Design and Implementation of Approximate Logarithmic Multipliers

A dissertation submitted in partial fulfillment of the requirements

for the degree of Master of Technology in

VLSI Design

By

ANUPAM HASSA PURTY

19MVD0005

Under the guidance of

Prof. Rajeev Pankaj Nelapati

Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

Department of Micro and Nano Electronics

School of Electronics Engineering
VIT, Vellore
Tamil Nadu 632014, India
May 2021

Certificate

This is to certify that the dissertation entitled ***Design and Implementation of Approximate Logarithmic Multipliers*** submitted by ***ANUPAM HASSA PURTY*** **(Reg. No. 19MVD0005)** to VIT Vellore, in partial fulfullment of the requirement for the award of the degree of **Master of Technology** in **VLSI Design** is a bona-fide work carried out under my supervision. The dissertation fulfills the requirements as per the regulations of this University and in my opinion meets the necessary standards for submission. The contents of this dissertation have not been submitted and will not be submitted either in part or in full, for the award of any other degree or diploma and the same is certified.

The thesis is satisfactory / unsatisfactory

Examiner

Signature: ....

Date:

(Seal of the School)

Approved by

Head of the Department

Department of Micro and Nano Electronics
School of Electronics Engineering
VIT University Vellore

Declaration

I hereby declare that the dissertation ***Design and Implementation of Approximate Logarithmic Multipliers*** submitted by me to the Department of Micro and Nano Electronics, School of Electronics Engineering VIT Vellore, in partial fulfillment of the requirements for the award of **Master of Technology** in **VLSI Design** is a bona-fide record of the work carried out by me under the supervision of ***Prof. Rajeev Pankaj Nelapati***. I further declare that the work reported in this dissertation, has not been submitted and will not be submitted, either in part or in full, for the award of any other degree or diploma of this institute or of any other institute or University.

Place : Vellore

Anupam Hassa Purty
(19MVD0005)

Acknowledgements

I wish to express my sincere thanks to **Chancellor Dr. G.Viswanathan, Vice President Mr. G.V.Selvam, Vice Chancellor Dr. Rambabu Kodali and Pro Vice-Chancellor Dr. S.Narayanan** for providing me an excellent academic environment and facilities for pursuing M.Tech. program. I am grateful to **Dr. Kittur Harish Mallikarjun, Dean of School of Electronics Engineering**, VIT Vellore and to **Dr. Kumaravel S, HOD(Department of Micro and Nano Electronics)**. I wish to express my sincere gratitude to **Dr. Rajeev Pankaj Nelapati**, Faculty Guide for providing me an opportunity to do my Thesis work. I am also thankful to VIT Vellore for providing software platforms (Modelsim and Quartus Prime) to carry out my project work and help me gain knowledge. I thank my family and friends who motivated me during the course of the project.

Place : Vellore

Anupam Hassa Purty
(19MVD0005)

Abstract

A lot of digital applications especially in signal processing often are very highly dependent on large number of multiplications which comes with high design complexity, delay and power consumption. There are solutions available in the market like adder-series, combination of ripple carry adders, adder trees and many more which are costly in terms of power and area. Logarithmic multipliers are an optimal solution as they are faster with low power but introduce a lot of errors. Still, they can be used in applications where speed and area is a higher priority than accuracy like in Digital Signal Processing applications. In this work a simple and efficient 16 bit logarithmic multiplier is presented which can achieve a better accuracy by an iterative method. This method introduces a trade-off between accuracy and area. Larger area leads to high power dissipation. Pipelining is used to reduce error by iteration. The hardware consists of basic adders and shifters which does not contribute to high design complexity. Also a 16 bit Non-Iterative logarithmic multiplier is presented which gives better accuracy than the previous one but with higher design complexity. The designs are implemented using Verilog Hardware Description Language and simulated in Cyclone IV FPGA Board. The multipliers are used to find product of two 16 bit operands with relatively very low error percentages. The Iterative Logarithmic Multiplier offers highest accuracy due to more number of iterations but at the cost more area and higher power dissipation. The Non-Iterative Logarithmic Multiplier shows better accuracy than the basic Iterative Logarithmic Multiplier with no iterations but at the cost of high power dissipation due to its design complexity.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

FPGA	Field Programmable Gate Array
ILM	Iterative Logarithmic Multiplier
NILM	Non-Iterative Logarithmic Multiplier

Chapter 1

Introduction

Multiplication is one of the most common and important operations in computer arithmetic. It consists of a long series of adder circuits which takes up lot of area and hence generating lot of delay and power. Examples: Look Ahead Carry Adders, Carry Save Adders, Adder trees etc. For high accuracy Digital Signal Processing applications most of the circuits use lot of designs which contributes to high design complexity. To achieve very low relative error more amount of error correction circuits are introduced which increases the area drastically. This results in very high power dissipation which may permanently damage the entire circuit. Hence accuracy plays a huge factor in the implementation of design which directly contributes to its area. For low accuracy applications where speed is the priority, we can reduce the area and design complexity by removing most of the gates and error correcting circuits hence resulting in very low Power dissipation and faster circuit. Examples: cryptography, machine learning etc. Approximate Computing is a method where it fulfils all the needs of any low accuracy applications. It results in very low design complexity hence easier to implement with the added benefit of a faster circuit with very low power dissipation. It can be also called as high energy efficient circuit. For Digital Signal Processing applications like video or image compression where product of input operands is used approximate multipliers can be implemented to achieve greater speeds with lesser power and design complexity. Using limited adder circuits to achieve approximate results gives a huge range of errors with high power dissipation. Hence, logarithmic multipliers are presented to improve the accuracy and reduce power delay product.

Truncated multipliers are extensively used in digital signal processing where the speed of the multiplication and the area and power-consumptions are important. However, as mentioned before, there are many applications in DSP where high accuracy is not important. The basic idea of these techniques is to discard some of the less significant partial products and to introduce a compensation circuit to reduce the approximation error Logarithmic multiplication introduces an operand conversion from integer number system into the logarithm number system (LNS). The multiplication of the two operands N1 and N2 is performed in three phases, calculating the operand logarithms, the addition of the operand logarithms and the calculation of the antilogarithm, which is equal to the multiple of the two original operands. The main advantage of this method is the substitution of the multiplication with addition, after the conversion of the operands into logarithms. LNS multipliers can be generally divided into two categories, one based on methods that use lookup tables and interpolations, and the other based on Mitchell's algorithm (MA), although there is a lookup-table approach in some of the MA-based methods. Generally, MA-based methods suppressed lookup tables due to hardware-area savings. However, this simple idea has a significant weakness: logarithm and anti-logarithm cannot be calculated exactly, so there is a need to approximate the logarithm and the antilogarithm.

Logarithmic number system is considered because it provides us with useful properties like reducing a complicated multiplications into smaller additions. Also it reduces the powers of any numbers to smaller multiplications. These properties can be used in a radix-2 multiplications like binary number system also. The only problem involves is the uses of logarithmic and anti-logarithmic converters. This concept was used in multipliers in the earlier days. Later Mitchell's multiplication algorithm is used which converts any number to an approximate logarithmic number by simple shifting of bits. This reduces the need of any logarithmic or anti logarithmic multipliers hence reducing design complexity drastically. This work involves the use of a modified version of the Mitchell's algorithm. An iterative method is applied for 16 bit multiplication which reduces the relative error. This achieved through pipelining[12]. Also a 16 bit Non-Iterative multiplier is also presented which offers higher accuracy and lesser area compared to the iterative logarithmic multiplier but at the cost of design complexity and power dissipation[9]. The final designs are implemented in Verilog Hardware Description Language and simulated in Cyclone IV FPGA board with the help of software platforms. Error comparisons are made for few sample inputs. Comparative analysis is also done for all the design metrics.

1.1 Outline of the Thesis

- Chapter 2 consists of Literature Survey of the papers related to this work.
- Chapter 3 shows all the previous techniques used for logarithmic multiplication.
- Chapter 4 has all the description and hardware design of an Iterative Logarithmic Multiplier.
- Chapter 5 has all the description and hardware design of an Iterative Logarithmic Multiplier.
- Chapter 6 shows a comparative analysis of error% and device utilization of both iterative and non iterative logarithmic multipliers.
- Chapter 7 concludes the entire work.

Chapter 2

Literature Survey

2.1 Computer Multiplication and Division Using Binary Logarithms

This paper introduces a method for simpler computer multiplication and division using binary logarithms. The logarithm of any binary number can be found out approximately by simple shifting and counting of bits[11]. Finally, just one addition or subtraction operation is required after shifting to obtain product and quotient respectively. Small errors can be introduced as it uses approximate results. A small example is shown below for division operation[11].
Example: Division of 3216 by 25

$$\begin{array}{ccr} 3216 & = & 110010010000 \\ 25 & = & 000000011001 \\ log\ 3216 & = & 1011.1001001 \\ log\ 25 & = & 0100.1001 \\ log\ 3216 - log\ 25 & = & 0111.0000001 \\ Result & = & 10000001 \\ & = & 129 \\ \text{Error} & = & 0.27\% \end{array}$$

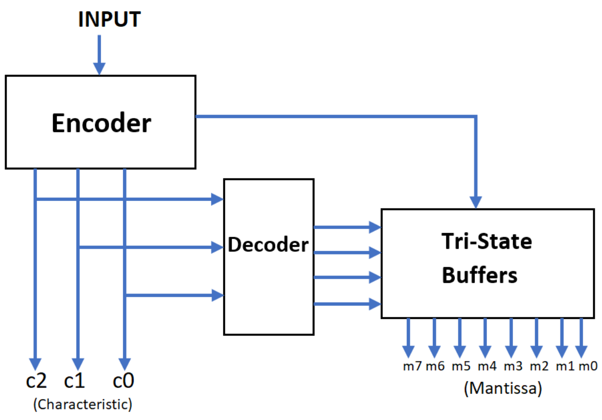
In the above example log of 3216 is obtained by first writing the bit position of most significant one which is 13 = 1011 and then placing the decimal point after the most significant one. The last few zeros are truncated to keep the number of bits same as in 3216. Similar operation is done for 25. As this is a division operation the binary log numbers are subtracted to get the result. The error obtained is very low. This is not true for all values. Consider the example below[11]:
Example: Division of 15 by 3

$$\begin{array}{ccr} 15 & = & 1111 \\ 3 & = & 0011 \\ log\ 15 & = & 11.111 \\ log\ 25 & = & 01.100 \\ log\ 15 - log\ 3 & = & 10.011 \\ Result & = & 101.1 \\ & = & 5.5 \\ \text{Error} & = & 10\% \end{array}$$

This shows that this method is not completely efficient for all numbers. In this method the only difference is multiplication operation is addition of binary logarithms. The multiplication error can be as large as −11.1% but this can be reduced by double or higher order precision. The division error can be as large as 12.5%. Errors for any particular type of operation cannot be further reduced by this method as the error rate increases with the number of bits[11].

2.2 CMOS VLSI Implementation of a Digital Logarithmic Multiplier

This paper introduces a practical approach for the implementation of a logarithmic multiplier. Approximate logarithm is obtained by simple shifting and counting of bits. General comparison is done where the approximate logarithms are stored in a Rom chip which leads to high design complexity. Further Logarithmic and Antilogarithmic converters are designed and implemented to produce approximate results which leads in larger area and power dissipation. The block diagram



(Source:[1])

Figure 2.1: Binary to Logarithm Conversion Circuit

of the binary to Logarithm module is shown in fig 2.1[13]. The input is fed to an encoder which generates the 3 bit characteristic of the logarithm by detecting the most significant 'one' bit position. The encoder output is fed to a control module which controls a bank of tristate buffers used to extract the bits to the right of the most significant 'one' of the input. Thus the output of the encoder[13].

2.3 An Efficient and Accurate Logarithmic Multiplier based on Operand Decomposition

This paper gives information of the disadvantages of using Mitchell's Algorithm as it introduces high errors due to piecewise straight-line approximation[13]. Operand decomposition method is proposed as a modification to the Mitchell's algorithm where the number of bits with the value '1' will be decreased in the multiplicands. This reduces the amount of approximation[1]. The operand decomposition of two n-bit binary numbers X and Y to four n-bit binary numbers A, B, C and D is performed using the following equations, A = X | Y, B = X & Y, C = X' & Y and D = X & Y' where | indicates OR operation, & indicates AND operation and ' indicates complement operation[10].

The product of the number X and Y is then computed from the decomposed operands using the

X =	00010010	=	18		Y =	00111100	=	60
A =	00111110	=	62		C =	00101100	=	44
B =	00010000	=	16		D =	00000010	=	02

equation :

$$\begin{aligned} X \times Y &= (C \times D) + (A \times B) \\ &= ((62 \times 16) + (44 \times 2)) \\ &= 1080 \end{aligned}$$

The approach increases the number of zero bits in the decomposed multiplications and hence decreases the switching power in the multiplication operation[10].

2.4 A Simple Pipelined Logarithmic Multiplier

This paper gives us an iterative approach to the previous versions of Mitchell's Algorithm. Pipelining is introduced for the implementation of an iterative multiplication[6]. The author claims to achieve relatively low error percentage[8]. Error is computed in every stage along with multiplication and then it is treated as input to the next stage parallelly[7]. This work is based on a trade-off between accuracy and area.

