DESIGN AND IMPLEMENTATION OF A DYNAMIC RANGE DETECTION BASED MULTIPLIER

A thesis submitted in partial fulfillment of the requirements

for the award of degree of

MASTER OF TECHNOLOGY

In

VLSI Design

Submitted By

SONIA VERMA

Roll No. 601261028

Under guidance of

Ms. Sakshi

Assistant Professor

Department of Electronics and Communication Engineering

THAPAR UNIVERSITY

(Established under the section 3 of UGC Act, 1956)

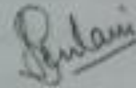
PATIALA – 147004 (PUNJAB)

July 2014

CERTIFICATE

I hereby declare that the work which is being presented in the thesis entitled, "Design and Implementation of Dynamic Range Detection Based Multiplier" in partial fulfillment of the requirement for the award of degree of Master of Technology in VLSI Design submitted in Electronics and communication Engineering Department of Thapar University, Patiala is an authentic record of my own work carried out under the supervision of Ms. Sakshi, Assistant Professor, ECED.

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

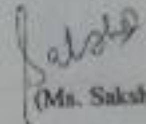


Date: 02/07/2014

(Sonia Verma)

Roll No: 601261028

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

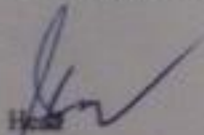


(Ms. Sakshi)

Assistant Professor

ECED, Thapar University

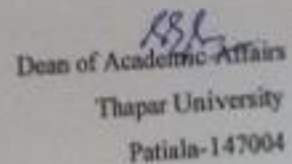
Countersigned by:



Head

ECED, Thapar University

Patiala-147004



Dean of Academic Affairs

Thapar University

Patiala-147004

ACKNOWLEDGEMENT

I would have never succeeded in completing my task without the cooperation, encouragement and help provided to me by various people. Words are often too less to reveals one's deep regards. I take this opportunity to express my profound sense of gratitude and respect to all those who helped me through the duration of this thesis. I acknowledge with gratitude and humility my indebtedness to **Ms. Sakshi, Assistant Professor,** Electronics and Communication Engineering Department, Thapar University, Patiala, under whose guidance I had the privilege to complete this thesis. I wish to express my deep gratitude towards her for providing individual guidance and support throughout the thesis work.

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

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

(SONIA)

ABSTRACT

In today's world of portable and super fast electronic gadgets, power and speed are of major concern. A good battery back up is desirable for a product. Multiplication is used in nearly every field whether it is Digital Signal Processing, image processing or arithmetic units in microprocessors and contributes largely to the total power consumption. On VLSI implementation level, the area also becomes quite important as more area means more system cost. So power, speed and area are three important parameters in VLSI Design. Increase in speed will lead to increased power consumption. Hence these parameters are always traded off.

In this thesis, an architecture for power efficient Dynamic Range Detection Based multiplier with compressors at the accumulation stage has been proposed. Dynamic Range Detection technique chooses the operand which on being taken as multiplier gives more partial product rows as zero. This will reduce the switching activity and hence power consumption is reduced. For partial product generation, Booth's recoding is used. Booth's Algorithm speeds up the multiplication as number of partial product rows is reduced. The level of reduction depends upon the radix r used. Wallace and Dadda algorithms are implemented at the accumulation stage which use half adders and full adders to sum up the partial product rows. In the Wallace algorithm, the partial product rows are reduced as soon as possible where as in Dadda minimum reduction is done at each level. Further compressors of different orders are used to sum up the partial product rows. These can compress three, four, five or seven rows to final two rows at a time and hence speed is increased as order of compressor increases. So carry save adders are used for partial product accumulation and carry propagating adder for adding last two rows.

These multipliers have been designed with the help of VHDL, simulated and synthesized on Xilinx ISE 14.5 targeted on Spartan 3E FPGA. Comparisons have been done in terms of delay, power and area.

