DESIGN AND IMPLEMENTATION OF
A HIGH SPEED PIPELINED FLOATING POINT MULTIPLIER

A thesis submitted in partial fulfillment of the requirements

for the award of degree of

MASTER OF TECHNOLOGY

In

VLSI Design and CAD

Submitted By

SHAIFALI

Roll No. 601061021

Under guidance of

Ms. Sakshi

Assistant Professor, ECED

T.U, Patiala

Department of Electronics and Communication Engineering

THAPAR UNIVERSITY, PATIALA

July 2012

CERTIFICATE

I hereby declare that the work which is being presented in the thesis entitled, **"Design and Implementation of A High Speed Pipelined Floating Point Multiplier"** in partial fulfillment of the requirement for the award of degree of MTech in VLSI Design & CAD 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.

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

Date: 06/06/2012

(SHAIFALI)
Roll No: 601061021

It is certified 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.Rajesh Khanna)
Professor & Head
ECED, Thapar University
Patiala-147004

Dean of Academic Affairs
Thapar University
Patiala-147004

i

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 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. 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.

(SHAIFALI)

ABSTRACT

A fast and energy-efficient multiplier is always needed in electronics systems i.e. DSP processors, image processing and arithmetic units in microprocessors. Multiplier is such an important element which contributes substantially to the total power consumption of the system. On VLSI implementation level, the area also becomes quite important as more area means more system cost. Speed is another key parameter while designing a multiplier for a specific application. These three parameters i.e. power, speed and area are always traded off. For DSP processors area and speed are of major concern. But sometimes, increasing the speed also increases the power consumption, so there is an upper bound of speed for a given power budget. For portable multimedia devices, low power and fast designs of multipliers are more important than area.

Since multiplication dominates the execution time of most DSP algorithms, so there is need for high speed multiplier. In this thesis, an architecture for a fast floating point multiplier compliant with the single precision IEEE 754 standard has been proposed. Verilog is used to implement a technology-independent pipelined design. The advantage of floating-point representation over fixed-point (and integer) representation is that it can support a much wider range of values. Because floating-point numbers are stored in sign-magnitude form, the multiplier needs only to deal with unsigned integer numbers and normalization. Integer multiplier using Modified Booth's algorithm and carry save adder is one way to increase the speed of multiplier. Modified Booth's algorithm reduces the number of partial products to be generated and is known as the fastest multiplication algorithm. Many researches on the multiplier architectures including array, parallel and pipelined multipliers have been pursued which shows that pipelining is the most widely used technique to reduce the propagation delays of digital circuits.

The design is simulated on Modelsim SE and synthesized on Xilinx ISE. The Thesis pays a significant attention to the analysis of multiplier in terms of pipelining and area so as to maximize throughput. After the significand multiplication exceptions like invalid number, infinity, overflow, underflow have also been considered in this multiplier.

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 Objective…….	2
	1.2 Thesis Organization……..……...	2
2.	**CHAPTER 2- Floating point Multiplier……**	4
	2.1 Floating Point Numbers……	4
	2.1.1 Normalization…….	5
	2.1.2 IEEE 754 Standard For Binary Floating-Point Arithmetic…	7
	2.1.2.1 Formats……..……..	8
	a) Single Precision……..	8
	b) Double Precision…….	9
	2.1.3 Exceptions……...	9
	2.1.3.1 Invalid Operation……	10
	2.1.3.2 Division by zero……..	11
	2.1.3.4 Underflow/Overflow……..	11
	2.2 Floating Point Multiplication Algorithm……..	13

3. **CHAPTER 3- Multiplication…..……** 15

3.1 Introduction…….…… 15

3.1.1 Block Diagram of Multiplier…….. 17

3.2 Generation of partial products……. 18

3.2.1 Radix-2 Booth's Algorithm ……... 19

3.2.2 Modified Booth's Algorithm ……. 20

a) Radix-4 Booth's Algorithm…….. 20

b) Radix-8 Booth's Algorithm…….. 22

3.2.3 Comparison of radix 2, radix 4 and radix 8 algorithm …….. 24

3.3 Partial Product Reduction…… 24

3.4 Final Stage Carry Propagate Adder …… 27

3.5 Pipelining……. 29

4. **CHAPTER 4- Field Programmable Gate Array.....** 32

4.1 Introduction to FPGA …… 32

4.2 FPGA for Floating Point Computations …… 33

4.3 FPGA Technology Trends…….. 33

4.4 FPGA Implementation……. 34

4.4.1 Overview of FPGA Design Flow…… 34

5. **CHAPTER 5-Implementation of IEEE-754 Standard Floating Point Multiplier……..** 40

5.1 Array Multiplier…… 40

5.1.1 Floating Point Representation…… 40

5.1.2 Multiplication…….. 41

5.1.3 Normalization…….…… 42

5.1.4 Block Diagram of Floating Point Mutiplier…..... 43

5.1.5 Simulation Results…….. 44

5.2 Booth Multiplier……..……. 45

5.2.1 Generation Of Partial Products…….. 45

5.2.2 Partial Product Reduction…….. 46

5.2.3 Final Stage Carry Propagate Adder……... 47

5.2.4 Pipelining……..... 48

5.2.4.1 Unpipelined Multiplier…….……. 49

5.2.4.2 3-stage pipelining……...… 50

5.2.4.3 5-stage pipelining……... 52

5.2.5 Simulation Results…… 54

5.2.5.1 Simulation Results for 5-stage Floating Point Multiplier……..……. 54

5.2.5.2 Simulation Results for floating point Multiplier with exceptions……. 55

6. **CHAPTER 6- Conclusion……** 56

6.1 Conclusion……. 56

6.2 Future Scope…….. 58

REFERENCES……. 59

LIST OF FIGURES

Figure No.	Title of Figure	Page No.
2.1	Single Precision Format for Floating Point Number….	8
2.2	Bit Double Precision Floating Point Format…….	9
2.3	Floating point multiplier block diagram……	14
3.1	Basic Multiplication……..	15
3.2	Block Diagram of Multiplier…….	18
3.3	Recoded Version of the multiplier…….	20
3.4	Example of Radix-2 Algorithm…….	20
3.5	Recoding in Radix 4……..	21
3.6	Example of Radix-4 Algorithm…….	22
3.7	Example of Radix- 8 Algorithm……	23
3.8	Partial products multiplexer……...	24
3.9	CSA function in dot notation ……	25
3.10	Full-adder and half-adder blocks in dot notation……..	25
3.11	Block Diagram of 4:2 Compressor ……...	26
3.12	4:2 Compressor Design using Full Adders ……..	27
3.13	Ripple Carry Adder……..	28
3.14	Critical paths in a k-bit RCA……	28
3.15	Example of 5 stage pipeline……..	29
3.16	2 stage pipeline with respect to instruction pipeline…..	30
3.17	Floating point multiplier with pipelined stages…….	31
4.1	FPGA Design Flow……	35
5.1	IEEE single precision floating point format……..	40
5.2	4*4 bit Carry Save Multiplier……	42
5.3	Floating Point Mutiplier Architecture……	43

5.4 Simulation Results of Floating Point Multiplier…… 44

5.5 Floating Point Multiplier with 3 Pipeline Stages…….. 51

5.6 Floating Point Multiplier with 5-Pipeline Stage…… 53

5.7 Simulated Waveform of 5-stage Pipelined Multiplier... 54

5.8 Simulated Waveform of Multiplier with exceptions…. 55

LIST OF TABLES

Table No.	Title of Table	Page No.
2.1	Adjustments of exponents…….....	6
2.2	Normalisation……	6
2.3	Examples of Floating Point Numbers……...	7
2.4	Various basic formats of IEEE 754 standard……	8
2.5	Representation of Single Precision floating point numbers…..	10
2.6	Normalization effect on result's exponent and overflow/underflow detection……...	12
3.1	Recoding in Booth Radix 2-Algorithm…….	19
3.2	Recoding in Booth Radix 4-Algorithm…….	21
3.3	Recoding in Booth Radix 8-Algorithm…….	23
5.1	Synthesis Report of Floating Point Multiplier……..	43
5.2	DC synthesis result for multiplier…….	44
5.3	Recoding in Booth Radix-8 Algorithm…….	46
5.4	Synthesis Report of Partial Product Generation……...	46
5.5	Synthesis Report of Partial Product Accumulation……..	47
5.6	Synthesis Report of Final Stage Adder…….	48
5.7	Synthesis Report of Unpipelined Floating Point Multiplier….	49
5.8	DC synthesis result for unpipelined floating point multiplier..	49
5.9	Synthesis Report of 3-Stage Floating Point Multiplier…….	50
5.10	DC synthesis result for 3-stage pipelined multiplier……..…..	50
5.11	Synthesis Report of 5 Stage Floating Point Multiplier…….	52
5.12	DC Synthesis result for 5-stage pipelined multiplier.....…	52
6.1	Synthesis Report of on Xilinx Spartan 3E xc3s500E……...	57
6.2	Synthesis Report on Design Compiler……..	58

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. Currently, multiplication time is still the dominant factor in determining the instruction cycle time of a DSP chip. The demand for high speed processing has been increasing as a result of expanding computer and signal processing applications. Reducing the time delay and power consumption are very essential requirements for many applications. A system's performance is generally determined by the performance of the multiplier because the multiplier is generally the slowest element in the system.

In many DSP algorithms, the multiplier lies in the critical delay path and ultimately determines the performance of algorithm. The speed of multiplication operation is of great importance in DSP as well as in general processor. In the past multiplication was implemented generally with a sequence of addition, subtraction and shift operations. Multiplier is a fairly large block of a computing system. The amount of circuitry involved is directly proportional to the square of its resolution i.e a multiplier of size n bits has $O(n^2)$ gates.

Floating point is a way to represent numbers and do arithmetic in computing machines, ranging from simple calculators to computers. 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. In general, floating-point representations are slower than fixed-point representations, but they can handle a larger range of numbers[1]. Because mathematics with floating-point numbers requires a great deal of computing power,

many microprocessors come with a chip, called a floating point unit (FPU) , specialized for performing floating-point arithmetic. FPUs are also called math coprocessors and numeric coprocessors. Floating point units are widely used in a dynamic range of engineering and technology applications. This demands for the development of faster floating point arithmetic circuits.

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. Integer multiplier using Modified Booth's algorithm and carry save adder is one way to increase the speed of multiplier. Modified Booth's algorithm reduces the number of partial products to be generated and is known as the fastest multiplication algorithm. Many researches on the multiplier architectures including array, parallel and pipelined multipliers have been pursued which shows that pipelining is the most widely used technique to reduce the propagation delays of digital circuits.

In this thesis an 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 thesis is to increase the multiplier speed by minimizing the overall delay. IEEE 754 floating point format and various Booth recoding algorithms are studied. Partial products are generated using radix-8 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 THESIS ORGANIZATION

Chapter 1: Introduction about the multipliers and the objective of this thesis.