Table 2.1:
Average relative errors for a simple iterative logarithmic multiplier for 1,2 and 3 Correction Terms

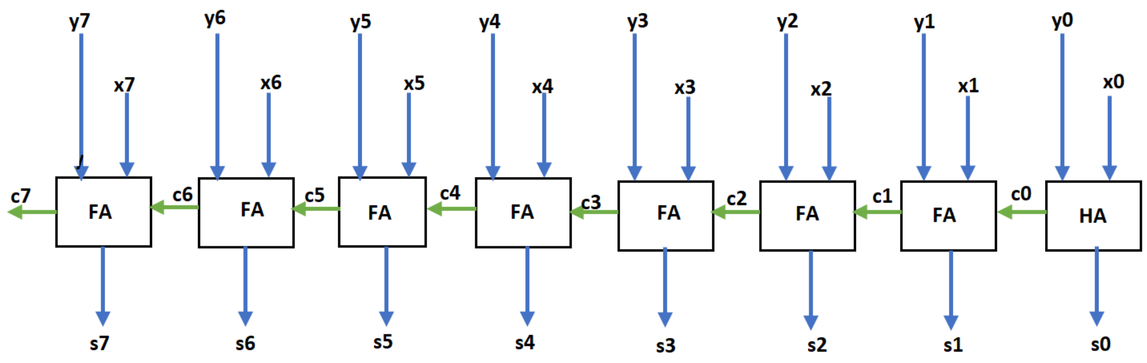
No. bits	BB	BB + 1 ECC	BB + 2 ECC	BB + 3 ECC
8	8.9131	0.8337	0.0708	0.0048
12	9.3692	0.9726	0.1029	0.0106
16	9.4124	0.9874	0.1070	0.1070

(Source : [8])

Table 2.1 shows that as the number of error correction circuits increases the error reduces. Here BB - Basic Block of the Multiplier and ECC - Error correction Circuit.

2.5 An efficient VLSI architecture for Iterative Logarithmic Multiplier

This paper introduces a seamless pipelined technique to the previous versions of Mitchell's algorithm based iterative logarithmic multiplier. This method gives hardware minimization at the same error cost than the previously reported architectures. The results show reduced hardware cost for larger applications like image processing and neural networks.
In fig 2.2 it is a combination of 3 input Full adders(FA) and two input a Half adder(HA) to construct a ripple carry adder which can be used for addition operation in the logarithmic multiplier. Carry(C) is continuously propagated while adding the bits. It can be placed in the critical path of the circuit to reduce delay.

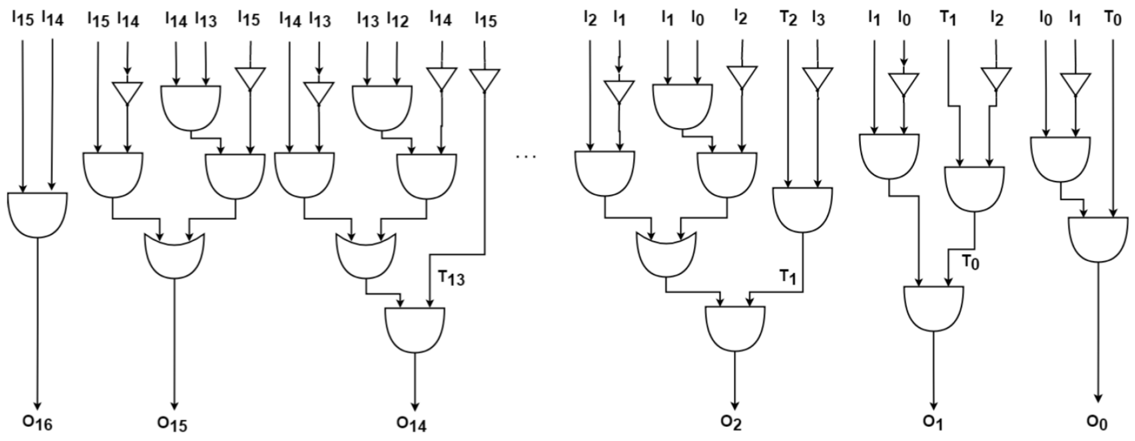


(Source : [12])

Figure 2.2: Block diagram of seamless pipelining of 8 bit ripple carry adder circuit

2.6 A Hardware-Efficient Logarithmic Multiplier with Improved Accuracy

This paper presents a novel method to find approximate logarithmic value of any number. This method rounds the input operand to the nearest power of two instead of the two smaller than or equal to that number. This method is further used to implement 16 bit logarithmic multiplier which show significant area savings with similar accuracy. The nearest power of 2 can be obtained from any number by using the nearest one detector[4] shown in the fig 2.3.

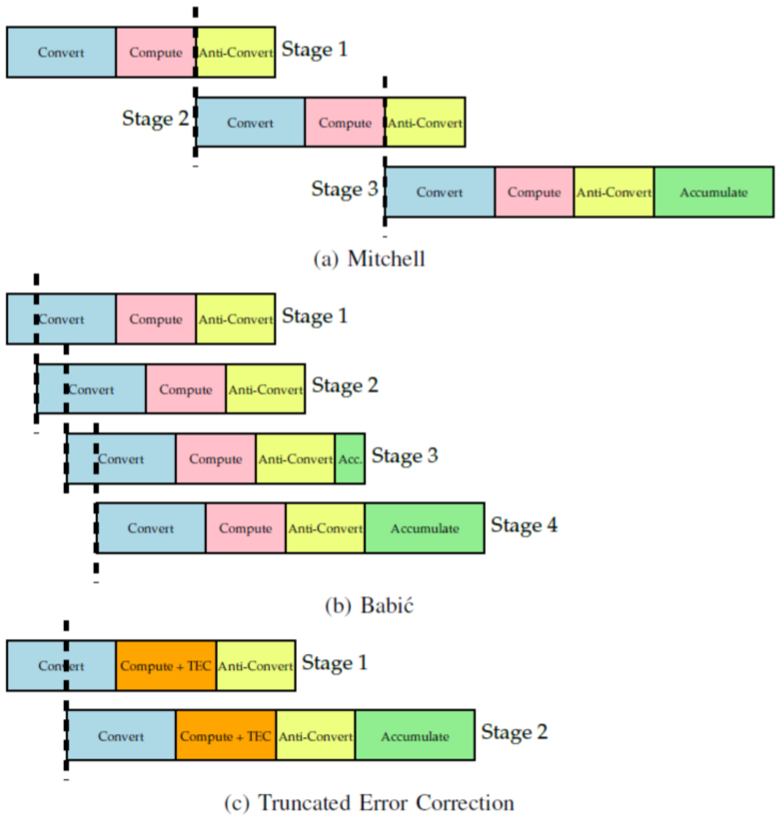


(Source : [4])

Figure 2.3: 16 bit Nearest One Detector Circuit

2.7 Truncated Error Correction for Flexible Approximate Multiplication

This paper proves that the first six mantissa mostly contributes to the accuracy of the logarithmic product. Truncated Error correction circuit is also mentioned which supports for higher precision. The figure 2.4 shows the comparison of previously proposed multipliers for different stages of computation[14]. It is clear that Mitchell's multiplier takes much more time. The truncated error correction approach computes different stages parallely hence saving a lot of time.

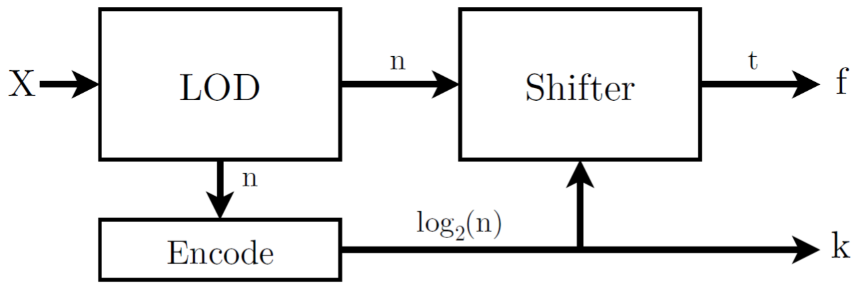


(Source : [14])

Figure 2.4: An illustration of different pipelined iterative approximate multipliers with roughly equivalent approximation errors

2.8 Truncated Logarithmic Approximation

In this paper shifters are presented which are used to compute and anti-convert mantissa bits from the output. The shifter can be easily constructed with the help of array of muxes with and gates. These shifters are almost similar to barrel shifters which are used in the designs. The paper also shows a simple circuit for logarithm generation[15], using an LOD(Leading one Detector) for detection of most significant one in the input. A simple left shifter is used to obtain mantissa bits(f) and encoder is used to obtain the characteristic(k). The complete diagram[15] is shown in fig 2.5.

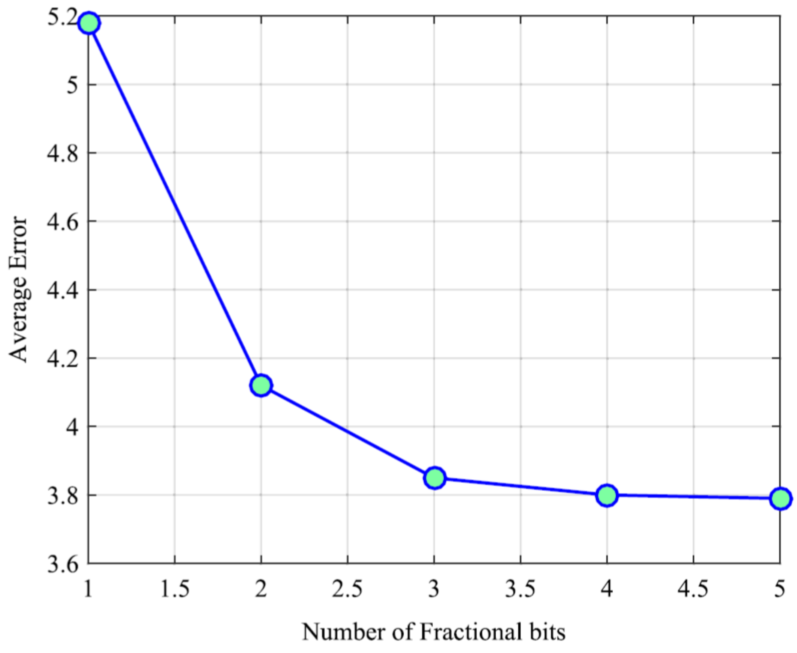


(Source : [15])

Figure 2.5: Logarithm Generation Block Diagram

2.9 An Improved Logarithmic Multiplier for Media Processing

This paper introduces a novel method for implementation of logarithmic multipliers by truncation of mantissa bits. It also proves that if carry bit is considered from the mantissa then it drastically improves the precision. A fractional predictor is also proposed which estimates the carry from the fractional part. The graph below shows the average error variation with the number of fractional bits prediction [2][3]. A comparative analysis is done for 5 fractional bits in the mantissa bits. It can be observed that as the error decreases with the number of fractional bits and is minimum for a bit size of '4' and there after it remains almost constant. Thus, it can be concluded that a fractional predictor with 4 bits is sufficient for carry.



(Source : [3])

Figure 2.6: Graph showing the average error variation with the number of fractional bits considered.

Chapter 3

Logarithmic Multiplication Methods

3.1 Motivation

Logarithmic Number System(LNS) can be used to reduce larger multiplications to basic additions and resuce larger powers of numbers into smaller multiplications like:

$$log(A \cdot B) = logA + logB \tag{3.1}$$

$$logA^2 = 2 \cdot logA \tag{3.2}$$

Example:

$$log(120) = log(10^2 \times 1.2)$$

Using (3.1) & (3.2)

$$= 2 \cdot log10 + log1.2$$

$$= 2 + 0.07198$$

Here 2 = **characteristic** and 0.7198 = **mantissa**
In a normal approach to get the product of two numbers, the logarithms of operands are taken to perform **radix-2 multiplication(log_2)**. Antilog is taken at the end of the product to get the final answer.

For binary number system:

$$\boldsymbol{log_2 N = k + log_2(1 + f)}$$

Where N = **Any decimal number**, k = **characteristic**, f = **mantissa or fractional part**
There are two types of LNS multipliers which are mostly considered :
(a) Look up Tables and Interpolations: Numbers are stored in memory or ROM in log and anti-log scale. Whenever any computation is required it is fed from the memory and is produced as result. If the expected is not stored in the ROM then the closest value(approximate) is placed as the result.
(b) Mitchell's Algorithm : Developed by John N. Mitchell in March 1962 where the approximate

logarithm of any decimal number can be obtained by simple shifting and counting of bits. This is followed by either addition or subtraction to find out product and quotient respectively.

3.2 Mitchell's Algorithm

The greatest advantage of using this algorithm is that it reduces a lot of design complexity and area hence resulting a lot of power savings. It creates an energy efficient design for any approach. It comes at a cost of introducing errors in the final product as it uses approximate logarithms[11]. Consider the following equation :

$$log_2(1+f) = f \qquad \textbf{(Approximation)} \tag{3.3}$$

Example
Using (3.3)

$$log_2(N_1 \cdot N_2) = k_1 + log_2(1+f_1) + k_2 + log_2(1+f_2)$$

$$\Rightarrow log_2(N_1 \cdot N_2) = k_1 + f_1 + k_2 + f_2$$

Approximate logarithms can be found be simple shifting and observation of bits. **Example** $13 = (1101)_2$

- Bit position of most significant one = $3rd\ position$ = $(011)_2$ = characteristic
- Put decimal after the most significant one
- So, approx $log 13 = (011.1010)_2$

Using the above shown method a comparison has been made with the accurate logarithms in table 3.1 which further shows that approximate logarithms calculated are very close to the true value. This can be further understood by the fig. 3.1.

