DESIGN AND IMPLEMENTATION OF SINGLE PRECISION FLOATING POINT MULTIPLIER USING DIVIDE & CONQUER ALGORITHM

A dissertation submitted in partial fulfillment of the requirements

for the award of degree of

MASTER OF TECHNOLOGY

In

VLSI Design

Submitted By

VISHAL SINGLA

Roll No. 601161005

Under guidance of

Ms. Sakshi

Assistant Professor, ECED

T.U, Patiala

Department of Electronics and Communication Engineering

THAPAR UNIVERSITY, PATIALA

July 2013

DECLARATION

I hereby declare that the work which is being presented in the dissertation entitled, **"Design and Implementation of Single Precision Floating Point Multiplier using Divide & Conquer Algorithm"** in partial fulfillment of the requirement for the award of 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 dissertation has not been submitted in any other University/Institute for the award of degree.

Vishal Singla

Date: 11/07/13

(VISHAL SINGLA)
Roll No: 601161005

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

Sakshi

(Ms. Sakshi)
Assistant Professor
ECED, Thapar University

Countersigned by:

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 torchbearer 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 dissertation. 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 dissertation. I wish to express my deep gratitude towards her for providing individual guidance and support throughout the dissertation work.

I convey my sincere thanks to **Head of the Department, Dr. Rajesh Khanna** as well as **PG Coordinator, Dr. Kulbir Singh, Assistant 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.

Vishal Singla

(VISHAL SINGLA)

ABSTRACT

Floating-point arithmetic is considered an esoteric subject by many people. This is rather surprising because floating-point is ubiquitous in computer systems. Almost every language has a floating-point data type; computers from PC's to supercomputers have floating-point accelerators; most compilers will be called upon to compile floating-point algorithms from time to time; and virtually every operating system must respond to floating-point exceptions such as overflow. The advantage of floating-point representation over fixed-point (and integer) representation is that it can support a much wider range of values. A system's performance is generally determined by the performance of the multiplier because the multiplier is generally the slowest element in the system. Floating point multiplication is much like integer multiplication. Because floating-point numbers are stored in sign-magnitude form, the multiplier needs only to deal with unsigned integer numbers.

In this thesis, architecture for a fast floating point multiplier compliant with the single precision IEEE 754 standard floating point multiplier has been designed. Different multipliers are designed using Verilog. Divide & Conquer technique and pipelining technique are used to design floating point multipliers. The designs are simulated on Modelsim SE 6.3f and synthesized on Xilinx ISE 8.2i targeted on FPGA on device Spartan 3E xc3s500E. This dissertation pays a significant attention to the analysis of multiplier in terms of area and delay.

TABLE OF CONTENTS

SR.NO.	CONTENTS	PAGE NO.
	Declaration……………………………………………………………..	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 Objective……………………………………………………….	2
	1.2 Dissertation Organization…………………….……………….	2
2.	**CHAPTER 2- Literature Review………………………………..**	3
3.	**CHAPTER 3- Floating point Multiplier…………………………….**	8
	3.1 Floating Point Numbers……………………………………………	8
	3.1.1 Normalization……………………………………………….	9
	3.1.2 IEEE 754 Standard For Binary Floating-Point Arithmetic…	11
	3.1.2.1 Formats……….…………………………………..	12
	a) Single Precision…………………………………..	12
	b) Double Precision………………………………….	13
	3.1.3 Exceptions…………………………………………………...	14
	3.1.3.1 Invalid Operation………………………………………	14
	3.1.3.2 Division by zero……………………………………..	15
	3.1.3.4 Underflow/Overflow…………………………………..	15

3.2 Floating Point Multiplication Algorithm………………………….. 17

4. **CHAPTER 4- Multiplier and Divide & Conquer Algorithm…….** 20

4.1 Introduction………………………………………………………… 20

4.1.1 Block Diagram of Multiplier……………………………….. 22

4.2 Generation of partial products……………………………………. 23

4.2.1 Radix-2 Booth's Algorithm ………………………………... 24

4.2.2 Modified Booth's Algorithm ………………………………. 25

a) Radix-4 Booth's Algorithm…………………………….. 25

b) Radix-8 Booth's Algorithm…………………………….. 27

4.2.3 Comparison of radix 2, radix 4 and radix 8 algorithm …….. 29

4.3 Partial Product Reduction…………………………………………. 29

4.3.1 Ripple Carry Adders (RCA)………………………………… 30

4.3.2 Carry Save Adder…………………………………………… 31

4.3.3 Carry Select Adders (CSLA)………………………………... 32

4.3.4 Carry Look Ahead Adder (CLA)……………………………. 33

4.4 Divide & Conquer Technique……………………………………... 34

4.5 Pipelining………………………………………………………….. 36

4.5.1 2-Stage Pipelinng……………………………………………. 37

4.5.2 3-Stage Pipelining…………………………………………… 37

5. **CHAPTER 5- RESULTS & CONCLUSION……………………..** 39

5.1 Array Multiplier…………………………………………………. 39

5.1.1 Synthesis results on Xilinx………………………………… 39

5.1.2 Simulation results………………………………………….. 40

5.2 Divide & Conquer Technique…………………………………… 40

5.2.1 Synthesis results on Xilinx………………………………… 40

5.2.2 Simulation results………………………………………….. 41

5.3 Divide & Conquer Technique with pipelining………………….. 42
5.3.1 Synthesis results on Xilinx………………………………… 42
5.3.2 Simulation results………………………………………….. 42
5.4 Conclusion……………………………………………………….. 43
5.5 Future Scope…………………………………………………….... 45

List of Publications…………………………………………………. 46
REFERENCES…………………………………………………... 47

LIST OF FIGURES

Figure No.	Title of Figure	Page No.
3.1	Single Precision format for floating point numbers…..	14
3.2	Bit double precision floating point format...…………	15
3.3	Floating point multiplier block diagram………….…..	20
4.1	Basic Multiplication………………………………..	22
4.2	Block Diagram of Multiplier………………………….	25
4.3	Recoded Version of the multiplier……………………	27
4.4	Example of Radix-2 Algorithm…………………..…..	27
4.5	Recoding of Radix-4………………………………..	28
4.6	Example of Radix-4 Algorithm……………………….	29
4.7	Example of Radix 8 Algorithm……………………….	30
4.8	Partial Product multiplexer……………………………	30
4.9	A 4-bit Ripple Carry Adder...…………………………	32
4.10	Carry Save adder tree……...…………………………..	33
4.11	A 16 bit carry select adder………………………...…..	34
4.12	4-BIT CLA Logic equations..………………...……….	35
4.13	Divide & Conquer Technique…………………………	36
4.14	Representation of divide & conquer technique………	37
4.15	5-stage pipeline with-respect-to instruction pipeline….	38
4.16	2-stage pipeline with-respect-to instruction pipeline….	39
4.17	3-stage pipeline with-respect-to instruction pipeline….	40
5.1	Simulation results of array multiplication…….……...	42
5.2	Simulation results of divide & conquer technique……	43
5.3	Simulation results of divide & conquer technique with pipelining……………….……………………………..	45

LIST OF TABLES

Table No.	Title of Table	Page No.
3.1	Adjustments of exponents……………………………..........	12
3.2	Normalization………………………………………………	12
3.3	Examples of Floating Point Numbers………………………...	13
3.4	Various basic formats of IEEE 754 standard…………………	14
3.5	Representation of Single Precision floating point numbers…..	16
3.6	Normalization effect on result's exponent and overflow/underflow detection………………………………...	19
4.1	Recoding in Booth Radix 2-Algorithm………………………	26
4.2	Recoding in Booth Radix 4-Algorithm………………………	28
4.3	Recoding in Booth Radix 8-Algorithm………………………	29
4.4	Categorization of adders w.r.t delay time and capacity……...	32
5.1	Synthesis Report of Array Multiplication……………………	41
5.2	Synthesis Report of multiplier using divide and conquer technique………………………………………………...	43
5.3	Synthesis Report of multiplier using divide and conquer technique with pipelining……………………………………	44
5.4	Comparison of Synthesis Report of different multipliers…….	46

ABBREVIATIONS

CLBs	Configurable logic blocks
CPA	Carry-propagate Adder
CSA	Carry save adder
DSP	Digital signal processing
FFT	Fast Fourier transform
FA	Full adder
FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
IEEE	Institute of Electrical and Electronics Engineers
ISE	Integrated Software Environment
LSD	least significant digit
LUT	Look-Up Table
MAC	Multiply and accumulate
MBA	Modified Booth's Algorithm
MBE	Modified Booth Encoding
PAR	Place and Route
RCA	Ripple-carry Adder
RISC	Reduced instruction set computing
VHDL	Very High Speed Integrated Circuits HDL
VLSI	Very Large Scale Integration

CHAPTER 1
INTRODUCTION

Multipliers are key components of many high performance systems such as FIR filters, microprocessors, digital signal processors, etc. Multiplication based operations such as multiply and accumulate (MAC) and inner product are among some of the frequently used computation-intensive arithmetic functions currently implemented in many digital signal processing (DSP) applications such as convolution, Fast Fourier Transform (FFT), filtering and in microprocessors in its arithmetic and logic unit. Since multiplication dominates the execution time of most DSP algorithms, so there is a need of high speed multiplier. A system's performance is generally determined by the performance of the multiplier because the multiplier is generally the slowest element in the system. However, area and speed are usually conflicting constraints hence improving speed results mostly in larger areas.

Latency and throughput are the two major concerns from delay perspective in multiplication algorithms. Latency is the real delay of computing a function, a measure of how long the inputs to a device are stable is the final result available on outputs. Throughput is the measure of how many multiplications can be performed in a given period of time. Floating-point arithmetic is considered an esoteric subject by many people. This is rather surprising because floating-point is ubiquitous in computer systems. Almost every language has a floating-point data type; computers from PC's to supercomputers have floating-point accelerators; most compilers will be called upon to compile floating-point algorithms from time to time; and virtually every operating system must respond to floating-point exceptions such as overflow. The advantage of floating-point representation over fixed-point (and integer) representation is that it can support a much wider range of values. The floating-point format needs slightly more storage (to encode the position of the radix point), so when stored in the same space, floating-point numbers achieve their greater range at the expense of slightly less precision. Floating-point representation, in particular the standard IEEE format, is by far the most common way of representing an approximation to real numbers in computers because it is efficiently handled in most large computer processors.

Floating point multiplication is much like integer multiplication. Because floating-point numbers are stored in sign-magnitude form, the multiplier needs only to deal with unsigned integer numbers and normalization.
In this dissertation architecture for a fast floating point multiplier compliant with the single precision IEEE 754 standard has been proposed. To attain a generic design, Verilog hardware description language was used for design entry of the entire multiplier unit as it presents a tremendous productivity improvement for circuit designers and descriptions of large circuits can be written in a relatively compact and concise form.

1.1 OBJECTIVE

The primary objective of this dissertation is to increase the multiplier speed by minimizing the overall delay. IEEE 754 floating point format and various Booth recoding algorithms are studied. Using Divide & Conquer technique multiplicands are divided into two parts and then partial products are generated using radix-4 modified booth's algorithm and they are reduced using carry save adder and ripple carry adder respectively. Finally pipelining stages are introduced further to reduce the delay.

1.2 DISSERTATION ORGANIZATION

Chapter 1: Introduction about the multipliers and the objective of this dissertation.
Chapter 2: Literature Review
Chapter 3: Discusses about the floating point numbers, its formats and various exceptions held by floating point numbers.
Chapter 4: Starts with the introduction of multipliers and its block diagram. Discusses various Booth Recoding Algorithm's for partial product generation, Carry save adders for partial product reduction and ripple carry adder for final carry propagate adder. Divide & Conquer technique is also explained with examples in this chapter. Further gives a brief introduction of pipelining.
Chapter 5: Shows the simulation and synthesis results of floating point multiplier using array multiplier. Discusses about the various algorithm's used in floating point multiplier. Shows the simulation and synthesis results of different floating point multipliers and also derives the conclusion and tells about future scope.