Chapter 2: Discusses about the floating point numbers, its formats and various exceptions held by floating point numbers.

Chapter 3: 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. Further gives a brief introduction of pipelining.

Chapter 4: This chapter is mainly concerned with FPGA and its design flow.

Chapter 5: Shows the simulation and synthesis results of floating point multiplier using array multiplier. Discusses about the various algorithm's and detailed pipelined architecture used in floating point multiplier. Shows the simulation and synthesis results of generation of partial products, reduction of partial products, final carry propagate adder, 3-stage pipelined floating point multiplier and 5-stage pipelined floating point multiplier.

Chapter 6: Derives the Conclusion and tells about future scope.

CHAPTER 2

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.

2.1 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.

2.1.1 Normalization

Sign

The sign of a binary floating-point number is represented by a single bit. A 1 bit indicates a negative number, and a 0 bit indicates a positive 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 2.1 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 2.1: 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 1.234567×10^3 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 1.101101×2^3 by moving the decimal point 3 positions to the left, and multiplying by 2^3. Here are some examples of normalizations in table 2.2:

Table 2.2: Normalisation

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 2.3:

Table 2.3: 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

2.1.2 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]

2.1.2.1 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 2.4.

Table 2.4: 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. The storage layout for single-precision is show in figure 2.1:

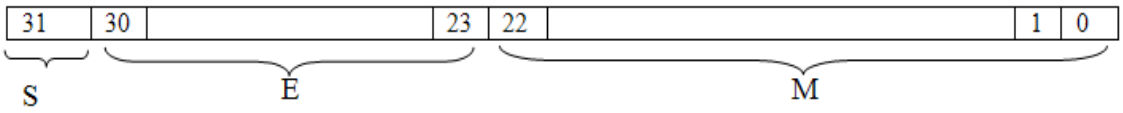

Figure 2.1: Single Precision Format for Floating Point Numbers[2]

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 plus a sign bit. Double precision floating point values take the form as shown in Figure 2.2.

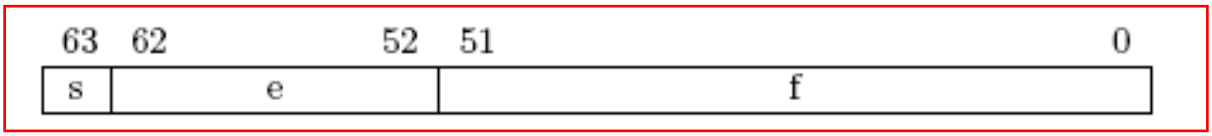

Figure 2.2: Bit Double Precision Floating Point Format

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.

2.1.3 Exceptions

The IEEE standard defines five types of exceptions that should be signalled through a one bit status flag when encountered. Table 2.5 shows the various exceptions.

2.1.3.1 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 2.5: 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

2.1.3.2 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.

2.1.3.3 Underflow/ Overflow

Two events cause the underflow exception to be signalled, 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 signalled whenever the result exceeds the maximum value that can be represented due to the restricted exponent range. It is not signalled 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 2.6 summarizes the Eresult different values and the effect of normalization on it[3].

Table 2.6: 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

2.2 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[2]

The block diagram of floating point multiplier is shown in figure 2.3:

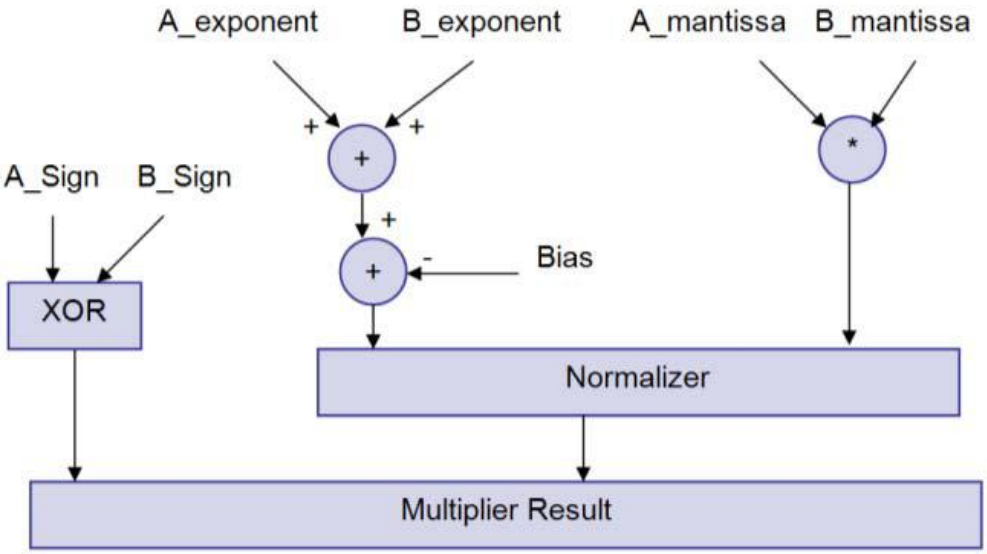

Figure 2.3: Floating point multiplier block diagram[2]

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 3

MULTIPLICATION

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

3.1 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 3.1:

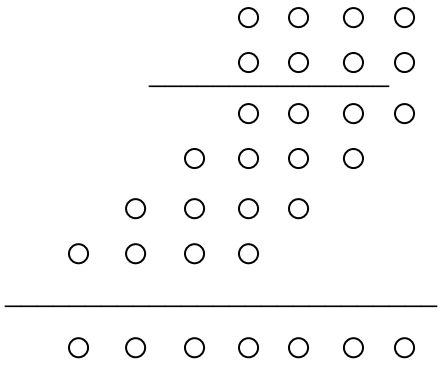

Figure 3.1: Basic Multiplication

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 $x= x_{n-1}\ldots. x_1x_0$ and the multiplier $y= y_{n-1}\ldots.y_1y_0$ 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.

100000101

<u>- 01111111</u>

10000110

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]

3.1.1 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. Block diagram of multiplier is shown in figure 3.2:

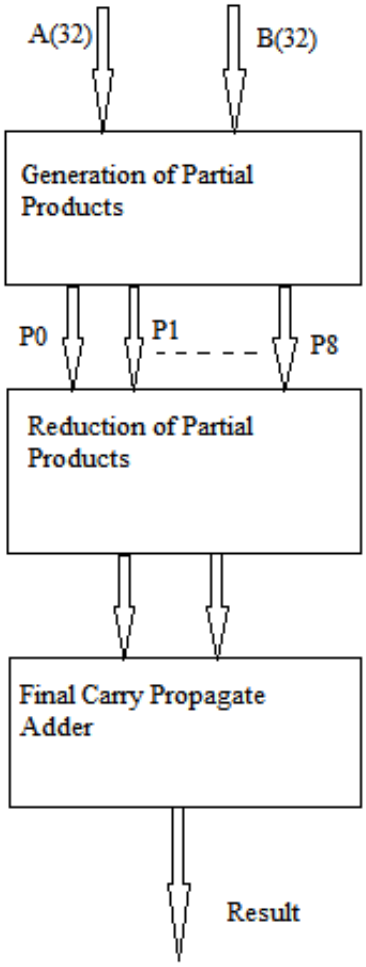

Figure 3.2: Block Diagram of Multiplier[5]

3.2 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:

3.2.1 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}... x_1x_0 are examined in order to yield ith bit, y_i of the recoded multiplier, y_{n-1} y_{n-2} ... 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. Table 3.1 presents a summary on the recoding method used by the Booth's theorem.

Table 3.1: Recoding in Booth Radix 2 Algorithm[7]

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 3.3 shows the recoded version of the multiplier and figure 3.4 shows the multiplication using above technique.

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

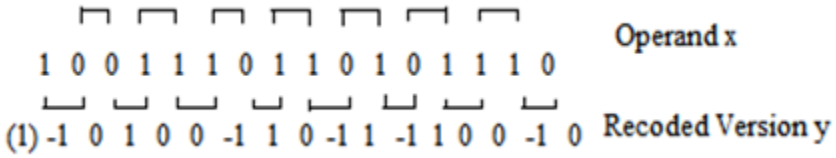

Figure 3.3: Recoded Version of the multiplier

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 3.4: 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.

3.2.2 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 3.5.

- 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.

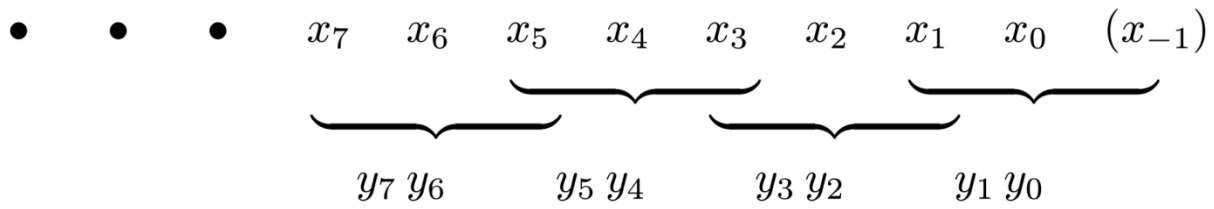

Figure 3.5: Recoding in Radix 4

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. Recoding in Booth's Radix-4 is shown in table 3.2.

Table 3.2: Recoding in Booth Radix-4 Algorithm [7]

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 decrease. Figure 3.6 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 3.6: 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=±3 requires the computation of 3x, and therefore a full addition. Each quartet is codified as a signed-digit using the table 3.3:

Table 3.3: Recoding in Booth Radix-8 Algorithm [7]

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 3.7:

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 3.7: 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 3.8:

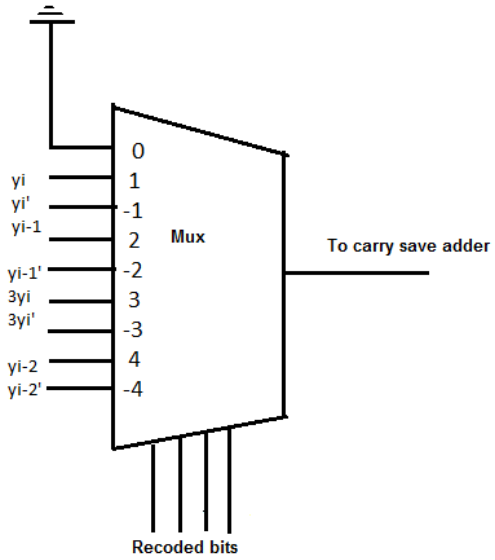

Fig 3.8: Partial products multiplexer

3.2.3 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.

3.3 PARTIAL PRODUCT REDUCTION

Efficient implementation of a digital multiplier on the method of the addition of partial product array bits. Since each shifted version of the multiplicand will give a delay proportional to the width of the multiplicand, the multiplier blocks will require a large amount of time to perform the operation if conventional adders were used to implement the addition. Hence partial products are reduced using a technique called carry save addition, which allows successive additions in one global step.