TABLE OF CONTENTS

SR.NO.	CONTENTS	PAGE No .
	Certificate…………………………………………………………..	i
	Acknowledgement……………………………………………………..	ii
	Abstract……………………………………………………………...	iii
	Table of contents……………………………………………………	iv
	List of figures………………………………………………………..	vi
	List of Tables………………………………………………………..	viii
	Abbreviations………………………………………………………..	ix
1.	**CHAPTER 1- Introduction…………………………………………**	1
	1.1 Classification of multipliers……………….…………………….	2
	1.2 Objective……………………….……………………………...	4
	1.3 Thesis Organisation………………………………………….....	4
2.	**CHAPTER 2- Literature Review……...…………………………….**	5
3.	**CHAPTER 3- Multiplication…..……………………………………**	8
	3.1 Introduction……………….…………………………………………	8
	3.1.1 Block Diagram of Multiplier………………………………..	9
	3.2 DRD Unit………………………………………………………..	10
	3.3 Generation of partial products…………………………………….	11
	3.2.1 Radix-2 Booth's Algorithm ………………………………...	12
	3.2.2 Radix-4 Booth's Algorithm ………………………………..	13
	3.2.3 Radix-8 Booth's Algorithm…………………………………	15
	3.2.4 Comparison of radix 2, radix 4 and radix 8 algorithm……...	16
	3.4 Partial Product Reduction…………………………………….....	17
	3.3.1 3:2 Compressor…………………………………………...	17
	3.3.2 4:2 Compressor…………………………………………...	18
	3.3.3 5:2 Compressor…………………………………………..…	18
	3.3.4 7:2 Compressor…………………………………………...	18
	3.5 Final Stage Adder …………………………………………………	20
	3.5.1 Ripple Carry Adder………………………………………….	20
	3.5.2 Carry Look Ahead Adder……………………………………	21

3.6 Tree Multipliers …………………………………………………….. 22
3.6.1 Dadda Tree Algorithm………………………………………... 22
3.6.2 Wallace Tree Algorithm……………………………………… 23
4. **CHAPTER 4-Field Programmable Gate Arrays……………...……** 25
4.1 Introduction……..………………………………………………… 25
4.2 FPGA Technology Trends.…………..……………………………. 25
4.3 FPGA Implementation…………………………………………….. 26
4.3.1 Overview of FPGA Design flow…………………………..... 26
5. **CHAPTER 5-Implementation of DRD Based Parallel Multiplier………………………………………………………...** 32
5.1 Design of Multiplier……………………………………………… 32
5.2 Results and Discussions……………..……………………………. 33
5.2.1 Simulation Results…………….……………………………. 33
5.2.2 Synthesis Results…………………………………………… 34
6. **CHAPTER 6- Conclusion………………………………………..** 36
6.1 Conclusion………………………………………………………. 36
6.2 Future Scope…………………………………………………….. 38
REFERENCES…………………………………………………... 40

LIST OF FIGURES

Figure No. **Title of Figure** **Page No.**

1.1 Classification of Multiplier……………………..3

3.1 Basic Multiplication……………………………..8

3.2 Block Diagram of Multiplier…………………...9

3.3 Dynamic Range Detection Unit.......................10

3.4 Recoded Version of multiplier………………….12

3.5 Example of Radix-2 Algorithm…………………13

3.6 Recoding in Radix-4……………………………13

3.7 Example of Radix-4 Algorithm......................14

3.8 Example of Radix-8 Algorithm………………..15

3.9 Partial products multiplexer...........................16

3.10 CSA function in dot notation...............................17

3.11 3:2 Compressor I/O Diagram...............................17

3.12 4:2 Compressor Architecture……………………18

3.13 5:2 Compressor Architecture……………………19

3.14 7:2 Compressor Architecture……………………19

3.15 Ripple Carry Adder……………………………..20

3.16 Critical paths in a k-bit Ripple
Carry Adder..21

3.17 4-BIT RLA logic equations…............................21

3.18 Dot diagram for an 8 by 8 Dadda
Multiplier……………..................................23

3.19 Dot diagram for an 8 by 8 Wallace
Multiplier……………………………………….24

4.1 FPGA Design flow……………………………..27

5.1 Multiplication of two 8-bit numbers…………...32

5.2 Simulation waveforms of 16×16 bit
multiplier with DRD…………………………….33

5.3 Simulation waveforms of 8×8 bit

Wallace tree multiplier……………………………34
5.4 Simulation waveforms of 8×8 bit
Dadda tree multiplier……………………………...34
6.1 Changes in Delay(ns) v/s order of
Compressor……………………………………...36
6.2 Changes in Power(mW) v/s order of
Compressor……………………………………...36
6.3 Delay(ns) comparison of 16 bit
Multiplier with and without DRD…………………37
6.4 Power(mW) comparison of 16-bit
multiplier with and without DRD………………..37
6.5 Delay comparison of Wallace
and Dadda tree multiplier…………………………….37