CHAPTER 2
LITERATURE REVIEW

Mohamed Al-Ashrafy et al. [1] described an efficient implementation of an IEEE 754 single precision floating point multiplier. The multiplier implementation handles the overflow and underflow cases. Rounding was not implemented to give more precision when using the multiplier in a Multiply and Accumulate (MAC) unit. With latency of three clock cycles the design achieves 301 MFLOPs. The multiplier was verified against Xilinx floating point multiplier core.

In order to enhance the performance of the multiplier, three pipelining stages are used to divide the critical path thus increasing the maximum operating frequency of the multiplier. The pipelining stages are imbedded at the following locations:

1. In the middle of the significand multiplier, and in the middle of the exponent adder (before the bias subtraction).
2. After the significand multiplier, and after the exponent adder.
3. At the floating point multiplier outputs (sign, exponent and mantissa bits).

Three pipelining stages mean that there is latency in the output by three clocks. 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.

Metin Mete Ozbilen et al. [2] presented a new floating-point multiplier which can perform a double-precision floating-point multiplication or two simultaneous single precision floating-point multiplications. Since in single precision floating-point multiplication two results were generated in parallel so, the multiplier's performance is almost doubled as compared to a conventional floating point multiplier.

One of the important aspects of the presented design method is that it can be applicable to all kinds of floating-point multipliers. The presented design is compared with a standard floating point multiplier via synthesis. The synthesis results showed that proposed design is 10% larger than conventional multiplier and critical path increment is only one or two gate delay.

Lamiaa S. A. Hamid et al. [3] 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 floating point multiplier unit is the multiplier block. Many algorithms have been introduced aiming to speed up the multiplier block. In this paper, they used a novel multiplication algorithm to optimize the multiplier block, where the operands to be multiplied were sliced into smaller blocks. This proposed multiplier is referred to as "Block Multiplier". The bottle neck of the adder/ subtractor module is the normalization block which was responsible for adjusting the result to the IEEE 754 normalized format after the result from addition/subtraction has been calculated.

Saroja.V Siddamal et al. [4] presented a hardware implementation of optimized IEEE 754 single precision floating-point multiplier. An improvement of area and delay was shown. They have discussed the combined methods to reduce area and increase speed of FP multiplier. The use of CSD multiplier which uses 12adders for mantissa multiplication which uses MCLA, which was 31.25% faster than CLA has resulted to a high speed multiplier. By selecting proper modules of reduced latency, area and delay, they have shown significant improvement in speed.

Zichu Qiet al. [5] proposed a four stage low-cost FMA design, which was capable of performing one double-precision or two single-precision operations in parallel. In order to support two single-precision in parallel with lower area cost, the traditional double-precision multiplier, shifter and adder, which contribute the major area of the FMA, were divided into two parts. Furthermore, a modified dual-path an algorithm was implemented to reduce the cycle time by classifying the exponent difference into three cases. Moreover, to reduce the cycle time of performing FADD instructions, the multiplier in the first stage was bypassed, and the resolution of hardware conflict was proposed to achieve a throughput of one instruction one cycle. The proposed FMA was also a lower power design, in which multiple clock gating cells were inserted to enable only one of the two paths and keep the multiplier stable in case of FADD instructions Compared with the conventional double-precision FMA, 13% delay was reduced with about 22% area increment. The increased area was mainly used to support parallel single-precision and reduce latency of FADD instructions.

Guy Even et al. [6] presented an IEEE floating-point multiplier capable of performing either single precision or Double-Precision multiplications. In single precision, the latency was two clock cycles and the throughput was one multiplication per clock cycle. In double-precision, the latency was three clock cycles and the throughput was one multiplication per two clock cycles. The multiplier saves hardware by using a half size multiplication array and by using the same rounding circuitry for both precisions.

Guillermo Marcus et al. [7] presented a Floating Point adder and a Floating Point multiplier. Both were available in single cycle and pipeline architectures and they were implemented in VHDL, were fully synthesizable with performance comparable to other available high speed implementations. The design was described as graphical schematics and VHDL code and both were freely available for general and educational use. This dual representation was very valuable as follows for easy navigation over all the components of the units that allows for a faster understanding of their interrelationships and the different aspects of a Floating Point operation. Various opportunities for extension and modifications were also presented.

Ahmet Akkas et al. [8] presented the quadruple precision multiplier in this paper take three cycles to perform quadruple precision multiplication and can produce a quadruple precision product every other cycle. It takes two cycles to perform two parallel double precision multiplications and can produce two double precision products every cycle. Compared to two double precision multipliers operating in parallel, the design presented in this paper has 5% more area and 30% more delay, but provides the ability to quickly perform quadruple precision multiplication.

Gokul Govindu et al. [9] showed that FPGA based floating-point architectures like the matrix multiplication employing efficient floating-point units can achieve a significant improvement in performance over that of processors. Moreover, performance per unit power expended, for designs on FPGAs was lower than that for designs on processors**.** We also showed the impact of the floating-point units on the overall design. And since, the floating-point units can be

resource/latency/energy dominant than the other resources like storage, control, etc., they have to be analyzed in the context of the overall architecture.
They show that FPGA based floating-point architectures like the matrix multiplication employing efficient floating-point units can achieve a significant improvement in performance over that of processors. Moreover, performance per unit power expended, for designs on FPGAs is lower than that for designs on processors. We also showed the impact of the floating-point units on the overall design. And since, the floating-point units can be resource/latency/energy dominant than the other resources like storage, control, etc., they have to be analyzed in the context of the overall architecture.

S. Cui et al. [10] presented a GAAs IEEE floating point standard single precision multiplex. A modified carry save array was used in conjunction with Booth's algorithm to reduce the partial product addition and interconnection. A special rounding technique called Trailing-1's Predictor was used to speed up the final addition and rounding. The combination of the fast arithmetic architecture and compact layout style achieves 4ns multiplication time with 3.5W power dissipation at 75°C giving 14 mW/Mhz The area was2.43mm by 3.77mm (excluding pads) and uses 28,000 transistors to give a density of 3056 transistors /mm square for 0.8-um GA As technology.

Yozo Hida et al. [11] proposed that quad-double number was an unevaluated sum of four IEEE double precision numbers, capable of representing at least 212 bits of significand. They present the algorithms for various arithmetic operations (including the four basic operations and various algebraic and transcendental operations) on quad-double numbers. The performance of the algorithms, implemented in C++, was also presented.

Loucas Louca et al. [12] presented that floating point operations were hard to implement on FPGAs because of the complexity of their algorithms. On the other hand, many scientific problems require floating point arithmetic with high levels of accuracy in their calculations. Therefore, we have explored FPGA implementations of addition and multiplication for IEEE single precision floating-point numbers. Customizations were performed where this was possible in order to save chip area, or get the most out of our prototype board. The implementations trade-

off area and speed for accuracy. The adder was a bit parallel adder, and the multiplier was a digit-serial multiplier. Prototypes have been implemented on AlteraFLEX8OOOs, and peak rates of 7MFlops for 32-bit addition and 2.3MFlops for 32-bit multiplication have been obtained.

CHAPTER 3
FLOATING POINT MULTIPLIER

A Floating point multiplier is the most common element in most digital applications such as digital filters, digital signal processors, data processors and control units. The present Floating Point Multiplier IP has three blocks sign calculator, exponent calculator, mantissa calculator, which works parallel and a normalization unit. The Multiplier is pipelined, so the first result appears after the latency period and then the result can be obtained after every clock cycle.

3.1 FLOATING POINT NUMBERS

The term floating point is derived from the fact that there is no fixed number of digits before and after the decimal point, that is, the decimal point can float. There are also representations in which the number of digits before and after the decimal point is set, called fixed-point representations. Fixed Point Representation is used in low cost products such as cellular telephones or hard disk controllers while Floating Point Numbers are used where performance is critical and cost is insignificant. For example Medical imaging system, X-rays, Ultrasound uses Floating Point format. Radar for navigation and guidance requires wide dynamic range that cannot be defined ahead of time. Almost every language has a floating-point data type; computers from PC's to supercomputers have floating-point accelerators; most compilers will be called upon to compile floating-point algorithms from time to time; and virtually every operating system must respond to floating-point exceptions such as overflow.

The advantage of floating-point representation over fixed-point (and integer) representation is that it can support a much wider range of values, shorter development cycle and higher precision. For example, a fixed point representation that has seven decimal digits, with the decimal point assumed to be positioned after the fifth digit, can represent the numbers 12345.67, 8765.43, 123.00, and so on, whereas a floating-point representation (such as the IEEE 754 decimal32 format) with seven decimal digits could in addition represent 1.234567, 123456.7, 0.00001234567, 1234567000000000, and so on. The floating-point format needs slightly more storage (to encode the position of the radix point), so when stored in the same space, floating-point numbers achieve their greater range. Over the years, several different floating-point

representations have been used in computers; however, for the last ten years the most commonly encountered representation is that defined by the IEEE Standard for Floating-Point Arithmetic (IEEE 754). It is a technical standard established by the Institute of Electrical and Electronics Engineers (IEEE) and the most widely used standard for floating-point computation, followed by many hardware and software implementations. Single precision representation occupies 32 bits: a sign bit, 8 bits for exponent and 23 for the mantissa. Floating-point representation, in particular the standard IEEE format, is by far the most common way of representing an approximation to real numbers in computers because it is efficiently handled in most large computer processors.

3.1.1 Normalization

Sign

The sign of a binary floating-point number is represented by a single bit.

0 -----Positive Number
1 -----Negative Number

Mantissa

Using -3.154×10^5 as an example, the sign is negative, the mantissa is 3.154, and the exponent is 5. The fractional portion of the mantissa is the sum of each digit multiplied by a power of 10:

.154 = 1/10 + 5/100 + 4/1000

A binary floating-point number is similar. For example, in the number $+11.1011 \times 2^3$**,** the sign is positive, the mantissa is 11.1011, and the exponent is 3. The fractional portion of the mantissa is the sum of successive powers of 2. In our example, it is expressed as:

.1011 = 1/2 + 0/4 + 1/8 + 1/16

Exponent

IEEE Short Real exponents are stored as 8-bit unsigned integers with a bias of 127. Let's use the number 1.101×2^5 as an example. The exponent (5) is added to 127 and the sum (132) is binary 10100010. Here are some examples of exponents, first shown in decimal, then adjusted, and finally in unsigned binary.

The binary exponent is unsigned, and therefore cannot be negative. The largest possible exponent is 128 when added to 127, it produces 255, the largest unsigned value represented by 8 bits. The approximate range is from 1.0×2^{-127} to $1.0 \times 2^{+128}$. Table 3.1 shows the adjustment of

exponent by adding the bias. For example, exponent of -10 is added with a bias of 127 to give the adjusted exponent 117.

Table 3.1: Adjustments of exponents

Exponent (E)	Adjusted (E + 127)	Binary
+5	132	10000100
0	127	01111111
-10	117	01110101
+128	255	11111111
-127	0	00000000

Before a floating-point binary number can be stored correctly, its mantissa must be normalized. The process is basically the same as when normalizing a floating-point decimal number. For example, decimal 1234.567 is normalized as 1.234567 x 10^3 by moving the decimal point so that only one digit appears before the decimal. The exponent expresses the number of positions the decimal point was moved left (positive exponent) or moved right (negative exponent).Similarly, the floating-point binary value 1101.101 is normalized as 1.101101 x 2^3 by moving the decimal point 3 positions to the left, and multiplying by 2^3. Here are some examples of normalizations in table 3.2:

Table 3.2: Normalization

Binary Value	Normalized As	Exponent
1101.101	1.101101	3
.00101	1.01	-3
1.0001	1.0001	0
10000011.0	1.0000011	7

In a normalized mantissa, the digit 1 always appears to the left of the decimal point. In fact, the leading 1 is omitted from the mantissa in the IEEE storage format because it is redundant.

Sign, exponent, and normalized mantissa are combined into the binary IEEE short real representation. The value 1.101 x 2^0 is stored as sign = 0 (positive), mantissa = 101, and

exponent = 01111111 (the exponent value is added to 127). The "1" to the left of the decimal point is dropped from the mantissa. Here are more examples in table 3.3:

Table 3.3: Examples of Floating Point Numbers

Binary Value	Biased Exponent	Sign, Exponent, Mantissa
-1.11	127	1 01111111 11000000000000000000000
+1101.101	130	0 10000010 10110100000000000000000
-.00101	124	1 01111100 01000000000000000000000
+100111.0	132	0 10000100 00111000000000000000000
+.0000001101011	120	0 01111000 10101100000000000000000

The advantages of normalizing floating-point numbers are:

1) The representation is unique, there is exactly one way to write a real number in such a form.