Carry save adder

The carry-save adder avoids carry propagation by treating the intermediate carries as outputs instead of advancing them to the next higher bit position, thus saving the carries for later propagation. A Carry-Save Adder is just a set of one-bit full adders, without any carry-chaining. Therefore, an n-bit CSA receives three n-bit operands, namely A(n-1)……A(0), B(n-1)……..…B(0), and Cin(n-1)……..…..Cin(0), and generates two n-bit result values, Sum(n-1)…..…..Sum(0) and Cout(n-1)…..…..Cout(0). The most important application of a carry-save adder is to calculate the partial products in integer multiplication. Figure 3.9 presents 3: 2 carry-save adder in dot notation. To specify more precisely how the various dots are related or obtained, enclose any three dots that form the inputs to a full-adder in a dashed box and to connect the sum and carry outputs of a full-adder by a diagonal line as shown in Figure 3.10. Occasionally, only two dots are combined to form a sum-bit and a carry-bit.

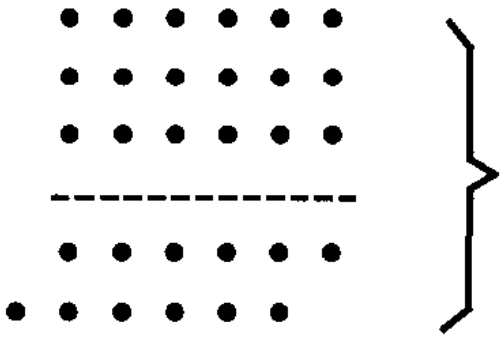

Figure 3.9: CSA function in dot notation[7]

Then the two dots are enclosed in a dashed box and the use of a half-adder is signified by a cross line on the diagonal line connecting its outputs shown in Fig 3.10.

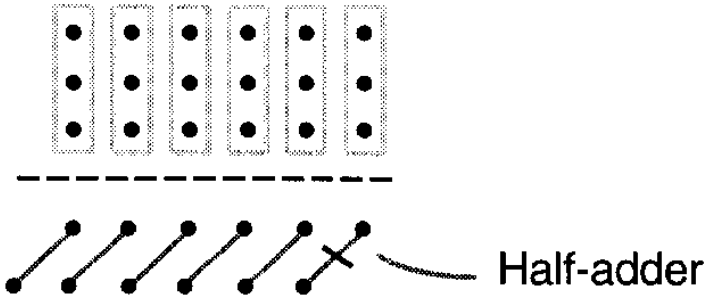

Figure 3.10: Specifying full-adder and half-adder blocks in dot notation[7]

4-2 compressors are also used as carry save adders. The 4-2 and 5-2 compressors have been widely employed in the high speed multipliers to lower the latency of the partial product accumulation stage. Owing to its regular interconnection, the 4-2 compressor is ideal for the partial products addition stage. The 4:2 compressor structure actually compresses five partial products bits into three. The architecture is connected in such a way that four of the inputs are coming from the same bit position of the weight j while one bit is fed from the neighboring position j-1(known as carry-in). The outputs of 4:2 compressor consists of one bit in the position j and two bits in the position j+1.This structure is called compressor since it compresses four partial products into two(while using one bit laterally connected between adjacent 4:2 compressors). Figure 3.11 shows the block diagram of 4-2 compressor. A 4-2 compressor can also be built using 3-2 compressors. It consists of two 3-2 compressors (full adders) in series and involves a critical path of 4 XOR delays as shown in Figure 3.12. The output Cout, being independent of the input Cin accelerates the carry save summation of the partial products.

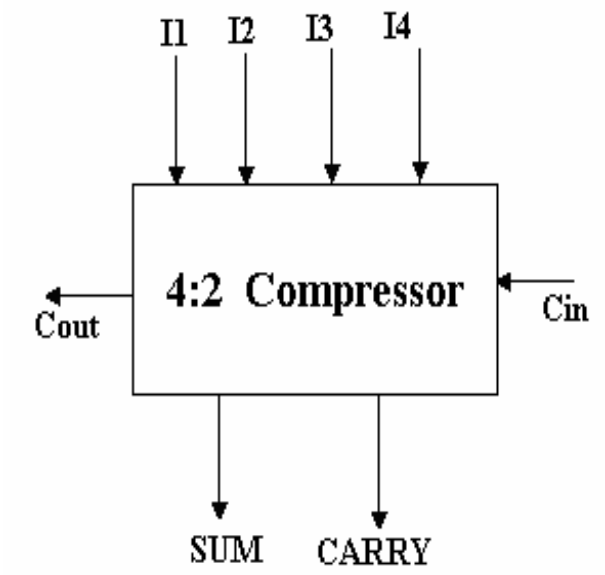

Figure 3.11: Block Diagram of 4:2 Compressor

4:2 compressor is made from 2 full adders. The final carry is saved and hence is called carry save adder. The delay of 4:2 compressor is equal that of 4 xor gates.

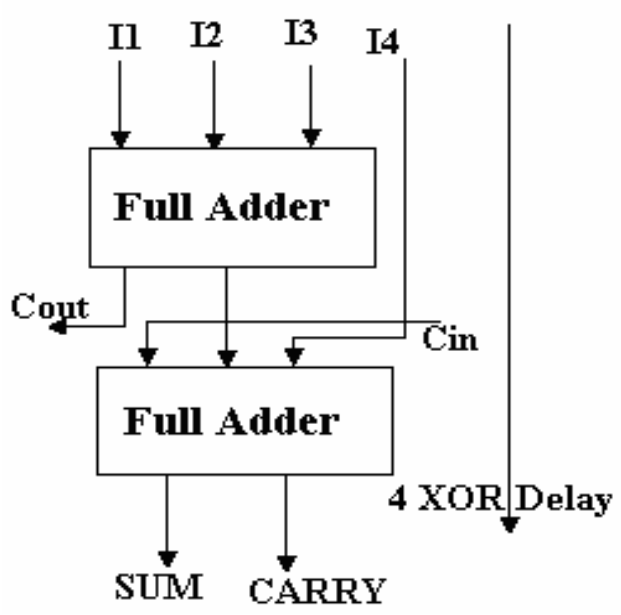

Figure 3.12: 4:2 Compressor Design using Full Adders[9]

Further the partial products accumulated through carry save adders are further reduced by using ripple carry adder. This adder is also used for adding the exponents.

3.4 FINAL STAGE CARRY PROPAGATE ADDER

The increase in the popularity of portable systems as well as the rapid growth of the power density in integrated circuits have made power dissipation one of the important design objectives, second only to performance. Because adders are one of the most widely used components in integrated circuits, designing efficient adders has been the goal of much research in VLSI design. This stage is also crucial for any multiplier because in this stage addition of large size operands is performed so in this stage fast carry propagate adders like Ripple Carry Adder can be used as per our requirement and is discussed as under.

Ripple Carry Adder

It is used to obtain the final sum and the output carry by adding the partial products from the carry save adders. It creates a logical circuit using multiple full adders to add N-bit numbers. Each full adder inputs a C_{in}, which is the C_{out} of the previous adder. This kind of adder is a ripple carry adder, since each carry bit "ripples" to the next full adder.

This unsigned adder is also 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). The result of this stage is called the intermediate exponent. The add operation is done on 8 bits, and there is no need for a quick result because most of the calculation time is spent in the significand multiplication process (multiplying 24 bits by 24 bits); thus we need a moderate exponent adder and a fast significand multiplier. An 8-bit ripple carry adder is used to add the two input exponents. As shown in Fig. 3.13 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).

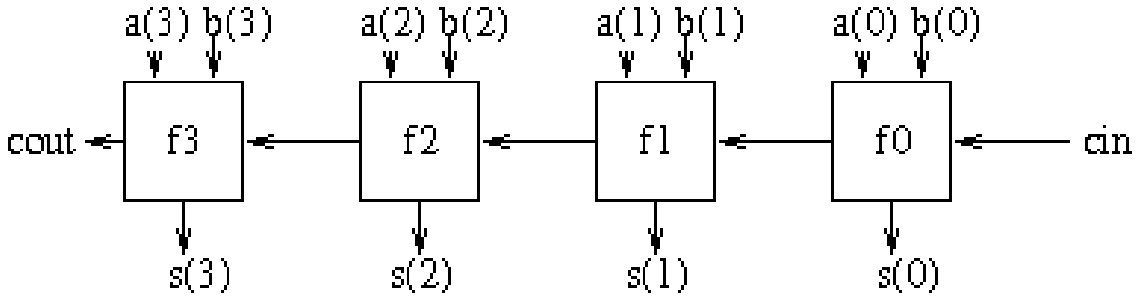

Figure 3.13: Ripple Carry Adder[2]

The latency of k-bit ripple-carry adder can be derived by considering the worst-case signal propagation path. As shown in Figure 3.14 the critical path usually begins at the x_0 or y_0 input proceeds through the carry-propagation chain to the leftmost FA and terminates at the s_{k-1} output. The critical path might begin at c_0 and/or terminate at c_k.

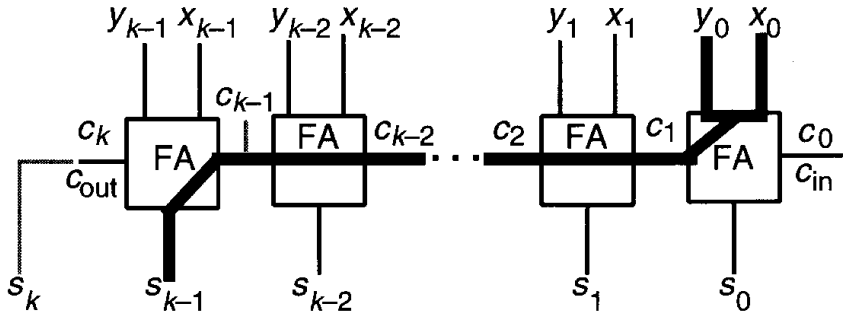

Figure 3.14: Critical paths in a k-bit RCA[7]

3.5 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. Figure 3.15 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 3.15: Example of 5 stage pipeline[10]

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:

3.5.1 2-Stage Pipelining

In 2 stage pipeline as shown in figure 3.16, 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	FI	EI						
Inst i+1		FI	EI					
Inst i+2			FI	EI				
Inst i+3				FI	EI			
Inst i+4					FI	EI		
Inst i+5						FI	EI	
Inst i+6							FI	EI

Figure 3.16: 2 stage pipeline with-respect-to instruction pipeline

3.5.2 3-Stage Pipeline

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 3.17:

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).