Algorithm 1 (Mitchell's Algorithm [11])

```
1. $N_1, N_2$ : $n-$bits binary multiplicands, $P_{approx} = 0$ : $2n-$bits approximate product
2. Calculate $k_1$ : leading one position of $N_1$
3. Calculate $k_2$ : leading one position of $N_2$
4. Calculate $f_1$ : shift $N_1$ to the left by $n - k_1$ bits
5. Calculate $f_2$ : shift $N_2$ to the left by $n - k_2$ bits
6. Calculate $k_{12} = k_1 + k_2$
7. Calculate $f_{12} = f_1 + f_2$
8. IF $f_{12} \geq 2^n \quad ie. \quad (f_1 + f_2 \geq 1)$ :
     • Calculate $k_{12} = k_{12} + 1$
     • Decode $k_{12}$ and insert $f_{12}$ in that position of $P_{approx}$
9. ELSE
     • Decode $k_{12}$ and insert $'1'$ in that position of $P_{approx}$
     • Append $f_{12}$ immediately after this one in $P_{approx}$
10. $N_1 \cdot N_2 = P_{approx}$
```

Table 3.1: Comparison of Approximate and Accurate Binary Logarithms

N	**N(binary)**	**Approx log_2N (binary)**	**Approx log_2N**	log_2N
1	000001	000.00000	0.00000	0.00000
2	000010	001.00000	1.00000	1.00000
3	000011	001.10000	1.50000	1.58496
4	000100	010.00000	2.00000	2.00000
5	000101	010.01000	2.25000	2.32192
6	000110	010.10000	2.50000	2.58496
7	000111	010.11000	2.75000	2.80735
8	001000	011.00000	3.00000	3.00000
9	001001	011.00100	3.12500	3.16992
10	001010	011.01000	3.25000	3.32192
11	001011	011.01100	3.37500	3.45943
12	001100	011.10000	3.50000	3.58496
13	001101	011.10100	3.62500	3.70043
14	001110	011.11000	3.75000	3.80735
15	001111	011.11100	3.87500	3.90689
16	010000	100.00000	4.00000	4.00000
17	010001	100.00010	4.06250	4.08746
18	010010	100.00100	4.12500	4.16992
19	010011	100.00110	4.18750	4.24792
20	010100	100.01000	4.25000	4.32192
21	010101	100.01010	4.31250	4.39231
22	010110	100.01100	4.37500	4.45943
23	010111	100.01110	4.43750	4.52356
24	011000	100.10000	4.50000	4.58496
25	011001	100.10010	4.56250	4.64385
26	011010	100.10100	4.62500	4.70043
27	011011	100.10110	4.68750	4.75488
28	011100	100.11000	4.75000	4.80735
29	011101	100.11010	4.81250	4.85798
30	011110	100.11100	4.87500	4.90689
31	011111	100.11110	4.93750	4.95419
32	100000	101.00000	5.00000	5.00000
33	100001	101.00001	5.03125	5.04439
34	100010	101.00010	5.06250	5.08746
35	100011	101.00011	5.09375	5.12928

(Source : [11])

3.3 Mathematical model

The Binary Representation of any number(N) can be represented as[8]:

$$\mathbf{N = 2^k \left(1 + \sum_{i=j}^{k-1} 2^{i-k} Z_i\right) = 2^k\,(1 + f)} \tag{3.4}$$

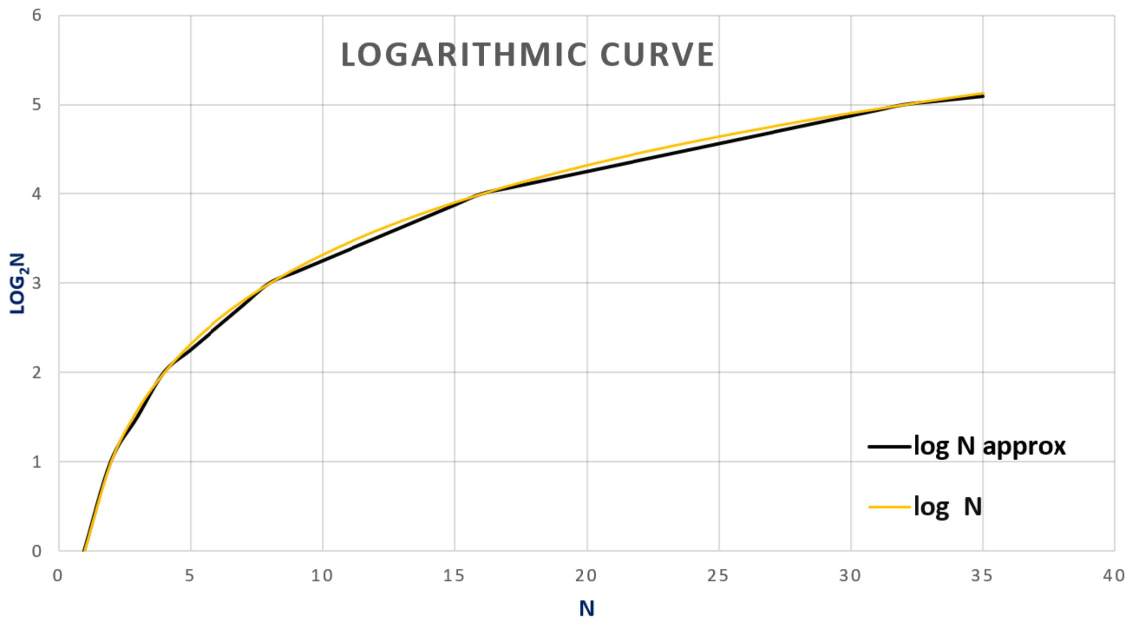
Where,
k = characteristic
Z_i =bit value at the i^{th} position
f = mantissa or fractional part
j = depends on number's precision (0 for integer numbers)
Now, taking log_2 on both sides:

$$log_2(N) = log_2\left(2^k(1+f)\right)$$

$$\Rightarrow log_2(N) = k + log_2(1+f)$$

Table 3.2: Multiplication of two 8 bit numbers using Mitchell's algorithm

$N_1 = 220 = 11011100$	$N_2 = 159 = 10011111$
$k_1 = 0111$	$k_2 = 0111$
$f_1 = 10111000$	$f_2 = 00111110$
$k_{12} = k_1 + k_2 = 1110$	
$f_{12} = f_1 + f_2 = 11110110$ (No Carry bit ie. $f_{12} < 2^8$)	
$P_{approx} = 111110110000000 = 32128$	
$P_{true} = 34980$	
$Error(\varepsilon) = (P_{true} - P_{approx})/P_{true} = 8.15\%$	



(Source : [11])

Figure 3.1: Graph showing comparison of Approximate and Accurate Binary Logarithms

From (3.4)

$$N = 2^k(1+f)$$

$$\Rightarrow f \cdot 2^k = N - 2^k \tag{3.5}$$

Using (3.4) and the product(P_{true}) of two numbers N_1 and N_2 can be written as :

$$P_{true} = N_1 \cdot N_2 = 2^{k_1}(1+f_1) \cdot 2^{k_2}(1+f_2)$$

$$\Rightarrow P_{true} = 2^{k_1+k_2}(1+f_1+f_2) + 2^{k_1+k_2}(f_1 \cdot f_2) \tag{3.6}$$

Using (3.5) in (3.6)

$$\mathbf{P_{true} = 2^{k_1+k_2} + (N_1 - 2^{k_1})2^{k_2} + (N_2 - 2^{k_2})2^{k_1} + (N_1 - 2^{k_1}) \cdot (N_2 - 2^{k_2})} \tag{3.7}$$

Here $(N_1 - 2^{k_1})$ and $(N_2 - 2^{k_2})$ are the mantissa or fractional part(f) of N_1 and N_2 respectively. So, consider: $(N_1 - 2^{k_1}) = f_1$ and $(N_1 - 2^{k_1}) = f_2$

Using the above results:

(a) If carry bit from mantissa is neglected : ie. $(f_1 + f_2 < 1)$

$$\mathbf{P_{true} = 2^{k_1+k_2} + f_1 \cdot 2^{k_2} + f_2 \cdot 2^{k_1} + f_1 \cdot f_2} \tag{3.8}$$

(b) If carry bit from mantissa is considered : ie. $(f_1 + f_2 > 1)$

From (3.5) and (3.6)

$$P_{true} = 2^{k_1+k_2+1}(1+f_1+f_2) + 2^{k_1+k_2}(f_1' \cdot f_2')$$

$$\Rightarrow \mathbf{P_{true} = f_1 \cdot 2^{k_2+1} + f_2 \cdot 2^{k_1+1} + f_1' \cdot f_2'} \tag{3.9}$$

where $'$ represents 2's complement

Chapter 4

Iterative Logarithmic Multiplier

4.1 Mathematical Analysis

Consider the equation (3.8) where the carry bit from the mantissa is neglected:

$$\mathbf{P_{true} = 2^{k_1+k_2} + f_1 \cdot 2^{k_2} + f_2 \cdot 2^{k_1} + f_1 \cdot f_2}$$

Where, $(N_1 - 2^{k_1}) = f_1$ and $(N_1 - 2^{k_1}) = f_2$
Let, $P^{(0)}_{approx} = 2^{k_1+k_2} + f_1 \cdot 2^{k_2} + f_2 \cdot 2^{k_1}$

$$\Rightarrow P_{true} = P^{(0)}_{approx} + f_1 \cdot f_2$$

$$\Rightarrow P_{true} = P^{(0)}_{approx} + \mathbf{Error}(\varepsilon^{(\mathbf{0})})$$

The product of any two numbers consists of an approximate term and an error term
For 1st **iteration:** Consider $\varepsilon^{(0)} = C^{(1)}$ where C = **Correction Term**[7]

$$\Rightarrow P_{true} = P^{(0)}_{approx} + C^{(1)} + \varepsilon^{(1)}$$

$$\Rightarrow P_{true} = P^{(1)}_{approx} + \varepsilon^{(1)}$$

For 2nd **iteration:** Consider $\varepsilon^{(1)} = C^{(2)}$

$$\Rightarrow P_{true} = P^{(1)}_{approx} + C^{(2)} + \varepsilon^{(2)}$$

$$\Rightarrow P_{true} = P^{(2)}_{approx} + \varepsilon^{(2)}$$

For n^{th} **iteration:** Consider $\varepsilon^{(n-1)} = C^{(n)}$ where $C^{(n)} = n^{th}$ Correction Term

$$\Rightarrow P_{true} = P^{(n-1)}_{approx} + C^{(n)} + \varepsilon^{(n)}$$

$$\Rightarrow P_{true} = P^{(n)}_{approx} + \varepsilon^{(n)}$$

Here,

$$\mathbf{P^{(n)}_{approx} = P^{(0)}_{approx} + C^1 + C^2 + \cdots + C^{(i)} = P^{(0)}_{approx} + \sum_{i=1}^{n} C^{(i)}} \tag{4.1}$$

Where,

$$\mathbf{P^{(0)}_{approx} = 2^{k_1+k_2} + f_1 \cdot 2^{k_2} + f_2 \cdot 2^{k_1}} \tag{4.2}$$

Algorithm 2 (Iterative Mitchell's based Algorithm)

1. N_1, N_2 : $n-$bits binary multiplicands,$P^{(0)}_{approx} = 0$: $2n-$bits first approximation, $C^{(i)}$ correction terms, $P_{approx} = 0$: $2n-$bits product
2. Calculate k_1 : leading one position of N_1
3. Calculate k_2 : leading one position of N_2
4. Calculate f_1: Copy all the bits from N_1 excluding leading one

5. Calculate f_2: Copy all the bits from N_2 excluding leading one
6. Calculate $f_1 \cdot 2^{k_2}$: shift f_1 to the left by k_2 bits
7. Calculate $f_2 \cdot 2^{k_1}$: shift f_2 to the left by k_1 bits
8. Calculate $k_{12} = k_1 + k_2$
9. Calculate $2^{k_1+k_2}$: decode k_{12}
10. Calculate $P^{(0)}_{approx}$: add $2^{k_1+k_2}$,$f_1 \cdot 2^{2^{k_2}}$ and $f_2 \cdot 2^{k_1}$
11. Repeat $i-$times or until $N_1 = 0$ or $N_2 = 0$
    - Set $N_1 = f_1, N_2 = f_2$
    - Repeat steps from 2 to 7
    - Calculate $C^{(i)}$: add $2^{k_1+k_2}$,$f_1 \cdot 2^{2^{k_2}}$ and $f_2 \cdot 2^{2^{k_1}}$
12. $P^{(n)}_{approx} = P^{(0)}_{approx} + \sum_{i=1}^{n} C^{(i)}$

4.1.1 Example: Multiplication of two 16-bit numbers by iteration

Table 4.1: Error Calculation for 0th Iteration of Iterative Logarithmic Multiplier

