraction of product is 1 LSB. So we did Area and power analysis for truncated multiplier with $h = 46$ and is compare with Normal Floating Point Unit in Table2.1. Fig.3.8 and 3.9 shows that Area and Power estimations decreases with increase in the no of truncated bits.

Fig.3.9 shows that most of the power dissipated in FPM is dissipated in fraction

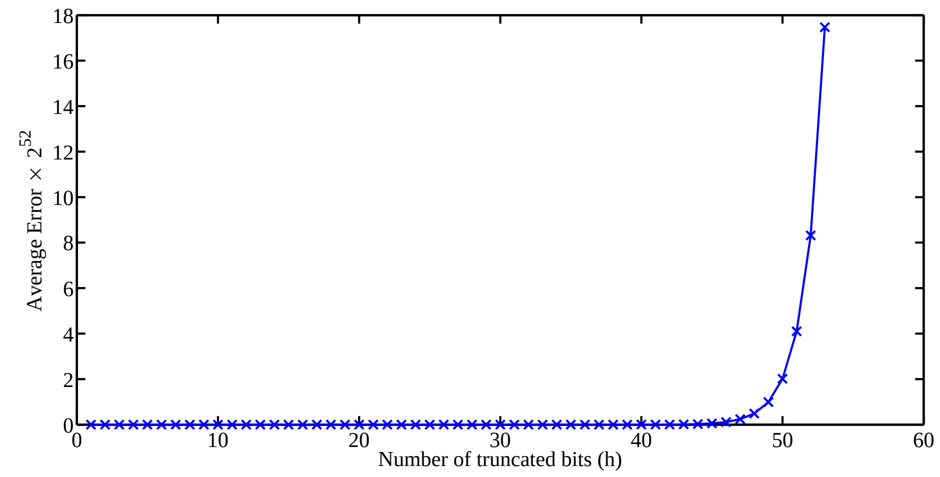

Figure 3.7: Average Error in Product Fraction in FPM with truncated multiplier

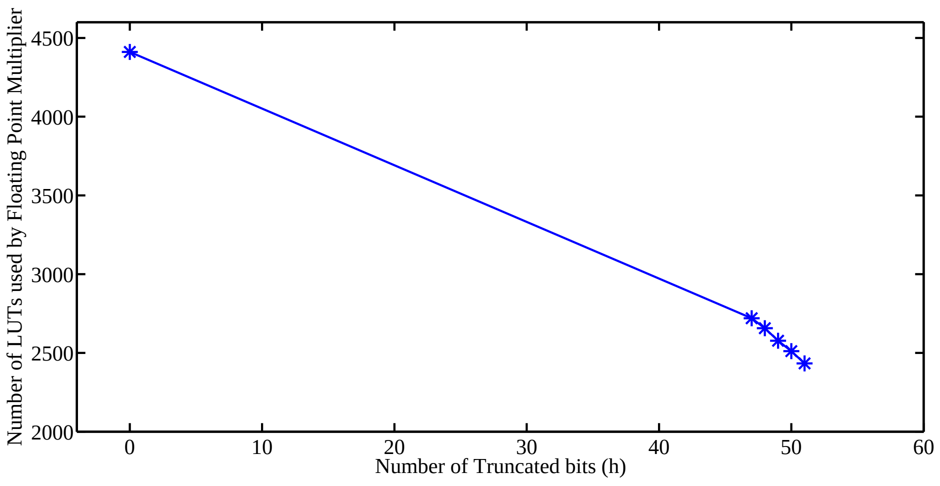

Figure 3.8: Decrease in area of FPM with Truncated Multiplier

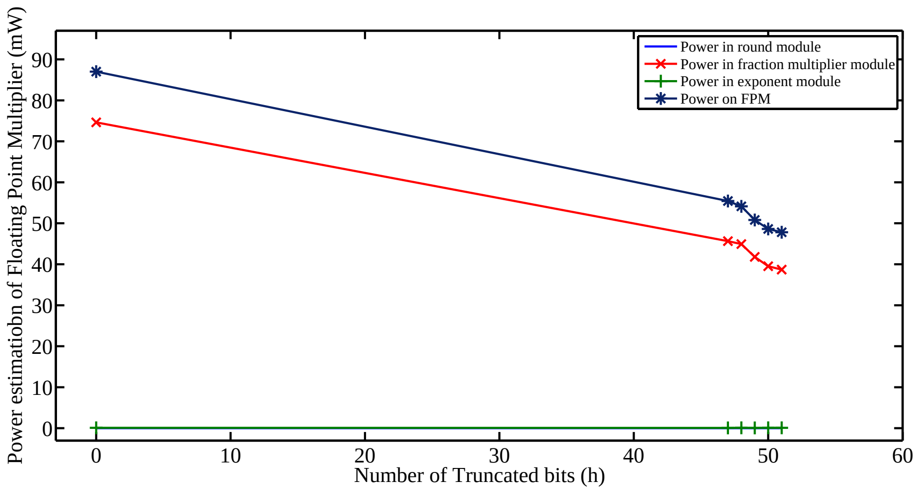

Figure 3.9: Decrease in Estimated Power of FPM with Truncated Multiplication.

multiplication.

Chapter 4

Floating Point Division

4.1 Introduction

Floating Point Division is one of the basic Floating Point arithmetic operations. The most complex part of Floating Point Division is calculation of quotient fraction. Division is most complex operation among multiplication, addition and division. In this chapter I will discuss the implementation of Floating Point Division using Newton-Phonograph method. I will discuss how we can gain improvements in area and power by replacing the Full Precision Multiplier with truncated Multiplier in division architecture.

4.2 Binary Division

There are two different approaches to the development of algorithms for high speed division. The more conventional approach uses add/subtract and shift operations, while the second relies on multiplication. The operation count in first approach is linearly proportional to the word size. The number of steps in the second approach(Division through Multiplication) is logarithmically proportional to the word size, but each individual step is more complex.

The most well known algorithm of the first type is the SRT division and High radix division. In general these algorithms(SRT) result in less area and power but latency for high width division is high compared to second approach. So for division between long binary number we use Division through Multiplication algorithms. So in this section we discuss further about division though multiplication.

