DESIGN AND IMPLEMENTATION OF EFFICIENT ADDER BASED FLOATING POINT MULTIPLIER

Dissertation submitted in partial fulfilment of the
requirements for the award of degree of

MASTER OF TECHNOLOGY

In

VLSI Design

Submitted By

Lokesh Bhardwaj
Roll No. 601261016

Under the guidance of

Ms. Sakshi
Assistant Professor, ECED
Thapar University
Patiala

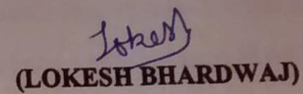
Department of Electronics and Communication Engineering
THAPAR UNIVERSITY
(Established under the section 3 of UGC Act, 1956)
PATIALA-147004 (PUNJAB)

JULY, 2014

DECLARATION

I hereby declare that the work which is being presented in the thesis entitled, **"Design and Implementation of Efficient Adder based Floating Point Multiplier"** in partial fulfilment of the requirement for the award degree of master of technology in VLSI Design submitted in Electronics and Communication Engineering Department of Thapar University Patiala, is an authentic record of my own work carried out under the supervision of Ms. Sakshi, Assistant Professor, ECED and refers other researcher's work which are duly listed in the reference section.

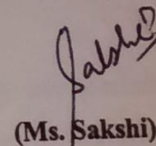
The matter presented in this thesis has not been submitted in any other University/Institute for the award of degree.



Date: 10/07/14 **(LOKESH BHARDWAJ)**

Roll No: 601261016

It is to certify that the above statement made by the student is correct to the best of my knowledge and belief.

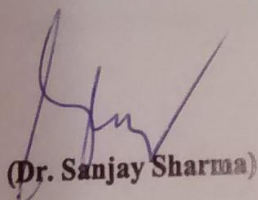


(Ms. Sakshi)

Assistant Professor

ECED, Thapar University

Countersigned by:

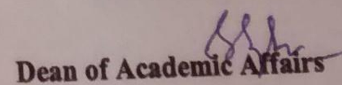


(Dr. Sanjay Sharma)

Head

ECED, Thapar University

Patiala-147004



Dean of Academic Affairs

Thapar University

Patiala- 147004

ACKNOWLEDGEMENT

To discover, analyze and to present something new is to venture on an untrodden path towards and unexplored destination is an arduous adventure unless one gets a true torch bearer to show the way. I would have never succeeded in completing my task without the cooperation, encouragement and help provided to me by various people. Words are often too less to reveals one′s deep regards. I take this opportunity to express my profound sense of gratitude and respect to all those who helped me through the duration of this thesis. I acknowledge with gratitude and humility my indebtedness to **Ms. Sakshi, Assistant Professor,** Electronics and Communication Engineering Department, Thapar University, Patiala, under whose guidance I had the privilege to complete this thesis. I wish to express my deep gratitude towards her for providing individual guidance and support throughout the thesis work.

I convey my sincere thanks to **Head of the Department, Dr. Sanjay Sharma** as well as **PG Coordinator, Dr. Kulbir Singh, Associate Professor,** Electronics and Communication Engineering Department, entire faculty and staff of Electronics and Communication Engineering Department for their encouragement and cooperation.

My greatest thanks are to all who wished me success especially my parents. Above all I render my gratitude to the Almighty who bestowed self-confidence, ability and strength in me to complete this work for not letting me down at the time of crisis and showing me the silver lining in the dark clouds. I do not find enough words with which I can express my feelings of thanks to my dear friends for their help, inspiration and moral support which went a long way in successful competition of the present study.

(LOKESH BHARDWAJ)

ABSTRACT

Multipliers are widely used in DSP processors, filters, communications systems etc. A high speed multiplier is always desirable. Floating point numbers provide larger range and better accuracy than fixed point numbers. In fixed point number representation, digits after and before the decimal is fixed while in floating representation the decimal point is not fixed, it can float. Floating point multipliers are used for calculating product of two floating point numbers. There are different steps involved in calculation of product of two floating point numbers. The major steps involved are calculation of sign, significand and exponent. For the significand, adders are required for accumulating the partial products. Adders are also used in calculation of exponent. The maximum time of multiplication is elapsed in accumulating the partial product and at the final addition stage to get the significand product. So, the multiplication time can be reduced if the partial products are accumulated with the help of fast adders.

This thesis aims to implement a single precision floating point multiplier using the fastest adder. For this, initially different adders are compared in terms of delay and then on the basis of the results obtained, the floating point multiplier is designed using the fastest adder. The multiplier is also designed using different adders at its different stages and their results are compared on the basis of delay and area. The modules have been designed in Verilog HDL, simulated and synthesized using Xilinx 14.5.

TABLE OF CONTENTS

Sr. No.	Contents	Page No.
	Certificate	I
	Acknowledgement	II
	Abstract	III
	Table of contents	IV
	List of figures	VII
	List of tables	IX
	Abbreviations	X
1.	**CHAPTER 1- Introduction**	1
	1.1 Thesis organization	2
2.	**CHAPTER 2- Literature review**	3
3.	**CHAPTER 3- Floating point multiplier**	7
	3.1 Standard for binary floating point arithmetic	7
	3.1.1 Formats for floating point numbers	7
	1. Single precision	8
	2. Double precision	9
	3. Quadruple precision	9
	3.1.2 Operations	10
	3.1.3 Rounding rules	10
	3.1.4 Exceptions	10
	3.2 Steps in floating point multiplication	11
	3.2.1 Calculation of sign bit	12
	3.2.2 Calculation of exponent	12
	3.2.3 Calculation of significand	13

1. Partial product generation 13
2. Partial product reduction 16
3. Final stage adder 17
3.2.4 Normalizing the number 18
3.2.5 Rounding the number 18
3.2.6 Checking the exceptions 19
3.3 Adders used in floating point multiplication 20
3.3.1 Half adder 20
3.3.2 Full adder 21
3.3.3 Ripple carry adder 22
3.3.4 Carry look-ahead adder 23
3.3.5 Carry select adder 24
3.3.6 Carry skip adder 25
3.3.7 Carry save adder 26
4. **CHAPTER 4- FPGA implementation using xilinx** 27
4.1 Design entry 27
4.2 Behavioral simulation 27
4.3 Design synthesis 28
4.4 Implementing the design 29
4.5 Generating programming file 31
4.6 Analyzing design using chipscope pro 31
5. **CHAPTER 5- Implementation results and conclusion** 33
5.1 Adders 33
5.2 Single precision floating point multiplier 34
5.2.1 Power calculation using Xpower estimator 36
5.2.2 Simulation results 36
5.2.3 Implementation using chipscope pro analyser 37

5.3 Conclusion 39

5.4 Future scope 39

REFERENCES **40**

LIST OF FIGURES

Figure Number	Content	Page No.
2.1	Technique for vedic multiplication	6
3.1	Representation of single precision floating point number	8
3.2	Representation of double precision floating point number	9
3.3	Representation of quadruplele precision floating point number	10
3.4	Block diagram for floating point multiplier	12
3.5	Block diagram for exponent calculation	13
3.6	Block diagram for significand calculation	13
3.7	Pairing of bits in booths algorithm	15
3.8	3:2 Compressor	16
3.9	Wallace tree structure	18
3.10	Half adder	21
3.11	Full Adder	22
3.12	Ripple carry adder	22
3.13	Carry Look-ahead adder	23
3.14	Carry Select Adder	24
3.15	Carry Skip Adder	25
3.16	Tree structure using carry save adder	26
4.1	FPGA design flow	28

4.2	Steps in synthesis process	29
4.3	Different files generated in implementation process	30
5.1	Comparison of 8,16 and 32 bit adders on the basis of delay	34
5.2	Delay comparison for single precision multiplier	35
5.3	Simulation result for floating point multiplier	36
5.4	Block diagram for FPGA implementation	37
5.5	Output results on chipscope pro analyser	38

LIST OF TABLES

Table Number	Content	Page No.
3.1	IEEE basic formats for floating point numbers	7
3.2	Special number representations for single and double precision	9
3.3	IEEE rounding modes	11
3.4	Operations involved in radix-2 booth algorithm	15
3.5	Operations involved in radix-4 booths algorithm	16
3.6	Truth table for half adder	20
3.7	Truth table for full adder	21
5.1	Synthesis report for various adders	34
5.2	Synthesis report for single precision floating point multiplier	35
5.3	Power for single precision floating pint multiplier	37

ABBREVIATIONS

ASIC	Application specific integrated circuit
CPU	Central processing unit
DSP	Digital signal processor
FPGA	Field programmable gate array
HDL	Hardware description language
IEEE	Institute of Electrical and Electronics Engineers
ISE	Integrated Software Environment
LSB	Least significant digit
LUT	Look up table
MAC	Multiply and accumulate
MSB	Most significant digit
RTL	Register transfer level
RAM	Random access memory
VLSI	Very large scale integration
XST	Xilinx synthesis technology

Chapter 1

INTRODUCTION

Multipliers are widely used in many high performance systems. They are used as small blocks in large systems like Digital signal processors, filters, communication systems, microprocessors etc. DSP"s have operations like convolution operations, transform operations, filtering operations etc. All these operations need multiplication unit. So multiplier plays an important role in DSP"s. In older microprocessors, multiplication is performed by repeated addition but in recent microprocessors multiplication instructions are available. In communication systems for baseband signal transmission multiplier is used. Filter contains large number of multipliers, so use of fast multipliers is necessary for high speed applications. Multipliers also find application in 3D graphics and in video processing for displaying streamline images.The time taken by multiplier in calculating final result are very important and plays major role in determining its performance. In DSP chips multiplication time plays important role in calculating instruction cycle time. So there is a need of high speed multiplier.

A multiplier is a digital circuit, which is used to multiply two numbers. Digital multiplier multiplies two numbers in a similar manner as it is done in mathematics i.e. addition of shifted partial products. Multiplication of unsigned numbers can be done by using AND gates and full adders. AND gates are used for generating the partial products and full adders are used for adding the generated partial products. For signed number multiplication the negative numbers are first converted into its 2"s complement representation. This makes sure that the all the partial products are positive.There are three types of digital multipliers i.e. serial, parallel and serial-parallel multiplier. In serial multiplier, both the operands are entered serially, therefore it requires less area. Due to less area requirement of serial multiplier it has less hardware cost and low power consumption. But the speed of the serial multiplier is poor because the operands are entered serially. In parallel multiplier operation is carried out in parallelwhich increases the speed of the multiplier. But parallel multipliers occupy larger area and more complex as compared to serial multipliers. High power consumption is also one problem in parallel multipliers. Parallel multipliers can further classified into array multipliers and tree multipliers. Most widely used array multipliers are Braun multiplier, booth multiplier, Robertson multiplier, Baugh-Wooley multiplier etc. Tree multipliers like

Wallace tree multiplier, dada tree multiplier etc. are also used for carrying out fast multiplication. Serial-parallel multipliers are used to take advantage of high speed operation of parallel multiplier and small area of serial multiplier. In serial-parallel multipliers, one operand is entered serially while other is stored in parallel. This requires less area and also enhances the speed of the multiplier.