N_1	0011 0101 0010 0100	13604
N_2	0101 1110 1000 0001	24193
k_1	1101	13
k_2	1110	14
f_1	1 0101 0010 0100	$5,412$
f_2	01 1110 1000 0001	$7,809$
$f_1 \cdot 2^{k_2}$	101 0100 1001 0000 0000 0000 0000	$8,86,70,208$
$f_2 \cdot 2^{k_1}$	11 1101 0000 0010 0000 0000 0000	$6,39,71,328$
$k_{12} = k_1 + k_2$	1 1011	27
$2^{k_1+k_2}$	1000 0000 0000 0000 0000 0000 0000	$13,42,17,728$
$P^{(0)}_{approx} = f_1 \cdot 2^{k_2} + f_2 \cdot 2^{k_1} + 2^{k_1+k_2}$	1 0001 0001 1001 0010 0000 0000 0000	$28,68,59,264$
P_{true}	1 0011 1001 1101 1111 1111 0010 0100	$32,91,21,572$
$Error(\varepsilon^{(0)}) = (P_{true} - P^{(0)}_{approx})/P_{true}$		12.84%

Table 4.2: Error Calculation for 1st Iteration of Iterative Logarithmic Multiplier

$N_1^{(1)} = f_1$	0001 0101 0010 0100	$5,412$
$N_2^{(1)} = f_2$	0001 1110 1000 0001	$7,809$
$k_1^{(1)}$	1100	12
$k_2^{(1)}$	1100	12
$f_1^{(1)}$	0101 0010 0100	$1,316$
$f_2^{(1)}$	1110 1000 0001	$3,713$
$f_1^{(1)} \cdot 2^{k_2^{(1)}}$	101 0010 0100 0000 0000 0000	$53,90,336$
$f_2^{(1)} \cdot 2^{k_1^{(1)}}$	1110 1000 0001 0000 0000 0000	$1,52,08,448$
$k_{12}^{(1)} = k_1^{(1)} + k_2^{(1)}$	1 1000	24
$2^{k_1^{(1)}+k_2^{(1)}}$	1 0000 0000 0000 0000 0000 0000	$1,67,77,216$
$C^{(1)} = 2^{k_1^{(1)}+k_2^{(1)}} + f_1^{(1)} \cdot 2^{k_2^{(1)}} + f_2^{(1)} \cdot 2^{k_1^{(1)}}$	100011 1010 0101 0000 0000 0000	$3,73,76,000$
$P^{(1)}_{approx} = P^{(0)}_{approx} + C^{(1)}$	1 0011 0101 0011 0111 0000 0000 0000	$32,42,35,264$
P_{true}	1 0011 1001 1101 1111 1111 0010 0100	$32,91,21,572$
$Error(\varepsilon^{(1)}) = (P_{true} - P^{(1)}_{approx})/P_{true}$		1.48%

Table 4.3: Error Calculation for 2nd Iteration of Iterative Logarithmic Multiplier

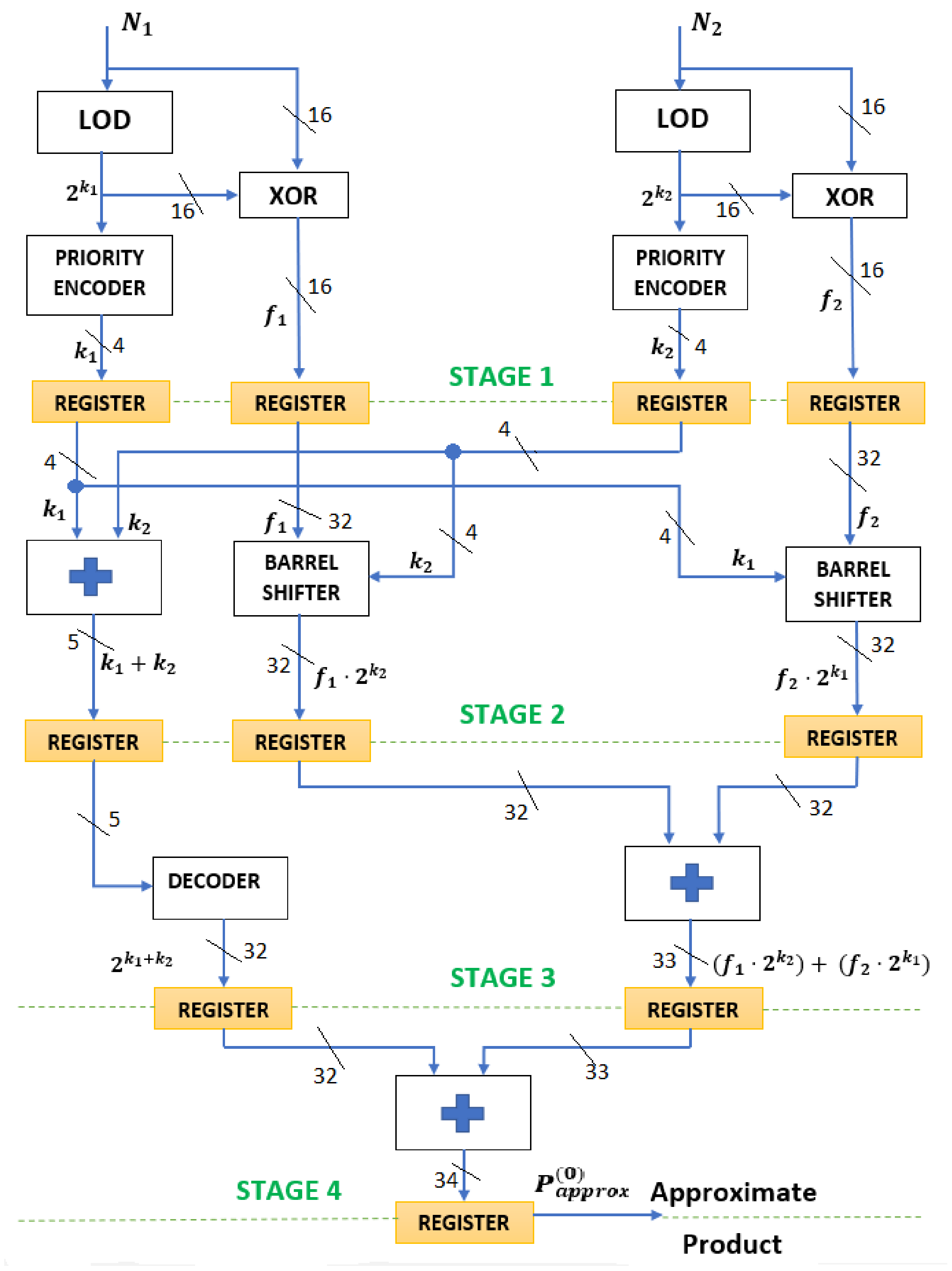
$N_1^{(2)} = f_1^{(1)}$	0000 0101 0010 0100	$1,316$
$N_2^{(2)} = f_2^{(1)}$	0000 1110 1000 0001	$3,713$
$k_1^{(2)}$	1010	10
$k_2^{(2)}$	1011	11
$f_1^{(2)}$	01 0010 0100	292
$f_2^{(2)}$	110 1000 0001	$1,665$
$f_1^{(2)} \cdot 2^{k_2^{(2)}}$	1001 0010 0000 0000 0000	$5,98,016$
$f_2^{(2)} \cdot 2^{k_1^{(2)}}$	1 1010 0000 0100 0000 0000	$17,04,960$
$k_{12}^{(2)} = k_1^{(2)} + k_2^{(2)}$	1 0101	21
$2^{k_1^{(2)}+k_2^{(2)}}$	10 0000 0000 0000 0000 0000	$20,97,152$
$C^{(2)} = 2^{k_1^{(2)}+k_2^{(2)}} + f_1^{(2)} \cdot 2^{k_2^{(2)}} + f_2^{(1)} \cdot 2^{k_1^{(2)}}$	100 0011 0010 0100 0000 0000	$44,00,128$
$P_{approx}^{(2)} = P_{approx}^{(1)} + C^{(2)}$	1 0011 1001 0110 1001 0100 0000 0000	$32,86,35,392$
P_{true}	1 0011 1001 1101 1111 1111 0010 0100	$32,91,21,572$
$Error(\varepsilon^{(2)}) = (P_{true} - P_{approx}^{(2)})/P_{true}$		0.14%

4.2 Hardware Design

4.2.1 Basic Block

The architecture is based on the equation(4.2)

$$P^{(0)}_{approx} = 2^{k_1+k_2} + f_1 \cdot 2^{k_2} + f_2 \cdot 2^{k_1}$$



(Source : [7])

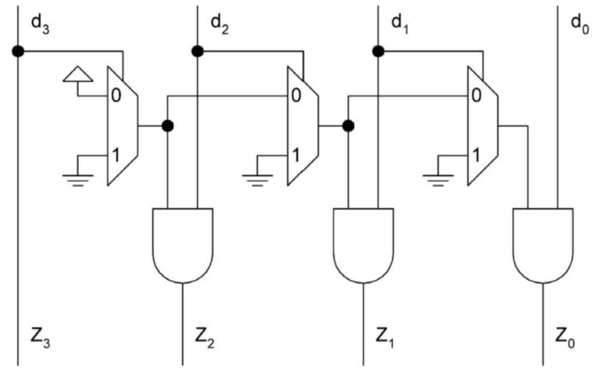
Figure 4.1: Block Diagram of Iterative Logarithmic Multiplier for Basic Block

4.2.2 Leading One Detector(LOD)

It is used to find the most significant $'1'$ from the input.[5]
Example:

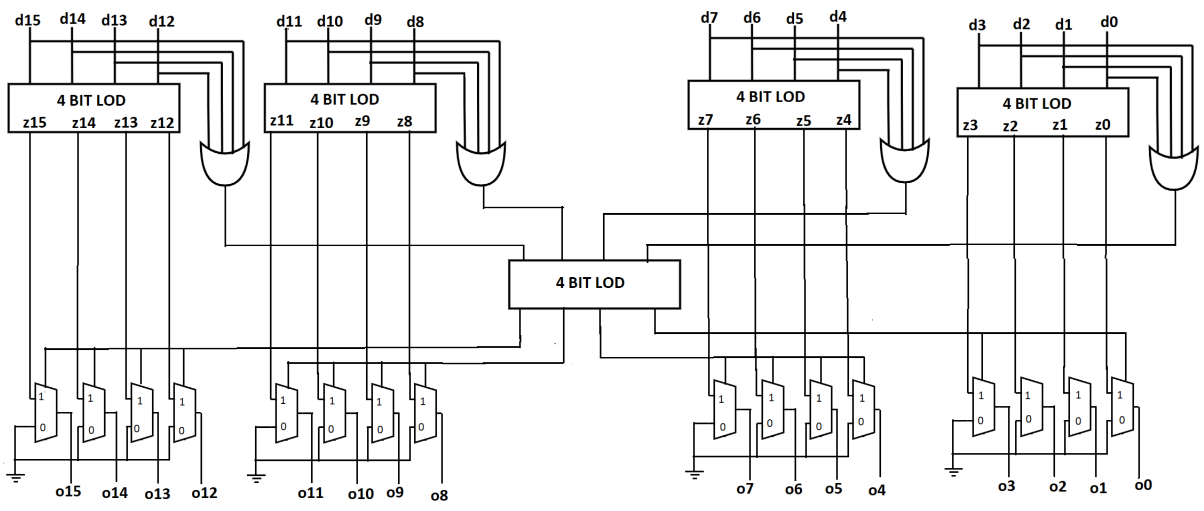
$$\begin{aligned} \text{Input} =& \quad 0101\ 1110\ 1000\ 0001 \\ \text{Output} =& \quad 0100\ 0000\ 0000\ 0000 \end{aligned}$$

The architecture is given below for 4-bit and 16-bit LOD which is used in fig.4.1



(Source : [7])

Figure 4.2: 4-bit Leading One Detector Circuit

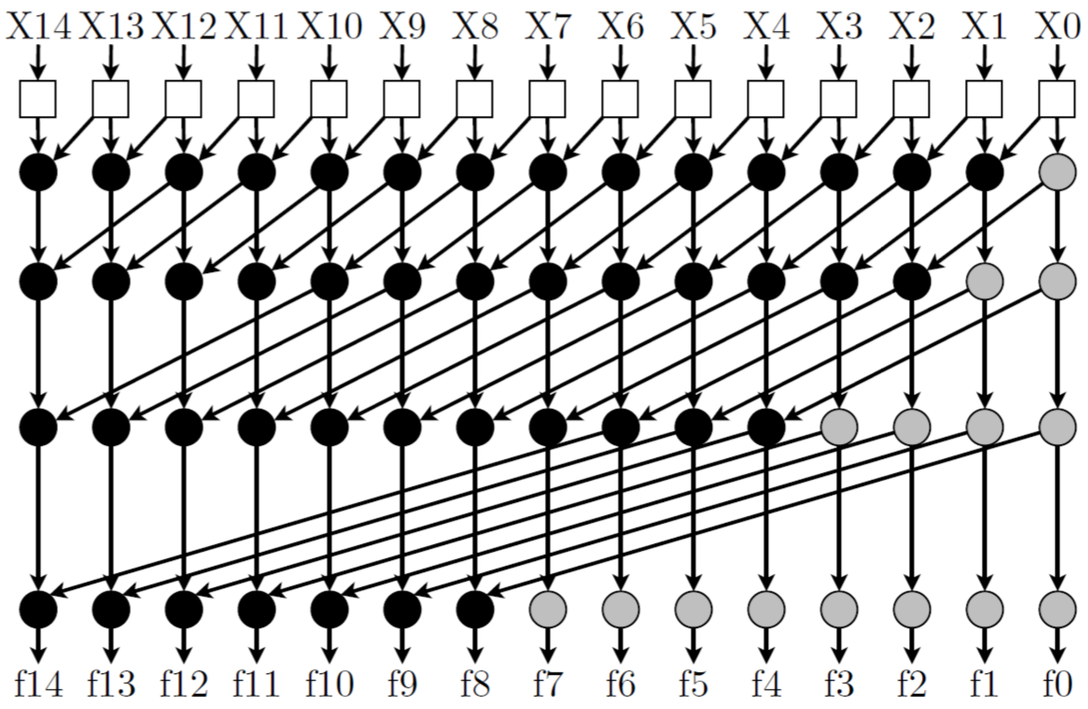


(Source : [7])

Figure 4.3: 16-bit Leading One Detector Block Diagram

4.2.3 Barrel Shifter

It is a purely combinational circuit which is used to shift data bit left or right. This is achieved by using array of 2 X 1 muxes with and gates. In the fig. 4.1, 32 X 5 barrel shifter is used where 5 are the select lines which does logical left shift operation. In the figure below a simple 15 X 4 barrel shifter is shown. **Black nodes represent two-input multiplexers,** which are controlled by the (unsigned binary) shift amount. Some internal simplification occurs in the shifter due to the fact that constant 0 values are propagated into the least significant bits of a shifted output. To account for this, **grey nodes represent AND gates whose second input is connected to the inverted shift signal**[15].