2) It's easy to compare two normalized numbers, you separately test the sign, exponent and mantissa.

3) In a normalized form, a fixed size mantissa will use all the 'digit cells' to store significant digits.

4) The IEEE and DEC normalization conditions makes the representation always start with a 1-bit, this bit can be omitted, and its place used for data. The omitted bit is called the "hidden bit".

3.1.2 IEEE 754 Standard for Binary Floating-Point Arithmetic

The IEEE (Institute of Electrical and Electronics Engineers) has produced a Standard to define floating-point representation and arithmetic. The standard brought out by the IEEE come to be known as IEEE 754. The IEEE 754 Standard for Floating-Point Arithmetic is the most widely-used standard for floating-point computation, and is followed by many hardware (CPU and FPU) and software implementations. Many computer languages allow or require that some or all arithmetic should be carried out using IEEE 754 formats.

The standard specifies:

- Basic and extended floating-point number formats
- Add, subtract, multiply, divide, square root, remainder, and compare operations

- Conversions between integer and floating-point formats
- Conversions between different floating-point formats
- Conversions between basic format floating-point numbers and decimal strings
- Floating-point exceptions and their handling, including non-numbers

When it comes to their precision and width in bits, the standard defines two groups: basic and extended format [3]

3.1.2.1 Formats

The standard defines five basic formats, named using their base and the number of bits used to encode them. There are three binary floating-point formats (which can be encoded using 32, 64, or 128 bits) and two decimal floating-point formats (which can be encoded using 64 or 128 bits). The first two binary formats are the 'Single Precision' and 'Double Precision' formats of IEEE 754-1985, and the third is often called 'quad'; the decimal formats are similarly often called 'double' and 'quad'. The formats are shown in table 3.4.

Table 3.4: Various basic formats of IEEE 754 standard

parameter → **format name**	***B*** **Base**	***P*** **(bits or digits)**	***Emax***
Binary32	2	23+1 bits	+127
Binary64	2	52+1 bits	+1023
Binary128	2	112+1 bits	+16383
Decimal64	10	16 digits	+384
Decimal128	10	34 digits	+6144

a) Single Precision

The most significant bit starts from the left. The three basic components are the sign, exponent, and mantissa (fraction) [1]. The storage layout for single-precision is show in figure 3.1:

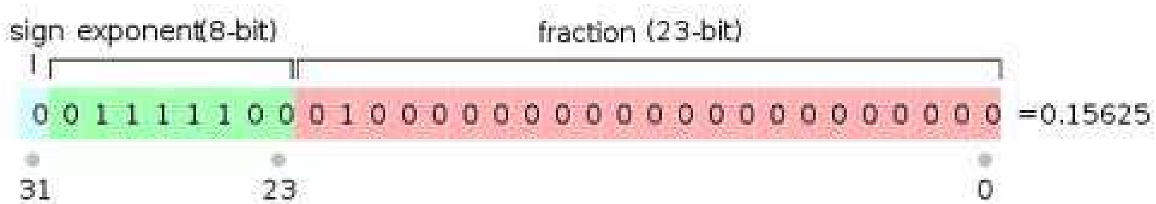

Figure 3.1: Single Precision Format for Floating Point Numbers [1]

The number represented by the single-precision format is:

Value = $(-1)^s\ 2^{E-127} * 1.M$(normalized) when E>0

Where $M = m_{22}\ 2^{-1} + m_{21}\ 2^{-2} + m_{20}\ 2^{-3} + \ldots + m_1\ 2^{-22} + m_0\ 2^{-23}$;

Bias = 127.

s= sign (0 is positive, 1 is negative)

E =biased exponent; E_{max} = 255; E_{min}= 0. E=255 and E=0 are used to represent special values.

e =unbiased exponent; e = E-127(bias)

A bias of 127 is added to the actual exponent to make negative exponents possible without using a sign bit. So for example if the value 100 is stored in the exponent placeholder, the exponent is actually -27(100 – 127). Not the whole range of E is used to represent numbers. The leading fraction bit before the decimal point is actually implicit (not given) and can be 1 or 0 depending on the exponent and therefore saving one bit.

b) Double Precision

The double precision format helps overcome the problems of single precision floating point. Using twice the space, the double precision format has an 11-bit excess-1023 exponent and a 53 bit mantissa (fraction) plus a sign bit [1]. Double precision floating point values take the form as shown in Figure 3.2.

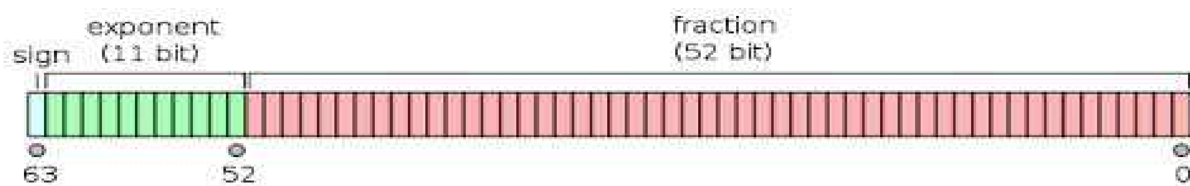

Figure 3.2: Bit Double Precision Floating Point Format [1]

In order to improve accuracy during long chains of computations involving double precision floating point numbers, Intel designed the extended precision format. The extended precision format uses 80 bits. Twelve of the additional 16 bits are appended to the mantissa; four of the additional bits are appended to the end of the exponent. Although the 32 bit ("single") and 64 bit ("double") formats are by far the most common, the standard actually allows for many different precision levels. Computer hardware (for example, the Intel Pentium series and the Motorola

68000 series) often provides an 80 bit extended precision format, with a 15 bit exponent, a 64 bit significand, and no hidden bit.

3.1.3 Exceptions

The IEEE standard defines five types of exceptions that should be signalled through a one bit status flag when encountered. Table 3.5 shows the various exceptions.

3.1.3.1 Invalid Operation

Some arithmetic operations are invalid, such as a division by zero or square root of a negative number. The result of an invalid operation shall be a NaN (Not a number). There are two types of NaN, quiet NaN (QNaN) and signaling NaN (SNaN). They have the following format, where s is the sign bit:

QNaN = s 11111111 10000000000000000000000

SNaN = s 11111111 00000000000000000000001

Table 3.5: Representation of Single Precision floating point numbers

Sign(s)	Exponent(e)	Fraction	Value
0	00000000	00000000000000000000000	+0 (positive zero)
1	00000000	00000000000000000000000	-0 (negative zero)
1	00000000	10000000000000000000000	-2^{0-127}x0.(2^{-1})= -2^{0-127} x 0.5
0	00000000	00000000000000000000001	$+2^{0-127}$x0.(2^{-23}) (smallest value)
0	00000001	01000000000000000000000	$+2^{1-127}$x1.(2^{-2})= $+2^{1-127}$ x 1.25
0	10000001	00000000000000000000000	$+2^{129-127}$ x1.0= 4
0	11111111	00000000000000000000000	+ infinity
1	11111111	00000000000000000000000	- infinity
0	11111111	10000000000000000000000	Not a Number(NaN)
1	11111111	10000100010000000001100	Not a Number(NaN)

The result of every invalid operation shall be a NaN string with a QNaN or SNaN exception. The SNaN string can never be the result of any operation, only the SNaN exception can be signalled and this happens whenever one of the input operand is a SNaN string otherwise the QNaN exception will be signalled. The SNaN exception can for example be used to signal operations with uninitialized operands, if we set the uninitialized operands to SNaN. Under default exception handling, any operation signaling an invalid operation exception and for which a floating-point result is to be delivered shall deliver a quiet NaN. Signaling NaNs shall be reserved operands that, under default exception handling, signal the invalid operation exception for every general-computational and signaling-computational operation.

The following are some arithmetic operations which are invalid operations and that give as a result a QNaN string and that signal a QNaN exception:

- Any operation on a NaN
- Addition or subtraction: $\infty + (-\infty)$
- Multiplication: $\pm 0 \times \pm \infty$
- Division: $\pm 0 / \pm 0$ or $\pm \infty / \pm \infty$
- Square root: if the operand is less than zero

3.1.3.2 Division by Zero

In mathematics, a division is called a division by zero if the divisor is zero. Such a division can be formally expressed as a/0 where a is the dividend. Whether this expression can be assigned a well-defined value depends upon the mathematical setting. In ordinary (real number) arithmetic, the expression has no meaning. In computer programming, integer division by zero may cause a program to terminate or, as in the case of floating point numbers, may result in a special not-a-number value. The division of any number by zero other than zero itself gives infinity as a result. The addition or multiplication of two numbers may also give infinity as a result. So to differentiate between the two cases, a divide-by-zero exception was implemented.

3.1.3.3 Underflow/ Overflow

Two events cause the underflow exception to be signaled, tininess and loss of accuracy. Tininess is detected after or before rounding when a result lies between ±2Emin. Loss of accuracy is detected when the result is simply inexact or only when a renormalizations loss occurs. The

implementer has the choice to choose how these events are detected. They should be the same for all operations. The implemented FPU core signals an underflow exception whenever tininess is detected after rounding and at the same time the result is inexact. The overflow exception is signaled whenever the result exceeds the maximum value that can be represented due to the restricted exponent range. It is not signaled when one of the operands is infinity, because infinity arithmetic is always exact. Division by zero also doesn't trigger this exception.

Underflow/Overflow detection:
Overflow/underflow means that the result's exponent is too large/small to be represented in the exponent field. The exponent of the result must be 8 bits in size, and must be between 1 and 254 otherwise the value is not a normalized one. An overflow may occur while adding the two exponents or during normalization. Overflow due to exponent addition may be compensated during subtraction of the bias; resulting in a normal output value (normal operation). An underflow may occur while subtracting the bias to form the intermediate exponent. If the intermediate exponent < 0 then it's an underflow that can never be compensated; if the intermediate exponent = 0 then it's an underflow that may be compensated during normalization by adding 1 to it.

When an overflow occurs an overflow flag signal goes high and the result turns to ±Infinity (sign determined according to the sign of the floating point multiplier inputs). When an underflow occurs an underflow flag signal goes high and the result turns to ±Zero (sign determined according to the sign of the floating point multiplier inputs). Denormalized numbers are signaled to Zero with the appropriate sign calculated from the inputs and an underflow flag is raised. Assume that E1 and E2 are the exponents of the two numbers A and B respectively; the result's exponent is calculated by Eresult = E1 + E2 – 127. E1 and E2 can have the values from 1 to 254; resulting in Eresult having values from -125 (2-127) to 381 (508-127); but for normalized numbers, Eresult can only have the values from 1 to 254. Table 3.6 summarizes the Eresult different values and the effect of normalization on it [3].

Table 3.6: Normalization effect on result's exponent and overflow/underflow detection

Eresult	Category	Comments
$-125 \leq$ Eresult < 0	Underflow	Can't be compensated during Normalization
Eresult = 0	Zero	May turn to normalized number during normalization (by adding 1 to it)
1 < Eresult < 254	Normalized Number	May result in overflow during Normalization
$255 \leq$ Eresult	Overflow	Can't be compensated

3.2 FLOATING POINT MULTIPLICATION ALGORITHM

Normalized floating point numbers have the form of $Z= (-1^S) * 2^{(E - Bias)} * (1.M)$. To multiply two floating point numbers the following procedure is followed:

1) Multiplication; i.e. (1.M1*1.M2) : This unit is responsible for multiplying the unsigned significand and placing the decimal point in the multiplication product. The result of significand multiplication is known as intermediate product (IP). The unsigned significand multiplication is done on 24 bit. Multiplier performance should be taken into consideration so as not to affect the whole multiplier's performance.
2) Placing the decimal point in the result.
3) Adding the exponents; i.e. (E1 + E2 – *Bias*): This unsigned adder is responsible for adding the exponent of the first input to the exponent of the second input and subtracting the Bias (127) from the addition result (i.e.A_exponent + B_exponent - Bias). An 8-bit ripple carry adder is used to add the two input exponents. A ripple carry adder is a chain of cascaded full adders and one half adder; each full adder has three inputs (A, B, Ci) and two outputs (S, Co). The carry out (Co) of each adder is fed to the next full adder (i.e each carry bit "ripples" to the next full adder). The addition process produces an 8 bit sum (S7 to S0) and a carry bit (Co,7). These bits are concatenated to form a 9 bit addition result (S8 to S0) from which the Bias is subtracted. The Bias is subtracted using an array of ripple borrow subtractors. The subtractor logic can be optimized if one of its inputs is a constant value, where the Bias is constant (127|10 = 001111111|2).