Fixed point and floating point representations are used for representing numbers. In fixed point number representation digits after and before the decimal is fixed while in floating representation the decimal point is not fixed, it can float hence given the name floating point. Floating point numbers are used for representing extremely large and small values. For example if we have to represent the distance between sun and earth or we have to represent mass of electron, then there is a need of floating point numbers. Although the floating point representation is slower than fixed point representation but it provides larger range and better accuracy. Today many microprocessors come with floating point units, for performing floating point arithmetic.

In today"s world there is a requirement of fast speed, low power small equipment"s. Fast speed can be achieved by decreasing the delay from to input. Area can be reduced by decreasing the sizes of transistors used in making gates. Power dissipation also decreases with area. But as we know that we cannot improve speed, area and power simultaneously. So there is a trade-off between these three parameters. Sometimes product of these two quantities is used for estimating performance of the system. Power delay product is an important factor for measuring system performance.

1.1 Organisation of the report

The report is organized as follows:

- Chapter 2- Describe the applications of the multiplier and gives a brief about the floating point numbers.
- Chapter 3- Discusses the IEEE 754 standard, Floating point multiplication algorithm and the adders used in floating point multiplication.
- Chapter 4- FPGA flow using Xilinx is discussed in this chapter.
- Chapter 5- Implementation results are described and report is concluded.

CHAPTER 2

LITERATURE REVIEW

SR. Kuang et. al. [1] proposed a low power variable latency floating point multiplier which is suitable for embedded floating point applications. The architecture splits the significand multiplier into upper and lower parts, and predicts the carry bit, sticky bit and significand product from upper part. Experimental result show that proposed multiplier can save respectable power and energy when compared to the fast multipliers at the expense of slight area and acceptable delay overheads.

Y. He et. al. [2] proposed a new redundant booth encoding scheme, in which the idea was to polarize two adjacent booth encoded digits to directly form an RB partial product to avoid the hard multiple of high radix booth encoding without incurring any correction vector. Synthesis results show that the RB multiplier designed with proposed booth encoding algorithm exhibit higher speed and less energy-delay product than the existing multiplication algorithms.

C. Nagendra et. al. [3] surveyed several classes of parallel, synchronous adders based on their power, delay and area characteristics. The adder studied include the linear time ripple carry adder and Manchester carry chain adders, the square root time carry skip and carry select adders, the logarithmic time carry look-ahead adder and its variations, and the constant time signed-digit and carry save adders. All the adders have been simulated using HSPICE.

The ELM adder was found to be best among the two‟s complement adders for entire range of precision studied.

J. Kaur et. al. [4] used compressors for increasing the speed of the multiplier. The conventional multiplier was compared with the multiplier containing compressors. They found that the use of compressors not only reduced the vertical critical path but also the stage operations. The multiplier with compressors also reduced time delay of the multiplier.

M. Al-Ashrafy et. al. [5] presented an efficient implementation of an IEEE 754 single precision floating point multiplier. The multiplier implementation handles the overflow and underflow cases. To give more precision to the multiplier in a multiply and accumulate (MAC) unit rounding was not implemented. The design achieved 301 MFLOPs with latency of three clock cycles. The multiplier was verified against Xilinx floating point multiplier core. The performance of the multiplier was enhanced using pipelining stages. These are used

to divide the critical path and increase the maximum operating frequency of the multiplier.The pipeline stages are used at different locations i.e. in the middle of significant multiplier, in the middle of exponent adder, after the significant multiplier and the exponent adder and finally at the outputs.The synthesis tool “retiming” option was used so that the synthesizer uses its optimization logic to better place the pipelining registers across the critical path.

R.P.P. Singh et. al. [6] presented performance analysis of different fast adders. They compared different adders on the basis of three parameters i.e. area, speed and power consumption. They also presented a design methodology of hybrid carry lookahead/carry skip adders (CLSKAs) which is combination of carry look-ahead adder and carry skip adder.In conventional carry skip adder, each block consists of ripple carry adder and skip logic is used after each block to generate carry for next block. The speed of operation depends on carry propagation from previous block to next block. In CLSKAs, they used a carry look-ahead logic in each block to generate carry for next block. The modified carry skip adders presented in the paper provided better speed and power consumption as compared to conventional carry skip adder and other adders like ripple carry adder, carry look-ahead adder, Ling adder and carry select adder. The modified Carry Skip adder architecture (hybrid carry lookahead/carry skip adders) with fix block size proposed by them gave better result than other adders in terms of speed but required more area and power consumption.

P. Gurjar et. al. [7] described the construction of high speed adder circuit using Hardware Description Language (HDL). The motivation behind this is that an adder is a very basic building block of arithmetic and plays an important role in determining the performance of central processing unit (CPU). They have simulated and synthesised the various adders like full adder, ripple carry adder, carry look-ahead adder, carry skip adder, carry select adder and carry save adder. The simulated results are verified and functionality of high speed adders was compared on parameters like speed, area etc. The result obtained for 8-bit, 16-bit and 32-bit are shown in figure 2.1 below. They concluded that carry save adder is the best adder in terms of speed and area consumption.

A. Jain et. al. [8] proposed architecture for a fast 32-bit floating point multiplier. The design was intended to make a multiplier faster by reducing the delay caused by the propagation of carry by implementing adders having the least power delay constant. After comparing various adders on basis of power and delay, they concluded that Kogge-Stone adder is the fastest.The main idea behind this design is to increase the speed of multiplier by reducing delayat every

stage using optimal the adder design. They used Kogge-stone adder at each stage of the multiplier to increase the speed of the multiplier.

G. Renxi et. al. [9] implemented high speed floating point multiplier. In the design of the floating point multiplier, they used new Radix-4 Booth‟s encoding algorithm for generating the partial products and the improved 4:2 compression structure for reducing the partial products. The final sum and carry vectors generated by compressors are added by a carry look-ahead adder to obtain the final product. The timing simulation results show that the floating point multiplier can be stably run at the frequency 80 MHz.

M.A. Franklin et. al. [10] compared the performance of six adders design with respect to their average delays. They have shown that asynchronous adders (32 or 64-bit) with a hybrid structure (e.g. carry select adders) run 24% faster than simple ripple carry adders. They have shown the results that most hybrid adders (in addition to ripple carry adders) have variable (data dependent) delays and this variability can be exploited through use of an asynchronous design. Simulation results shows that a 64-bit carry select adder runs faster than its ripple carry counterparts. They also noted that an adder design which is well suited to the clock environment (CSA) may not be a good option in the asynchronous environment.

M.K. Jaiswal et. al. [11] proposed a hardware efficient approach for implementing a fully pipelined large integer multipliers and further extended it to quadruple floating point multiplication. Good results are obtained when they compared their design with the best large integer multipliers and also QP floating point multipliers. The proposed work has shown attractive area efficient designs for several large integer multipliers, and it can be extended for even much larger integer multipliers. The proposed architectures achieved high performance with smaller area and shorter latency.

L.S.A. Hamid et. al. [12] implemented a high speed generic multiplier and adder/subtractor single precision floating point units. In order to achieve a high maximum operating frequency for both the multiplier and the adder/ subtractor units, each unit was optimized separately by optimizing its bottle neck block. The bottle neck of the floatingpoint multiplier unit is the multiplier block. While the bottle neck of the adder/subtractor module is the normalization block which was responsible for adjusting the result to the IEEE 754normalized format after the result from addition/subtraction has been calculated. Many algorithmshave been introduced aiming to speed up the multiplier block. In this paper, they used anovel multiplication algorithm to optimize the multiplier block, where the operands tobe multiplied were sliced into smaller blocks. They referred the proposedmultiplieras "Block Multiplier".

A. Kahne et. al. [13] used vedic multiplication technique to implement floating point multiplier. In this paper the Urdhvatriyakbhyam sutra technique was used for multiplication of mantissa. The underflow and overflow cases were also handled. Multiplication technique used in vedic maths is shown in figure below. The use of less LUTs verified that the hardware requirement is reduced, thereby reducing the power consumption.

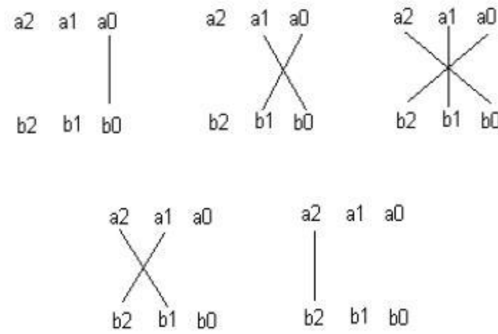


Figure 2.1: The vedic multiplication method [11]

K.L.S. Swee et. al. [14] compared performance of radix-2, radix-4, radix-8, radix-16 and radix-32 booth encoding multipliers. They found that as the radix number of multiplier is increased both the area and delay performance degraded due to increase in complexity of the multiplier. The result obtained by them shows that the radix-4 booth multiplier gives best performance in in terms of area and speed constraints.

CHAPTER 3

FLOATING POINT MULTIPLIER

This chapter describes the IEEE 754 standard for floating point arithmetic, the various steps involved in floating point multiplication and the adders which can be used at different places in floating point multiplier.

3.1 Standard for binary Floating-Point Arithmetic(IEEE 754)

The IEEE 754 standard was developed in 1985 by IEEE. The latest version of this standard is IEEE 754-2008. The standard defines arithmetic formats and interchange formats, the operation in the specified formats, rounding modes and exception handling.

3.1.1 Formats for floating point numbers

The IEEE standard defines five formats for floating point numbers. There are three binary formats and two decimal formats. The three binary formats are known as single, double and quadruple precision while the decimal formats are called as double and quad. These five formats are also called as basic formats. Table 3.1 shows the basic formats used for floating point numbers

Table 3.1: IEEE basic formats for floating point numbers

Basic formats	Base value	Bits or digits
Binary32	2	23 bits
Binary64	2	52 bits
Binary128	2	112 bits
Decimal64	10	16 digits
Decimal128	10	34 digits

The standard also specifies the extended formats. These formats are used for increasing the precision and range of the exponent of the basic formats. Using these formats user can specify the precision and exponent range. The standard also defines the interchange formats which are used for the exchange of the data. Below three basic binary formats are discussed:

1. Single precision: In single precision 32 bits are used for representing a number. The most significant bit is used for storing sign of the number, next lower 8 bits are used for storing exponent and next 23 bits are used for storing mantissa of the number. Figure 3.1 shows the single precision representation of floating point number. For representing exponent a bias value of 127 is added to actual exponent, to make negative numbers possible without using extra bit for sign. Some special numbers are defined for the single precision representation. Table 3.2 shows all these numbers which are possible for single precision representation. These special numbers are used for checking exceptions.

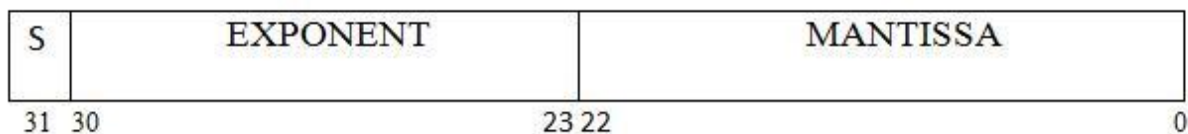


Figure 3.1: Representation of single precision floating point number

Below the example shows how the floating point number is represented in single precision format.