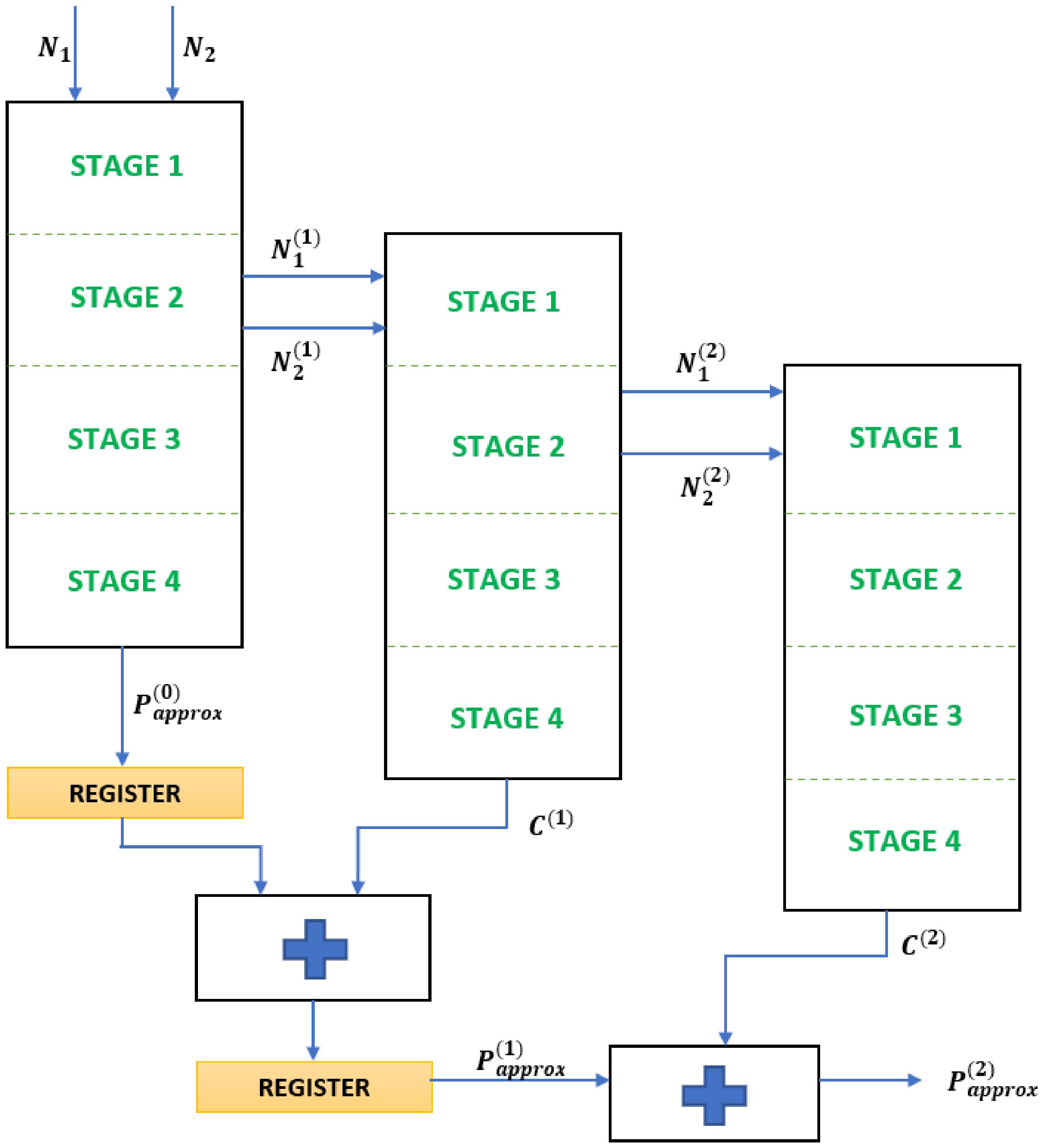
(Source : [15])

Figure 4.4: 15 bit Barrel Shifter illustration for Left Shift Operation

4.2.4 Final Block

The architecture is based on the equation(4.1)

$$\mathbf{P^{(n)}_{approx} = P^{(0)}_{approx} + C^1 + C^2 + \cdots + C^{(i)} = P^{(0)}_{approx} + \sum_{i=1}^{n} C^{(i)}}$$



(Source : [7])

Figure 4.5: Iterative Logarithmic Multiplier Block Diagram

Chapter 5

Non - Iterative Logarithmic Multiplier

5.1 Mathematical Analysis

Consider the equation (3.9) where the carry bit from the mantissa is considered[3]:

$$\mathbf{P_{true} = f_1 \cdot 2^{k_2+1} + f_2 \cdot 2^{k_1+1} + f_1' \cdot f_2'} \tag{5.1}$$

Where, $(N_1 - 2^{k_1}) = f_1$ and $(N_1 - 2^{k_1}) = f_2$ and $'$ represents 2's complement
Let:

$$\mathbf{P_{approx} = f_1 \cdot 2^{k_2+1} + f_2 \cdot 2^{k_1+1}} \tag{5.2}$$

$$Error\ (\varepsilon) = f_1' \cdot f_2'$$

So,

$$\Rightarrow P_{true} = P_{approx} + \mathbf{Error}(\varepsilon)$$

As the Error(ε) is in 2's complement form, it is neglected here for simpler hardware design.

Algorithm 3 (Non- Iterative Mitchell's based algorithm considering carry)

1. N_1, N_2 : $n-$bits binary multiplicands, $P_{approx} = 0$: $2n-$bits product
2. Calculate k_1: leading one position of N_1
3. Calculate k_2: leading one position of N_2
4. Calculate f_1: Copy all the bits from N_1 excluding leading one
5. Calculate f_2 : Copy all the bits from N_2 excluding leading one
6. Calculate $f_1 \cdot 2^{k_2+1}$: shift f_1 to the left by $k_2 + 1$ bits
7. Calculate $f_2 \cdot 2^{k_1+1}$: shift f_1 to the left by $k_1 + 1$ bits
8. Calculate P_{approx} : add $f_1 \cdot 2^{k_2+1}$ and $f_2 \cdot 2^{k_1+1}$

5.1.1 Example: Multiplication of two 16 bit numbers

For better comparison same inputs are taken from table 4.1

Table 5.1: Error Calculation for Non-Iterative Logarithmic Multiplier

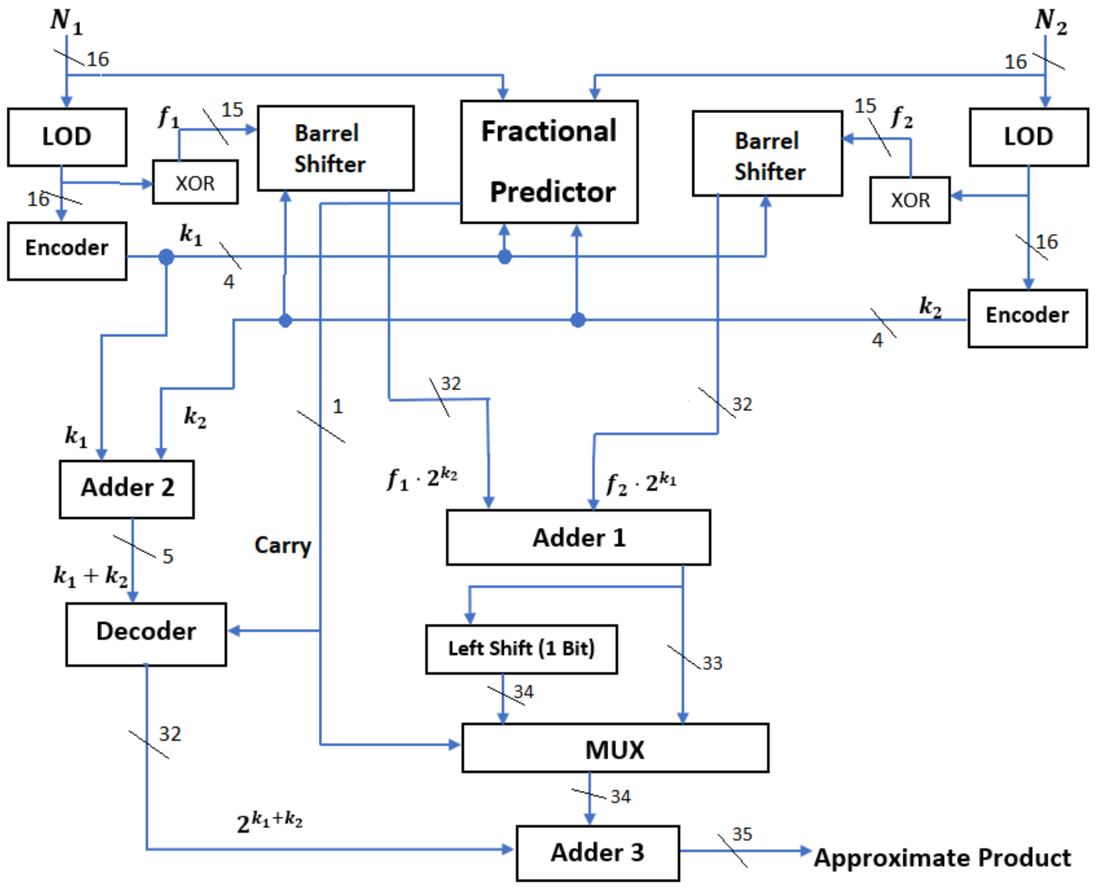
N_1	0011 0101 0010 0100	13604
N_2	0101 1110 1000 0001	24193
k_1	1101	13
k_2	1110	14
f_1	1 0101 0010 0100	$5,412$
f_2	01 1110 1000 0001	$7,809$
$f_1 \cdot 2^{k_2+1}$	1010 1001 0010 0000 0000 0000 0000	$17,73,40,416$
$f_2 \cdot 2^{k_1+1}$	111 1010 0000 0100 0000 0000 0000	$12,79,42,656$
P_{approx}	1 0010 0011 0010 0100 0000 0000 0000	$30,52,83,072$
P_{true}	1 0011 1001 1101 1111 1111 0010 0100	$32,91,21,572$
$Error(\varepsilon)$		7.24%

5.2 Hardware Design

The architecture is based on both the equations (4.2) and (5.2) respectively depending on the carry bit from the mantissa.

For No Carry: $\mathbf{P_{approx} = 2^{k_1+k_2} + f_1 \cdot 2^{k_2} + f_2 \cdot 2^{k_1}}$

For Carry: $\mathbf{P_{approx} = f_1 \cdot 2^{k_2+1} + f_2 \cdot 2^{k_1+1}}$



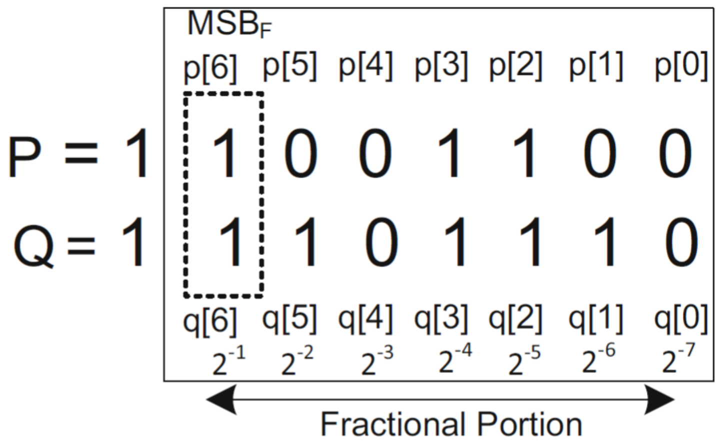
(Source: [3])

Figure 5.1: Non-Iterative Logarithmic Multiplier Block Diagram

The design is almost same as the Iterative Logarithmic Multiplier - Basic Block(fig 4.1). The only new block introduced here is Fractional Predictor which has been explained below.