4) Obtaining the sign; i.e. s1 xor s2: Multiplying two numbers results in a negative sign number if one of the multiplied numbers is of a negative value. By the aid of a truth table we find that this can be obtained by XORing the sign of two inputs.

5) Normalizing the result; i.e. obtaining 1 at the MSB of the results' significand: The result of the significand multiplication (intermediate product) must be normalized to have a leading '1' just to the left of the decimal point (i.e. in the bit 46 in the intermediate product). Since the inputs are normalized numbers then the intermediate product has the leading one at bit 46 or 47.

   a) If the leading one is at bit 46 (i.e. to the left of the decimal point) then the intermediate product is already a normalized number and no shift is needed.

   b) If the leading one is at bit 47 then the intermediate product is shifted to the right and the exponent is incremented by 1.

6) Rounding the result to fit in the available bits

7) Checking for underflow/overflow occurrence [1]

The block diagram of floating point multiplier is shown in figure 3.3:

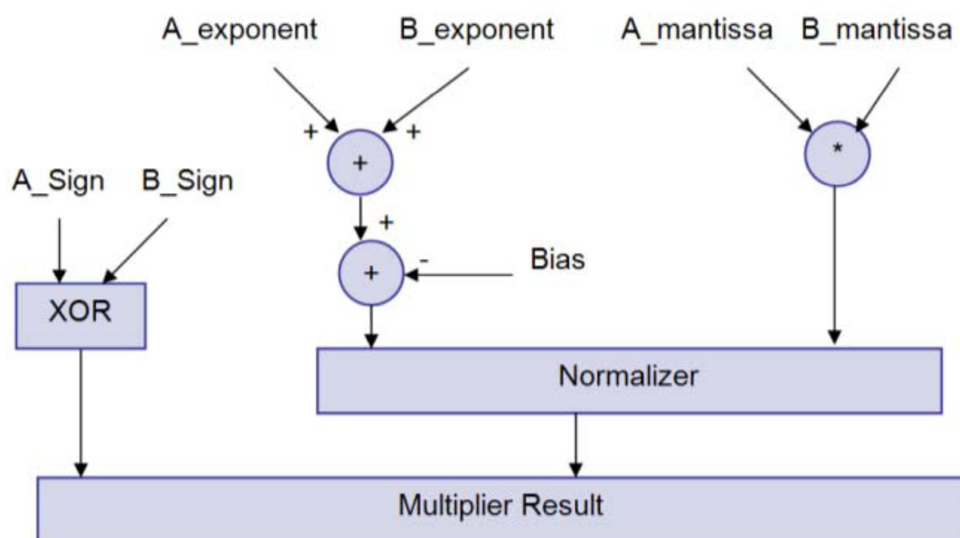

Figure 3.3: Floating point multiplier block diagram [1]

Multiplying two numbers in floating point format is done by

1) Adding the exponent of the two numbers then subtracting the bias from their result
2) Multiplying the significand of the two numbers
3) Calculating the sign by XORing the sign of the two numbers.

In order to represent the multiplication result as a normalized number there should be 1 in the MSB of the result (leading one).

CHAPTER 4

MULTIPLIER AND DIVIDE & CONQUER ALGORITHM

This chapter deals with the various algorithms of booth's recoding and the basic principles, architectures of adders that exist for binary addition.

4.1 INTRODUCTION

Multiplication is a mathematical operation that at its simplest is an abbreviated process of adding an integer a specified number of times. A number (multiplicand) is added to itself a number of times as specified by another number(multiplier) to form a result(product). In elementary schools, students learn to multiply by placing the multiplicand on top of the multiplier. The multiplicand is then multiplied by each digit of the multiplier beginning with the rightmost, least significant digit (LSD). Intermediate results (partial-products) are placed one atop the other, offset by one digit to align digits of the same weight. The final product is determined by summation of all the partial-products. Basic multiplication technique is shown in figure 4.1:

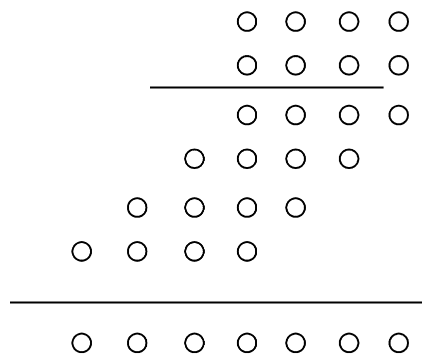

Figure 4.1: Basic Multiplication

In the binary number system the digits, called bits, are limited to the set [0,1]. The result of multiplying any binary number by a single binary bit is either 0, or the original number. This makes forming the intermediate partial-products simple and efficient. Summing these partial-products is the time consuming task for binary multipliers. The two main categories of binary multiplication include signed and unsigned numbers. Digit multiplication is a series of bit shifts and series of bit additions, where the two numbers, the multiplicand and the multiplier are

combined into the result. Considering the bit representation of the multiplicand $x= x_{n-1} \dots x_1x_0$ and the multiplier $y= y_{n-1} \dots y_1y_0$ in order to form the product up to n shifted copies of the multiplicand are to be added for unsigned multiplication. The entire process of multiplication is divided in 3 par5ts.

1) Generate the Partial Products
2) Partial Product Reduction.
3) Final stage Carry Propagate Adder

Normalized floating point numbers have the form of $Z= (-1^S) * 2^{(E - Bias)} * (1.M)$. To multiply two floating point numbers the following is done:

Consider a floating point representation similar to the IEEE 754 single precision floating point format, but with a reduced number of mantissa bits (only 4) while still retaining the hidden '1' bit for normalized numbers[2]:

A = 0 10000100 0100 = 40, B = 1 10000001 1110 = -7.5

To multiply A and B

1. Multiply significand:

```
     1.0100
   × 1.1110
      00000
     10100
    10100
   10100
  10100
 1001011000
```

2. Place the decimal point: 10.01011000

3. Add exponents:

```
  10000100
+ 10000001
 100000101
```

The exponent representing the two numbers is already shifted/biased by the bias value (127) and is not the true exponent; i.e. EA = EA-true + bias and EB = EB-true + bias And

$E_A + E_B = E_{A\text{-true}} + E_{B\text{-true}} + 2$ bias

So we should subtract the bias from the resultant exponent otherwise the bias will be added twice.

$$\begin{array}{r} 100000101 \\ \underline{-\ 01111111} \\ 10000110 \end{array}$$

4. Obtain the sign bit and put the result together:

1 10000110 10.01011000

5. Normalize the result so that there is a 1 just before the radix point (decimal point). Moving the radix point one place to the left increments the exponent by 1; Moving one place to the right decrements the exponent by 1.

1 10000110 10.01011000 (before normalizing)

1 10000111 1.001011000 (normalized)

The result is (without the hidden bit):

1 10000111 00101100

6. The mantissa bits are more than 4 bits (mantissa available bits); rounding is needed. If the truncation rounding mode is applied then the stored value is: 1 10000111 0010 [2]

4.1.1 Block Diagram of Multiplier

In order to achieve signed number multiplication Partial Products are generated. After generation of partial products they are reduced using adders. For generation of Partial Products Booth's recoding algorithm is used and for the accumulation of partial products carry save adder is used. Ripple carry adder is used to generate the final sum and carry. For partial product generation Radix 2, Radix 4 and Radix 8 Booth's recoding algorithms are studied. The Booth multiplier makes use of booth encoding algorithm in order to reduce partial products by considering certain bits at a time, thereby achieving speed advantage over other multiplier architectures [13]. Block diagram of multiplier is shown in figure 4.2:

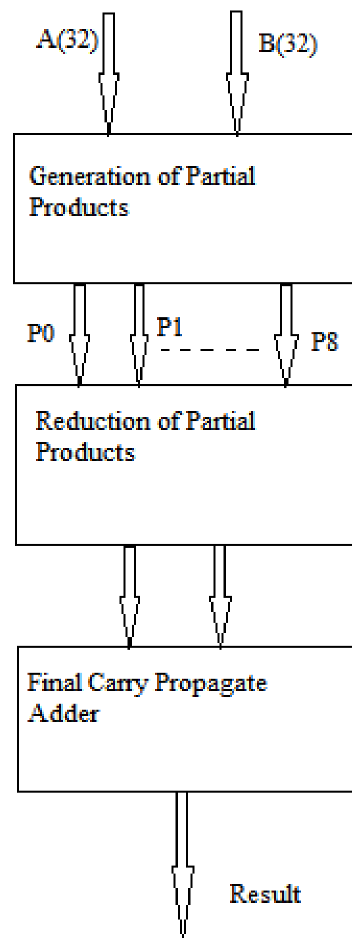

Figure 4.2: Block Diagram of Multiplier [13]

4.2 GENERATION OF PARTIAL PRODUCTS

In the digital multiplication, as in initial step, one needs to generate n shifted copies of the multiplicand, which may be added in the coming stage. The value of the multiplier bit determines whether the shifted copy is to be added or not: if the ith bit of the multiplier is '1', then the shifted copy of the multiplicand is added. If the bit is '0' it is not added. The logical AND gate can implement this operation and the resulting values are called partial products. Conventional array multipliers, like the Braun multiplier and Baugh Wooley multiplier achieve comparatively good performance but they require large area of silicon, unlike the add-shift algorithms, which require less hardware and exhibit poorer performance. Here booth algorithm is used for the generation of partial products.

Booth Multiplier

The Booth multiplier makes use of Booth encoding algorithm in order to reduce the number of partial products by considering certain number of bits of the multiplier at a time, thereby achieving a speed advantage over other multiplier architectures. This algorithm is valid for both signed and unsigned numbers. It can handle signed binary multiplication by using 2's complement representation. Various algorithms of booth recoding are discussed as under:

4.2.1 Radix-2 Booth's Algorithm

In order to start the algorithm, an imaginary 0 is appended to the right of the multiplier. Subsequently, the current bit x_i and the previous bit x_{i-1} of the multiplier, $x_{n-1}\ x_{n-2}\ldots x_1x_0$ are examined in order to yield ith bit, y_i of the recoded multiplier, $y_{n-1}\ y_{n-2}\ldots y_1y_0$. At this point, the previous bit x_{i-1} serves only as a reference bit. At its turn, x_{i-1} will be recoded to yield y_{i-1}, with x_{i-2} acting as the reference bit. For i=0, its corresponding reference bit x_{-1} is defined to be zero [14]. Table 4.1 presents a summary on the recoding method used by the Booth's theorem.

Table 4.1: Recoding in Booth Radix 2 Algorithm [14]

x_i	x_{i-1}	**Operation**	**Comments**	y_i
0	0	Shift only	String of zeroes	0
1	1	Shift only	String of ones	0
1	0	Subtract and shift	Beginning of a string of ones	-1
0	1	Add and shift	End of a string of ones	1

· Recoding multiplier- $\mathbf{x_{n-1}\ x_{n-2}\ldots x_1x_0}$ in SD (sign digit) code

· Recoded multiplier- $\mathbf{y_{n-1}\ y_{n-2}\ldots y_1y_0}$

· $\mathbf{x_i\ x_{i-1}}$ of multiplier examined to generate yi

· Previous bit – x_{i-1} - only reference bit

· i=0 - reference bit $\mathbf{x_{-1}=0}$

· Simple recoding **-** $\mathbf{y_i = x_{i-1}\ x_i}$

Example in figure 4.3 shows the recoded version of the multiplier and figure 4.4 shows the multiplication using above technique.