Suppose the number is 1050.25

First the number is converted into binary.

Binary representation of the number = 10000011010.01

Now the decimal point is shifted to get the leading „1". In this case decimal point is shifted 10 times to get leading one. After 10 shifts number can be represented as

1.000001101001

M is value after the decimal point, So the value of mantissa can be written as

M= 00000110100100000000000

Now value 10 is added to the 127 to get the correct exponent value. The binary value corresponding to 137 is the exponent value. The number is positive so the value of S is 0.

S= 0

E= 10001001

The floating point representation for 1050.25 is

01000100100000110100100000000000

2. Double precision: Double precision takes twice the size of single precision. It uses 11 bits for exponent and 53 bits are used for mantissa part. Figure 3.2 shows the bits arrangement for double precision floating point number. The small increase in exponent and large increase in mantissa increases the range and accuracy for a representing a number.This format can be extended to improve accuracy. 80 bits are used for extended format. 12 bits are appended to the mantissa and 4 bits are added to the exponent. 80 bit extended format with 15 bit exponent and 64 bit significand is used in computer hardware‟s like intel Pentium series and Motorola 6800 series. Special numbers associated with double precision floating point representation are show below in Table 3.2. These special numbers are used for detection of exceptions.

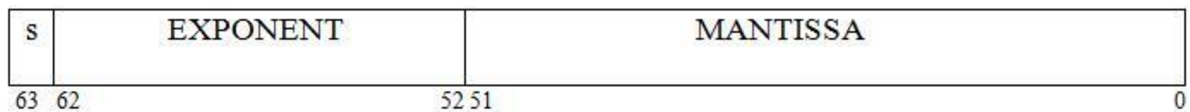


Figure 3.2: Representation of double precision floating point number

Table 3.2 Special number representations for single and double precision

Single precision		**Double precision**		Number
Exponent	Mantissa	Exponent	Mantissa	
0	0	0	0	0
0	Non zero	0	Non zero	Denormalized
1-254	Any value	1-2046	Any value	Normalized
255	0	2047	0	Infinity
255	Non zero	2047	Non zero	NaN

3. Quadruple precision: In quadruple precision 128 bits are used for representing a number. 112 bits are used formantissa, 15 bits are used for exponent and most significant digit is used for sign. This format is not only used for representing the numbers but also for computations of double precision results. When overflow occurs in double precision computations then

quadruple precision format is used. Below Figure 3.3 shows the bits arrangement for quadruple precision floating point number.

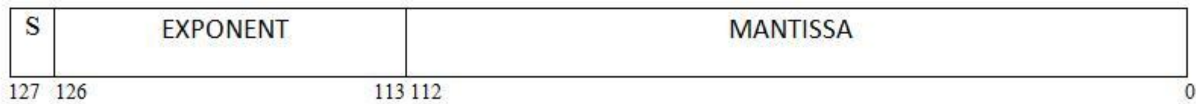


Figure 3.3: Representation of quadruple precision floating point number

3.1.2 Operations

The standard includes all the arithmetic operations like add, subtract, multiply, divide, square root etc. The standard defines conversion between formats, their scaling and quantization.It also defines functions related to sign like copying and manipulating the sign. Comparisons and total ordering functions are also defined in it. Testing and setting of flags can also be done. Some other miscellaneous operations are also used in this standard.

3.1.3 Rounding rules

Five rounding rules are defined in IEEE 754 standard. These are round to nearest that ties to even, round to nearest that ties away from zero, round towards zero, round towards positive infinity, round towards negative infinity. The first two are round to nearest modes and rest three are direct rounding modes. In round to nearest ties to even, if the number falls midway it is rounded to nearest value with an even LSB. This rounding mode is by default for binary floating point numbers.In round to nearest ties away from zero, if the number falls midway it is rounded to nearest value above for positive numbers and below for negative numbers. Round towards zeromode directly rounds towards zero while round towards positive infinity and round towards negative infinity rounds towards positive and negative infinity. Table 3.3 shows how positive and negative numbers are rounded in all five formats.

3.1.4 Exceptions

There are five exceptions which are defined in this standard. These five exceptions are invalid operation, divide by zero, overflow, underflow and inexact. Invalid exception occurs when operation that is not defined in standard is performed. Like if square root of a negative number is taken then this exception occurs. When infinite result is obtained by operation on finite operands then divide by zero exception occurs. Overflow occurs when the value to be represented is too large to be fit in exponent field. Underflow occurs when the value to be

represented is too small to be fit in exponent field. If the value of the exponent is 8 bit in size and between 1 and 254 then the value is normalized. Inexact exception occurs when the obtained result is by default correct.

Table 3.3 IEEE rounding modes

Rounding mode	**+6.5**	**+7.5**	**-6.5**	**-7.5**
To nearest, ties to even	+7.0	+7.0	-7.0	-7.0
To nearest,Away from zero	+7.0	+8.0	-7.0	-8.0
Towards 0	+6.0	+7.0	-6.0	-7.0
Towards +infinity	+7.0	+8.0	-6.0	-7.0
Towards –infinity	+6.0	+7.0	-7.0	-8.0

3.2 Steps in floating point multiplication

Floating multiplication is similar to the integer multiplication, in addition it involves some extra steps. The multiplication of mantissa part is similar to unsigned integer multiplication. But exponent calculation, sign calculation and normalization are extra operations that must be carried out.The general equation for a normalized floating point number can be written as

$$Z = (-1^{s}) * 2^{E-Bias} * (1.M) \quad (1)$$

Where M is mantissa part, E is exponent part and S is sign bit of the number. If two numbers of this kind are multiplied than different steps are carried out for obtaining the correct result. After multiplication of two numbers the equation for resultant number can be written as

$$Z = (-1^{S1\ XOR\ S2}) * (2^{E1+E2-Bias}) * (1.M1 * 1.M2) \quad (2)$$

where S1 and S2 are signs of two numbers, E1 and E2 are exponents of two numbers and M1 and M2 are mantissas of two numbers. Figure 3.4 shows the complete block diagram for floating point multiplier. The various steps involved in calculating the product are calculation of sign, calculation of exponent and calculation of significand. The steps include are described below.

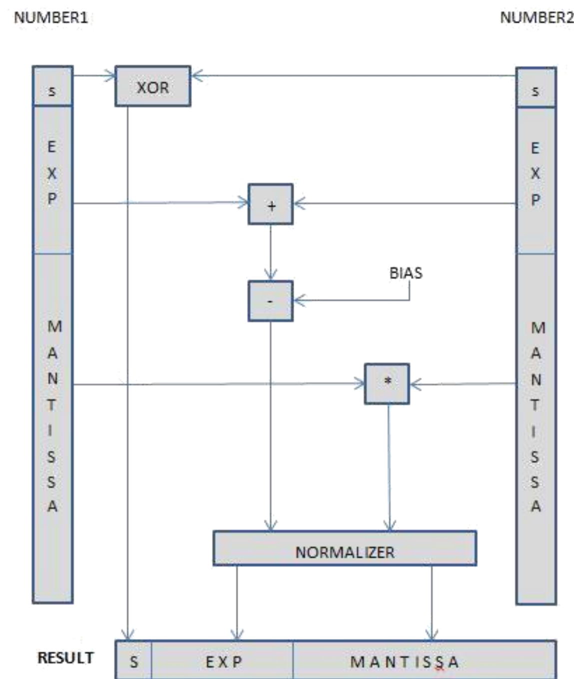


Figure 3.4: Block diagram for floating point multiplier

3.2.1 Calculation of sign bit

The sign bit of the product is calculated by performing exclusive-or operation on the sign bit of two operands. If both numbers are positive or negative then the result is positive. But if the numbers are of opposite sign then result will be negative. The similar operation is performed by an ex-or gate. A zero sign bit represent positive number and a one sign bit represent negative number.

3.2.2 Calculation of exponent

Exponent is calculated by adding exponents of two operands and then subtracting the bias. Initially the operands are in biased notation, so the bias is added twice. Hence a bias is subtracted to get the correct result. Bias value is fixed i.e. 127 for single precision floating point multiplier. So we require two adders, one for adding the exponents and second for adding the intermediate result and 2‟s complement of the bias. Bias can also be subtracted by using ripple borrow subtractors. Resultant exponent can be written as

$$E = E1+E2- bias$$

Figure 3.5 shows the block diagram for exponent calculation

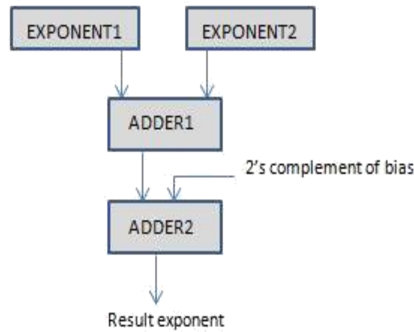


Figure 3.5: Block diagram for exponent calculation

3.2.3 Calculation of significand

Significand is calculated by unsigned multiplication of two numbers after appending „1" to mantissa of each number, because this 1 is hidden. The calculation of significand involves different steps. First partial products are generated, then partial products are passed through different compressors. At last stage carry propagate or other fast adder is used to compute the final result. Figure 3.6 shows the different steps involved in calculation of significand.

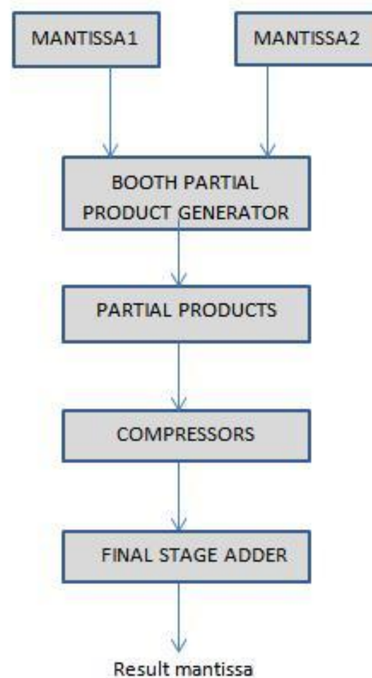


Figure 3.6: Block diagram for significand calculation

The steps for calculating significand are described below:

1. Partial product generation: Partial products can be generated by multiplying the multiplicand by a „0" or „1". If a „0" is multiplied then the result is string of zeros so there is

no need to add that partial product. But if a „1‟ is multiplied then the result is the multiplicand itself so that partial product must be added. Multiplication can be done by using AND gates because the AND gate performs the multiplication of the input bits applied to it. But it takes large amount of area. Hence for high performance systemsthis method of multiplication is not used. The operation of multiplication becomes fast if the generated partial products are reduced. Different algorithms are used for reducing the partial products. Partial products generated can be reduced by using booth‟s multiplier. In next section booth‟s multiplier is described.