5.3 Fractional Predictor

This block is used is used to detect the carry bit from the mantissa[3][2]. Fig 5.1 is a combination of both equation 4.2 and 5.2. Depending on the carry bit P_{approx} is generated at the end. The carry signal acts as an active low enable to the decoder ie. when carry signal is high it turns of the decoder. It also acts as control signal to the mux. If the carry signal is high the shifted output propagated to adder 3. Consider the figure shown below. P and Q are two multiplicands as input.



(Source: [3])

Figure 5.2: Illustration of Carry bit Detection in Mantissa

The leading ones are neglected as they are the characteristic to the product. The carry is 1 if p[6] and q[6] are one. If any of them if zero, then it depend on p[5] and q[5]. This dependency continues till p[0] and q[0]. The carry detection logic till 3 fractional bits is shown below:

$$Carry1 = p[6]\ \&\ q[6]$$

$$Carry2 = Carry1 \mid ((p[6] \mid q[6])\ \&\ (p[5]\ \&\ q[5]))$$

$$Carry3 = Carry2 \mid ((p[6] \mid q[6])\ \&\ (p[5] \mid q[5])\ \&\ (p[4]\ \&\ q[4]))$$

Here & refers AND operation and | refers OR operation.

Chapter 6

Implementation and Results

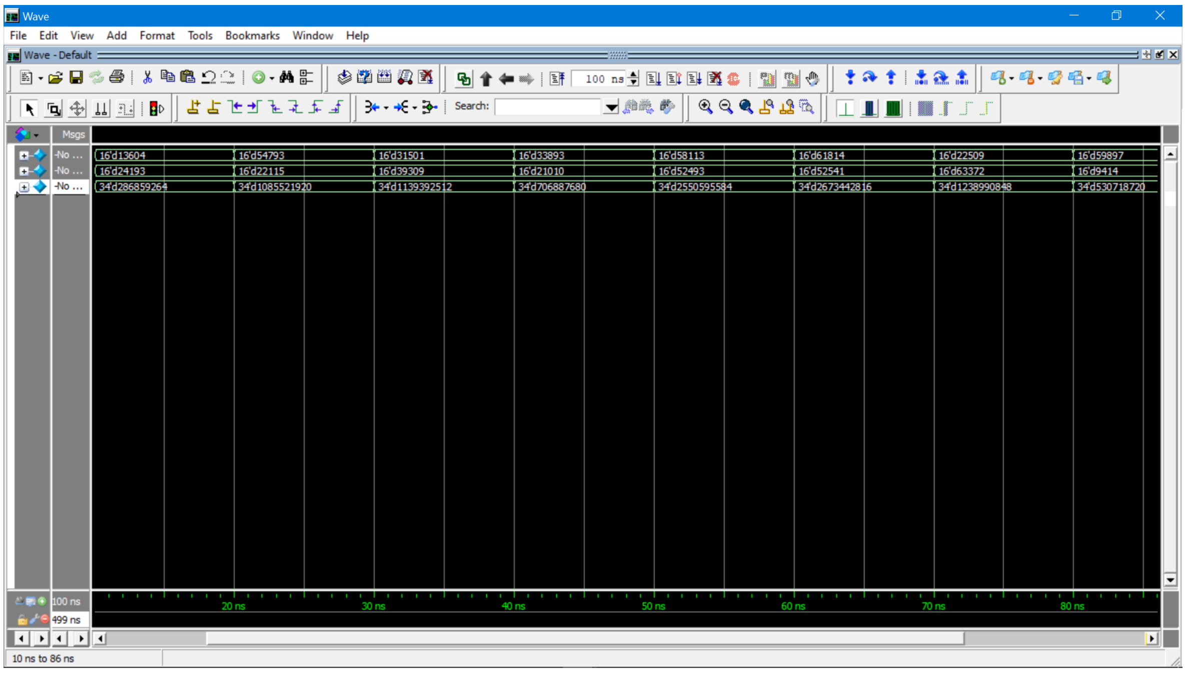
The designs in the previous chapters were simulated and verified in ***Modelsim*** by Mentor Graphics Pvt. Ltd. and synthesized and implemented in an FPGA Board using ***Intel Quartus Prime***. Below are the specifications given for the device used.

Table 6.1: Cyclone IV E FPGA Device Specifications

Family	Cyclone IV E
Name	EP4CE115F29C7
Core Voltage	1.2V
Logic Elements	114480
User I/Os	529
Memory Bits	3981312
PLL	4

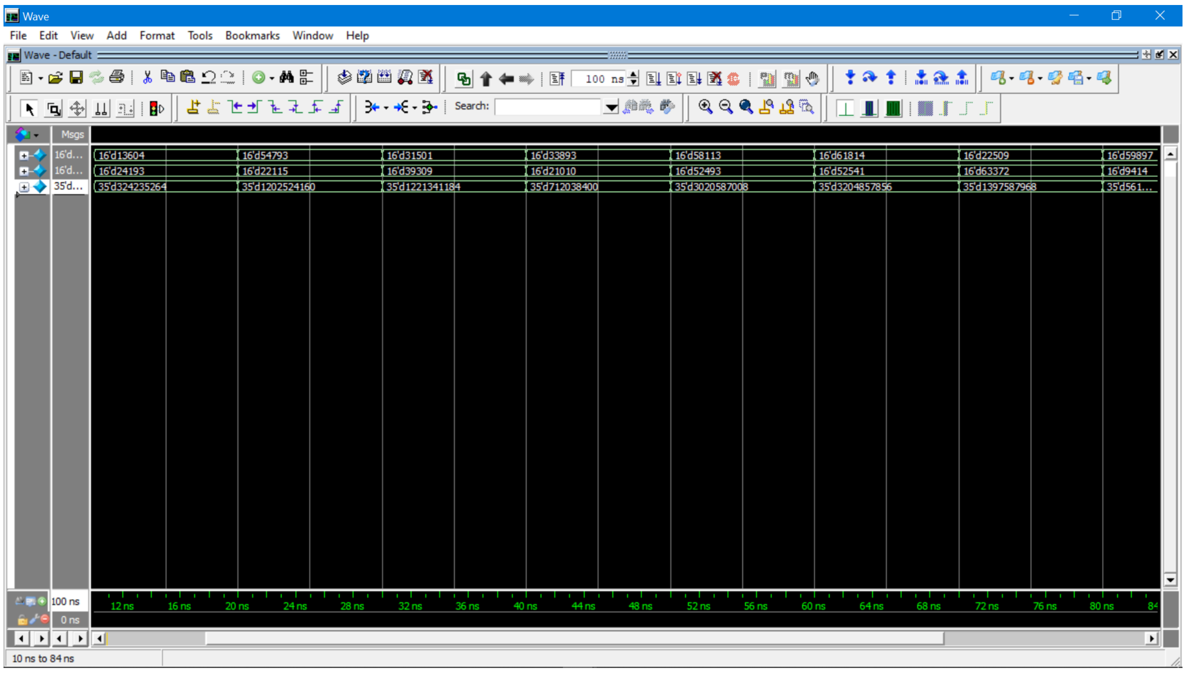
(Source : Quartus Prime Software)

6.1 Iterative Logarithmic Multiplier



```
16'd13604      16'd54793       16'd31501       16'd33893      16'd58113       16'd61814       16'd22509       16'd59897
16'd24193      16'd22115       16'd39309       16'd21010      16'd52493       16'd52541       16'd63372       16'd9414
34'd286859264  34'd1085521920  34'd1139392512  34'd706887680  34'd2550595584  34'd2673442816  34'd1238990848  34'd530718720
```

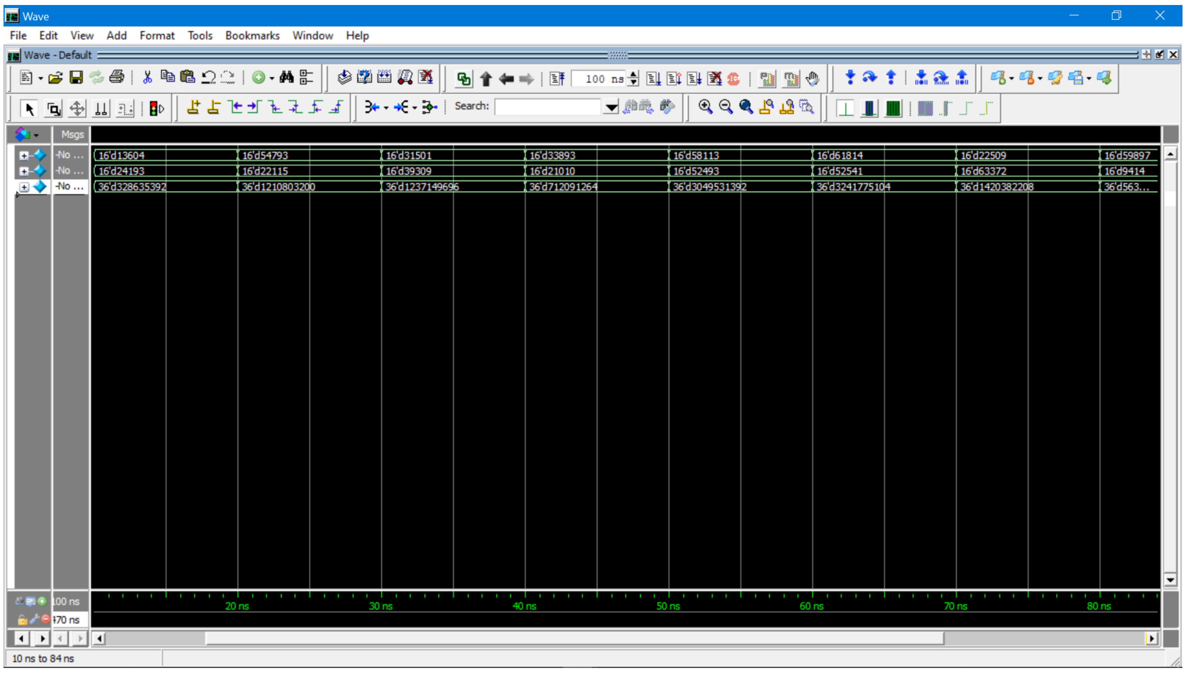
Figure 6.1: Modelsim Simulation of 0^{th} Iteration of Iterative Logarithmic Multiplier



```
Msgs
16'd...  16'd13604     16'd54793      16'd31501      16'd33893     16'd58113      16'd61814      16'd22509      16'd59897
16'd...  16'd24193     16'd22115      16'd39309      16'd21010     16'd52493      16'd52541      16'd63372      16'd9414
35'd...  35'd324235264 35'd1202524160 35'd1221341184 35'd712038400 35'd3020587008 35'd3204857856 35'd1397587968 35'd561...

100 ns   12 ns  16 ns  20 ns  24 ns  28 ns  32 ns  36 ns  40 ns  44 ns  48 ns  52 ns  56 ns  60 ns  64 ns  68 ns  72 ns  76 ns  80 ns
0 ns
10 ns to 84 ns
```

Figure 6.2: Modelsim Simulation of 1st Iteration of Iterative Logarithmic Multiplier



```
Msgs
-No ...  16'd13604     16'd54793      16'd31501      16'd33893     16'd58113      16'd61814      16'd22509      16'd59897
-No ...  16'd24193     16'd22115      16'd39309      16'd21010     16'd52493      16'd52541      16'd63372      16'd9414
-No ...  36'd328635392 36'd1210803200 36'd1237149696 36'd712091264 36'd3049531392 36'd3241775104 36'd1420382208 36'd563...

100 ns   20 ns  30 ns  40 ns  50 ns  60 ns  70 ns  80 ns
170 ns
10 ns to 84 ns
```

Figure 6.3: Modelsim Simulation of 2nd Iteration of Iterative Logarithmic Multiplier

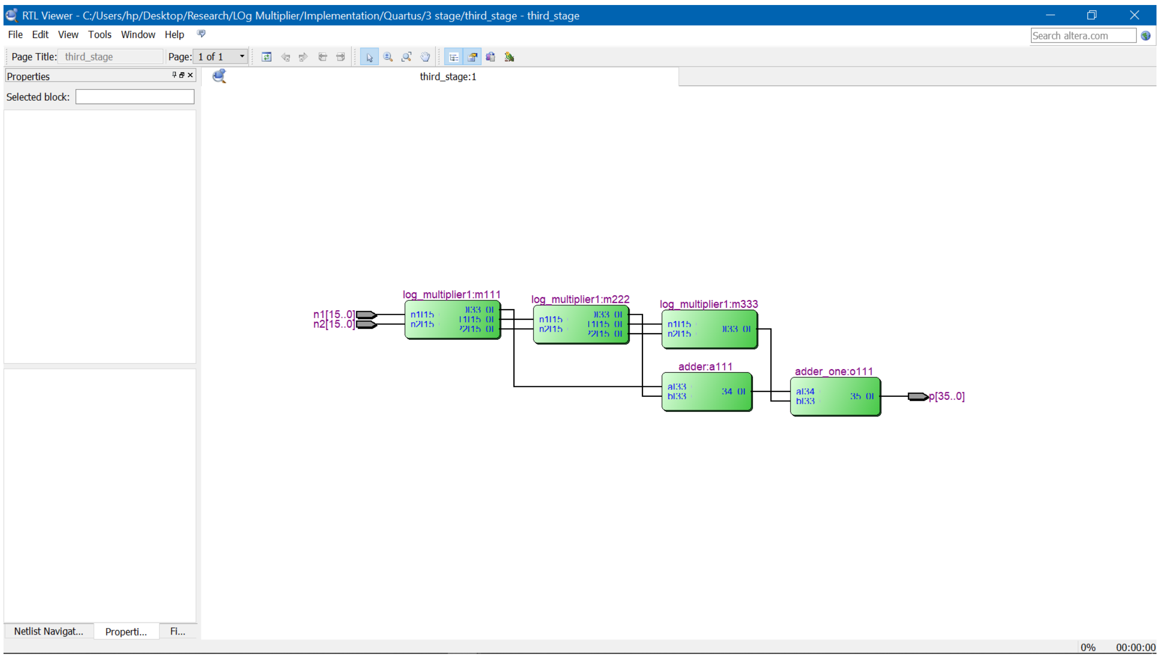


Figure 6.4: Implementation of Iterative Logarithmic Multiplier on Quartus Prime

Table 6.2: Multiplication Results of Iterative Logarithmic Multiplier in Modelsim