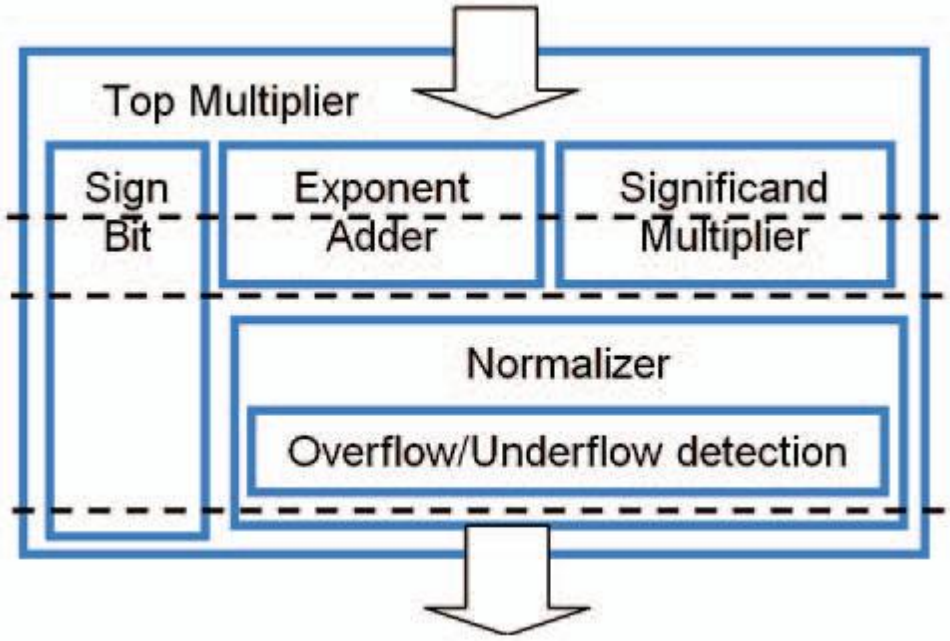

Figure 3.17: Floating point multiplier with pipelined stages[2]

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

CHAPTER 4

FIELD PROGRAMMABLE GATE ARRAY

This chapter introduces about the FPGA concepts and FPGA Synthesis Flow. An FPGA is a device that consists of thousands or even millions of transistors connected to perform logic functions. They perform functions from simple addition and subtraction to complex digital filtering and error detection and correction.

4.1 INTRODUCTION TO FPGA

A field programmable gate array (FPGA) is a semiconductor device that can be configured by the customer or the designer after manufacturing hence the name "field- programmable". Field Programmable gate arrays (FPGAs) are truly revolutionary devices that blend the benefits of both hardware and software. FPGAs are programmed using a logic circuit diagram or a source code in Hardware Description Language (HDL) to specify how the chip will work. They can be used to implement any logical function that an Application Specific Integrated Circuit (ASIC) could perform but the ability to update the functionality after shipping offers advantages for many applications. FPGAs contain programmable logic components called "logic blocks", and a hierarchy of reconfigurable interconnects that allow the blocks to be "wired together" somewhat like a one chip programmable breadboard. Logic blocks can be configured to perform complex combinational functions or merely simple logic gates like AND and XOR. In most FPGAs, the logic block also includes memory elements, which may be simple flip flops or more complete blocks of memory.

FPGAs blend the benefits of both hardware and software. They implement circuits just like hardware performing huge power, area and performance benefits over softwares, yet can be reprogrammed cheaply and easily to implement a wide range of tasks. Just like computer hardware, FPGAs implement computations spatially, simultaneously computing millions of operations in resources distributed across a silicon chip. Such systems can be hundreds of times faster than microprocessor-based designs. However unlike in ASICs, these computations are programmed into a chip, not permanently frozen by the manufacturing process. This means that an FPGA based system can be programmed and reprogrammed

many times. FPGAs are being incorporated as central processing elements in many applications such as consumer electronics, automotive, image/video processing military/aerospace, base stations, networking/communications, super computing and wireless applications.

4.2 FPGA FOR FLOATING POINT

With gate counts approaching ten million gates, FPGA's are quickly becoming suitable for major floating point computations. However, to date, few comprehensive tools that allow for floating point unit trade offs have been developed. Most commercial and academic floating point libraries provide only a small number of floating point modules with fixed parameters of bit-width, area and speed. Due to these limitations, user designs must be modified to accommodate the available units. The balance between FPGA floating point unit resources and performance is influenced by subtle context and design requirements. Generally, implementation requirements are characterized by throughput, latency and area.

1. FPGAs are often used in place of software to take advantage of inherent parallelism and specialization. For data intensive applications, data throughput is critical.
2. If floating point computation is in a dependent loop, computation latency could be an overall performance bottleneck.

4.3 FPGA TECHNOLOGY TRENDS

- General trend is bigger and faster.
- This is being achieved by increases in device density through even smaller fabrication process technology.
- New generations of FPGAs are geared towards implementing entire systems on a single device.
- Features such as RAM, dedicated arithmetic hardware, clock management and transceivers are available in addition to the main programmable logic.
- FPGAs are also available with the embedded processors (embedded in silicon or as cores within the programmable logic fabric).

4.4 FPGA IMPLEMENTATION

The FPGA that is used for the implementation of the circuit is the Xilinx Spartan 3E (Family), XC3S5000 (Device). The working environment/tool for the design is the Xilinx ISE 8.2i is used for FPGA Design flow of Verilog code.

4.4.1 Overview of FPGA Design Flow

As the FPGA architecture evolves and its complexity increases. Today, most FPGA vendors provide a fairly complete set of design tools that allows automatic synthesis and compilation from design specifications in hardware specification languages, such as Verilog or VHDL, all the way down to a bit stream to program FPGA chips. A typical FPGA design flow includes the steps and components shown in Figure 4.1. Inputs to the design flow typically include the HDL specification of the design, design constraints, and specification of target FPGA devices . We further elaborate on these components of the design input in the following: Design constraints typically include the expected operating frequencies of different clocks, the delay bounds of the signal path delays from input pads to output pads (I/O delay), from the input pads to registers (setup time), and from registers to output pads (clock-to-output delay). In some cases, delays between some specific pairs of registers may be constrained.

The second design input component is the choice of FPGA device. Each FPGA vendor typically provides a wide range of FPGA devices, with different performance, cost, and power tradeoffs. The selection of target device may be an iterative process. The designer may start with a small (low capacity) device with a nominal speed-grade. But, if synthesis effort fails to map the design into the target device, the designer has to upgrade to a high-capacity device. Similarly, if the synthesis result fails to meet the operating frequency, he has to upgrade to a device with higher speed-grade. In both the cases, the cost of the FPGA device will increase in some cases by 50% or even by 100%. This clearly underscores the need to have better synthesis tools since their quality directly impacts the performance and cost of FPGA designs [17].

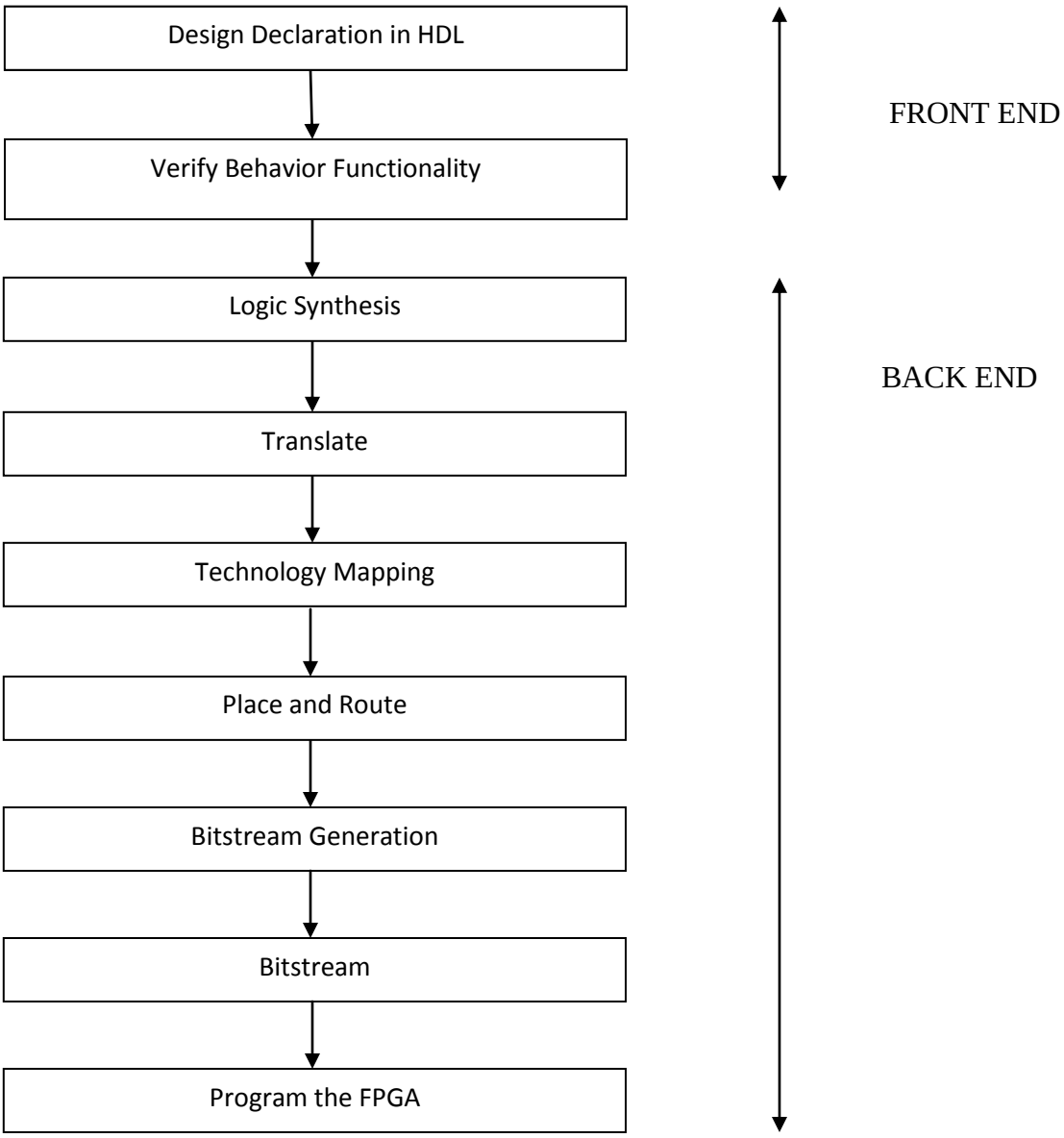

Figure 4.1: FPGA Design Flow

Design Entity

The basic architecture of the system is designed in this step which is coded in a Hardware description Language like Verilog or VHDL. A design module is split into two parts, each of which is called a design unit in Verilog. The module declaration represents the external interface to the design module. The module internals represents the internal description of the design module-its behavior, its structure, or a mixture of both.