4.2.1 Division Through Multiplication

4.2.1.1 Newton-Phonograph Method

In this method we first calculate the reciprocal of divisor D and then multiply it by the dividend to form the final quotient [1]. The reciprocal of D can be calculated using the Newton-Phonograph iteration method [8]. This is a method of finding the zero of a given function $f(x)$, where a zero of $f(x)$ is the solution of $f(x) = 0$. Let x_0 be the first approximation and x_i be the estimate for the zero at the ith step. The next estimate, x_{i+1}, is calculated from

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \tag{4.1}$$

where $f'(x)$ is the derivative of $f(x)$ with respect to x. For the function $f(x) = 1/x - D$, which has a zero at $x = 1/D$, $f'(x) = -1/x^2$, yielding

$$x_{i+1} = x_i(2 - D.x_i). \tag{4.2}$$

x_{i+1} converges to the reciprocal of D. Every iteration the error becomes roughly proportional to the square of the previous error. Hence, the number of significant figures of accuracy approximately doubles itself, which provides the property of quadratically convergence to the method.

The error of initial approximation is assumed to be E_{x_i}, such that:

$$E_{x_i} = x_i - 1/X \tag{4.3}$$

which can be written as;

$$x_i = E_{x_i} + 1/X \tag{4.4}$$

Inserting the obtained x_i into eqn.(19) will yield us

$$x_{i+1} = 1/X - XE_{x_i}^2 \tag{4.5}$$

Similar to eqn.(21), the error of reciprocal result maintained by first NR iteration can be defined as:

$$x_{i+1} = E_{x_{i+1}} + 1/X \tag{4.6}$$

Equalizing the equations (22) and (23), we obtain

$$E_{x_{i+1}} = -XE_{x_i}^2 \tag{4.7}$$

which proves us that the absolute error degrades quadratically in each NR iteration as it is proportional to the square of one previous error.

We may reduce the required number of iterations by reading the first approximation from table. This table is stored in a ROM, accepts the j most significant digits of D, and produces an approximation to $\frac{1}{D}$.