Sl No.	N_1	N_2	P_{true}	P_{approx}	P_{approx} **(1st Iteration)**	P_{approx} **(2nd Iteration)**
1	13604	24193	329121572	286859264	324235264	328635392
2	54793	22115	1211747195	1085521920	1202524160	1210803200
3	31501	39309	1238272809	1139392512	1221341184	1237149696
4	33893	21010	712091930	706887680	712038400	712091264
5	58113	52493	3050525709	2550595584	3020587008	3049531392
6	61814	52541	3247769374	2673442816	3204857856	3241775104
7	22509	63372	1426440348	1238990848	1397587968	1420382208
8	59897	9414	563870358	530718720	561742848	563691648
9	33989	53930	1833026770	1807187968	1832085504	1832979712

Table 6.3: Error% Comparison for Iterative Logarithmic Multiplier

Sl No.	N_1	N_2	P_{true}	**Error%**	**Error% (1st Iteration)**	**Error% (2nd Iteration)**
1	13604	24193	329121572	12.84	1.48	0.14
2	54793	22115	1211747195	10.41	0.76	0.07
3	31501	39309	1238272809	7.34	1.36	0.09
4	33893	21010	712091930	0.73	0.007	0.00009
5	58113	52493	3050525709	16.38	0.98	0.03
6	61814	52541	3247769374	17.68	1.32	0.18
7	22509	63372	1426440348	13.14	2.02	0.42
8	59897	9414	563870358	5.87	0.37	0.03
9	33989	53930	1833026770	1.4	0.05	0.002

Based on the table 6.3, an error comparison chart has been plotted below. Here Error1 = Error%($1^{st} Iteration$) and Error2 = Error%($2^{nd} Iteration$)

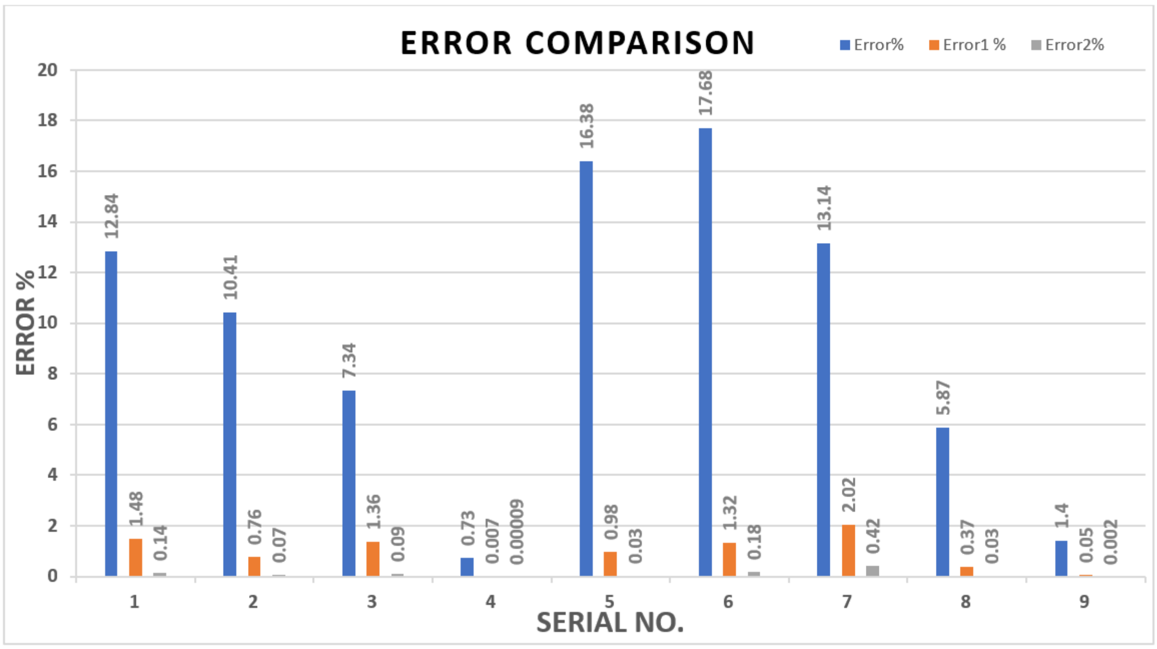
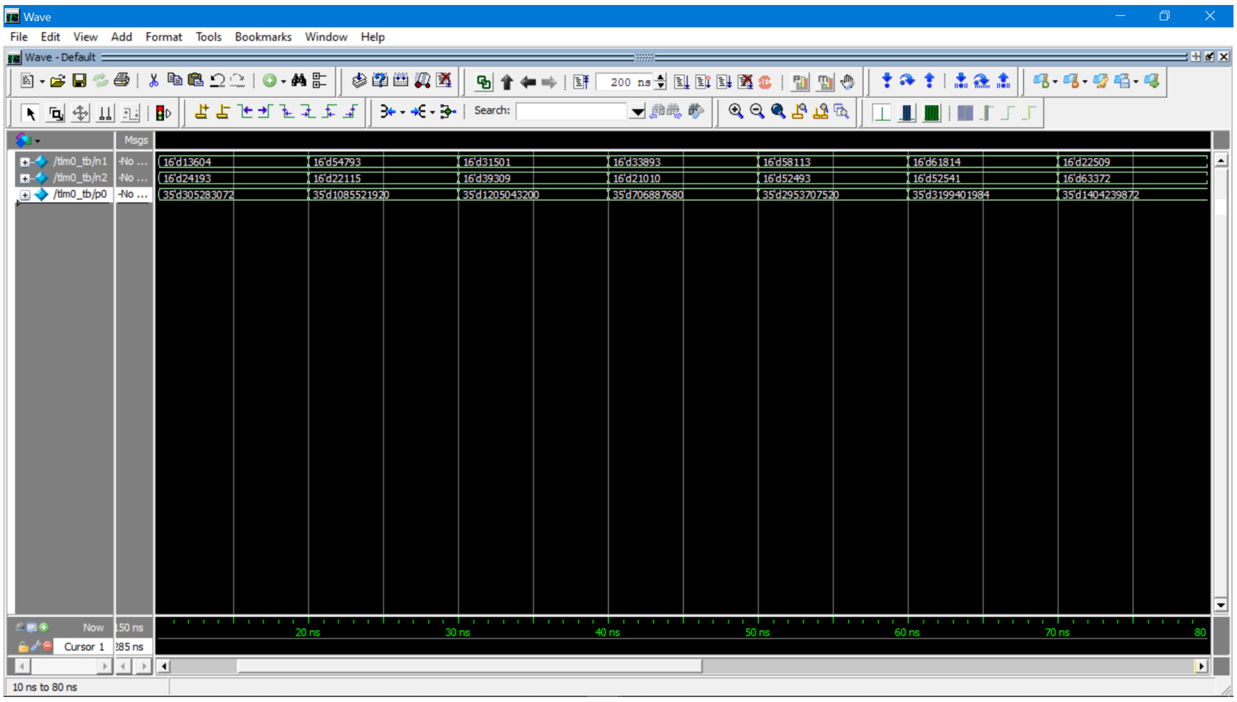


Figure 6.5: Error% Comparison Chart for Iterative Logarithmic Multiplier

6.2 Non-Iterative Logarithmic Multiplier



```
/tlm0_tb/n1  -No ...  16'd13604       16'd54793        16'd31501        16'd33893       16'd58113        16'd61814        16'd22509
/tlm0_tb/n2  -No ...  16'd24193       16'd22115        16'd39309        16'd21010       16'd52493        16'd52541        16'd63372
/tlm0_tb/p0  -No ...  35'd305283072   35'd1085521920   35'd1205043200   35'd706887680   35'd2953707520   35'd3199401984   35'd1404239872
                      20 ns  30 ns  40 ns  50 ns  60 ns  70 ns  80
```

Figure 6.6: Modelsim Simulation of Non-Iterative Logarithmic Multiplier

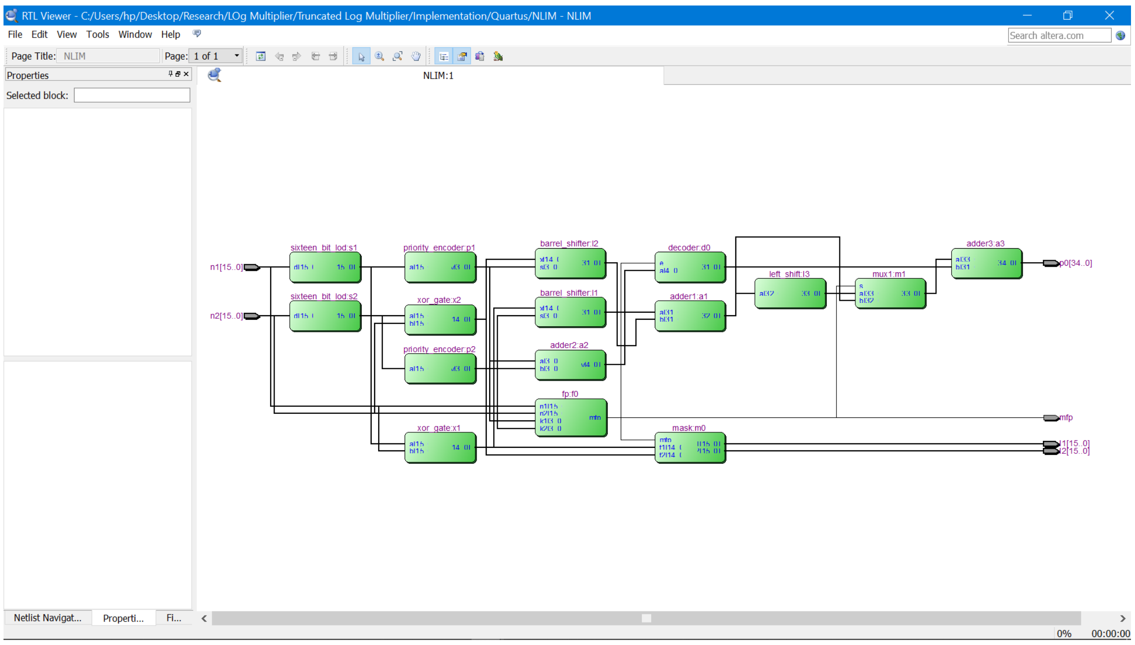


Figure 6.7: Non-Iterative Logarithmic Multiplier Implementation in Quartus Prime

Table 6.4: Comparison of Products for Non-Iterative and Iterative Logarithmic Multiplier in Modelsim

SLNo.	N_1	N_2	Carry	P_{true}	ILM	Non-ILM
1	13604	24193	1	329121572	286859264	305283072
2	54793	22115	0	1211747195	1085521920	1085521920
3	31501	39309	1	1238272809	1139392512	1205043200
4	33893	21010	0	712091930	706887680	706887680
5	58113	52493	1	3050525709	2550595584	2953707520
6	61814	52541	1	3247769374	2673442816	3199401984
7	22509	63372	1	1426440348	1238990848	1404239872
8	59897	9414	0	563870358	530718720	530718720
9	33989	53930	0	1833026770	1807187968	1807187968

ILM - Iterative Logarithmic Multiplier
Non-LIM - Non Iterative Logarithmic Multiplier
A direct comparison is made between Non-ILM and 1st Iteration of ILM.