Booth's multiplier

Booth‟s multiplier uses booth‟s algorithm. The algorithm was proposed by A.D. Booth in 1951. Using this algorithm multiplication time can be reduced. The exact amount of time reduction depends on bit pattern of the multiplier. If the multiplier has a stream of 1‟s, the number of addition required is minimized. This algorithm can be used with both signed and unsigned number. It uses 2‟s complement representation for handling the signed numbers. There are different versions of booth‟s algorithm. These are radix-2, radix-4, radix-8 etc. Radix-2 and radix-4 booth‟s algorithm are explained below:

Radix-2 booth's algorithm

In this algorithm first a zero is appended to the LSB of the multiplier. Then two adjacent bits of the multiplier starting from the LSB and appended zero are examined. Depending upon the two bits a specific operation is carried out which is shown in table 2. In this manner 1 bit of the multiplier number is eliminated in each pass. The multiplication procedure carried out in this algorithm is based on the underlying fact that k long sequence of 1‟s is equivalent to k-1 long sequence of 0‟s. This replacement of 1‟s by 0‟s helps in reducing the partial products.The reduction in partial products speeds up the operation of multiplier. In radix-2 booth‟s algorithm we need to generate only two operations. But this algorithm becomes inefficient when there are isolated ones. This drawback can be overcome by modified booths algorithm. The other disadvantage of this algorithm is that it is convenient to use it when only synchronous multiplier is designed. The problem associated with radix-2 booths algorithm can be overcome if three adjacent bits re processed rather than two bits. As the processed bits increased the total number of cycles in obtaining the multiplication gets reduced. So the as the radix number increases the speed of the multiplier is also increased.

Table 3.4 Operations involved in radix-2 booth algorithm

X(i)	X(i-1)	Operation on multiplicand
0	0	One bit shift to the right only
0	1	Add multiplicand to existing sum of partial products and then shift result right by one bit
1	0	Subtract multiplicand from the current sum of partial products and then shift result right by one bit
1	1	One bit shift to the right only

Radix-4 Booth's algorithm

The radix-4 booth algorithm generates partial products depending on the three bit pairs of the multiplier. The pairing starts after appending a zero to the LSB. The 3 bits pairs are formed in such a way that lowest bit is from the previous pair and other two bits are next two bit of the multiplier as shown in figure 3.7. Pairing is carried out until the MSB of the number is achieved. Based upon pairing of these three bits different multiples of multiplicand is generated. These multiples may be any of {+A,-A,-2A,+2A,0A} these values depending upon the table as shown in table 3.5. The value +A is simply the multiplicand. The value +2A is obtained by shifting the multiplicand by one to left. –A is generated by taking two‟s complement of the multiplicand. While -2A is obtained by shifting and two‟s complementing the multiplicand. In this way the partial products generated are reduces to half. The main aim of the booth‟s algorithm is to generate more and more zero‟s in the design

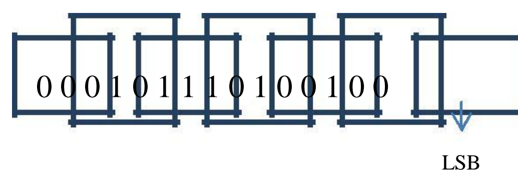


Figure 3.7 Pairing of bits in booths algorithm

Table 3.5 Operations involved in radix-4 booths algorithm

Bit Pairs	Operation on multiplicand
000	0
001	+A
010	+A
011	+2A
100	-2A
101	-A
110	-A
111	0

Radix-8 booth's algorithm

In radix-8 algorithm 4 bits are processed simultaneously. It is similar to radix-4 algorithm but 4 bits are paired instead of three. Multiplier using this algorithm generates less partial products than radix-4 algorithm. The computation of partial products becomes difficult as the operation carried out becomes difficult. We need to generate +1,+2,+3,+4,-1,-2,-3,-4. The generation of 3A is very difficult because there is a requirement of a full adder. First 2A is calculated then A is added to obtain the correct result. In radix-8 booth"s algorithm less number of partial products are generated but large number of operations are required.. So the complexity increases. But its speed is fastest among radix-2 and radix-4.

The disadvantage of radix-2 booth"s algorithm is that it becomes inefficient when there are isolated ones. In some cases it requires more operations than usual multiplication. As the radix number increases more operations are required. Therefore radix-8 booths algorithm increases the complexity. So radix-4 algorithm gives the best results without increasing the complexity. With this algorithm partial products can be reduced to almost half. This will increase the performance of the multiplier with limited operations.

2. Partial product reduction: Partial products can be reduced by using compressors. Compressors are nothing but carry save adders that generates separate bit strings for sum and the carry. In carry save adder the carry bit is not propagated rather generated for each bit.

This makes operation faster than ripple carry adder. Figure 3.8 shows a 3:2 compressor. In 3:2 compressor sum and carry bit is calculated for each set of 3 input bits.

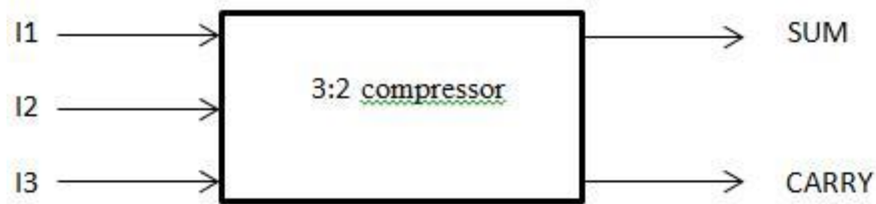


Figure 3.8 3:2 Compressor

The 4:2 and 5:2 compressors are also widely used in high speed multipliers to accumulate partial products. The structure of compressors makes them faster than simple adders for adding the partial products. Compressors are also used in tree structures. Compressors combined in different configurations to form different tree structures. In Wallace tree structure different inputs are applied to the different compressor. Then outputs of these compressors are further given to other compressors. This process continues until the output only has 2 numbers, one is final sum bit and other is final carry bit string. This two bit string must be added by carry propagate adder or any other fast adder. Figure 3.9 shows the structure of the of a Wallace tree. Here eight inputs are applied. The first six sets of input are applied two first stage carry save adders. Remaining two inputs are applied to the next stage carry save adders with first stage outputs. The third stage carry save adder produces two bit strings that are given to the carry propagate adder. In this way Wallace tree reduces partial products at each stage.

Other tree structure which is mostly used is dadda tree structure. Dadda tree do as few reductions as possible, due to this it has less expensive reduction phase but requires large number of adders. In dadda tree first three wires of same weight are applied to full adders. The result will be two outputs, one with same weight as of inputs and other with higher order. If the two wires of same weight left and current number of outputs with that weight is equal to 2 or modulo3 then they are given to a half adder otherwise they are passed to next layer. If a single wire is left then it is connected to next layer. Some other trees are also used like overturned stair tree, 4:2 tree etc.

3. Final stage adder: Final stage adder is used to add last two bit strings. This adder plays very important role in determining the performance of the multiplier, as the size of this adder is almost twice the size of the operands to be multiplied. If a fast adder is used then speed of the multiplier can be increased.

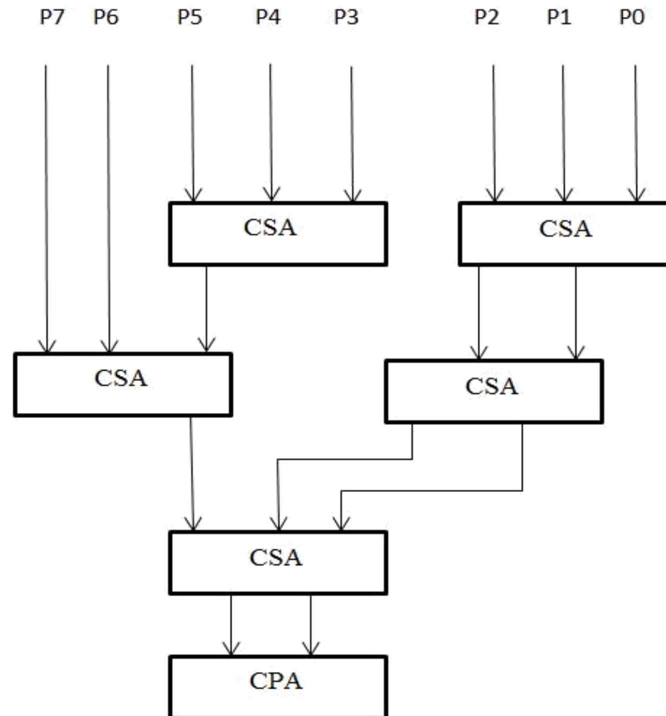


Figure 3.9 Wallace tree structure

3.2.4 Normalizing the number

Normalizing means getting a „1" at MSB of the significand. This „1" must be present at left of decimal point i.e. after 46th bit of the significand. So a „1" is detected and exponent is adjusted accordingly. Two cases may arise. A one may be either at 46th bit or at 47th bit of final 48 bit final result. If leading „1" is at 46th bit the number is already a normalized number and no shift is needed. While if the leading „1" is at 47th bit then result is shifted to the right and exponent is incremented by one. So the correct result is obtained only after normalizing the number. After normalization rounding is done to make result more accurate by discarding some bits.

3.2.5 Rounding the number

Precision is lost when bits are shifted in normalizing the number. So rounding is implemented to get the best result from the given bits.Various rounding modes are used to get accurate result. Rounding modes described by IEEE standard areround to nearest ties to even, round to nearest ties away from zero, round towards zero, round towards positive infinity, round towards negative infinity. Round to nearest even mode is by default mode for floating point numbers.

3.2.6 Checking the exceptions

Exceptions occur when the resultant number after multiplication is out of range for that particular representation. These exceptions can be checked by raising the flag register. The exceptions in IEEE standard are invalid operation, divide by zero, overflow, underflow and inexact.

Example:-

For simplicity the multiplicand and multiplier are taken as same

First number: 1050.25

Representation in single precision format: 01000100100000110100100000000000 where

S1 = 0

E1 = 10001001

M1 = 00000110100100000000000

Second number: 1050.25

Representation in single precision format: 01000100100000110100100000000000 where

S2= 0

E2= 10001001

M2= 00000110100100000000000

The value of the final product can be calculated as

S= S1 XOR S2; 0 XOR 0 = 0

E = E1+E2-*bias*; 10001001+10001001-*bias*= 10010011

M= (1.M1)*(1.M2); (1.00000110100100000000000)*(1.00000110100100000000000)

This multiplication yields final value of 52 bits. But the significant 23 bits are used and after normalization the value for resultant mantissa is 00001101010010110001000. The final representation for resultant product is

01001001100001101010010110001000 where

S= 0 E= 10010011 M= 00001101010010110001000

3.3 Adders used in floating point multiplication

Addition is the basic operation that is crucial to processing the fundamental arithmetic operations. It is used extensively in many VLSI design paradigms and by far the most frequently used operation in general-purpose system and in application-specific processors. The operations of subtraction, multiplication and division usually rely on operation of addition, so addition is very important part of the arithmetic units. Floating point multipliers also use adders. The adders are used to add the exponents, to add the generated partial products and at the final stage. Adders are also used for designing the compressors to increase the speed of multiplication operations. These adders play an important role in performance of the multiplier. The time elapsed in adding the partial product"s is the critical parameter for measuring the performance of the multiplier. Different adders are which can be used in floating point multiplier are described below

3.3.1 Half adder

A half adder is a digital circuit that is used to perform two bits addition. It has two inputs and two outputs. The inputs are bits to be added and outputs are resultant sum of inputs and carry generated. The truth table for the half adder is shown below in Table 3.6. in which value for output sum and carry for each input combination is given. The simplest half adder design is shown in figure 3.10 which uses an XOR gate for calculating the sum while AND gate for calculating carry. The half adder is a basic building block in digital systems which is used in other digital blocks as a sub-block. For example using two half adders and a OR gate a full adder can be designed.