4.2.1.2 Goldschdmidt's algorithm

Goldschmidt division uses an iterative process to repeatedly multiply both the dividend and divisor by a common factor R_i to converge the divisor, D, to 1 as the dividend, N, converges to the quotient Q [1].

$$Q = \frac{NR_1R_2R_3....}{DR_1R_2R_3....} \tag{4.8}$$

The steps for Goldschmidt division are:

- Generate an estimate for the multiplication factor R_i
- Multiply the dividend and divisor by R_i
- If the divisor is sufficiently close to 1, return the dividend, otherwise, loop to step 1.

Assuming N/D has been scaled so that $0 < D < 1$, each R_i is derived from D_i: D_{i+1} and N_{i+1} are derived from R_i, D_i and N_i

$$R_i = 2 - D_i \tag{4.9}$$

D_{i+1} and N_{i+1} are derived from R_i, D_i and N_i

$$D_{i+1} = D_i \times R_i \tag{4.10}$$

$$N_{i+1} = N_i \times R_i. \tag{4.11}$$

$R_i = 2 - D_i$ is equivalent to finding two's complement to D_i. So totally two multiplications and one two's complement is required for each iteration. Here also number iterations can be reduced if can find a good initial approximation for $\frac{1}{D}$.

4.3 Division Implementation

Newton-Phonograph method is used to implement division among fractions of floating point numbers. As discussed above Newton-Phonograph method does division in two steps, 1) Inverse calculation of divisor, 2) Multiply it with dividend. In this section we discuss an architecture to compute division.

4.3.1 Determining the Table Look-up Values

A good initial approximation gives result in less no of iterations, hence less latency for the division. For good initial estimate we have used a modified piecewise linear approximation based on the first-order Taylor expansion is used [7].

The modification of the operand and the determination of the ROM values are carried out based on Taylor series expansion. The operand, X, is a 64-bit normalized double precision floating-point number in the range of $1 \leq X < 2$. The hidden-one and the least significand 52-bit of X represents mantissa. The 53-bit mantissa represented as:

$$X_{mantissa} = [1.x_1 x_2 x_3 \ldots . x_{52}] \tag{4.12}$$

Approximated version of the Taylor series expansion by truncating the series after the first derivative term can be represented as :

$$f(x_{i+1}) = f(x_i) + f'(x)(x_{i+1} - x_i)$$

In order that the reciprocal function X^{-1} to be represented by Taylor series expansion, the operand, X , can be split into two parts from m^{th} bit to $2m^{th}$, where $m < 26$, such that [4]:

$$X_{m\_1} = [1.x_1 x_2 x_3 \ldots . x_m]$$

$$X_{m\_2} = [0.x_{m+1} x_{m+2} \ldots . x_{2m}] \times 2^{-m} \tag{4.13}$$

$$X_{mantissa} \approx X_{m\_1} + X_{m\_2}$$

The initial reciprocal approximation, X^{-1}, computed by the following equation

$$X^{-1} = (X_{m\_1} + 2^{-m-1})^{-1} - (X_{m\_1} + 2^{-m-1})^{-2}(X_{m\_2} - 2^{-m-1}) \tag{4.14}$$

where, $f(x_i) = (X_{m\_1} + 2^{-m-1})^{-1}$, $f(x_{i+1}) = X^{-1}$ and

$$\begin{aligned} x_{i+1} - x_i &= X - (X_{m\_1} + 2^{-m-1})^{-1} \\ &= (X_{m\_2} - 2^{m-1}). \end{aligned} \tag{4.15}$$

By rewriting eqn.(6), X^{-1} can also expressed as

$$X^{-1} = (X_{m\_1} + 2^{-m-1})^{-2}[(X_{m\_1} + 2^{-m-1}) - (X_{m\_2} - 2^{-m-1})]. \tag{4.16}$$

In the above equation the first term $(X_{m\_1}+2^{-m-1})^{-2}$, will be read from ROM as a constant term. The first m bit of the mantissa will constitute as the selecting bits of table look-up procedure. The remaining term of eqn.(10), $[(X_{m\_1}+2^{-m-1})-(X_{m\_2}-2^{-m-1})]$, will be formed by operand modifier. The operation of operand modifier is bit wise inversion of the bits from $(m+1)^{th}$ to $2m^{th}$; where from 1^{st} to m^{th} remain unchanged.