Behavioral Simulation

After the design phase, create a test bench waveform containing input stimulus to verify the functionality of the verilog code module using a simulation software i.e. Modelsim SE for different inputs to generate outputs and if it verifies then proceed further, otherwise modifications and necessary corrections will be done in the HDL code. This is called as the behavioral simulation.

Design Synthesis

After the correct simulations results, the design is then synthesized. During synthesis, the Xilinx ISE tool does the following operations:

a) HDL Compilation: The tool compiles all the sub-modules in the main module if any and then checks the syntax of the code written for the design.

b) Design Hierarchy Analysis: Analysis the hierarchy of the design.

c) HDL Synthesis: The process which translates VHDL or Verilog code into a device netlist formate, i.e. a complete circuit with logical elements such as Multiplexer, Adder/subtractors, counters, registers, flip flops Latches, Comparators, XORs, tristate buffers, decoders, etc. for the design. If the design contains more than one sub designs, ex. to implement a processor, we need a CPU as one design element and RAM as another and so on, and then the synthesis process generates netlist for each design element. Synthesis process will check code syntax and analyze the hierarchy of the design which ensures that the design is optimized for the design architecture, the designer has selected. The resulting netlist is saved to an NGC (Native Generic Circuit) file (for Xilinx® Synthesis Technology (XST)).

d) Advanced HDL Synthesis: Low Level synthesis: The blocks synthesized in the HDL synthesis and the Advanced HDL synthesis are further defined in terms of the low level blocks such as buffers, lookup tables. It also optimizes the logic entities in the design by eliminating the redundant logic, if any. The tool then generates a 'netlist' file (NGC file) and then optimizes it. The final netIist output file has an extension of .ngc. This NGC file contains both the design data and the constraints. The optimization goal can be pre-specified to be the faster speed of operation or the minimum area of implementation before running this process. The level optimization effort can also be specified. The higher the effort, the more optimized is the design but higher effort can also be specified. The higher the effort, the more

optimized is the design but higher effort requires larger CPU time (i.e. the design time) because multiple optimization algorithms are tried to get the best result for the target architecture.

Design Implementation

The design implementation process consists of the following sub processes:

1. Translation: The Translate process combines all the input netlists and constraints to a logic design file. This information is saved an a NGD (Native Generic Database) file. This can be done using NGD Build program and .ngd file describes the logical design reduced to the Xilinx device primitive cells. Here, defining constraints is nothing but, assigning the ports in the design to the physical elements (ex. pins, switches, buttons etc) of the targeted device and specifying time requirements of the design. This information is stored in a file named UCF (User Constraints File). Tools used to create or modify the UCF are PACE, Constraint Editor Etc.

2. Mapping: The Map process is run after the Translate process is complete. Map process divides the whole circuit with logical elements into sub blocks such that they can be fit into the FPGA logic blocks. That means map process fits the logic defined by the NGD file into the targeted FPGA elements (Combinational Logic Blocks (CLB), Input Output Blocks (lOB)) and generates an NCD (Native Circuit Description) file which physically represents the design mapped to the components of FPGA. MAP program is used for this purpose.

3. Place and Route: Place and Route (PAR) program is used for this process. The place and route process places the sub blocks from the map process into logic blocks according to the constraints and connects the logic blocks. Example if a sub block is placed in a logic block which is very near to IO pin, then it may save the time but it may affect some other constraint. So tradeoff between all the constraints is taken account by the place and route process .The PAR tool takes the mapped NCD file as input and produces a completely routed NCD file as output. Output NCD file consist the routing information.

4. Bitstream Generation: The collection of binary data used to program the reconfigurable logic device is most commonly referred to as a "bitstream," although this is somewhat misleading because the data are no more bit oriented than that of an instruction set processor and there is generally no "streaming." While in an instruction set processor the configuration

data are in fact continuously streamed into the internal units, they are typically loaded into the reconfigurable logic device only once during an initial setup phase.

5. Functional Simulation: Post-Translate (functional) simulation can be performed prior to mapping of the design. This simulation process allows the user to verify that the design has been synthesized correctly and any differences due to the lower level of abstraction can be identified.

6. Static timing analysis: Three types of static timing analysis can be performed that are:

(i) Post-fit Static timing analysis: The timing results of the Post-fit process can be analyzed. The Analyze Post-Fit Static timing process opens the timing Analyzer window, which interactively select timing paths in design for tracing.

(ii) Post-Map Static Timing Analysis: Analyze the timing results of the Map process. Post Map timing reports can be very useful in evaluating timing performance. Although route delays are not accounted for the logic delays can provide valuable information about the design. If logic delays account for a significant portion (>50%) of the total allowable delay of a path, the path may not be able to meet your timing requirements when routing delays are added. Routing delays typically account for 45% to 65% of the total path delays. By identifying problem paths, redesign the logic paths to use fewer levels of logic, tag the paths for specialized routing resources, move to a faster device, or allocate more time for the path. If logic-only-delays account for much less (35%) than the total allowable delay for a path or timing constraint, then the place-and-route software can use very low placement effort levels. In these cases, reducing effort levels allow to decrease runtimes while still meeting performance requirements.

(iii) Post Place and Route Static Timing Analysis: Analyze the timing results of the Post-Place and Route process. Post-PAR timing reports incorporate all delays to provide a comprehensive timing summary. If a placed and routed design has met all of timing constraints, then proceed by creating configuration data and downloading a device. On the other hand, identify problems and the timing reports, try fixing the problems by increasing the placer effort level, using re-entrant routing, or using multi-pass place and route. Redesign the logic paths to use fewer levels of logic, tag the paths for specialized routing resources, move to a faster device, or allocate more time for the paths.

(iv) Timing Simulation: Perform Post-Place and Route simulation after the design has been

and routed. After the design has been through all of the Xilinx implementation tools, a timing simulation netlist can be created. This simulation process allows to see how the design will behave in the circuit. Before performing this simulation it will benefit to create a test bench or test fixture to apply stimulus to the design. After this a .ncd file will be created that is used for the generation of power results of the design.

CHAPTER 5

IMPLEMENTATION OF IEEE-754 STANDARD FLOATING POINT MULTIPLIER

This chapter introduces about the implementation and the simulation and synthesis results of floating point multiplier 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: 6.4a

5.1 ARRAY MULTIPLIER

5.1.1 Floating Point Representation

Figure 5.1 shows the the IEEE 754 single precision binary format representation; it consists of a one bit sign (S), an eight bit exponent (E), and a twenty three bit fraction (M or Mantissa). An extra bit is added to the fraction to form what is called the significand. If the exponent is greater than 0 and smaller than 255, and there is 1 in the MSB of the significand then the number is said to be a normalized number; in this case the real number is represented by (1)

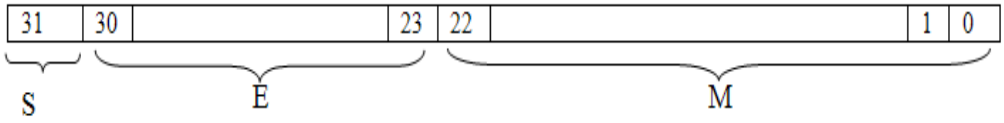

Figure 5.1. IEEE single precision floating point format[2]

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

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.

5.1.2 Multiplication

This unit is responsible for multiplying the unsigned significand and placing the decimal point in the multiplication product. The result of significand multiplication will be called the 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. A 24x24 bit carry save multiplier architecture is used as it has a moderate speed with a simple architecture. In the carry save multiplier, the carry bits are passed diagonally downwards (i.e. the carry bit is propagated to the next stage). Partial products are made by ANDing the inputs together and passing them to the appropriate adder. Carry save multiplier has three main stages:

1) The first stage is an array of half adders.
2) The middle stages are arrays of full adders. The number of middle stages is equal to the significand size minus two.
3) The last stage is an array of ripple carry adders. This stage is called the vector merging stage.

The number of adders (Half adders and Full adders) in each stage is equal to the significand size minus one. For example, a 4x4 carry save multiplier is shown in Fig. 5.2 and it has the following stages:

1) The first stage consists of three half adders.
2) Two middle stages; each consists of three full adders.
3) The vector merging stage consists of one half adder and two full adders.

The decimal point is between bits 45 and 46 in the significand multiplier result. The multiplication time taken by the carry save multiplier is determined by its critical path. The critical path starts at the AND gate of the first partial products (i.e. a1b0 and a0b1), passes through the carry logic of the first half adder and the carry logic of the first full adder of the middle stages, then passes through all the vector merging adders. The critical path is marked in bold in Figure 5.2.

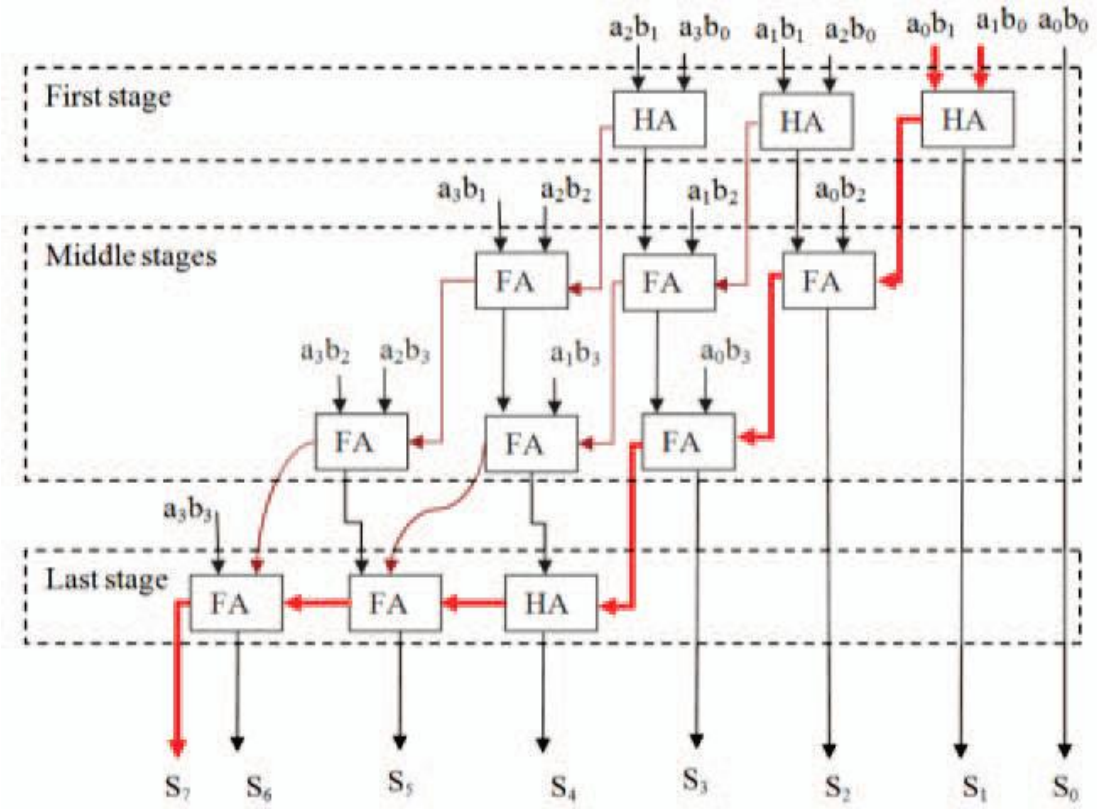

Figure 5.2: 4x4 bit Carry Save multiplier[2]

In Figure 5.2:

1- Partial product: aibj = ai and bj

2- HA: half adder

3- FA: full adder

5.1.3 Normalization

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[2].