Table 3.6 Truth table for half adder

Input		Output	
A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

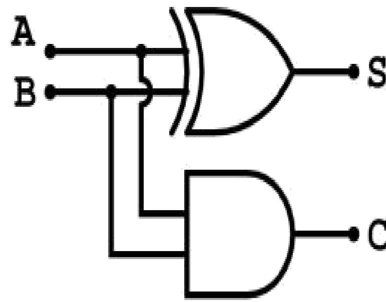


Figure 3.10: Half adder [7]

3.3.2 Full adder

A Full Adder (FA) is a digital circuit that performs an addition operation on three binary digits. The full adder produces a sum and a carry value. It is different from half adder because it accounts for values carried in as well as out. A one-bit full adder adds three one bit numbers, often written as A, B, and C where A, B are the operands and C is a bit carried in (in theory from past addition). The circuit produces two outputs sum and carry represented by the signals S(sum) and Co (Carry). The truth table for full adder is shown below in table 3.7.

Table 3.7 Truth table for full adder

A	B	C_{in}	C_{out}	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

The Boolean equations for calculating the sum and carry value for the full adder are written below.

$$S = A \text{ XOR } B \text{ XOR } C \quad (3)$$

$$Co = (A \text{ AND } B) \text{ OR } (B \text{ AND } C) \text{ OR } (C \text{ AND } A) \quad (4)$$

The logic diagram for the full adder is shown below in figure 3.11.

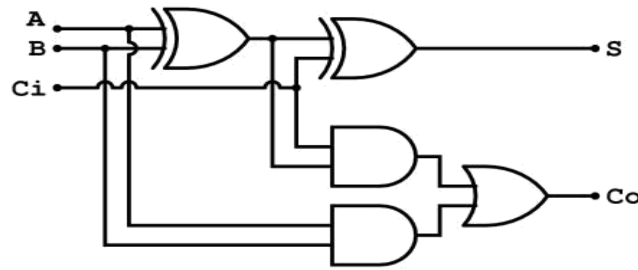


Figure 3.11: Full Adder [7]

3.3.3 Ripple carry adder

Ripple carry adder is nothing but the cascading of full adders in series i.e. carry from previous full adder is connected as input carry for the next stage. Full adder is a basic building block of Ripple carry adder. Therefore, to design n-bit parallel adder, there is a requirement of n full adders. The major limitation of Ripple carry adder is that as the bit length goes on increases, delay also increases. Therefore, Ripple carry adder is not suitable if large number bits are to be added.The increase in delay is due to carry propagation in ripple carry adder. The worst case delay in ripple carry adder occurs when carry ripples from least significant bit to most significant bit. This is given by

$$T = (n-1)tc + ts \tag{5}$$

Where tc is the delay of carry stage of full adder and ts is the delay to compute sum of last stage. Therefore situations when high speed performance is crucial and the minimum amount of hardware is under performing, using RCA in arithmetic operation would be detrimental. The circuit diagram for 4-bit RCA is shown below in figure 3.12

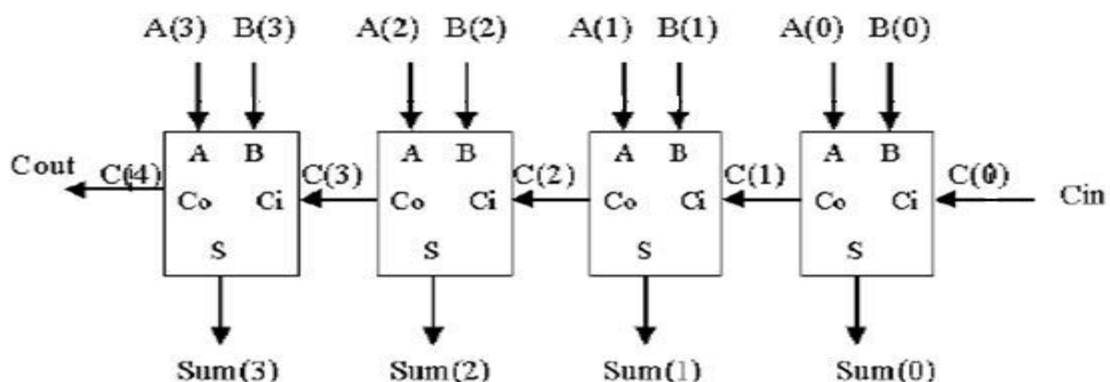


Figure 3.12 Ripple Carry adder [7]

3.3.4 Carry look-ahead adder

Look-ahead carry adder speed up the operation of addition, because in this algorithm carry for the next stages is calculated in advance based on input signals. The delay time of carry look-ahead exhibits logarithmic dependency on size of adder which decreases the propagation delay of carry signal. The carry look-ahead adder has propagation time in O(log n) and area requirement in O(n log n).If X and Y are two inputs, c(i) is initial carry, S(i) and C(i+1) are the output sum and carry respectively, then Boolean expression for calculating next carry and addition is:

$P(i) = X(i) \text{ XOR } Y(i)$ (Carry Propagation) (6)

$G(i) = X(i) \text{ AND } Y(i)$ (Carry Generate) (7)

$S(i) = P(i) \text{ XOR } c(i)$ (Final Sum) (8)

$C(i+1) = G(i) \text{ OR } (P(i) \text{ AND } C(i))$ (Final Carry) (9)

If X(i)= Y(i) then carry of 1 is generated if X(i)= Y(i)= 1,carry of 0 is generated when X(i)= Y(i)= 0. If X(i) is not equal to Y(i) then carry is said to be propagated. The structure of CLA for 4bit is shown below in Figure 3.13 below. In this structure partial full adders are used which are used to generate sum output and propagate and generate signals. Using propagate and generate signals carry output is calculated.

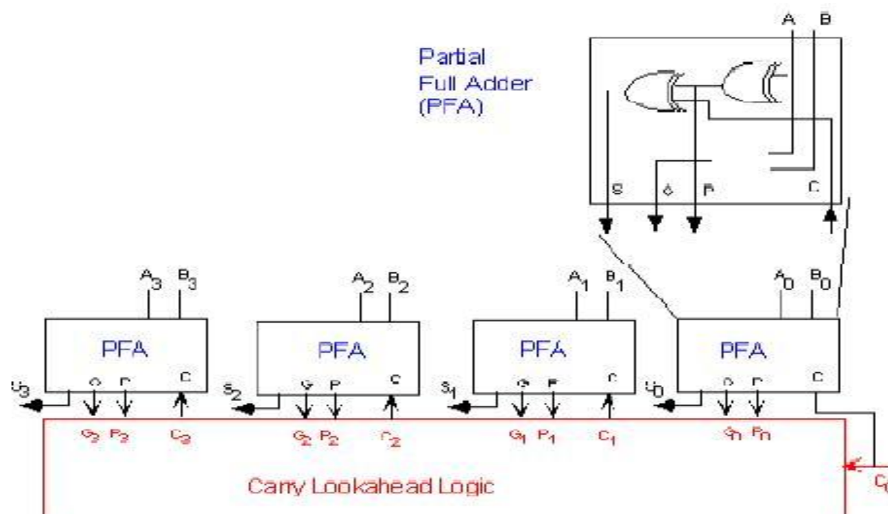


Figure 3.13 Carry Look-ahead adder [7]

3.3.5 Carry select adder

The concept of the carry-select adder is to compute alternative results in parallel and subsequently selecting the correct result with single or multiple stage hierarchical techniques.The increase in the speed performance of the carry-select adder is at the cost of increase in its area. In carry-select adder both sum and carry bits are calculated for the two alternatives: input carry “0” and “1”. Once the carry-in is delivered, the correct computation is chosen (using a MUX) to produce the desired output. Therefore instead of waiting for the carry-in to calculate the sum, the sum is correctly output as soon as the carry-in gets there. Time taken to compute the sum is then avoided which results in a good improvement in speed. But the area requirement of carry select adder increases because two ripple carry adders are used one for carry in value „1‟ and other for carry in value „0‟.Carry-select adders can be divided into equal or unequal sections. Figure 3.14 shows the implementation of an 8 bits carry-select adder with 4-bit sections. For each section, shown in figure, the calculation of two sums is accomplished using two 4-bit ripple-carry adders. One of these adders is fed with a 0 as carry-in whereas the other is fed a 1. Then using a multiplexer, depending on the real carryout of the previous section, the correct sum is chosen. Similarly, the carryout of the section is computed twice and chosen depending of the carryout of the previous section. Below figure shows 8 bit carry select adder.

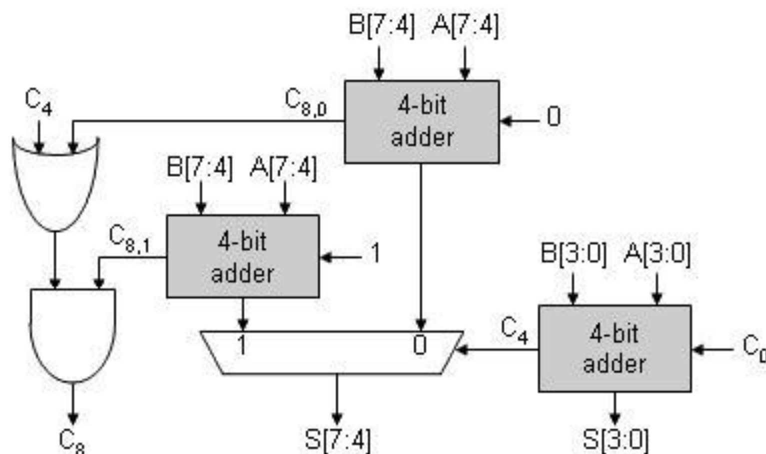


Figure 3.14: Carry Select Adder [7]

This concept can be expanded to any length for example a 16-bits carry-select adder can be composed of four sections each section is computed twice for both values of input carry.

Further each of these sections can composed of two 4-bits ripple-carry adders. This is referred as linear expansion. Sometimes carry select adder and carry look-ahead adder are combined to increase the performance to take advantages of both the adders. The combination of two adders is termed as hybrid adders. Carry select adder is also a hybrid adder because ripple carry adders are used in it.