Multiplier 0011110011(0) recoded as 0100010101 - 4 instead of 6 add/subtracts.

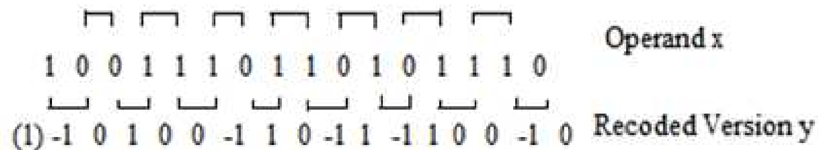

Figure 4.3: Recoded Version of the multiplier

A	1	0	1	1				-5
X	* 0	0	0	1				1
Y	0	0	1	-1				recoded multiplier
Add –A	0	1	0	1				
Shift	0	0	1	0	1			
Add A	+ 1	0	1	1				
	1	1	0	1	1			
Shift	1	1	1	0	1	1		
Shift	1	1	1	1	0	1	1	-5

Figure 4.4: Example of Radix-2 Algorithm

The multiplication procedure uses recoding of the 2's compliment multiplier with the underlying fact that a k- long sequence of 1's is equivalent to a (k-1) long sequence of zero's. This replacement of string of 1's by 0's help reduce the partial products.

4.2.2 Modified Booth's Algorithm

The Modified Booth Encoding (MBE) or Modified Booth's Algorithm (MBA) was proposed by O. L. Macsorley in 1961. It is widely used to increase the speed and to reduce the area of multiplier. Radix-4 and Radix-8 modified booth's algorithm is discussed as under:

a) Radix-4 Booth's Algorithm

The booth encoding algorithm is a bit-pair encoding algorithm that generates partial products which are multiples of the multiplicand. The booth algorithm shifts and/or complements the

multiplicand (X operand) based on the bit patterns of the multiplier (Y operand). Essentially, three multiplier bits [Y (i+1), Y (i) and Y (i-1)] are encoded into nine bits that are used to select multiples of the multiplicand {-2X, -X, 0, +X, +2X}. The three multiplier bits consist of a new bit pair [Y (i+1), Y (i)] and the leftmost bit from the previously encoded bit pair [Y (i-1)] as shown in figure 4.5.

- Separately: x_{i-2} and x_{i-3} recoded into y_{i-2} and y_{i-3} - x_{i-4} serves as reference bit.
- Groups of 3 bits each overlap - rightmost being x_1 x_0 (x_{-1}), next x_3 x_2 (x_1), and so on.
- Bits x_i and x_{i-1} recoded into y_i and y_{i-1} - x_{i-2} serves as reference bit.

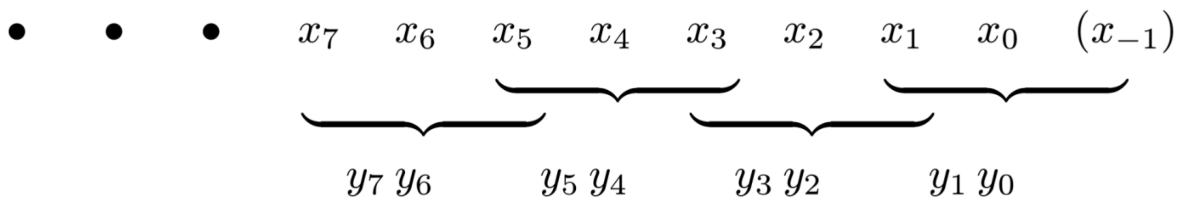

Figure 4.5: Recoding in Radix-4

The modified Booth's algorithm (radix-4 recoding) starts by appending a zero to the right of x0 (multiplier LSB). Triplets are taken beginning at position x_{-1} and continuing to the MSB with one bit overlapping between adjacent triplets. If the number of bits in X (excluding x_{-1}) is odd, the sign (MSB) is extended one position to ensure that the last triplet contains 3 bits. In every step we will get a signed digit that will multiply the multiplicand to generate a partial product entering the reduction tree [14]. Recoding in Booth's Radix-4 is shown in table 4.2.

Table 4.2: Recoding in Booth Radix-4 Algorithm [14]

x_i	x_{i-1}	x_{i-2}	y_i	y_{i-1}	Operation	Comments
0	0	0	0	0	+0	String of zeroes
0	1	0	0	1	+A	A single 1
1	0	0	-1	0	-2A	Beginning of 1's
1	1	0	0	-1	-A	Beginning of 1's
0	0	1	0	1	+A	End of 1's
0	1	1	1	0	+2A	End of 1's
1	0	1	0	-1	-A	A single 0
1	1	1	0	0	+0	String of 1's

This recoding scheme applied to a parallel multiplier halves the number of partial products so the multiplication time decreases. Figure 4.6 shows the multiplication using radix-4 booth's algorithm.

A			01	00	01			17
X			11	01	11			-9
Y			-A	+2A	-A			recoded multiplier operation
Add –A	+		10	11	11			
2 bit shift		1	11	10	11	11		
Add 2A	+	0	10	00	10			
			01	11	01	11		
2 bit shift			00	01	11	01	11	
Add -A			10	11	11			
			11	01	10	01	11	-153

Figure 4.6: Example of radix 4 Algorithm

b) Radix-8 Booth's Algorithm

Recoding extended to 3 bits at a time - overlapping groups of 4 bits each. Only n/3 partial products generated - multiple 3A needed - more complex basic step. Example: recoding 010(1) yields $y_i\ y_{i-1}\ y_{i-2}$=011. Technique for simplifying generation and accumulation of ±3A exists.

Radix-8 recoding applies the same algorithm as radix-4, but now we take quartets of bits instead of triplets. Consequently, a multiplier based on this radix-8 scheme generates fewer partial products than a radix-4 multiplier, but the computation of each partial product is more complex. In particular, a partial product corresponding to an encoding $x=\pm 3$ requires the computation of 3x, and therefore a full addition [14]. Each quartet is codified as a signed-digit using the table 4.3:

Table 4.3: Recoding in Booth Radix-8 Algorithm [14]

Quartet value	**Signed-digit value**	**Quartet value**	**Signed-digit value**
0000	0	1000	-4
0001	+1	1001	-3
0010	+1	1010	-3
0011	+2	1011	-2

0100	+2	1100	-2
0101	+3	1101	-1
0110	+3	1110	-1
0111	+4	1111	0

Example for Radix 8 is shown in figure 4.7:

A	00	01	00	01			+17
X	00	00	10	10			+10
Add A	00	10	00	10			
3 bit shift	00	00	10	00	10		
Add A	00	01	00	01			
	00	01	01	01	01	0	+170

Figure 4.7: Example of Radix 8 Algorithm

Here we have an odd multiple of the multiplicand 3Y, which is not immediately available. To generate it we need to perform this previous add: 2Y+Y=3Y. The multiplication of two binary numbers, 24-bit length, 2s-complement and using the algorithm with radix-8 recoding of the multiplier presents the following features:

- Radix-8 recoding of the multiplier implies a reduction in the number of digits to 8.
- The partial products multiplexer must choose one out of nine possibilities depending on the value of the corresponding signed-digit, as shown in figure 4.8:

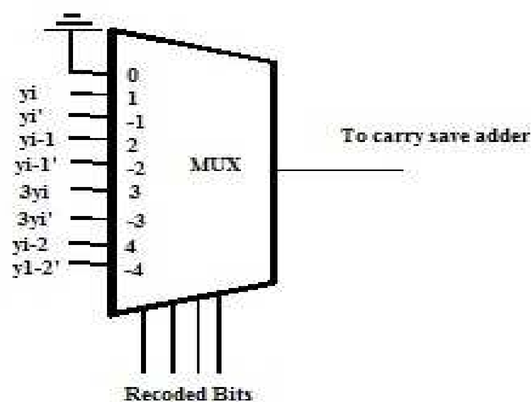

Fig 4.8: **Partial products multiplexer**

4.2.3 Comparison of radix 2, radix 4 and radix 8 algorithm

- The shortcoming of Radix 2 Booth algorithm is that it becomes inefficient when there are isolated 1's. For example, 001010101(decimal 85) gets reduced to 01-11-11-11-1(decimal 85), requiring eight instead of four operations.001010101(0) recoded as 011111111, requiring 8 instead of 4 operations. This problem can be overcome by using high radix Booth algorithms.
- As we move towards Radix 8 less number of partial products are generated but more number of operations are required to generate {+1,+2, +3, +4, -1, -2, -3, -4}. In Radix 4 we need to save {+2, -2, +1, -1}.
- Speed of Radix 8 is highest among Radix 2, 4 and 8 but the complexity increases.

4.3 PARTIAL PRODUCT REDUCTION

Addition is the most common and often used arithmetic operation on microprocessor, digital signal processor, especially digital computers. Also, it serves as a building block for synthesis all other arithmetic operations. Therefore, regarding the efficient implementation of an arithmetic unit, the binary adder structures become a very critical hardware unit. In any book on computer arithmetic, someone looks that there exists a large number of different circuit architectures with different performance characteristics and widely used in the practice. Although many researches dealing with the binary adder structures have been done, the studies based on their comparative performance analysis are only a few. In this dissertation, qualitative evaluations of the classified binary adder architectures are given. Among the huge member of the adders we wrote VHDL (Hardware Description Language) code for Ripple-carry, Carry-select and Carry-look ahead to emphasize the common performance properties belong to their classes. In the following section, we give a brief description of the studied adder architectures.

With respect to asymptotic delay time and area complexity, the binary adder architectures can be categorized into four primary classes as given in Table 4.4. The given results in the table are the highest exponent term of the exact formulas, very complex for the high bit lengths of the operands.

The first class consists of the very slow ripple-carry adder with the smallest area.

In the second class, the carry-skip, carry-select adders with multiple levels have small area requirements and shortened computation times. From the third class, the carry-look ahead adder and from the fourth class, the parallel prefix adder represents the fastest addition schemes with the largest area complexities.

Table 4.4: Categorization of adders w.r.t delay time and capacity

Complex(A)	**Delay(T)**	**Product (A x T)**	**Adder Class Schemes**
O(n)	O(n)	O(n2)	Ripple Carry
O(n)	O(n)		Carry select Carry-skip Carry-Inc
O(n)	O(logn)	O(n logn)	Carry look ahead

4.3.1 Ripple Carry Adders (RCA)

The well-known adder architecture, ripple carry adder is composed of cascaded full adders for n-bit adder, as shown in figure 4.9. It is constructed by cascading full adder blocks in series. The carry out of one stage is fed directly to the carry-in of the next stage. For an n-bit parallel adder it requires n full adders.

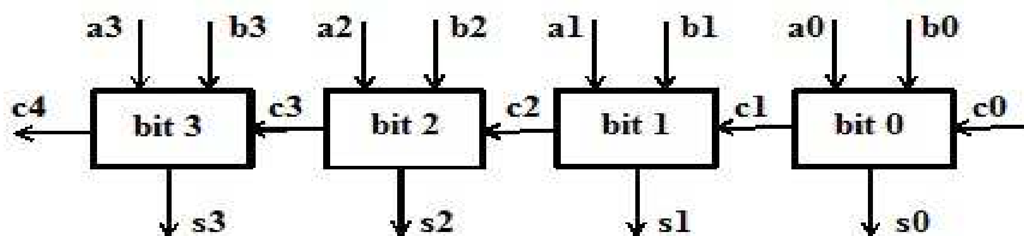

Figure 4.9 A 4-bit Ripple Carry Adder

- Not very efficient when large number bit numbers are used.
- Delay increases linearly with bit length.