Table 6.5: Error% Comparison of Non Iterative Logarithmic Multiplier and 1stiteration of Iterative Logarithmic Multiplier

SLNo.	N_1	N_2	Carry	P_{true}	Error%(ILM)	Error%(Non-ILM)
1	13604	24193	1	329121572	12.84	7.24
2	54793	22115	0	1211747195	10.41	10.41
3	31501	39309	1	1238272809	7.34	2.68
4	33893	21010	0	712091930	0.73	0.73
5	58113	52493	1	3050525709	16.38	3.17
6	61814	52541	1	3247769374	17.68	1.48
7	22509	63372	1	1426440348	13.14	1.55
8	59897	9414	0	563870358	5.87	5.87
9	33989	53930	0	1833026770	1.40	1.40

The below chart shows the comparison between Non-ILIM and 1st Iteration of ILM. It can be

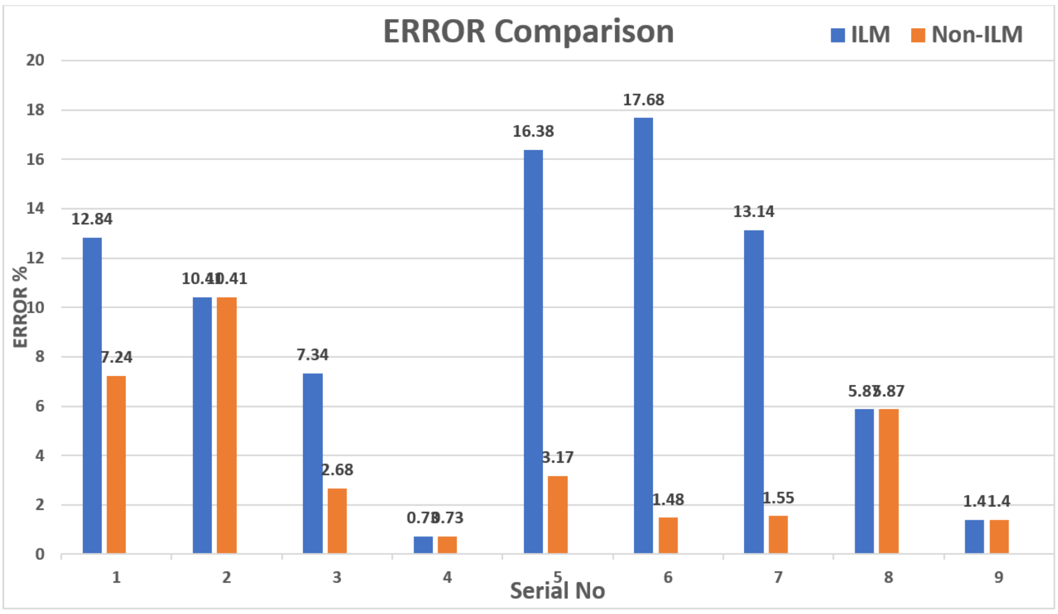


Figure 6.8: Error% Comparison Chart of Non Iterative Logarithmic Multiplier and 1stiteration of Iterative Logarithmic Multiplier

clearly observed that whenever there is no carry from the mantissa bits the error remains same. Whereas there is significant improvement in accuracy when there is carry.

6.3 Results Comparison

Table 6.6 shows the comparison of product of two 16-bit numbers obtained in modelsim between the Non Logarithmic Multiplier(Non-ILM) and all the iterations of Iterative Logarithmic Multiplier(ILM).Also an error% comparison of both the Logarithmic Multipliers is shown in table 6.7 based on table 6.6 .

Table 6.6: Comparison of Multiplication Results for Non Iterative Logarithmic Multiplier and all iterations of Iterative Logarithmic Multiplier simulated in Modelsim

Sl No	N_1	N_2	P_{true}	**Non-ILM**	**ILM**		
				P_{approx}	P_{approx} **(No Iteration)**	P_{approx} **(1st Iteration)**	P_{approx} **(2nd Iteration)**
1	13604	24193	329121572	305283072	286859264	324235264	328635392
2	54793	22115	1211747195	1085521920	1085521920	1202524160	1210803200
3	31501	39309	1238272809	1205043200	1139392512	1221341184	1237149696
4	33893	21010	712091930	706887680	706887680	712038400	712091264
5	58113	52493	3050525709	2953707520	2550595584	3020587008	3049531392
6	61814	52541	3247769374	3199401984	2673442816	3204857856	3241775104
7	22509	63372	1426440348	1404239872	1238990848	1397587968	1420382208
8	59897	9414	563870358	530718720	530718720	561742848	563691648
9	33989	53930	1833026770	1807187968	1807187968	1832085504	1832979712

Table 6.7: Error% Comparison of Non Iterative Logarithmic Multiplier and all iterations of Iterative Logarithmic Multiplier simulated in Modelsim

Sl No	N_1	N_2	**Mantissa Carry Bit**	**Non-ILM**	**ILM**		
				Error%	**Error% (No Iteration)**	**Error% (1st Iteration)**	**Error% (2nd Iteration)**
1	13604	24193	1	7.24	12.84	1.48	0.14
2	54793	22115	0	10.41	10.41	0.76	0.07
3	31501	39309	1	2.68	7.34	1.36	0.09
4	33893	21010	0	0.73	0.73	0.007	0.00009
5	58113	52493	1	3.17	16.38	0.98	0.03
6	61814	52541	1	1.48	17.68	1.32	0.18
7	22509	63372	1	1.55	13.14	2.02	0.42
8	59897	9414	0	5.87	5.87	0.37	0.03
9	33989	53930	0	1.4	1.4	0.05	0.002

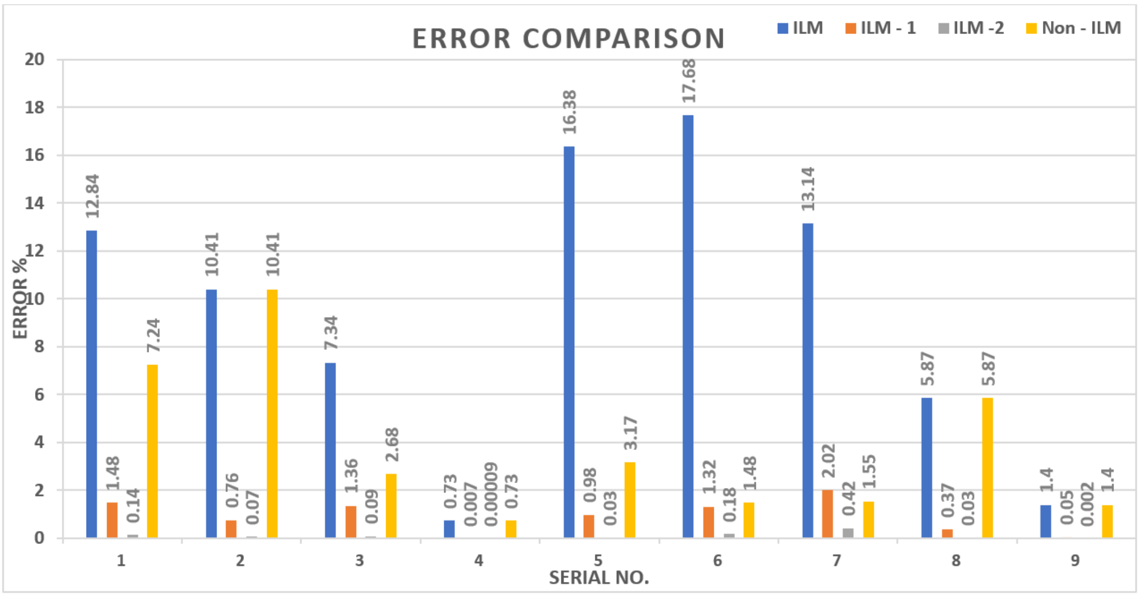


Figure 6.9: Error% Comparison Chart of Non Iterative Logarithmic Multiplier and all iterations of Iterative Logarithmic Multiplier

Figure 6.9 shows the comparison between the Non Iterative Logarithmic Multiplier vs. all the iterations of Iterative Logarithmic multiplier.Here, ILM-1 = First iteration of ILM and ILM-2 = Second Iteration of ILM
Table 6.8 shows the comparison of the design metrics of both the multipliers. The area used in

Table 6.8: Comparative Analysis of Area, Power and Delay of Iterative and Non-Iterative Logarithmic Multipliers

	Logic Elements Used	**Current (mA)**	**Power (mW)**	**Min. Propagation Delay(ns)**	**Max. Propagation Delay(ns)**
ILM	341	15.81	140.44	7.940	28.371
ILM-1	717	20.57	146.15	12.966	32.295
ILM-2	1088	25.92	152.60	18.583	33.723
Non-ILM	481	42.23	172.51	15.877	32.635

the design is shown in terms of logic elements as in this work, FPGA is used. The maximum logic elements available in this device is 114480.

Chapter 7

Conclusion

It is clear from the results that in case of Iterative Logarithmic Multiplier the error keeps on decreasing as the number of iterations increase. It is safe to assume that for a $16-$bit multiplication maximum of two to three iterations is enough if accuracy is the priority. Clearly from table 6.3, after the second iteration, the error is less than 1%. Fig. 6.2 shows that there is a drastic improvement in accuracy in the first iteration. After the second iteration there is not much of a reduction in error. In this design the precision comes with a trade off in power, area and delay. From table 6.6 it is clear that as the design complexity increases for higher precision or as the number of iterations increases, it contributes to larger area which results in more power and delay. Also it can be observed that except the area or the logic elements used there is not much of an increase in power and delay between the first and second iterations. It depends on the requirement as to which design is to be used. For energy efficient applications, a simple Iterative Logarithmic Multiplier is advised and for high precision applications Iterative Logarithmic Multiplier with two or more iterations is advised.

For an accurate comparison same inputs are taken for iterative and non iterative logarithmic multipliers. The non-iterative logarithmic multiplier is dependent on the carry bit from the mantissa to obtain better precision. In Table 6.5 carry bit is also shown to understand the error better. The comparison is made between the products of both multipliers without any iteration. Here it is clear that whenever the carry bit is 0 the error remains same, but when the carry bit is 1 there is huge improvement in precision. The same results are depicted in fig. 6.4 where the error% is decreased to huge amount whenever the carry from the mantissa bit is 1.

Finally a comparative analysis is done for non-iterative logarithmic multiplier and all the iterations of the iterative logarithmic multiplier. In fig. 6.5 it is clear that the error remain same when there is no carry from the mantissa bits. But when carry is 1, the error from the non-iterative logarithmic multiplier is closer to the first iteration of the iterative logarithmic multiplier. Surely when it comes to precision the second iteration leads all the way. Along with this it can be observed from table 6.6 that non-iterative logarithmic multiplier lies somewhere in between the 0^{th} iteration and first iteration of the iterative logarithmic multiplier in terms of maximum propagation delay and area or logic elements used. Due to higher design complexity the current and power is significantly high as compared to other forms of multipliers. The propagation delay is somewhat closer to the first iteration of the iterative logarithmic multiplier. It is safe to conclude that for minimum area and energy efficient applications the basic iterative logarithmic multiplier is recommended. For maximum precision, the more number of iterations the better. The non-iterative logarithmic multiplier offers somewhat a balance between reduced area and high accuracy applications.

Bibliography

[1] K. H. Abed and R. E. Siferd. Cmos vlsi implementation of a low-power logarithmic converter. *IEEE Transactions on Computers*, 52(11):1421–1433, 2003.

[2] S. E. Ahmed, S. Kadam, and M. Srinivas. An iterative logarithmic multiplier with improved precision. In *2016 IEEE 23nd Symposium on Computer Arithmetic (ARITH)*, pages 104–111. IEEE, 2016.

[3] S. E. Ahmed and M. Srinivas. An improved logarithmic multiplier for media processing. *Journal of Signal Processing Systems*, 91(6):561–574, 2019.

[4] M. S. Ansari, B. F. Cockburn, and J. Han. A hardware-efficient logarithmic multiplier with improved accuracy. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 928–931. IEEE, 2019.

[5] M. S. Ansari, S. Gandhi, B. F. Cockburn, and J. Han. Fast and low-power leading-one detectors for energy-efficient logarithmic computing. *IET Computers & Digital Techniques*, 2021.

[6] Z. Babič, A. Avramovič, and P. Bulič. An iterative mitchell's algorithm based multiplier. In *2008 IEEE International Symposium on Signal Processing and Information Technology*, pages 303–308. IEEE, 2008.

[7] Z. Babić, A. Avramović, and P. Bulić. An iterative logarithmic multiplier. *Microprocessors and Microsystems*, 35(1):23–33, 2011.

[8] P. Bulić, Z. Babić, and A. Avramović. A simple pipelined logarithmic multiplier. In *2010 IEEE International Conference on Computer Design*, pages 235–240. IEEE, 2010.

[9] W. Liu, J. Xu, D. Wang, C. Wang, P. Montuschi, and F. Lombardi. Design and evaluation of approximate logarithmic multipliers for low power error-tolerant applications. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 65(9):2856–2868, 2018.

[10] V. Mahalingam and N. Ranganathan. An efficient and accurate logarithmic multiplier based on operand decomposition. In *19th International Conference on VLSI Design held jointly with 5th International Conference on Embedded Systems Design (VLSID'06)*, pages 6–pp. IEEE, 2006.

[11] J. N. Mitchell. Computer multiplication and division using binary logarithms. *IRE Transactions on Electronic Computers*, (4):512–517, 1962.

[12] D. Nandan, J. Kanungo, and A. Mahajan. An efficient vlsi architecture for iterative logarithmic multiplier. In *2017 4th international conference on signal processing and integrated networks (SPIN)*, pages 419–423. IEEE, 2017.

[13] S. Ramaswamy and R. E. Siferd. Cmos vlsi implementation of a digital logarithmic multiplier. In *Proceedings of the IEEE 1996 National Aerospace and Electronics Conference NAECON 1996*, volume 1, pages 291–294. IEEE, 1996.

[14] M. B. Sullivan and E. E. Swartzlander. Truncated error correction for flexible approximate multiplication. In *2012 Conference Record of the Forty Sixth Asilomar Conference on Signals, Systems and Computers (ASILOMAR)*, pages 355–359. IEEE, 2012.

[15] M. B. Sullivan and E. E. Swartzlander. Truncated logarithmic approximation. In *2013 IEEE 21st Symposium on Computer Arithmetic*, pages 191–198. IEEE, 2013.