3.3.6 Carry skip adder

In ripple carry adder the carry propagates through different adders. If carry propagation is accelerated then speed of the adder can be increased. This is accomplished by using Carry-Propagate signals within a group of bits. If all the propagate signals within the group are 1 then the carry bypasses the entire group. The condition for the carry to bypass the entire group is

$$P(i, i+3) = P(i+3).P(i+2).P(i+1).P(i) \tag{10}$$

Where P(i,i+3) is group propagate signal while Pi is individual propagate signal. Figure 3.15 shows a carry skip adder. Here group of four full adders are taken. If all the propagate signals from the full adders are 1 then carry can be bypassed. The output from the AND gate is ORed with C(i+4) which is carry generated signal to produce a group carry output.

$$\text{Carry} = C(i+4) + P(i, i+3).Ci \tag{11}$$

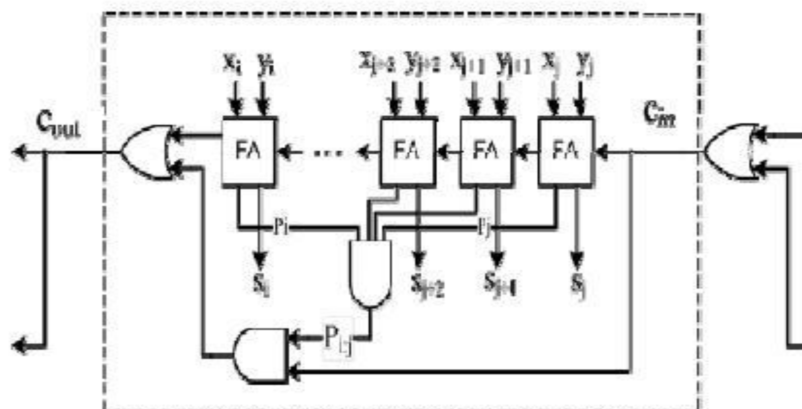


Figure 3.15 Carry Skip Adder [7]

As shown in the figure 3.15, if P(i,i+3) = 0, then the carry out of the group is determined by the value of C(i+4). However if P(i,i+3) = 1 and the carry-in bit Ci is = 1, then the group carry-in is automatically send to the next group of adders. The name “carry-skip” is due to the fact that if the condition P(i,i+3).C(i) is true then the carry-in bit skips the block entirely.

3.3.7 Carry save adder

Carry save adder are based on the idea that a full adder really has three inputs and produces two outputs. While it usually associates the third input with a carry in, it could equally well be used as a regular value. The full adder can be used as 3:2 reduction networks. It takes three bits from 3 words, adds them and produces two bits, one for sum and other for carry bits wide. An n-bit carry save adder can be built by using n separate adders. The name carry save arises from the fact that we save the carry out words instead of using it immediately to calculate the final sum. Carry-save adders are useful in situations where we need to add more than two numbers. Since the design automatically avoids the delay in carry out bits. Carry save adders connected in various configurations are called as trees.Different tree structures are Wallace tree, dadda tree, overturned stairs tree, 4:2 compressor tree. Below figure 3.16 show the Wallace tree configuration. Carry save adders are used in adding partial products.

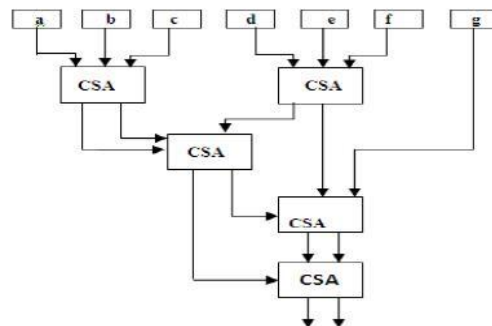


Figure 3.16 Tree structure using carry save adder [7]

Carry save adders are used as compressors. Accumulation of partial products becomes easy when compressors are used because the value for sum and carry strings is evaluated simultaneously. These strings are further given as input to other compressors. And at last only two bit strings remains. There is need of only one carry propagate adder at last stage to add last two strings of sum and carry.

CHAPTER 4

FPGA IMPLEMENTATION USING XILINX

FPGAs are semiconductor devices which has extremely useful property of field programming. Field programming means user or designer can reprogram it after its manufacturing. FPGA contains configurable logic blocks, input/output blocks and programmable interconnections. They also contain storing elements like flip flops or blocks of memory. The FPGA implementation ofdesigns are done to check their functionality on actual hardware. The cost of implementation and design cycle time of ASICs are large therefore the bigger designs are first checked on FPGA and if they give satisfactory results then there ASIC implementation is done. Figure 4.1 shows the different steps involved in the FPGA implementation. It includes design entry through HDL, behaviour simulation, logic synthesis, design implementation, bit stream generation and finally programing the FPGA. Xilinx provides all the functions necessary for implementing a design on FPGA. The xilinx ISE suite includes ISIM simulator for functional and timing simulation, XST for synthesis applications, planahead tool for floor planning, Xpower analyzer for estimating power, iMPACT for directly configuring FPGA device and chipscope pro analyser for debug and verification purposes.The different steps involved in FPGA implementation using xilinx are described below:

4.1 Design entry:

This is the first step in FPGA implementation. In design entry source files are created for representing the given design. The source file can be HDL file such as Verilog or VHDL, schematic file, embedded processor file, or EDIF file.

4.2 Behavioral simulation:

Behavioral simulation is performed for checking the design functionality. Simulation can be performed on HDL source files, HDL test benches, waveform files, Simulation-only HDL source files. To perform the simulation xilinx either uses ISIM or modelsim simulator. Simulation can also be done after synthesizing the design. The simulation after place and route is called timing simulation.

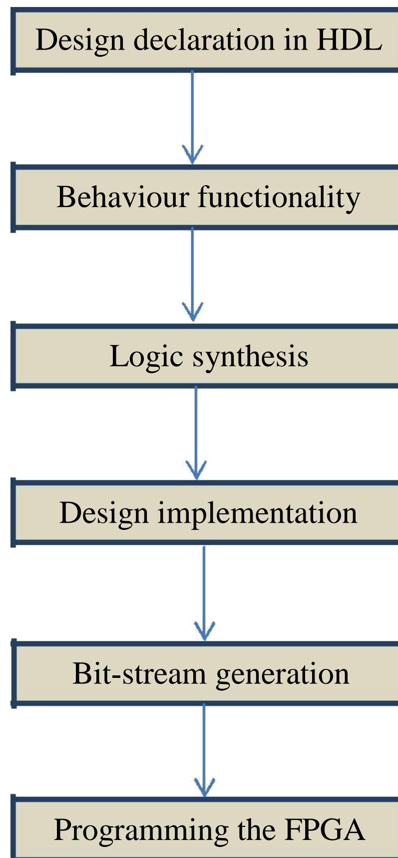


Figure 4.1 FPGA design flow

4.3 Design synthesis:

Xilinx uses XST(xilinx synthesis technology) synthesizer for synthesizing VHDL and Verilog codes. It generates netlist file for given design. This file contains logic design data and constraints. The netlist file has an extension .NGC and serves as input for the translate process. This file is the result of three steps i.e. HDL parsing, HDL synthesis and low level optimization. In HDL parsing, code is checked for syntax errors. In HDL synthesis part code is analysed and inferred to basic macros like RAM, multiplexers, adders etc. This is done for efficient implementation of technology. In HDL synthesis part FSM recognition is also done. Synthesizer chooses one of the several FSM encoding algorithms according to the optimization goal for efficient implementation. In low level optimization the macros are transferred into technology specific components such as carry logic, shift registers, clock buffers etc. This step also contains timing optimization, technology mapping and register replication. So the NGC file is generated after HDL parsing, HDL synthesis and low level optimization. Figure 4.2 shows the complete process of synthesis from HDL code to NGC file generation.

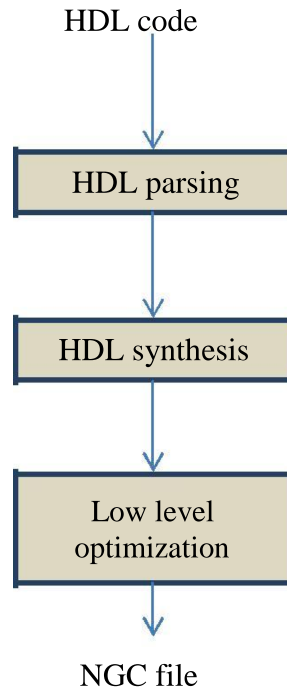


Figure 4.2 Steps in synthesis process

Besides generating NGC file XST generates RTL schematic, technology schematic and synthesis report.

- RTL schematic: This is the register transfer level representation of the pre optimized design. In this representation basic building blocks like adders, AND gate, OR gate etc. are used. This view helps in discovering issues in starting phase of design because the design is still not mapped on the target device.

- Technology schematic: This is the representation of NGC file in terms of logic elements of the targeted device like LUTs, I/O buffers etc. The technology schematic is generated after optimization of the design.

- Synthesis report: This report is the result of entire synthesis run. It contains area, delay and timing estimations. This report consists of several other reports like HDL synthesis report, advanced synthesis report, Device utilization summary, Partition resource summary, timing report etc.

4.4 Implementing the design:

Using implementation process the design is transferred onto the desired device.
Implementation of design is done in three different steps. These steps are translate, map and

route. The output NGC file from synthesis process is taken as input for the design implementation process. After translate, map and place and routing operation routed NCD file is generated which is used to generate bit file. The generated bit file can be burned on the FPGA. Different files are generated in this whole process. Figure 4.2 shows different intermediate files generated in the process of implementation.

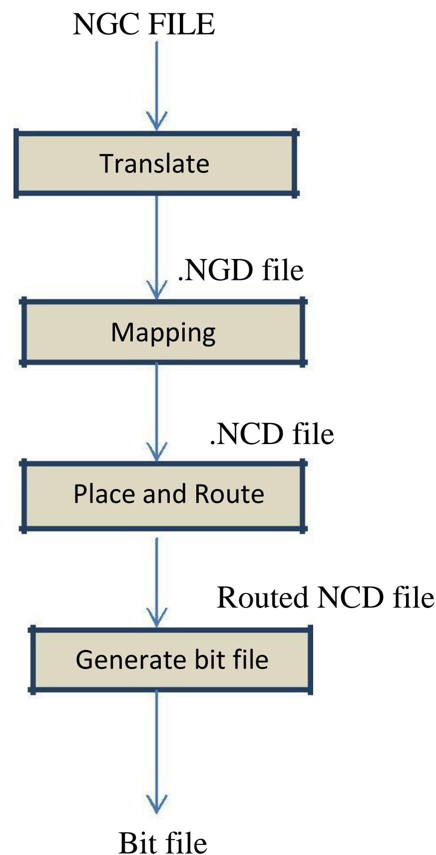


Figure 4.3 Different files generated in implementation process

Steps involved in the implementation process are written below:

- Translate: In this process all the input netlists and design constraints are combined and saved in a file called as native generic database file. The ports available in design are assigned to physical elements of the target device. Timing requirements of the design are also specified in translate process. Translate properties can also be changed by modifying them.
- Mapping: After translate process mapping is done. In mapping the circuit is divided into sub-blocks. The sub-blocks are made so that they can fit into FPGA sub-blocks. A file is generated called as native circuit description file. This file contains our design mapped into components of FPGA.