$$C=(X_{m\_1}+2^{-m-1})^{-2} \tag{4.17}$$

$$X'=[1.x_1x_2x_3...x_m\overline{x_{m+1}x_{m+2}x_{m+3}...x_{2m}}] \tag{4.18}$$

Therefore, the initial approximation of X^{-1} is computed with accuracy of $2m\pm3$ by multiplication of term C (read from ROM) with modified operand X'.

$$X^{-1}=C.X' \tag{4.19}$$

4.3.2 Division Unit Architecture

The division unit is shown in Fig.4.1. The first stage is to obtain the initial approximation for the reciprocal of X by the help of table look-up [7]. At the beginning, the first m bit from the mantissa of X is taken as the selection bits of the ROM. We have decided on , such that; $m=0$ for which the ROM size would be $2^{10}\times 20$ bits. It is desired to keep the ROM size minimum, in order to prevent the cost of hardware.

Here are the computations in every cycle

Cycle 1: The first twenty ($2m=20$) bits of X is supplied to the operand modifier unit. The operand modifier itself is constructed by ten inverters, in which the most significand ten bits stays the same, whereas the least significand ten bits are bit wise inverted.The ROM output of twenty bits are concatenated by thirty-four zeros in order to be suitable for 54 x 54 multiplication. Same procedure is applied to operand modifier output as well. These two 54-bit values constitute C and X' in eqn.(47), respectively.

Cycle 2: C and X' values are selected by the multiplexors MUX1 and MUX2 and

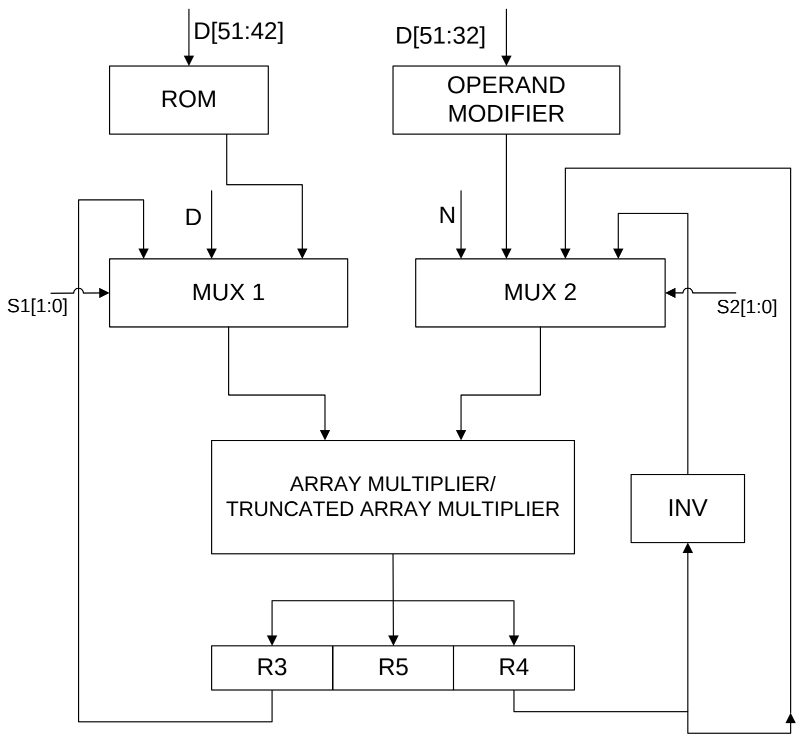

Figure 4.1: Division Unit

multiplied using Array Multiplier. The result containing the most significant 54-bits of the sum (around forty bits of them contain the true result, since two twenty-bit numbers are multiplied until now) are written into registers R3 and R4. At the end of the cycle, an initial approximation of reciprocal, i.e., x_i is obtained.

Cycle 3: In this clock cycle, the 52-bit mantissa of is selected by MUX1 because the original number is required to be multiplied with the initial approximation during Newton-Raphson iterations, as to carry out Xx_i operation, which is explicitly stated in eqn.(19). The selection of X via MUX1 and x_i via MUX2 are multiplied. The result is stored in R4.