5.1.4 Block Diagram of Floating Point Mutiplier

Figure 5.3 shows the block diagram of floating point multiplier. The hidden bit is added to the mantissa and multiplied using carry save multiplier. The result is then finally normalized and the output is calculated.

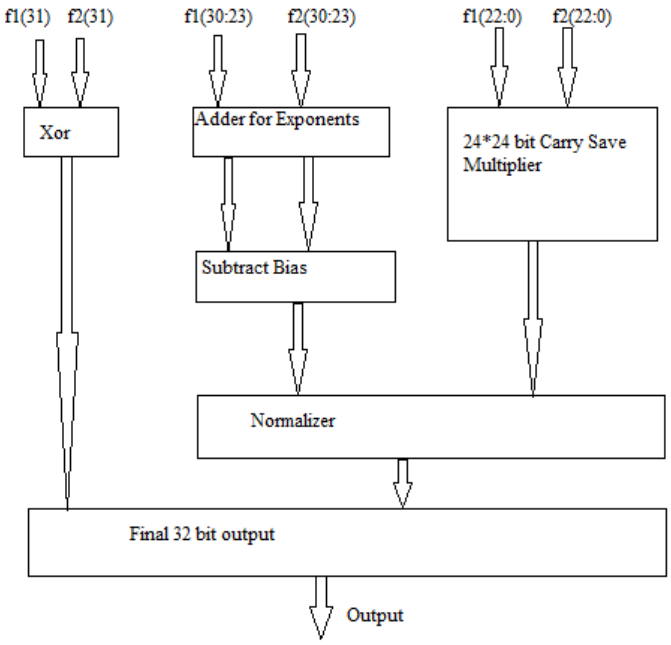

Figure 5.3: Floating Point Mutiplier Architecture

- Synthesis Results of floating point multiplication using array multiplier

a) Synthesis Results on Xilinx

Table 5.1 shows synthesis results of multiplication using array multiplier on Xilinx.

Table 5.1: Synthesis Report of Floating Point Multiplier

	Spartan 3E xc3s500E
No. of Slices	773/4656 (16%)
No. of LUTs	1508/9312 (16%)
Minimum Period	31.855ns
Maximum Frequency	31.392MHz

b) Synthesis Results on Design Compiler

Table 5.2 shows synthesis results of multiplication using array multiplier on Design Compiler.

Table 5.2: DC synthesis result for multiplier(1 unit= 1 NAND Gate)

Number of Nets	236
Combinational Area	53488.2304690 units
Noncombinational area	4799.691406 units
Total cell area	58286.894531 units
Total Dynamic Power	23.9935 mW
Cell Leakage Power	19.3128 nW

5.1.5 Simulation Results

Figure 5.4 shows the simulation results of multiplication using array multiplier.

f1: input data 32-bit

f2: input data 32-bit

clock: input clock

prod_f: Output product 32-bit

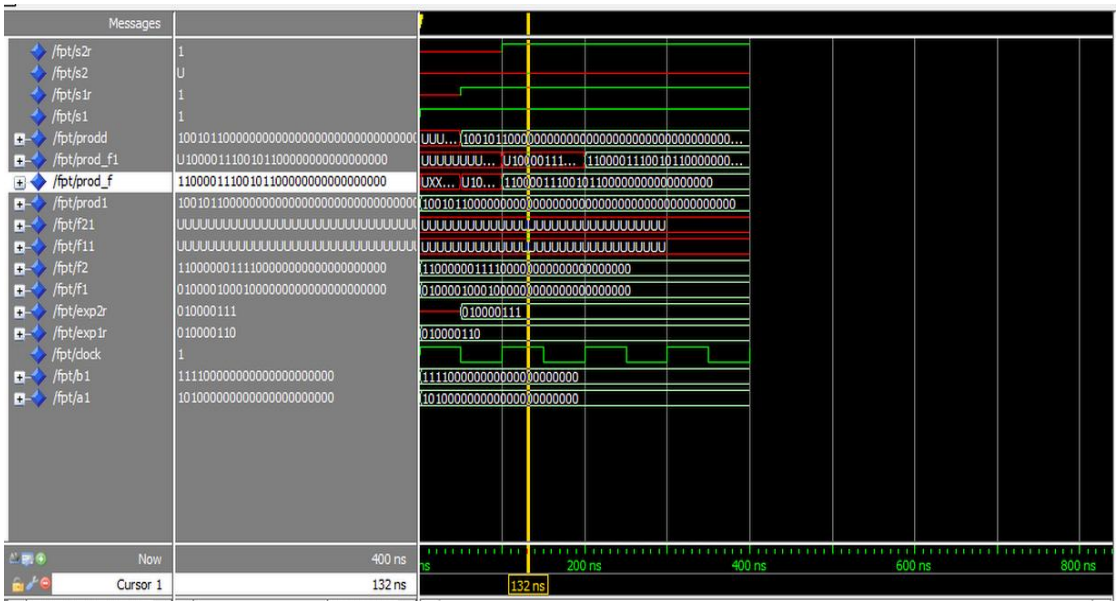

Figure 5.4: Simulation Results of Floating Point Multiplier

Consider the inputs:

f1= 01000010001000000000000000000000 (40)

f2= 11000000111100000000000000000000 (-7.5)

The output prod_f is:

prod_f = 11000011100101100000000000000000

5.2 BOOTH MULTIPLIER

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). To speed up this process it is divided in following stages:

5.2.1 Generation of Partial Products

For generating the partial products Radix-8 Modified Booth's Algorithm is used. In general, the higher the radix, the fewer the partial products generated. which translates into fewer cycles necessary to compute the product. For this reason, high radix recoding has found increasing use in modern high-speed arithmetic units and single chip multipliers. Since the multiplier and multiplicand comprises of 24 bits, this algorithm will generate 8 partial products.

Radix-8 Modified Booth's Algorithm

Recoding extended to 3 bits at a time - overlapping groups of 4 bits each. 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. Each quartet is codified as a signed-digit using the table 5.3:

Table 5.3: Recoding in Booth Radix-8 Algorithm [7]

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

- Synthesis Results of Partial Product Generation

a) Synthesis Results on FPGA

Table 5.4 shows the synthesis report of partial product generation.

Table 5.4: Synthesis Report of Partial Product Generation on Spartan 3E xc3s500E

	Using Radix-8 Booth's Encoding	Using Radix-4 Booth's Encoding[1]
No. of Slices	179/4656 (3%)	68/4656 (1 %)
No. of 4 input LUTs	327/9312 (3%)	123/9312 (1%)
Minimum Period	10.728 ns	11.306 ns
Maximum Frequency	93.211 MHz	88.44MHz

5.2.2 Partial Product Reduction

Multiplier blocks have been shown to require a small number of adders for multiplying one data sample with multiple, constant, coefficients. However, for high-speed applications carry-save adders are a better choice then carry propagation adder. 8 Partial products are generated using Radix-8 Modified Booth's Algorithm. They are reduced using 4:2 compressors.

Carry Save Adder

A Carry-Save Adder is just a set of one-bit full adders, without any carry-chaining. The most important application of a carry-save adder is to calculate the partial products in integer multiplication. 4:2 compressors are used as carry save adders. It consists of two 3:2 compressors (full adders) in series and involves a critical path of 4 XOR delays as shown in Figure 5.2. Owing to its regular interconnection, this compressor is ideal for the partial products addition stage. The 4:2 compressor structure actually compresses five partial products bits into three. Initially two 4:2 compressors are used to reduce each 4 partial products pair to generate the pair of sum and carry. Then these final 4 partial products generated from above two 4:2 compressors are further reduced to generate final sum and carry. 4:2 compressor is made from 2 full adders. The final carry is saved and hence is called carry save adder. The delay of 4:2 compressor is equal that of 4 xor gates.

- Synthesis Results of Partial Product Accumulation

a) Synthesis Results on FPGA

Table 5.5 shows the synthesis report on Xilinx for Partial Product Accumulation

Table 5.5: Synthesis Report of Partial Product Accumulation Spartan 3E xc3s500E

	Using 4:2 Compressors	Using Wallace tree[1]
No. of Slices	501/4656 (10%)	515/4656 (11%)
No. of 4 input LUTs	703/ 9312 (7%)	895/9312 (10%)
Minimum Period	5.184ns	13.708 ns
Maximum Frequency	192.905MHz	72.95MHz

5.2.3 Final Stage Carry Propagate Adder

Further the partial products generated through carry save adders are further reduced by using Ripple Carry Adder**.**

Ripple Carry Adder

Ripple Carry Adder is used to obtain the final sum and the output carry by adding the partial products from the carry save adders. It creates a logical circuit using multiple full adders to add

N-bit numbers. Each full adder inputs a C_{in}, which is the C_{out} of the previous adder. This kind of adder is a ripple carry adder, since each carry bit "ripples" to the next full adder. The 48-bit sum and carry outputs obtained from the partial product accumulator are added in the final stage adder to give the product of the mantissas.

This unsigned adder is also 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). The result of this stage is called the intermediate exponent.

- Synthesis Results of Final 48-bit Stage Carry Propagate adder

a) Synthesis Results on FPGA

Table 5.6 shows the synthesis report on Xilinx for Final 48-bit Stage Carry Propagate Adder

Table 5.6: Synthesis Report of Final Stage Adder Spartan 3E xc3s500E

	Using Ripple Carry Adder	Using Kogge-Stone Adder[1]
No. of Slices	150 /4656 (3%)	204/4656 (4%)
No. of 4 input LUTs	246 / 9312 (2%)	357/9312 (3%)
Minimum Period	8.235ns	18.946 ns
Maximum Frequency	121.438MHz	52.78MHz

5.2.4 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.

5.2.4.1 Unpipelined Multiplier

Table 5.7 and 5.8 shows the synthesis results on Xilinx and Design Compiler of Unpipelined floating point multiplier.