- Place and route: sub-blocks of map process are converted into logic blocks and connected in place and route step. This process takes NCD file as input and outputs the routed NCD file. Here placement and routing of blocks is done.

4.5 Generating programming file

In this process bit file is generated for particular xilinx device from the routed NCD file. The output bit file contains binary bits necessary to program the device. Sometimes this process is also called as bit-stream generation. The generated bit file is used to program the FPGA device.

4.6 Analysing design using chipscope

The FPGA designs are becoming more complex, due to need of faster designs and shorter design times. Debugging and verification is important factor in determining the complete design time. It takes almost 50% of the design time. But the xilinx chipscope pro software performs faster debugging and verification. It shrinks overall design by 25%. It is a powerful tool that is easy to use. It is used for debug, verification and inserting short signal sequences. Chipscope pro uses FPGA resources like block RAM for trigger and data storage, slice logic for trigger comparison. It uses three types of flowsi.e. core generator, core inserter and plan-ahead flow. The core inserter flow is similar to plan-ahead flow and provided in plan-ahead software. In core generator flow the core is instantiated in source HDL, while in core inserter flow the core is inserted into generated file after synthesis. Different types of cores are used by chipscope software like ICON core, ILA core, VIO core, IBA core. Some of them are explained below:

- VIO core: It defines and generates virtual i/os. It is used to apply stimulusand read outputs transition on the node we want to select. This core is used to generate virtual input and outputs. It provides options for synchronous and asynchronous inputs and outputs, where each can have width of 256 bits. There is also option of clock. It can be system clock or JTAG clock The outputs of the design which we want to implement are connected to the input of the VIO core and inputs of the design are connected to the output of the VIO core therefore the inputs are virtual LEDs and outputs are virtual DIP Switches. This core is controlled by the ICON core. So a control port is also provided. VIO core uses FPGA logic not RAM. It is only used in core generator flow.

- ICON core: It is used to control up to 15 capture cores. It acts as interface between JTAG interface and capture cores. The main function of this core is to control the other cores. It can be used in both the core generator and core inserter flows.
- ILA core: It is a capture core. It can be used to create custom triggers when activated causes data to be stored during circuit operations. Signals can be stored depending on the condition specified by used.A design can contain upto 15 ILA cores.
- Agilent trace core: It used to store large amount of data off chip or when customer uses agilent analyzer. ILA with agilent trace is similar to ILA exceptdata is captured off chip or by agilent trace port analyzer.

CHAPTER 5

IMPLEMENTATION RESULTS AND CONCLUSION

This chapter discusses about the implementation of various adders and the single precision floating point multiplier, and their synthesis and simulation results. Synthesis report describes actual hardware utilization and timing constraints of the design. Simulation results describe the behavioural functionality of the design. Implementation of the design is done to transfer the design on to actual hardware. The software used for the simulation, synthesis and implementation is xilinx. The working environment for all the designs is

- Tool version : ISE 14.5
- Optimization Goal : Speed
- Design Strategy : Balanced
- Family : Spartan 3E
- Device : XC3500E
- Speed : 4
- Package : FG320
- Simulator : ISIM
- Total slices: 4656
- Total LUTs: 9312

5.1 Adders

From the discussion of floating point multiplication algorithm it is clear that there is a requirement of adders at different stages of floating point multiplier. Adders are used in floating point multiplier for adding the exponent, accumulating the partial products and at the final stage. So there is a need of fast adders that can be used at different stages of multiplier. The basic adders are ripple carry adder, carry-look-ahead adder, carry skip adder, carry save adder and carry select adder. So all these adders of size 8, 16 and 32 bit are designed in Verilog HDL, simulated and synthesized in xilinx 14.5. The synthesis reports for all these adders are shown in table 5.1. Figure 5.1 shows the delay comparison of various adders.

Table 5.1 Synthesis report for 8,16 and 32-bit adders

S.No.	Adders (8,16 and 32 bits)	No. of slices (out of 4656)	Total Delay (in ns)
1,	Ripple carry	9/18/37	14.8/24.6/44.3
2.	Carry-look ahead	9/39/79	12.8/19.6/26.7
3.	Carry skip	13/28/56	13.9/18.9/29.0
4.	Carry-select	14/31/66	11.5/16.7/27.3
5.	Carry-save	13/29/61	13.7/23.2/42.2

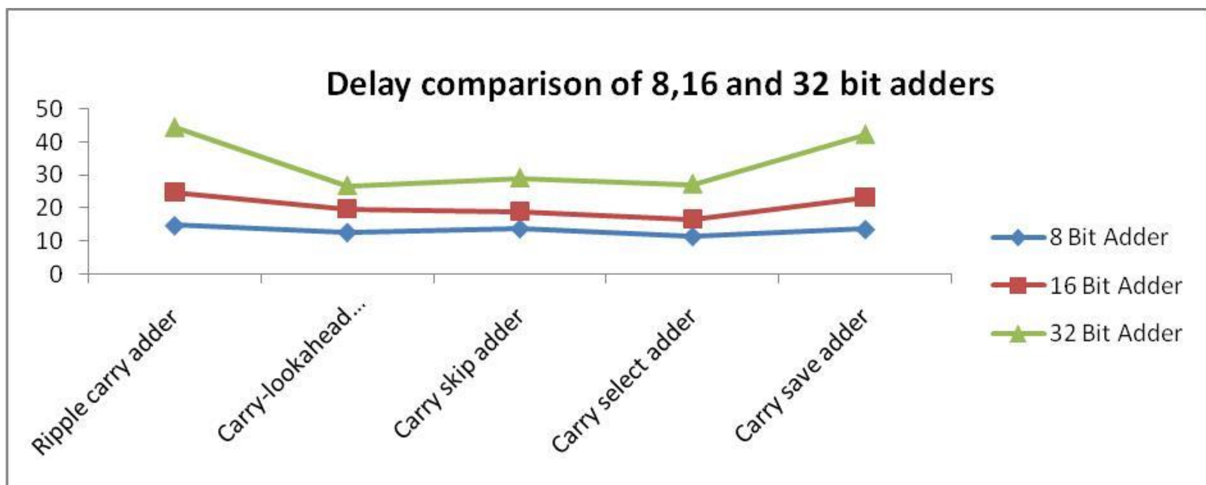


Figure 5.1 Comparison of 8,16 and 32 bit adders on the basis of delay

5.2 Single precision floating point multiplier

From the figure 5.1 it is clear that the carry select adder has the least delay, hence it is fastest of all adders. But this delay is at the cost of area. The size of the carry select adder is larger than other adders. So if we use carry select adder for floating point multiplier then its speed can be increased. But in previous literatures [8] carry save adder was used for floating point multiplication. So the floating multiplier first implemented using carry select adder at each stage and then using carry save adder at each stage. Figure 5.2 shows the delay comparison of single precision multiplier using carry select and carry save adders. The multiplier using carry

select adder did not gave improvement in delay so the multiplier is also designed using carry select adder at exponent stage and at the final stage while carry save adder for adding partial products. All these multipliers are designed in Verilog HDL, simulated and synthesized in xilinx 14.5. Table 5.2 shows the total delay and area utilization of the multiplier using three different configurations.

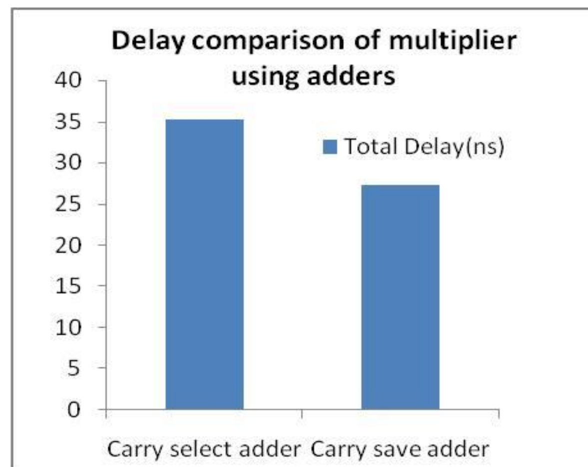


Figure 5.2 delay comparison for single precision multiplier

Table 5.2 Synthesis report for single precision floating point multiplier

S.No.	Multiplier using adders	Area utilization(No. of slices)	Total Delay
1.	Carry select Adder	1315/4656	35.222ns
2.	Carry save adder	977/4656	27.314ns
3.	Carry select for exponent and final stage while carry save for partial products	972/4656	27.600ns

From the synthesis report it is clear that the area utilization and total delay of the floating point multiplier with carry select adder stage is more than the multiplier with carry save adder at each stage. So if carry save adder is used at each stage of the multiplier then its gives better performance in terms of delay rather than using carry select adder at each stage.

5.2.1 Power calculation using Xpower estimator :

The power of the implemented design can be calculated by Xpower estimator software of xilinx. The power is calculated by providing values for toggle rate, clock speed and importing the mapped file of the implemented design for the specific device. We can also calculate power by modifying thermal parameters, clock and resource information, worst case voltages and connectivity parameters. Below table 5.3 shows total power for the three implemented multipliers.

Table 5.3 Total power for single precision floating point multiplier

S.No.	Multiplier using adders	Total power(in W)
1.	Carry select Adder	0.131
2.	Carry save adder	0.138
3.	Carry select for exponent and final stage while carry save for partial products	0.137

5.2.2 Simulation results for floating point multiplier

The design is simulated using ISIM simulator. Figure 5.3 shows the simulation results for single precision floating point multiplier. Considering the following inputs in single precision format

m1 = 01000100100000110100100000000000 (for 100ns) is the multiplicand

m2 = 01000100100000110100100000000000 (for 100ns) is the multiplier

clk = changing 0 to 1 (for 100ns with period 20ns) is the clock

The output obtained after calculating the sign, significand and the exponent using the steps described above should be

m3 = 01001001100001101010010110001000

The obtained results on the ISIM simulator are similar to the desired results as shown in figure below figure 5.3.