4.3.2 Carry Save Adder

One of the major speed enhancement techniques used in modern digital circuits is the ability to add numbers with minimal carry propagation. The basic idea is that three numbers can be reduced to 2, in a 3:2 compressor, by doing the addition while keeping the carries and the sum separate. This means that all of the columns can be added in parallel without relying on the result of the previous column, creating a two output "adder" with a time delay that is independent of the size of its inputs.

```
         10111001
         00101010
         00111001
Sum:     10101010
Carry:  00111001
Result:     100011100
```

The sum and carry can then be recombined in a normal addition to form the correct result. This process may seem more complicated and pointless in the above trivial example, but the power of this technique is that any amount of numbers can be added together in this manner. It is only the final recombination of the final carry and sum that requires a carry propagating addition. The use of a wallace tree, arranges the adder tree so that all of the output bits could be obtained while minimising the size of the circuit. However, in the case of multipliers, we know what the expected output size will be, and so we can set all of the input and output sizes to that value. We do not care about any overflowing sign bits, so they can be discarded and the carries can simply be shifted left to the correct alignment. All of the results can then be grouped together as one and continually reduced until we are left with two values. This is demonstrated by fig.4.10

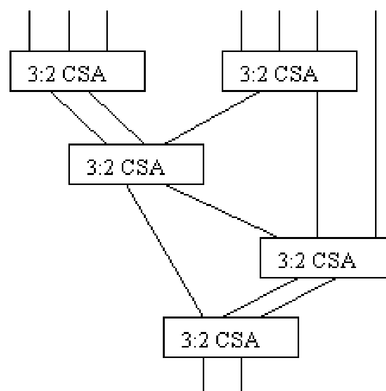

Fig 4.10: Carry-save adder tree for when overflowing carries from the MSB do not matter

This method may appear wasteful because a lot of bits in the first stages of the adder tree will be frozen to zero. However, these will be optimized during synthesis, and this technique seems to produce more favorable synthesis results than trying to code the design efficiently.

4.3.3 Carry Select Adders (CSLA)

An 8-bit **carry-select adder**, built as a cascade from a 1-bit full-adder, a 3-bit carry-select block, and a 4-bit carry-select adder. Click the input switches or type the 'a', 'b', 'c' bind keys to control the first-stage adder.

The problem of the ripple-carry adder is that each adder has to wait for the arrival of its carry-input signal before the actual addition can start. The basic idea of the carry-select adder is to use blocks of two ripple-carry adders, one of which is fed with a constant 0 carry-in while the other is fed with a constant 1 carry-in. Therefore, both blocks can calculate in parallel. When the actual carry-in signal for the block arrives, multiplexers are used to select the correct one of both pre calculated partial sums. Also, the resulting carry-out is selected and propagated to the next carry-select block.

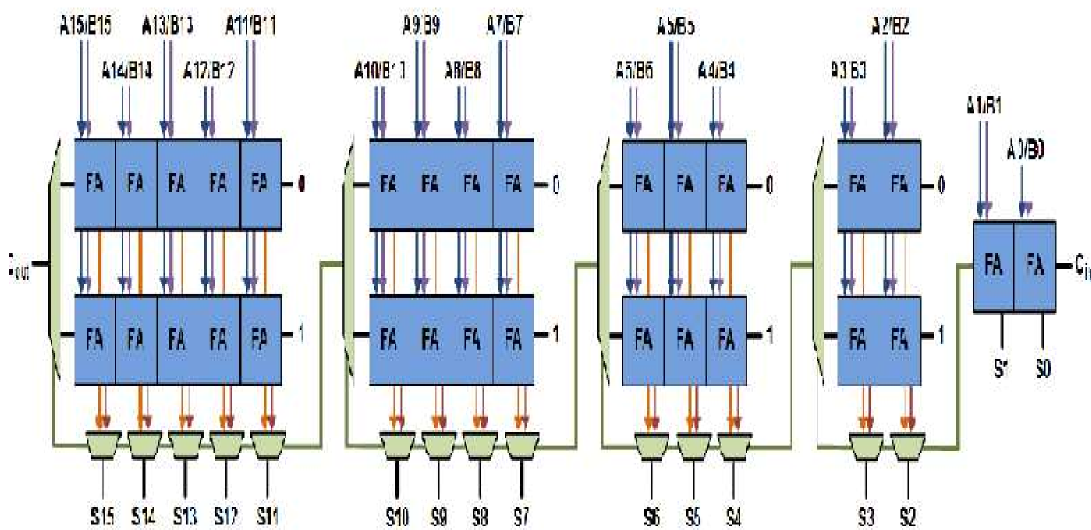

Figure 4.11: A 16 bit Carry Select Adder

In total, the carry propagation time through an n-bit adder block is reduced from O(n) to the number of stages times the delay of the multiplexers. Naturally, using n blocks of 1-bit carry-

select adders would incur a complexity of n multiplexers, again resulting in O(n) delay. Therefore, a partition with (slowly) increasing block-size is chosen. In the example, the first (least-significant) block consists of a simple full adder, followed by a 3-bit carry-select block, and finally a 4-bit carry-select block. A common choice for a 16-bit carry-select adder is to use a 6-4-3-2-1 bit partitioning. While the delay of the standard ripple-carry adder with n-bits is O(n).Example of 16-bit Carry Select adder is shown in figure 4.11:

4.3.4 Carry Look Ahead Adders (CLA)

Carry Look Ahead Adder can produce carries faster due to carry bits generated in parallel by an additional circuitry whenever inputs change. This technique uses carry by pass logic to speed up the carry propagation. Example of CLA logic is shown in figure 4.12:

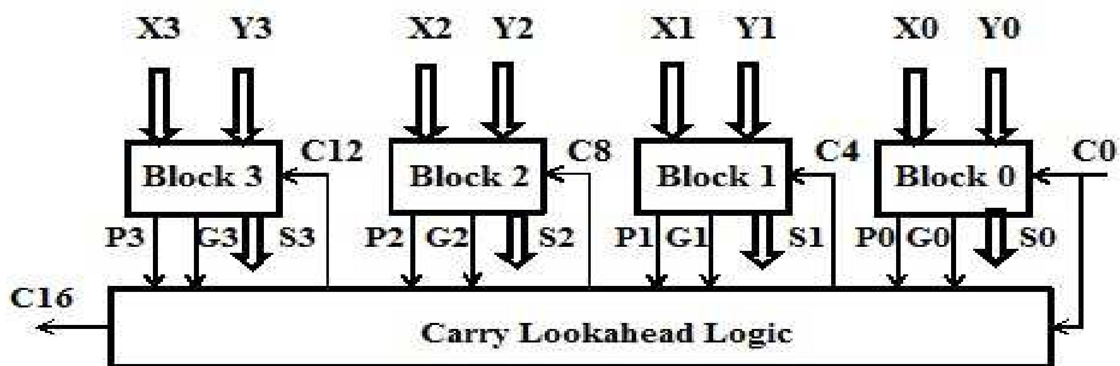

Figure 4.12: 4-BIT CLA Logic equations

Let **ai** and **bi** be the augends and addend inputs, **ci** the carry input, **si** and **ci+1**, the sum and carry-out to the **Ith** bit position. If the auxiliary functions, **pi** and **gi** called propagate and generate signals, the sum output respectively is defined as follows.

pi= ai+ bi　　　　gi= ai . bi

si= ai **xor** bi **xor** ci　　　　c**i+1**= gi + pi.ci .

- As we increase the no of bits in the Carry Look Ahead adders, the complexity increases because the no. of gates in the expression**Ci+1** increases. So practically it's not desirable to use the traditional CLA shown above because it increases the Space required and the power too.

- Instead we will use here Carry Look Ahead adder (less bits) in levels to create a larger CLA. Commonly smaller CLA may be taken as a 4-bit CLA. So we can define carry look ahead over a group of 4 bits.

4.4 DIVIDE & CONQUER TECHNIQUE

Divide and Conquer algorithms consist of two parts:

Divide: Smaller problems are solved recursively (except, of course, the base cases).

Conquer: The solution to the original problem is then formed from the solutions to the sub problems (that's the combining step).

The most well-known algorithm design strategy:

1. Divide instance of problem into two or more smaller instances
2. Solve smaller instances recursively
3. Obtain solution to original (larger) instance by combining these solutions

The tree diagram of Divide & Conquer Technique is shown in figure 4.13:

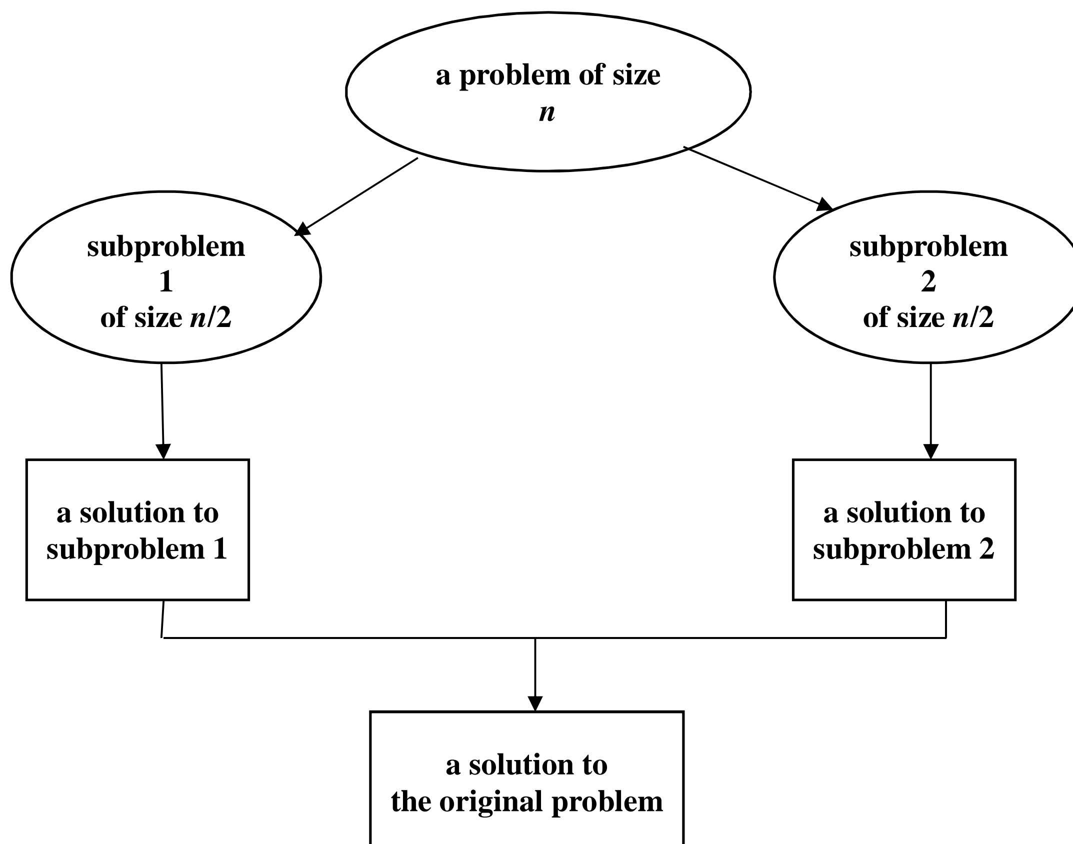

Fig 4.13: Divide & Conquer Technique

The Divide and Conquer technique is used to perform multiplications by performing a series of smaller sized multiplications and then adding these partial products. This is achieved by dividing the multiplied numbers into two parts. Hence, for multiplying two n-bit numbers A and B each of them can be expressed as $A = A_H \cdot k + A_L$ and $B = B_H \cdot k + B_L$ where A_H and B_H are the upper parts of each number, A_L and B_L are the lower parts while $k = 2^{n/2}$ **[15]**. This way the product of A and B is expressed as follows and explained in figure 4.14:

$$A \cdot B = (A_H \cdot k + A_L) \cdot (B_H \cdot k + B_L)$$

$$= A_H \cdot B_H \cdot k^2 + (A_H \cdot B_L + A_L \cdot B_H) \cdot k + A_L \cdot B_L$$

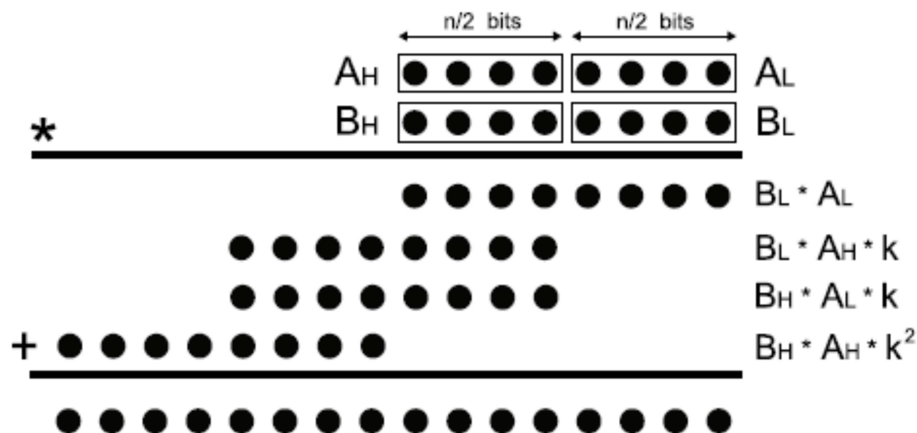

Figure 4.14: Representation of Divide-and-Conquer Technique for 8-bit words [15]

4.5 PIPELINING

A pipeline is a set of data processing elements connected in series, so that the output of one element is the input of the next one. It is divided into segments and each segment can execute its operation concurrently with the other segments. When a segment completes an operation, it passes the result to the next segment in the pipeline and fetches the next operation from the preceding segment. The final results of each instruction emerge at the end of the pipeline in rapid succession.