- Synthesis Results of Unpipelined Floating Point Multiplier

a) Synthesis Results on Xilinx

Table 5.7: Synthesis Report of Unpipelined Floating Point Multiplier Spartan 3E xc3s500E

	Our Floating Point Multiplier	Multiplier[1]
No. of Slices	1996 /4656 (42%)	1269/4656(27%)
No. of 4 input LUTs	4059 /9312 (43%)	2270/9312(24%)
Minimum Period	22.282ns	34.333 ns
Maximum Frequency	44.879MHz	29.126 MHz

b) Synthesis Results on Design Compiler

Table 5.8 shows the results of Unpipelined floating point multiplier on Design Compiler.

Table 5.8: DC synthesis result for Unpipelined Floating Point Multiplier.

(1 unit = 1 NAND Gate)

Combinational Area	169183.312500 units
Noncombinational area	11902.347656 units
Total cell area	181079.031250 units
Total Dynamic Power	54.4868 mW
Cell Leakage Power	60.3955 nW

5.2.4.2 3-Stage Pipelining

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:

i. After the Pre-processing of the Multiplicand and Multiplier.

ii. After subtracting the Bias and Compressing the partial Products to 4.

iii. After Normalization and Final Carry Propagate Adder.

- Synthesis Results of 3-stage Pipelined Floating Point Multiplier

a) Synthesis Results on Xilinx

Table 5.9 shows the synthesis report of 3-stage pipelined floating point multiplier on Xilinx.

Table 5.9: Synthesis Report of 3-Stage Floating Point Multiplier

	Spartan 3E xc3s500E
No. of Slices	2014/4656 (43%)
No. of Slice Flip Flops	244/9312 (2%)
No. of 4 input LUTs	3973/9312 (42%)
Minimum Period	13.272ns
Maximum Frequency	75.346MHz

b) Synthesis Results on Design Compiler

Table 5.10 shows the synthesis report of 3-stage pipelined floating point multiplier on Design Compiler.

Table 5.10: DC synthesis result for 3-stage pipelined floating point multiplier.

(1 unit = 1 NAND Gate)

Combinational Area	170536.265625 units
Noncombinational area	15480.262695 units
Total cell area	186009.968750 units
Total Dynamic Power	83.0874 mW
Cell Leakage Power	63.2998 nW

Figure 5.5 shows the various pipeline stages in the Multiplier and table 5.10 and 5.11 shows the synthesis results of 3-stage pipelined floating point multiplier.

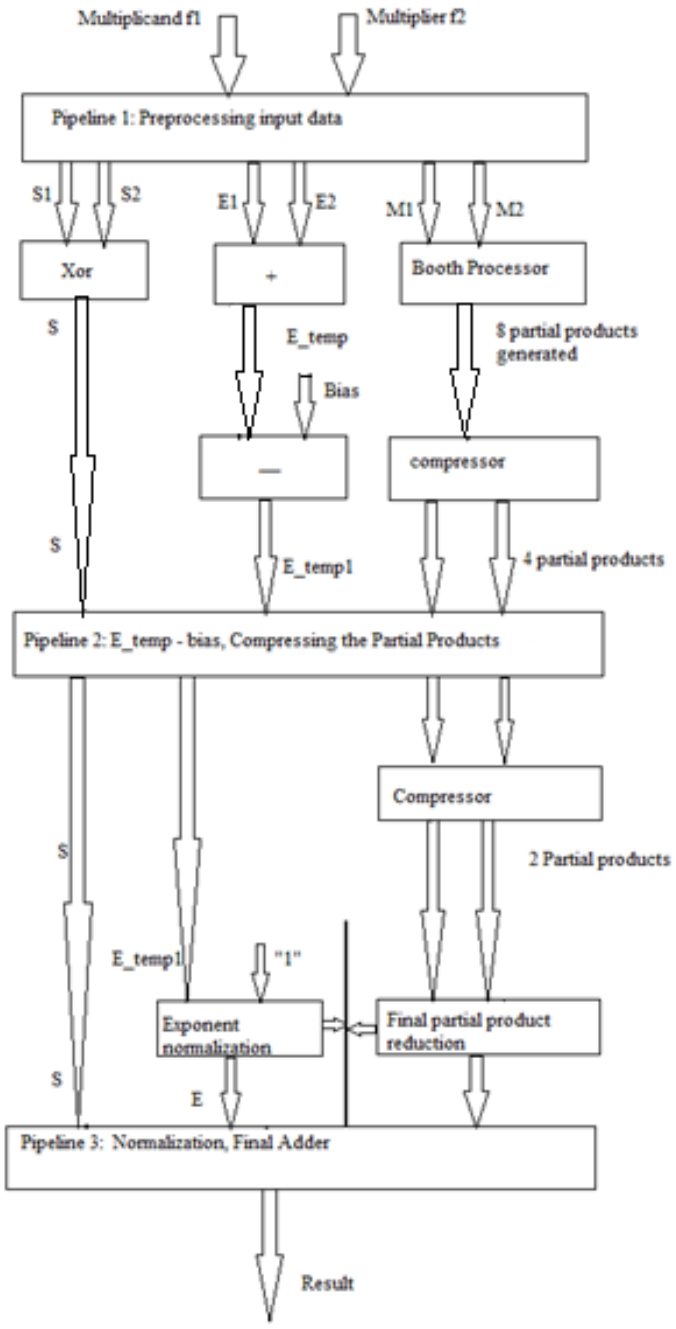

Figure 5.5: Floating Point Multiplier with 3 Pipeline Stages

5.2.4.3 5-Stage Pipelining

In order to enhance the performance of the multiplier, five 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:

i. After the Pre-processing of the Multiplicand and Multiplier.
ii. After the Exponent Adder and Generation of 8 Partial Products.
iii. After subtracting the Bias and Compressing the partial Products to 4.
iv. After Compressing the Partial Products to 2.
v. After Normalization and Final Carry Propagate Adder.

As the pipeline stages are increased, maximum operating frequency is also increased.

- Synthesis Results of 5-stage Pipelined Floating Multiplier

a) Synthesis Results on Xilinx

Table 5.11 shows the synthesis report of 5-stage pipelined floating point multiplier on Xilinx.

Table 5.11: Synthesis Report of 5 Stage Floating Point Multiplier

	Spartan 3E (xc3s500E)
No. of Slices	2045/4656 (43%)
No. of slice flip flops	545/9313 (5%)
No. of 4 input LUTs	3893/9312 (41%)
Minimum Period	12.154ns
Maximum Frequency	82.279MHz

b) Synthesis Results on Design Compiler

Table 5.12 shows the synthesis result for 5-stage pipelined floating point multiplier on DC.

Table 5.12: DC synthesis result for 5-stage pipelined multiplier (1 unit = 1 NAND Gate)

Combinational Area	173580.265625 units
Noncombinational area	36328.707031 units
Total cell area	209902.187500 units
Total Dynamic Power	96.6774 mW
Cell Leakage Power	71.9133 nW

Figure 5.6 shows the various pipeline stages in the Multiplier.

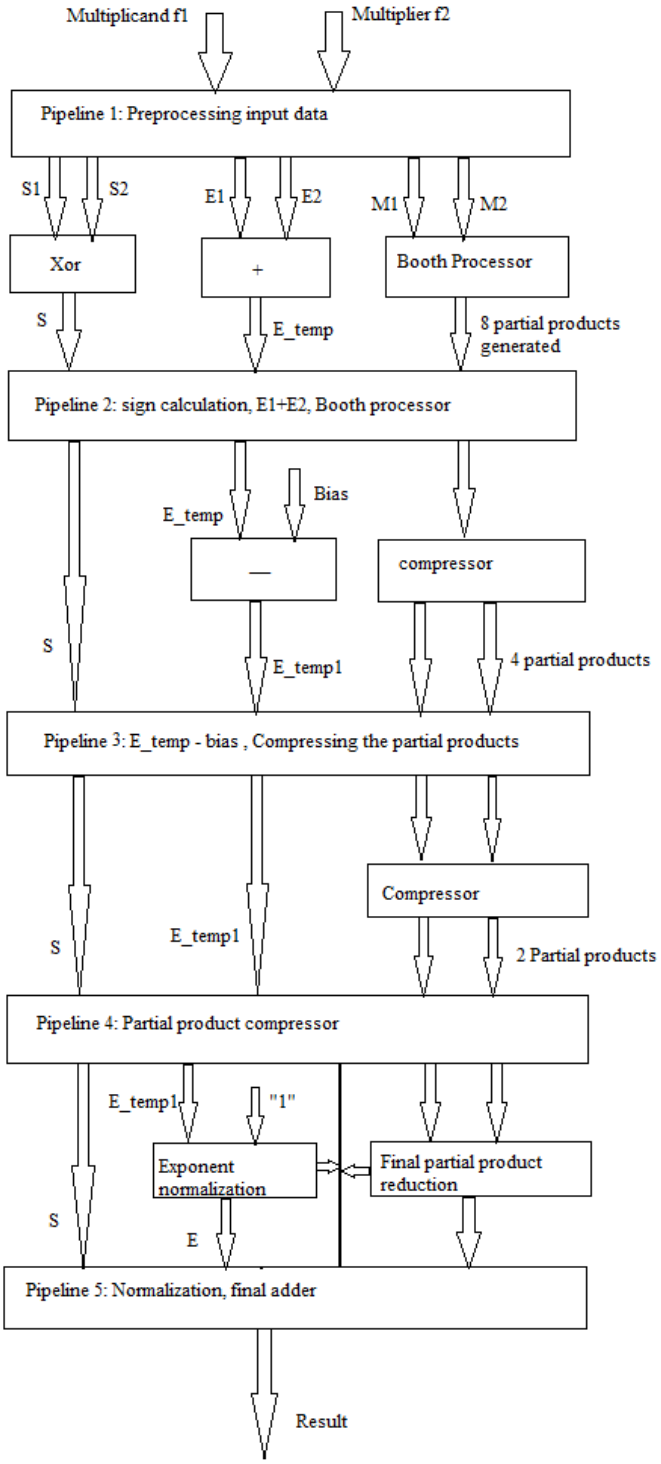

Figure 5.6: Floating Point Multiplier with 5-Pipeline Stage

5.2.5 Simulation Results

f1: input data 32-bit

f2: input data 32-bit

clock: input clock

prod_f: Output product 32-bit

s_inf_out: infinity exception

s_nan_out: not a number exception

s_ovf_out: overflow exception

s_und_out: underflow exception

5.2.5.1 Simulation Results for 5-stage Pipelined Floating Point Multiplier

Figure 5.7 shows the simulation results of 5-stage pipelined floating point multiplier.