Figure 5.3 Simulation result for floating point multiplier

5.2.3 Implementation using chipscope pro analyzer :

The design is verified by chipsope pro tool of xilinx. The code designed in Verilog HDL with necessary cores are instantiated in top module. VIO core is generated for providing virtual inputs and outputs. The outputs of the design are connected to the input of the VIO core and inputs of the design are connected to the output of the VIO core. This core is also provided with clock input to synchronize the operation. This same clock is also connected to the design block also. ICON core is generated for controlling the VIO core. The main function of the ICON core is to control the other cores. After generating the VIO and ICON core a top module is designed which instantiates both these cores and the design module. Then synthesis of top module is done. The RTL schematic for the top module is shown in figure 5.5. The figure shows that the connections are according to our desire. After that the top module is implemented on FPGA. The generated bit file is burned on the FPGA. The results can be verified by using chipscope anaylzer window. The output will be generated on the port

sync_in when inputs are given on async_out port. The value for the sync_in port we get after applying values to async_out is written below.

async_out= 0100011000101101011100001000000001000110001011010111000010000000

sync_in = 01001100111010110000001000000000

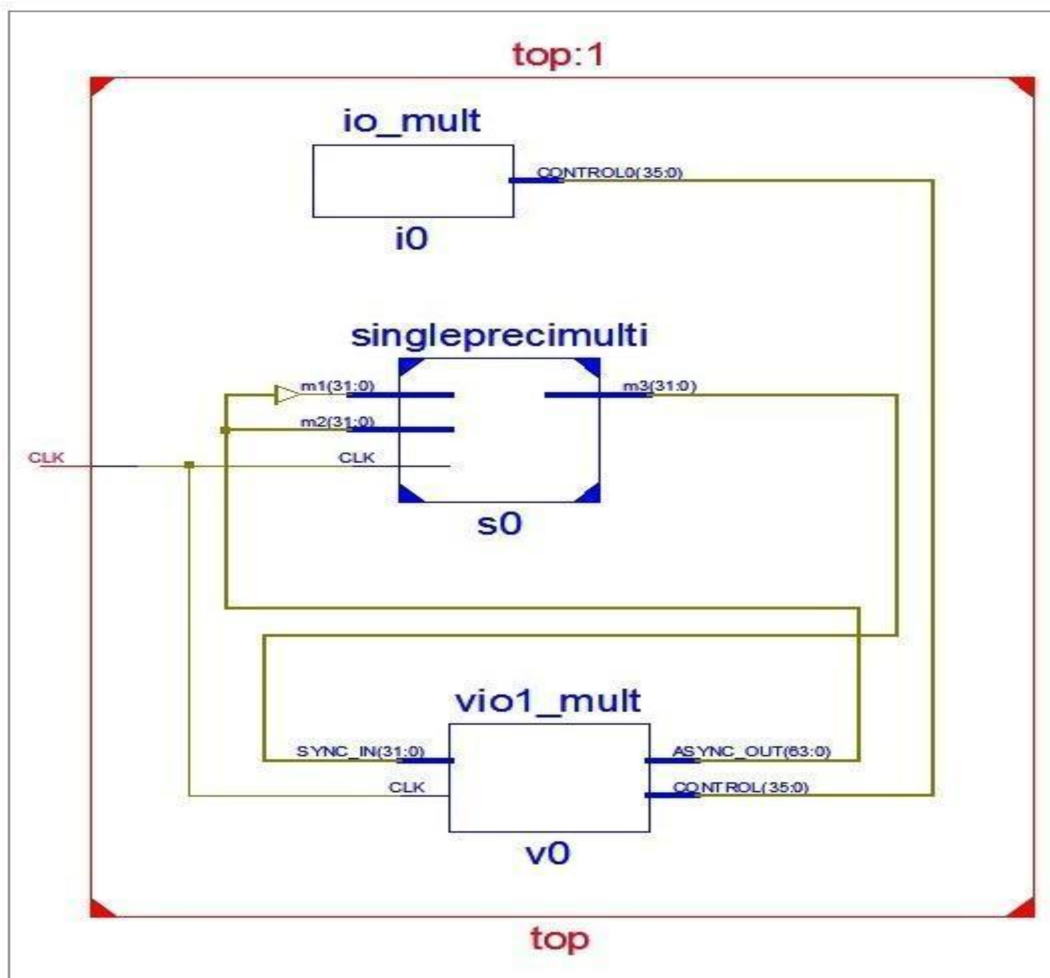


Figure 5.4 Block diagram for FPGA implementation

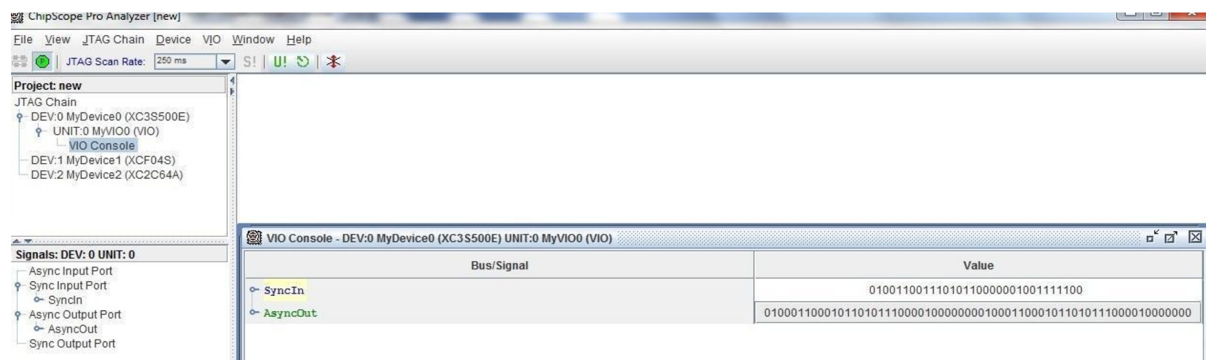


Figure 5.5 Output results on chipscope pro analyzer

5.3 Conclusion

In this thesis the floating point multiplier is implemented using carry select adder and carry save adders at each stage. First different adders are compared based on the total delay. Figure 5.1 shows that the carry select adder is fastest. So it can be used in the different stages of the multiplier. In earlier works[8] carry save adder was implemented in floating point multiplier. So both the multipliers are implemented and compared based on delay. Figure 5.2 shows the delay comparisons for both the multiplier. According to synthesis results, carry-select adder individually gave the best performance in terms of delay, for example 32 bit carry select adder provided 55% improvement in delay as compared to 32 bit carry save adder. But when carry select adder is used at each stage of single precision floating point multiplier, the performance of multiplier has degraded by 29% as compared to carry save adder. Hence carry select adder is better a choice for using as an adder individually but for multiplication applications carry save adder is preferred. From the delay and power results it is clear that the multiplier with carry select adder for exponent and final stage and carry save adder for significand calculation is also a better choice. No doubt this multiplier did not gave significant improvement, but we expect the significant change in double precision and quadruple precision floating point multipliers.

5.4 Future scope

The present work on floating point multiplier can be extended in various directions. Some of the suggestions are given below:

- Hybrid adders can be used to increase the speed of floating point multiplier.
- Dadda compressor tree structure can be used for adding the partial products.
- Different types of adders can be implemented in double and quadruple precision floating point multipliers also.

REFERENCES

[1] SR. Kuang, JP. Wang and HY. Hong, "Variable-Latency Floating-Point Multipliers for Low Power Applications," in *IEEE Transactions on Very Large Scale Integration Systems*, vol.18, no.10, Oct.2010.

[2] Y. He and C. Chang, "A New Binary Booth Encoding for Fast 2^n-Bit Multiplier Design," in *IEEE Transactions on Circuits and Systems-I*, vol.56, no.6, Jun.2009.

[3] C. Nagenndra, M.J. Irwin and R.M. Owens, "Area-Time-Power Tradeoffs in Parallel Adders," in *IEEE Transactions on Circuits and Systems-II*, vol.43, no.10, Oct.1996.

[4] J. Kaur, N.K. Gaahlan and P.shukla, "Delay-Power performance comparison of array multiplier in VLSI design," in *International Journal of Advanced Research in Computer Science and Electronics Engineering,* vol.1, no.3, May.2012.

[5] M. Al-Ashrafy, M. Salem and W. Anis, "An efficient implementation of floating point multiplier," in Proc. *Electronics,Communications and Photonics Conference(SIECPC)*, pp.1-5, 24-26 Apr.2011.

[6] R.P.P. Singh, P. Kumar and B. Singh, "Performance Analysis Of Fast Adders Using VHDL," in Proc. *International conference on Advances in Recent Technologies in Communication and Computing*, pp.189-193,27-28 Oct.2009.

[7] P. Gurjar, R. Solanki, P. Kansliwal and M. Vucha," VLSI implementation of adders for high speed ALU," in Proc. *India conference(INDICON),* pp.1-6,16-18 Dec.2011.

[8] A. Jain, B. Dash and A. Panda, "FPGA Design of a Fast 32-bit Floating Point Multiplier Unit," in Proc. *International Conference on Devices, Circuits and systems(ICDCS)*, pp.545-547,15-16 Mar.2012.

[9] G. Renxi,Z. Shangjun, Z. Hainman and M. xiaobi, "Hardware Implementation of a High Speed Floating Point Multiplier Based on FPGA," in Proc. *International Conference on Computer Science & Education(ICCSE),* pp.1902-1906,25-28 Jul.2009.

[10] M.A. Franklin and T. Pan, "Performance Comparison of Asynchronous Adders," in Proc. *International Symposium on* Advanced *Research in Asynchronous Circuits and Systems*, pp.117-125,3-5 Nov.1994.

[11] M.K. jaiswal and R.C.C Cheung, "Area-Efficient Architectures for Large Integer and Quadruple Precision Floating Point Multipliers," in Proc. *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp.25-28, Apr.29-May 1.2012.

[12] L.S.A. Hamid, K.A. Shehata and H. El-Ghitani, M. ElSaid, "Design of Generic Floating Point Multiplier and Adder/Subtractor Units" in Proc. *International Conference on Computer Modelling and Simulation*, pp.615-618,24-26 Mar.2010.

[13] A. Kanhe, S.K. Das and A.K. Singh, "Design and Implementation of Floating Point Multiplier based on Vedic Multiplication Technique," in Proc. *International Conference on* Communication, Information & Technology(ICCICT), pp. 1-4, 19-20 Oct.2012.

[14] K.L.S Swee and L.H Hiung, " Performance comparison review of Radix-based multiplier designs," in *4th International Conference on Intelligent and Advanced Systems(ICIAS)* , pp. 854-859, 12-14 Jun.2012.

[15] A. Gupta, S. Mandavalli and V.J. Mooney, "Low Probabilistic Floating Point Multiplier Design," in Proc. *IEEE Computer Society Annual S ymposium on VLSI(ISVLSI)*, pp.182-187, 4-6 Jul.2011.

[16] L. Louca, T.A. Cook and W. H. Johnson, "Implementation of IEEE Single Precision Floating Point Addition and Multiplication on FPGAs," in Proc. *IEEE Symposium on FPGAs for Custom Computing Machines,* pp.107-116, 17-19 Apr.1996.

[17] P. Karlstrom, A. Ehliar and D. Liu, "High Performance, Low Latency FPGA based Floating Point Adder and Multiplier Units in a Virtex 4," in Proc. *Norchip Conference,* pp.31-34, Nov.2006.

[18] Y. Kumar and R.K. Sharma, "Clock-less Design for Reconfigurable Floating Point Multiplier," in Proc. *International Conference on Computational Intelligence, Modeling and Simulation(CIMSiM),* pp. 222-226, 20-22 Sept.2011.

[19] SR. Kuang and J.P. Wang, “Design of Power-efficient Pipelined Truncated Multipliers with various Output Precisions,” in *IET Computer Digital Technology*, vol.1, no.2, Mar.2007.