Cycle 4: The result is stored in the register R4 is inverted and supplied back to the MUX2. This inversion is necessary to compute $(2 - Xx_i)$ in eqn.(19), which is obtained in the third clock cycle. $(2 - Xx_i)$ is selected by MUX2 and the initial approximation result, x_i , is selected by MUX1, which is available in register R3 and multiplied in this cycle. This concludes the first Newton-Phonograph iteration.

Cycle 5: Again, the 52-bit mantissa of is selected by MUX1 because the original number is required to be multiplied with the initial approximation during Newton-Raphson iterations, as to carry out Xx_i operation, which is explicitly stated in eqn.(19). The selection of X via MUX1 and x_i via MUX2 are multiplied. The result is stored in R4.

Cycle 6: The result is stored in the register R4, inverted and supplied back to the MUX2. $(2-Xx_i)$ is selected by MUX2 and the initial approximation result, x_i , is selected by MUX1, which is available in register R3 and multiplied in this cycle. This second Newton-Phonograph iteration is completed. Inversion of divisor is completed.

Cycle 7: Inversion of divisor is selected by MUX1 from R3 register, dividend is selected by MUX2 and the product is stored in R5.

4.4 Floating Point Division

Floating point division is very similar to floating point multiplication, yet floating point division is very less frequently used and computationally complicated due to its Division unit in palace of multiplier in Floating point multiplier. Division is the most complicated and computationally intensive among basic arithmetic operations addition, subtraction, multiplication and division. In the following section Division is implemented using Newto-Raphson method.

Fig.1.1 shows block diagram for Floating Point Division. This Floating Point Division unit is designed for double precision, normalized floating point numbers. The division of two floating point numbers can be represented by the following equation

$$(-1)^{sign_{quotient}}1.f_{quotient}.2^{e_{quotient}} = (-1)^{sign_1 \oplus sign_2}(1.f_1 \div 1.f_2).2^{(e_1-e_2+bias)}. \qquad (4.20)$$

Sign of the quotient is positive if both divisor and dividend are positive or both negative, negative otherwise. The algorithm and implementation of double precision Floating Point Division unit is explained below.