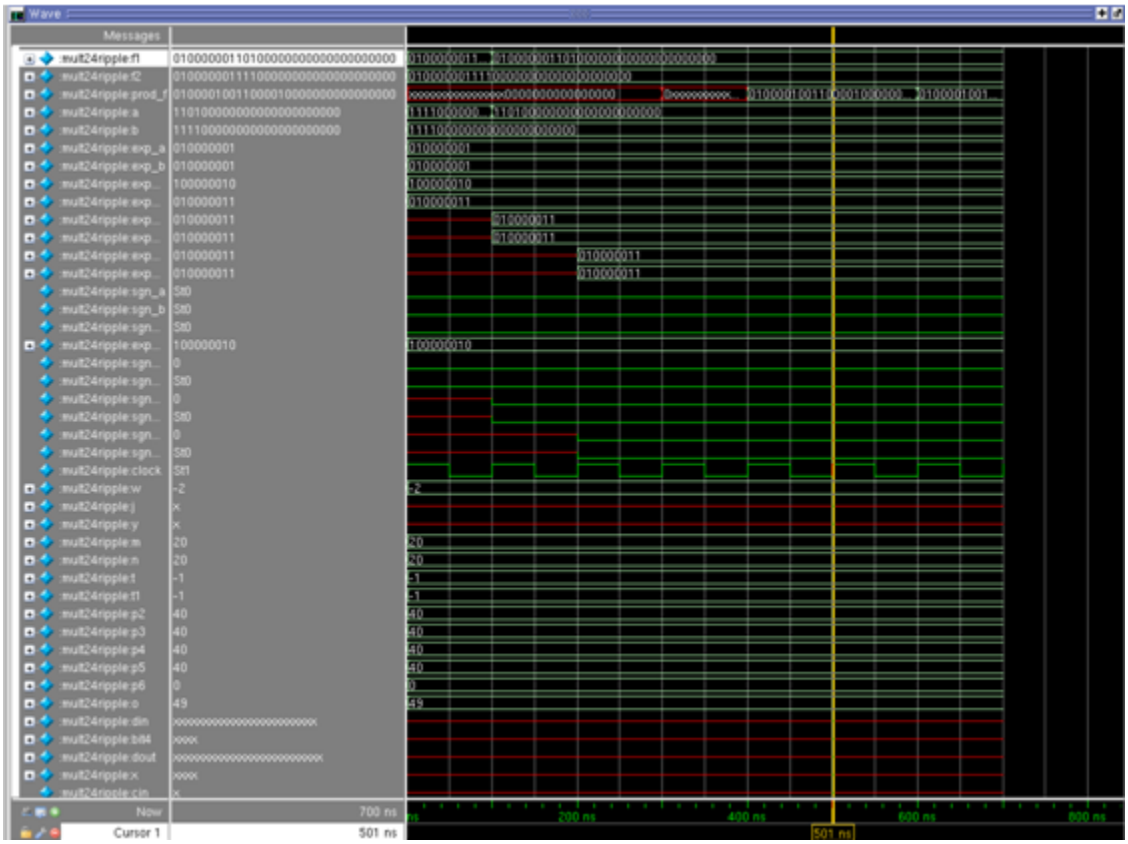

Figure 5.7: Simulated Waveform of 5-stage Pipelined Floating Point Multiplier

There is a latency of 5 clock cycles i.e. we will get the final output in 6th clock cycle. After that in every clock cycle we will, get the multiplication result.

On the 1st clock pulse Inputs f1 and f2 are fixed at 7.5. In IEEE 754 decimal 7.5 is represented as 0 10000001 11100000000000000000000. On 2nd clock f2 is given the same value but f1 is changed to 6.5 i.e 0 10000001 10100000000000000000000. After 5 clock cycles, i.e. on 6th clock period final product prod_f gets the value 0 10000100 11000010000000000000000 i.e 56.25. After next clock cycle it changes to 0 10000100 10000110000000000000000 i.e 48.75.

5.2.5.2 Simulation Results for Floating point Multiplier with exceptions

Consider the inputs to the floating point multiplier:

f1= 0 01111111 11100000000000000000000

f2= 0 11111110 11100000000000000000000

The output of the multiplier is:

Prod_f= 01111111100010001000000000110000

s_nan_out =1

Figure 5.8 shows the simulation results of floating point multiplier with exceptions.

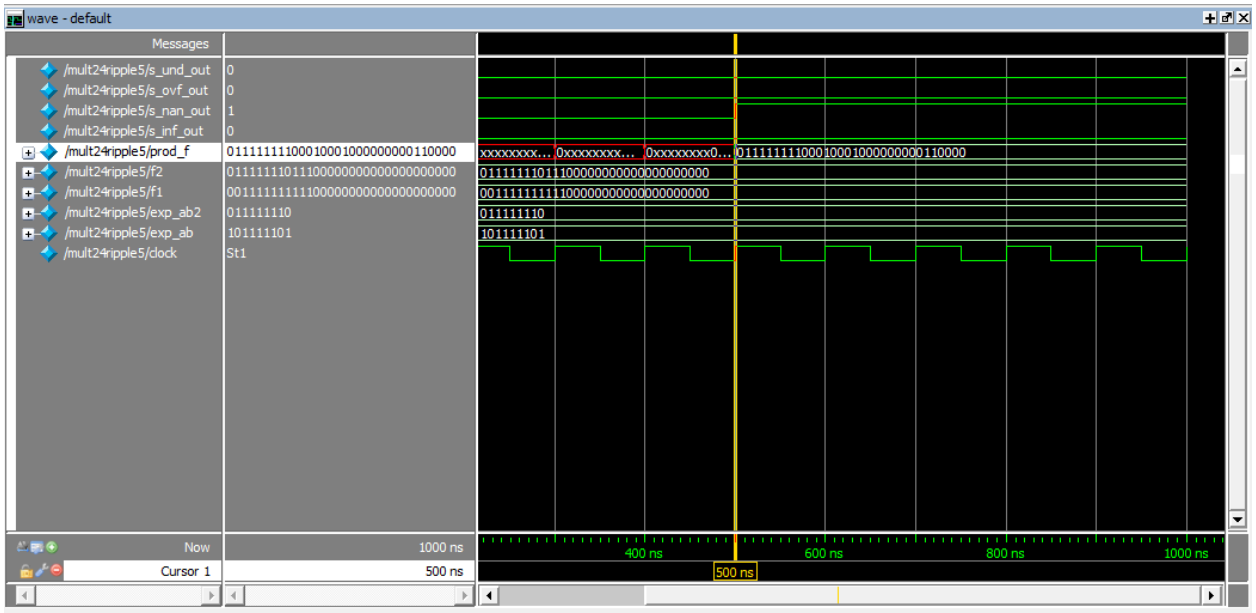

Figure 5.8: Simulated Waveform of Floating Point Multiplier with exceptions

CHAPTER 6

CONCLUSION

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

6.1 CONCLUSION

A new hardware implementation of a high speed floating point multiplier based on the IEEE-754 single precision format is developed based on pipeline technique. Verilog is used to implement a technology-independent pipelined design. 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 frequency generated is 31 MHz.

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-2 Booth's Algorithm, Radix-4 and Radix-8 Modified Booth's Algorithm are compared for the generation of partial products. Since the multiplier and multiplicand comprises of 24 bits, this algorithm will generate 8 partial products using Radix-8 Modified Booth's Algorithm. These 8 partial products 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, three and five pipelining stages are used to divide the critical path thus increasing the maximum operating frequency 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. For five stage pipelining, the pipelining stages are embedded at the following locations:

i. After the Pre-processing of the Multiplicand and Multiplier.
ii. After the Exponent Adder and Generation of 8 Partial Products.
iii. After subtracting the Bias and Compressing the partial Products to 4.
iv. After Compressing the Partial Products to 2.
v. After Normalization and Final Carry Propagate Adder.

Synthesis results of Unpipelined, 3-stage and 5-stage Pipelined Floating Point Multiplier on Xilinx Spartan 3E is shown in table 6.1. For an unpipelined multiplier we get an 60% increase in maximum frequency comparing with reference paper[1]. Further performance is improved by the pipelined stages in the multiplier. The maximum frequency obtained from 3-stage and 5-stage pipelined floating point multiplier is 75.346 MHz and 82.279 MHz respectively. Hence floating point multiplier achieves high throughput by insertion of pipeline stages but at the cost of area and power.

Table 6.1: Synthesis Report on Xilinx Spartan 3E xc3s500E

	Proposed Multiplier			Reference Multiplier[1]
	Unpipelined Floating Point Mulitiplier	3-Stage Pipelined Floating Point Multiplier	5-Stage Pipelined Floating Point Multiplier	Unpipelined Floating Point Multiplier
No. of Slices	1996/4656 (42%)	2014/4656 (43%)	2045/4656 (43%)	1269/4656(27%)
Minimum Period	22.282ns	13.272ns	12.154ns	34.333 ns
Maximum Frequency	44.879MHz	75.346MHz	82.279MHz	29.126 MHz

Synthesis Results on Design Compiler are in table 6.2. Extra logic circuitry with power penalty to achieve high throughput in pipelined multiplier.

Table 6.2: Synthesis Report on Design Compiler

(I unit= 1 Nand Gate)

	Unpipelined Floating Point Mulitiplier	3-Stage Pipelined Floating Point Multiplier	5-Stage Pipelined Floating Point Multiplier
Total area	181079.031250 units	186009.968750 units	209902.187500 units
Total Dynamic Power	54.4868 mW	83.0874 mW	96.6774 mW
Cell Leakage Power	60.3955 nW	63.2998 nW	71.9133 nW

The extra flip flops in 5-stage pipelining reduces the delay through the logic and hence pipelined code can operate at a higher frequency than the normal code. For 5-stage the ouput is generated after 5 clock cycles after the input is applied. But after 5 clock cycles we can get continuous stream of output.

6.2 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 lookahead 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.

REFERENCES

[1]. Anna Jain, Baisakhy Dash, Ajit Kumar Panda, Muchharla Suresh, "FPGA design of a fast 32-bit floating point multiplier unit", *2012 IEEE International Conference on Devices, Circuits and Systems, Coimbatore, India,March 15-16, pp 545 – 547,2012.*

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

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

[4]. Dhiraj Sangwan & Mahesh 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.*

[5]. Soojin Kim and Kyeongsoon 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.*

[6]. Young-Ho Seo, and Dong-Wook 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.*

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

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

[9]. Gong Renxi, Zhang Hainan, Meng Xiaobi,Gong Wenying, "Hardware Implementation of a High Speed Floating Point Multiplier Based on FPGA", *4th International Conference on Computer Science & Education,2009.*

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

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

[12]. N. Shirazi, A. Walters, and P. Athanas, "Quantitative Analysis of Floating Point Arithmetic on FPGA Based Custom Computing Machines", *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'95), pp.155–162, 1995.*

[13]. 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.*

[14]. Neil. H. E. Weste , "Principle of CMOS VLSI Design," Adison-Wesley, 1998.

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

[16]. Kavita Khare, R P Singh and Nilay 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.*

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

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

[19]. 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.*

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

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

[22]. Design Compiler User Guide v1999.10.

[23]. Creigton Asato, Christoph Di tzen and Suresh Dholakia, "A Datapath Multiplier with Automatic Insertion of Pipeline Stages", *IEEE Custom Integrated Circuits Conference,1989.*