The pipeline technique is widely used to improve the performance of digital circuits. As the number of pipeline stages is increased, the path delays of each stage are decreased and the

overall performance of the circuit is improved [16]. Figure 4.15 shows an example with-respect-to instruction pipelining.

Inst no.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock cycle	1	2	3	4	5	6	7

Figure 4.15: 5-stage pipeline with-respect-to instruction pipeline [16]

An instruction pipeline is a technique used in the design of computers and other digital electronic devices to increase their instruction throughput (the number of instructions that can be executed in a unit of time).The fundamental idea is to split the processing of a computer instruction into a series of independent steps, with storage at the end of each step. By breaking the logic into smaller pieces and inserting flip flops between the pieces of logic, the delay is reduced. In this way the clock period can be reduced. For example, the classic RISC pipeline is broken into five stages with a set of flip flops between each stage.

1. Instruction fetch
2. Instruction decode and register fetch
3. Execute
4. Memory access
5. Register write back

Apparently a greater number of stages always provide better performance. However:

- A greater number of stages increase the overhead in moving information between stages and synchronization between stages.
- With the number of stages the complexity of the cpu grows.

On the basis of stages pipelining is divided as:

4.5.1 2-Stage Pipelining

In 2 stage pipeline as shown in figure 4.16, 8 clock cycles are required for 2 instructions. If time required for each instruction is Tex, then execution time for 7 instructions with pipelining is (Tex/2)*8= 4*Tex.

Clock	1	2	3	4	5	6	7	8
Inst i	IF	IE						
Inst i+1		IF	IE					
Inst i+2			IF	IE				
Inst i+3				IF	IE			
Inst i+4					IF	IE		
Inst i+5						IF	IE	
Inst i+6							IF	IE

Figure 4.16: 2-stage pipeline with-respect-to instruction pipeline

4.5.2 3-Stage Pipelining

In order to enhance the performance of the multiplier, three pipelining stages are used to divide the critical path thus increasing the maximum operating frequency of the multiplier. The pipelining stages are embedded at the following locations as shown in figure 4.17:

Clock	1	2	3	4	5	6	7	8
Inst i	IF	ID	IE					
Inst i+1		IF	ID	IE				
Inst i+2			IF	ID	IE			
Inst i+3				IF	ID	IE		
Inst i+4					IF	ID	IE	
Inst i+5						IF	ID	IE

Figure 4.17: 3-stage pipeline with-respect-to instruction pipeline

1. In the middle of the significand multiplier, and in the middle of the exponent adder (before the bias subtraction).
2. After the significand multiplier, and after the exponent adder.
3. At the floating point multiplier outputs (sign, exponent and mantissa bits).

Three pipelining stages mean that there is latency in the output by three clocks.

CHAPTER 5

RESULTS & CONCLUSION

This chapter introduces about the implementation and the simulation and synthesis results of floating point multiplier synthesized by Xilinx ISE 8.2i and Synopsys Design Compiler Tool.

The Xilinx ISE 8.2i is used for implementation of all the circuits. The Working Environment for the design is :

- Target Device : XC3S500E-4FG320
- Tool Version : ISE 8.2i
- Optimization Goal : Speed
- Design Strategy : Balanced
- Total Slices : 4656
- Total LUTs : 9312
- Modelsim: 6.3f

5.1 ARRAY MULTIPLIER

The multiplier designed from this technique is simple multiplication between two floating point numbers. Synthesis results and simulation results are discussed further.

5.1.1 Synthesis Results on Xilinx

Synthesis Results of floating point multiplication using array multiplier. Table 5.1 shows synthesis results of multiplication using array multiplier on Xilinx:

Table 5.1: Synthesis Report of Floating Point Multiplier using Array Multiplication

Device Specifications	Spartan 3E xc3s500E
No. of Slices	665/4656 (12.5%)
No. of LUTs	1163/9312 (12.5%)
Delay	102.944 ns

5.1.2 <u>Simulation Results</u>

Figure 5.1 shows the simulation results of multiplication using array multiplier.

a: input data 32-bit

b: input data 32-bit

final_result: Output product 32-bit

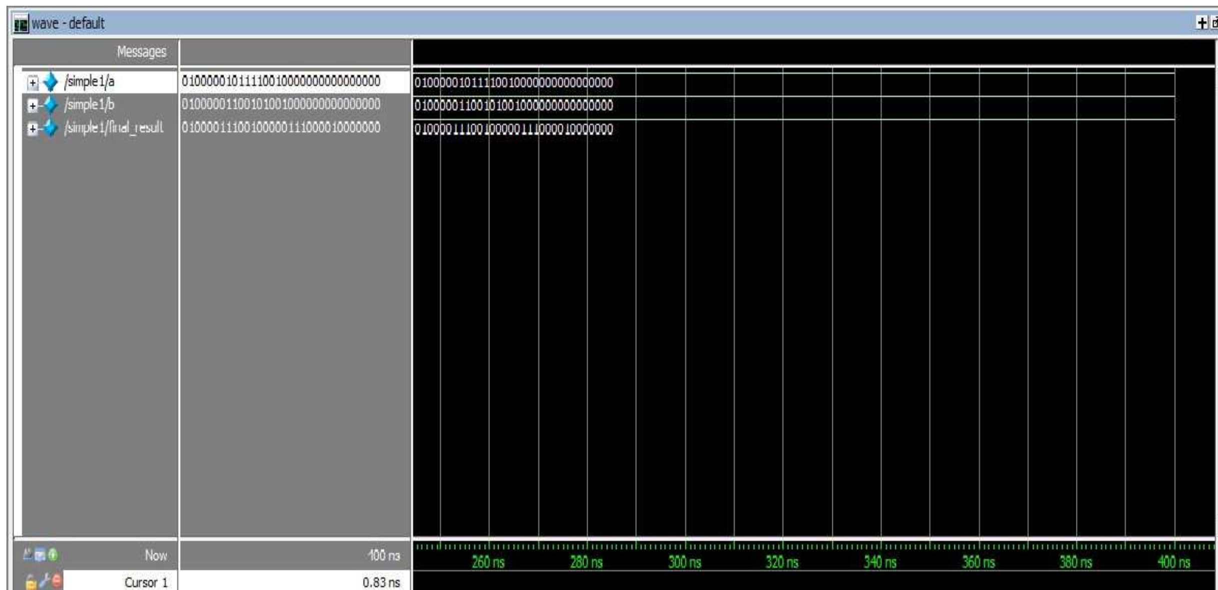

Figure 5.1: Simulation Results of Floating Point Multiplier using Array multiplication

Consider the inputs:

a= 01000001011110010000000000000000

b= 01000001011010010000000000000000

The output final_result is:

final_result= 01000011100100000111000010000000

5.2 DIVIDE & CONQUER TECHNIQUE

The multiplier is designed by using Divide & Conquer technique in which two multiplicands are divided and then multiplied to each other. Synthesis results and simulation results are shown in table 5.2 and figure 5.2 resp.:

5.2.1 <u>Synthesis Results on Xilinx</u>

The synthesis results for floating point multiplier designed by using Divide & Conquer technique is shown in table 5.2:

Table 5.2: Synthesis Report of Floating Point Multiplier using Divide & Conquer Technique

Device Specifications	Spartan 3E xc3s500E
No. of Slices	1281/4656 (25%)
No. of LUTs	2337/9312 (25%)
Delay	87.96 ns

5.2.2 Simulation Results

Figure 5.2 shows the simulation results of multiplication using Divide & Conquer technique.

a: input data 32-bit

b: input data 32-bit

final_result: Output product 32-bit

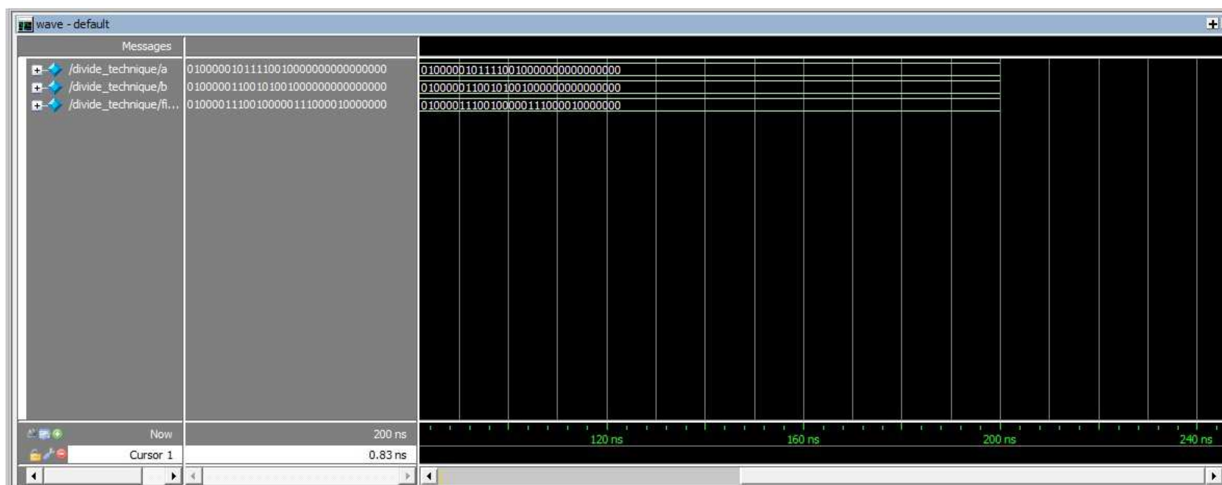

Figure 5.2: Simulation Results of Floating Point Multiplier using Divide & Conquer technique

Consider the inputs:

a= 01000001011110010000000000000000

b= 01000001011010010000000000000000

The output final_result is:

final_result= 01000011100100000111000010000000

5.3 DIVIDE & CONQUER TECHNIQUE WITH PIPELINING

The multiplier is designed by using Divide & Conquer technique also includes the pipelining in the intermediate multipliers used during floating point multiplication. Synthesis results and simulation results are shown in table 5.3 and figure 5.3 resp.:

5.3.1 Synthesis Results on Xilinx

The synthesis results for floating point multiplier designed by using Divide & Conquer technique with pipelining is shown in table 5.3:

Table 5.3: Synthesis Report of Floating Point Multiplier using Divide & Conquer Technique with Pipelining

Device Specifications	Spartan 3E xc3s500E
No. of Slices	1523/4656 (30%)
No. of LUTs	2809/9312 (30%)
Delay	8.02 ns

5.3.2 Simulation Results

Figure 5.3 shows the simulation results of multiplication using Divide & Conquer technique.

a: input data 32-bit

b: input data 32-bit

final_result: Output product 32-bit

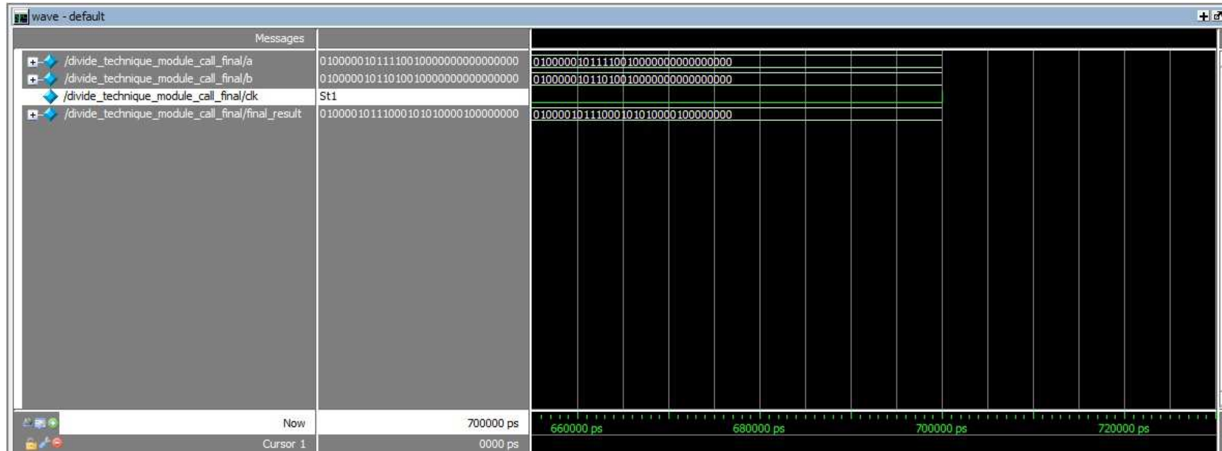

Figure 5.3: Simulation Results of Floating Point Multiplier using Divide & Conquer technique with Pipelining

Consider the inputs:

a= 01000001011111001000000000000000

b= 01000001011010010000000000000000

The output final_result is:

final_result= 01000011100100000111000010000000

This chapter concludes what have been done in dissertation and what can be done in future.