Exponent Subtraction

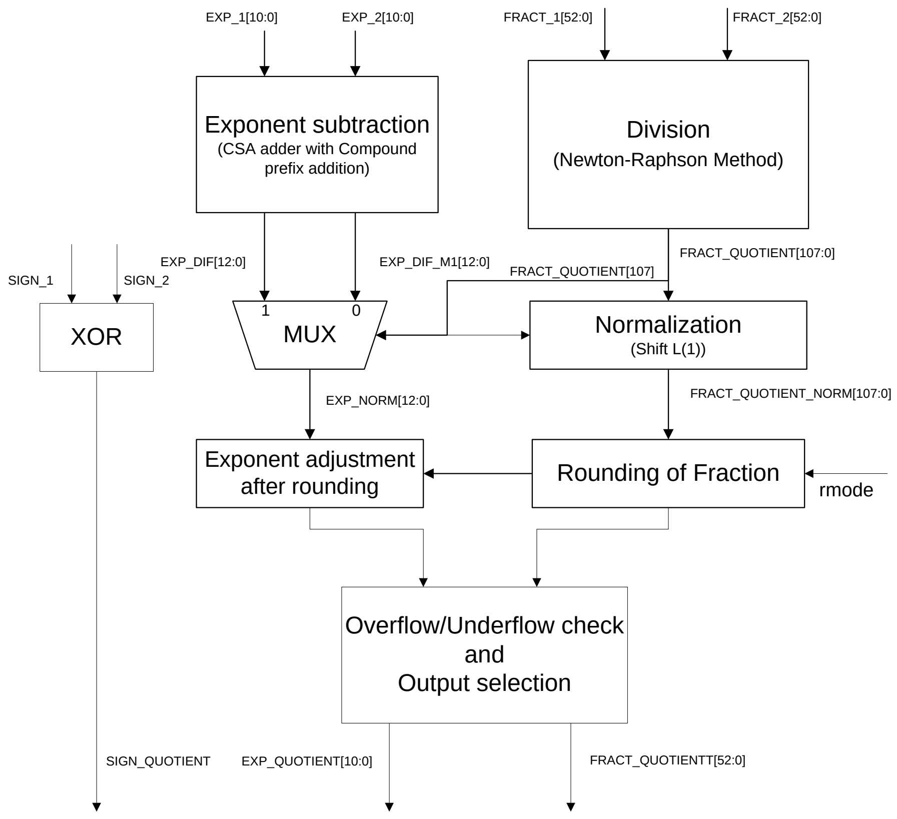

Figure 4.2: Block Diagram for Floating Point Division

1. As shown in the eqn.(1) we need to subtract the exponent of divisor, from exponent of dividend and add *bias*(1023 for 64 bit floating point multiplier) value to the difference. This can be expressed in the following equation

$$EXP\_DIF[12] = EXP\_1[10:0] - EXP\_2[10:0] + 10'd\,1023 \tag{4.21}$$

2. A CS(Carry Save Adder) used to calculate the difference of EXP_1,EXP_2 and add 1023. Compressed the three input adder into two input adder using 3:2 compressor(Full Adder), and added them with the compound prefix adder to calculate EXP_DIF and $EX\_DIF\_M1(EXP\_DIF - 1)$ concurrently. This way we can reduce one subtraction operation during normalization to adjust the exponent.

Fraction Division

1. Newton-Phonograph method is used to find the quotient. Newton-Phonograph method

first reciprocal of divisor is obtained and is multiplied with dividend. In sec.3 Newton-Phonograph method for quotient calculation is discussed in more detailed.

Normalization and Rounding

1. Both input fractions are normalized, hence $f_1 \in [1,2)$, $f_2 \in [1,2)$ and $f_{quotientt} = f_1 \div f_2 \in [1/2,2)$. A normalization step(shift left) is needed to bring the fraction $f_{quotient}$ into [1,2). exponent is adjusted by selecting the EXP_DIF_M1 if left shift is needed to normalize, EXP_DIV otherwise.

2. The normalized fraction is rounded with standard IEEE rounding modes Round-to-Zero, Round-to-Nearest-Even, Round-to-plus-Infinite and Round-to-minus-Infinite.

4.5 Division with Truncated Multipliers

In the above Newton-Raphson Division Algorithm, during the repetitive iterations only 54 MSB bits of the product term is used as input to the multiplier in next cycle. The computation of 54 LSB bits does not effect the result. By using a truncated multiplier in place of full precision multiplier we can save both Area and Power [9]. The truncation of LSB bits will introduce some error into the computation. And Newton-Raphson Division algorithm is a self correcting algorithm, So the effect of error introduced by truncated multiplier is further reduced by the algorithm itself. In the next section error analysis and Area and Power estimations for several Floating point multipliers with different width truncated array multipliers.

4.6 Results

Fig.4.3 shows that maximum error in first 54 bits due to truncation is zero till $h = 48$. hence we did the area and power analysis only for $h \geq 48$

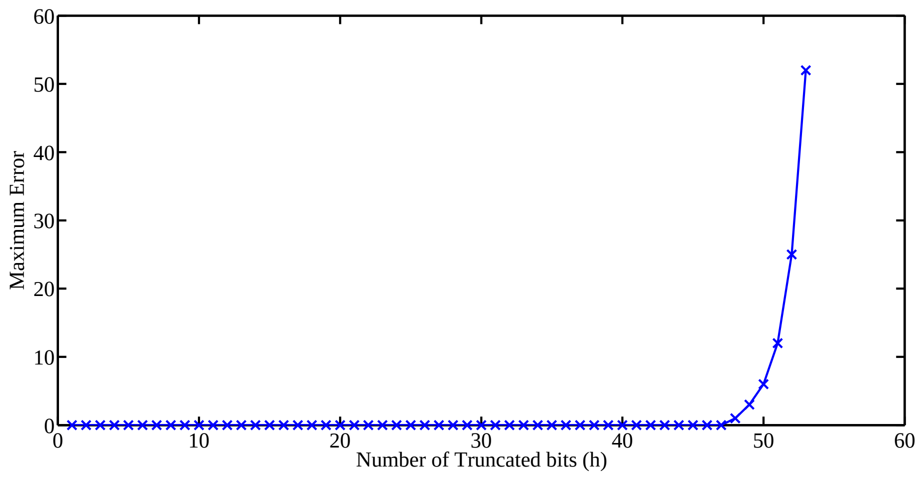

Figure 4.3: Expected Error in 54 bit Truncated Multiplier with *h* truncated bits

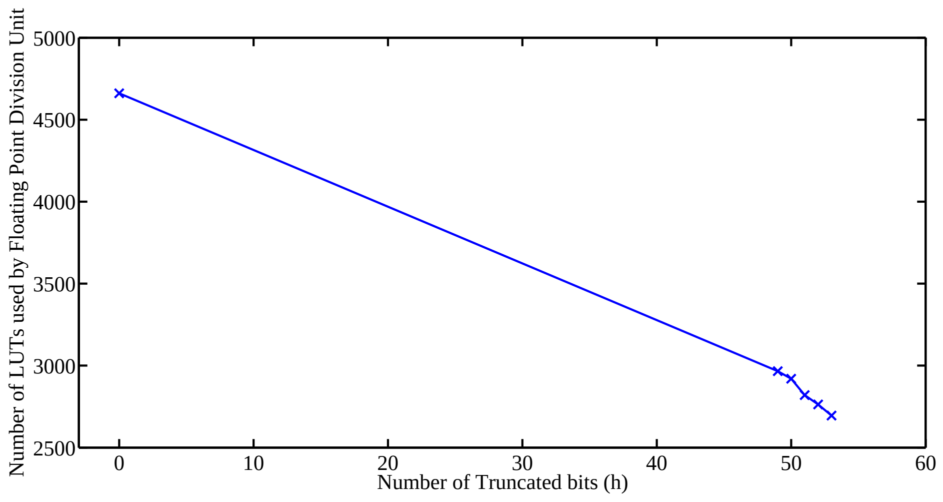

Figure 4.4: Area Reduction in FPM due to Truncated multipliers

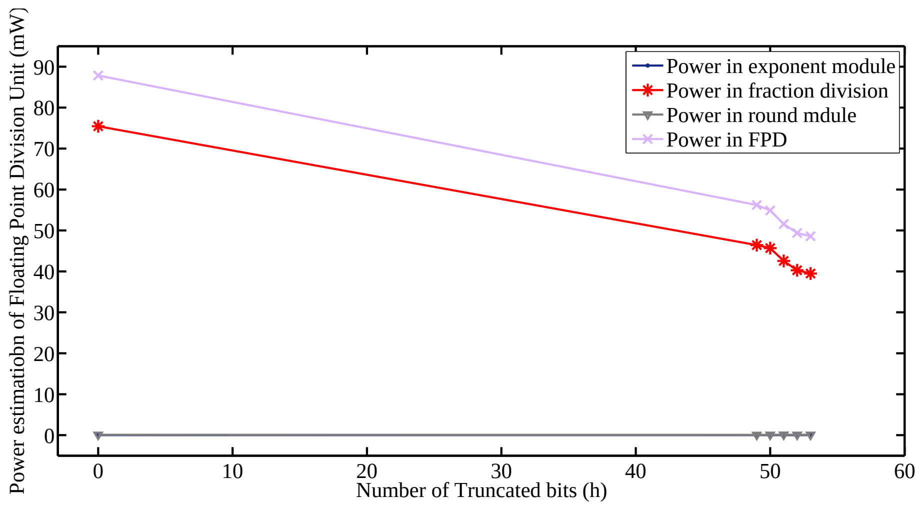

Table 4.1: Power Reduction in FPM due to Truncated multipliers

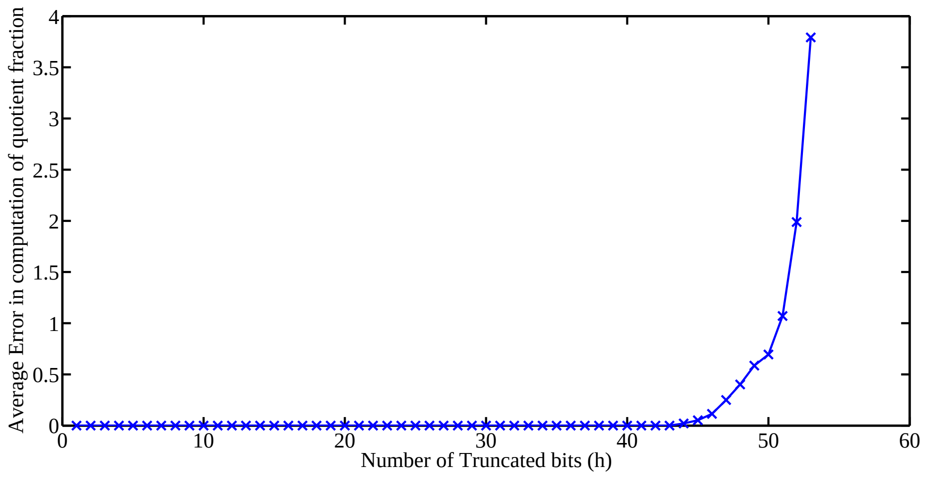

Figure 4.5: Average Error Introduced into Quotient fraction

Average error in quotient fraction computation is plotted w.r.t. number of truncated bits (h) for 500 random inputs. Fig.4.5 shows that average error in case of FPD is less than the average error in case of FPM. This is due to self correcting nature of Newton-Raphson method.

Chapter 5

Conclusion

In this report several approximations for Floating Point Arithmetic operations are discussed. For Floating Point Addition reducing the size of barrel right shifter to 54 bits from 118 bits reduces the area and power estimation of barrel right shifter by half, but it introduces maximum error of $1ulp$. But reduction of no of selection signal of barrel shifter introduces large errors compared to other approximation methods.

Truncated multipliers for Floating Point Multiplication and Floating Point Division introduces very little error till number of truncated bits, $h \approx 47$, and results in Good area and power savings. But when number of truncated bits larger than 47, error increases more steeply with respect to number of truncated bits(h). Error in case of Floating Point Division is less compared to error in Floating Point Multiplication for same no of truncated bits. This is due to self correcting nature of Newton-Raphson method. So the truncated multiplier can be used in functional approximation of other non liner functions also, in place of full precision multiplier.