5.4 CONCLUSION

A new hardware implementation of a high speed floating point multiplier based on the IEEE-754 single precision format is developed based on Divide & Conquer technique and pipeline technique. IEEE Standard for Floating-Point Arithmetic (IEEE 754) is a technical standard established by the Institute of Electrical and Electronics Engineers (IEEE) and the most widely used standard for floating-point computation, followed by many hardware and software implementations.

First array multiplier is used for significand multiplication. For exponent addition ripple carry adder is used and the result is finally normalized and 32 bit output is calculated. The maximum combinational delay for this multiplier is 102.94 ns.

Further in next algorithm for floating point multiplier instead of using Carry Save Multipliers for significand multiplication, Booth's Recoding scheme is used to generate the partial products. Radix-4 Booth Algorithm is used for the generation of partial products. After that Divide & conquer technique is used to divide both mantissa into two parts. Since the multiplier and multiplicand comprises of 12 bits in two parts, it makes 4 multiplications each multiplication will generate 6 partial products using Radix-4 Modified Booth's Algorithm. These 6 partial products of each multiplication are reduced by using carry save adders. Further the partial products accumulated through carry save adders are further reduced by using Ripple Carry Adder**.** The result of the significand multiplication (intermediate product) is normalized to have a leading '1' just to the left of the decimal point.

In order to enhance the performance of the multiplier, pipelining is used to divide the critical path thus decreasing the maximum combinational delay of the multiplier. The operation of multiplier is divided in sequence subtasks which can be executed concurrently with other stages and flip flops are inserted between subtasks.

Synthesis results of Array multiplier, multiplier using divide & conquer technique and multiplier using divide & conquer technique with pipelining floating Point Multiplier on Xilinx Spartan 3E is shown in table 5.4. The maximum combinational delay obtained from array multiplier, multiplier using divide & conquer technique and multiplier using divide & conquer technique with pipelined structure is 102.94 ns, 87.96 ns and 8.02 ns respectively. Hence floating point multiplier achieves high throughput by insertion of pipeline stages but at the cost of area and delay.

Table 5.4: Comparison of Synthesis Report of different multipliers

	Array Multiplication	**Divide & Conquer Technique**	**Divide & Conquer Technique with pipeline**
No. of Slices	665/4656 (12.5%)	1281/4656 (25%)	1523/4656 (30%)
No. of LUTs	1163/9312 (12.5%)	2337/9312 (25%)	2809/9312 (30%)
Delay	102.94 ns	87.96	8.02 ns

5.5 FUTURE SCOPE

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

1. In order to enhance the performance higher order compressors like 7:2, 9:2 can be used to accumulate the partial products.
2. In place of ripple carry adder, other adders such as carry select adder and carry look ahead adder can be used to increase the performance.

The ability to construct high performance multipliers provide many other interesting possibilities. A double precision IEEE Floating Point Multiplier can be designed. Multiplication intensive applications, such as DSP or graphics, could benefit significantly from several high performance multipliers on the same chip. A single very high throughput multiplier, or several multipliers working in parallel on the same chip, could open up new possibilities such as single chip video signal processors.

List of Publications

- V. Singla and Sakshi "FPGA Implementation of Divide & Conquer Technique based Floating Point Multiplier" in proc. *International Conference on Electronics Communication and Information Technology (ICECIT)*, Oct. 04-05, 2013.

REFERENCES

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

[2] M.M. Ozbilen and M. Gok, "A single/double precision floating-point multiplier design for multimedia applications", in proc. *International Conference on Electrical and Electronics Engineering ,* vol. 9, no. 1, 5-8 Nov. 2009, pp. II-352 - II-356.

[3] Lamiaa S. A. Hamid, Khaled A. Shehata, Hassan El-Ghitani and Mohamed ElSaid, "Design of Generic Floating Point Multiplierand Adder/Subtractor Units", in Proc. *International Conference on Computer Modelling and Simulation*, 24-26 March, 2010, pp. 615-618.

[4] S.V Siddamal, R.M Banakar, B .C. Jinaga, "Design of High-Speed Floating Point Multiplier", in proc. 4th *IEEE International Symposium on Electronic Design, Test and Applications*, 23-25 Jan. 2008, pp. 285-289.

[5] Zichu Qi , Qi Guo, Ge Zhang, Xiangku Li, Weiwu Hu, "Design of Low-Cost High-performance Floating-point Fused Multiply-Add with Reduced Power", in proc. *23rd International Conference on VLSI Design,* 3-7 Jan. 2010, pp. 206-211.

[6] G. Even, S.M. Mueller, P.M Seidel, "A Dual Mode IEEE Multiplier", in proc. *Second Annual IEEE International Conference on Innovative Systems in Silicon*, 8-10 Oct. 1997, pp. 282-289.

[7] G. Marcus, P. Hinojosa, A. Avila, J. Nolazco-Flores, "A Fully Synthesizable Single-Precision, Floating-Point Adder/Substractor and Multiplier in VHDL for General and Educational Use", *in proc. 5th IEEE International Caracas Conference on Devices, Circuits and Systems*, Vol. 1, 3-5 Nov. 2004, pp. 319-323.

[8] A. Akkas, M.J. Schulte, "A Quadruple Precision and Dual Double Precision Floating-Point Multiplier", *in proc. Eoromicro Symposium on Digital System Design*, 1-6 Sept. 2003, pp. 76-81.

[9] G. Govindu, L. Zhuo, S. Choi, V. Prasanna, "Analysis of High-performance Floating-point Arithmetic on FPGAs", *in proc. 18th International Symposium on Parallel and Distributed Processing*, 26-30 April 2004.

[10] S. Cui. N. Burgess, M. Liebelt, K. Eshraghian, "A GA As IEEE Floating Point Standard Single Precision Multiplier", *in proc. 12th Symposium on Computer Arithmetic*, 19-21 July 1995 pp. 91-97.

[11] Y. Hida, Xiaoye S. Li, D.H. Bailey, "Algorithms for Quad-Double Precision Floating Point Arithmetic", *in proc. 15th IEEE Symposium on Computer Arithmetic*, 11-13 June 2001, pp. 155-162.

[12] L. Louca, T.A. Cook, 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,* 17-19 April, 1996, pp. 107-116.

[13] S. Kim and K. Cho, "Design of High-speed Modified Booth Multipliers Operating at GHz Ranges", *World Academy of Science, Engineering and Technology, Issue 61, pp 1-4, January 2010.*

[14] Behrooz Parhami, "Computer Arithmetic, Algorithms and Hardware Design," Oxford University Press, 2000.

[15] K. Manolopoulos, D. Reisis, V.A. Chouliaras, "An Efficient Multiple Precision Floating-Point Multiplier", *in proc. 18th IEEE International Conference on Electronics, Circuits and Systems*, 11-14 Dec. 2011, pp. 153-156.

[16] M.Morris Mano Computer System Architecture, 3rd edition.

[17] A. Jain, B. Dash, A.K. Panda, M. Suresh, "FPGA design of a fast 32-bit floating point multiplier unit", *in proc. IEEE International Conference on Devices, Circuits and Systems, Coimbatore, India, March 15-16 2012, pp 545 – 547.*

[18] IEEE 754-2008, IEEE Standard for Floating-Point Arithmetic, 2008.

[19] D. Sangwan & M.K. Yadav,”Design and Implementation of Adder/Subtractor and Multiplication Units for Floating-Point Arithmetic”, *International Journal of Electronics Engineering, 2(1), pp. 197-203, 2010.*

[20] Young-Ho Seo, and D. Kim, “A New VLSI Architecture of Parallel Multiplier–Accumulator Based on Radix-2 Modified Booth Algorithm”, *IEEE transactions on very large scale integration (vlsi) systems, vol. 18, no. 2, February 2010.*

[21] K.C. Chang, “Digital system design with VHDL and Synthesis,” An integrated Approach IEEE Computer Society, pp 408-437, 1999.

[22] G. Renxi, Z. Hainan, M. Xiaobi, G. Wenying, “Hardware Implementation of a High Speed Floating Point Multiplier Based on FPGA”, in proc. *4th International Conference on Computer Science & Education, 2009.*

[23] J. Hennessy and D. Patterson, “Computer Architecture: A Quantitative Approach,” 4th edition, Morgan Kaufmann 2007.

[24] N. Shirazi, A. Walters, and P. Athanas, “Quantitative Analysis of Floating Point Arithmetic on FPGA Based Custom Computing Machines”, *in proc. IEEE Symposium on FPGAs for Custom Computing Machines (FCCM’95), 1995, pp.155–162.*

[25] L. Louca, T. A. Cook, and W. H. Johnson, “Implementation of IEEE Single Precision Floating Point Addition and Multiplication on FPGAs,” *Proceedings of 83 the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM’96), pp. 107–116, 1996.*

[26] Neil. H. E. Weste , “Principle of CMOS VLSI Design,” Adison-Wesley, 1998.

[27] A. D. Booth, “A Signed Binary Multiplication Technique,” Quarterly J.mechan appl. Math 4, Oxford University Press, pp. 236-240, 1951.

[28] K. Khare, R P Singh and N. Khare, "Comparison of Pipelined IEEE-754 standard floating point multiplier with unpipelined multiplier," *Journal of Scientific and Industrial Research, Vol-65, November 2006, pp. 900-904.*

[29] D.Chen. J.Cong, and P. Pan, "FPGA Design Automation: A Survey," Foundations and Trends in Electronic Design Automation, 2006.

[30] Hallin, T. G. and Flynn M. J., "Pipelining of Arithmetic Functions", *IEEE Transactions of computers, 1972, pp. 880-885.*

[31] B. Lee and N. Burgess, "Parameterisable Floating-point Operations on FPGA", *Conference Record of the Thirty-Sixth Asilomar Conference on Signals, Systems, and Computers, 2002.*

[32] J. Rabaey, B. Nikolicand and A. Chandrakasan, "Digital Integrated Circuits: A Design Perspective," Prentice Hall 2003.

[33] B. Fagin and C. Renard, "Field Programmable Gate Arrays and Floating Point Arithmetic", *IEEE Transactions on VLSI, vol. 2, no. 3, 1994, pp. 365–367.*

[34] Design Compiler User Guide v1999.10.

[35] C. Asato, C.Di Tzen and S. Dholakia, "A Datapath Multiplier with Automatic Insertion of Pipeline Stages", *IEEE Custom Integrated Circuits Conference, 1989.*