Bibliography

[1] Israel Korean, “Co muter Arithmetic Algorithms”, University Press, 2003. 5, 4.2.1.1, 4.2.1.2

[2] P.M. Sideline, Guy Even, “Delay-Optimized Implementation of IEEE Floating-Point Addition”, *IEEE Trans. Computers.*, vol. 53, no. 2, pp. 97-113, Feb.2004 1, 2, 2.2.1

[3] P. Farmwald, “On the Design of High Performance Digital Arithmetic Units,” PhD thesis, Stanford Univ., Aug. 1981. 2.2.1, 2.2.1

[4] R. Brent and H. King, “A Regular Layout for Parallel Adders,” IEEE Trans. Computers, vol. 31, no. 3, pp. 260-264, Mar. 1982. 4, 2.2.4

[5] Institute of Electrical and Electronics Engineers, New York, NY. ANSI/IEEE 754-1985 Standard for Binary Floating-Point Arithmetic, 1985

[6] Manuel DE la Guiana Solar, “Energy Efficient Design of Truncated Multipliers”, *P.D thesis,* Department of Electronic and Computer Science, University of Limerick, Sept.2011

[7] Smut Habakkuk, Ahmed Akkas, “Design and Implementation of Reciprocal Unit Using Table Look-up and Newton-Raphson Iteration” 4.3.1, 4.3.2

[8] A. Bjorck G. Dahlquist and N. Anderson eds. Numerical Methods. Prentice-Hall Inc., 1974. 4.2.1.1

[9] E. George Walters III, Michael J. Schulte, “Efficient Function Approximation Using Truncated Multipliers and Squarers”, Proceedings of the 17th IEEE Symposium on Computer Arithmetic (ARITH’05), 2005. 4.5

[10] A. Tyagi, “A Reduced-Area Scheme for Carry-Select Adders,” IEEE Trans. Computers, vol. 42, no. 10, Oct. 1993.

4, 2.2.4