LIST OF TABLES

Table No.	Title of Table	Page No.
3.1	Recoding in Booth Radix-2 algorithm……………….	12
3.2	Recoding in Booth Radix-4 algorithm……………….	14
3.3	Recoding in Booth Radix-8 algorithm……………….	15
5.1	Various multipliers designed and implemented……...	33
5.2	Delay and Area comparison of multipliers without using DRD… ……………………………………...	34
5.3	No. of slices and delay of multiplier designs without using DRD………………………………………....	35
5.4	Power consumption of multiplier designs with and without using DRD…………………………………...	35
5.5	Area and delay comparison of Wallace and Dadda Multiplier…………………………………………….	35

ABBREVIATIONS

CSA	Carry save adder
DSP	Digital signal processing
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
MBE	Modified Booth Encoding
RCA	Ripple-carry Adder
VHDL	Very High Speed Integrated Circuits HDL
VLSI	Very Large Scale Integration
DRD	Dynamic Range Detection
MSB	Most Significant Bit
LSB	Least Significant Bit
CPA	Carry Propagating Adder

CHAPTER 1

INTRODUCTION

Multiplication is a mathematical operation in which an integer is added a specified number of times. Multipliers are indispensable in modern electronic systems where high speed calculations are required such as FIR filters, digital signal processors and microprocessors etc. Multiplication based operations such as multiply and accumulate (MAC) and inner product are among some of the frequently used computations[1]. Currently, time taken by multiplication operation is still the dominant factor in determining the instruction cycle time of a DSP chip and the demand for high speed processing has been increasing because of expanding computing and signal processing applications. Also the multiplier is generally the slowest element in the system.

The systems which typically require low power consumption such as Portable multimedia and digital signal processing (DSP) with shorter design cycle and flexible processing ability are becoming popular over the last few years. Many multimedia and DSP applications are highly multiplication intensive, hence the performance and power consumption of these systems are dominated by multipliers. Unfortunately, portable devices mostly operate with stand-alone batteries, but multipliers consumes large amount of power. Multiplication algorithms are needed by Digital signal processing systems to implement DSP algorithms such as filtering where the multiplication algorithm lies within the critical path. Performance in most cases strongly depends on the effectiveness of the hardware used for computing multiplications as multiplication is massively used in these environments for instance in signal processing applications, multimedia, and 3D graphics. Consequently, it is of vital importance to develop power efficient multipliers to develop a high-performance and low-power portable multi-media and DSP systems. So reducing the time delay and power consumption of multipliers are very essential requirements for many applications. In the past implementation of multiplication operation was done generally with a sequence of addition, subtraction and shift operations. Multiplier is quite a 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.

As the scale of integration keeps growing due to miniaturization, more and more sophisticated signal processing systems are being implemented on a VLSI chip. These signal

processing applications not only demand great computation speed and capacity but also consume considerable amount of energy. While the speed and Area remain to be the two major design tools, the higher speed results into enlarged power consumption, thus, low power architectures will be the choice of the future. The need for low-power VLSI system arises from two main forces. First, with the steady growth of operating frequency and processing capacity per chip, large currents have to be delivered and the heat due to large power consumption must be removed by proper cooling techniques. Second, battery life in portable electronic devices is limited. Low power design directly leads to prolonged operation time in these portable devices. This has given way to the growth of new circuit algorithms, with the plan of reducing the power consumption of multiplication algorithms with having high-speed structures and appropriate performance. The multiplier is fairly large block of a computing system. The standard method of multiplying two *n*-digit numbers requires n^2 multiplications

With advances in technology, many researchers have tried and are trying to design multipliers which offer either of the following design targets: high speed, low power consumption, less area or even combination of them in one multiplier. However the fact remains that the area and the speed are the two conflicting performance constraints. Hence, innovating increased speed always result in larger area[2].

1.1 Classification of Multiplier

There are three kinds of multiplier as shown in fig. 1.1

a) Serial multiplier

b) Parallel multiplier

c) Serial-Parallel multiplier (SPM)

In serial multiplication algorithms, sequential circuits are used with feedbacks. The inner products are sequentially produced and added serially. But speed of serial multipliers is very less as compared to parallel multipliers because every operation is done serially and there is no parallel operation. Serial Multiplier adds each of the bits of the multiplicand sequentially and the process is repeated for each of the multiplier bits. Both the operands are entered in a serial form. Only one adder is used to add the $m \times n$ number of partial products where m and n are number of bits of multiplicand and multiplier respectively. Serial Multipliers are used where area and power are of utmost importance and where delay can be ignored. Bit-serial

arithmetic is attractive in view of its smaller pin count, reduced wire length, and lower floor space requirements in VLSI[3].

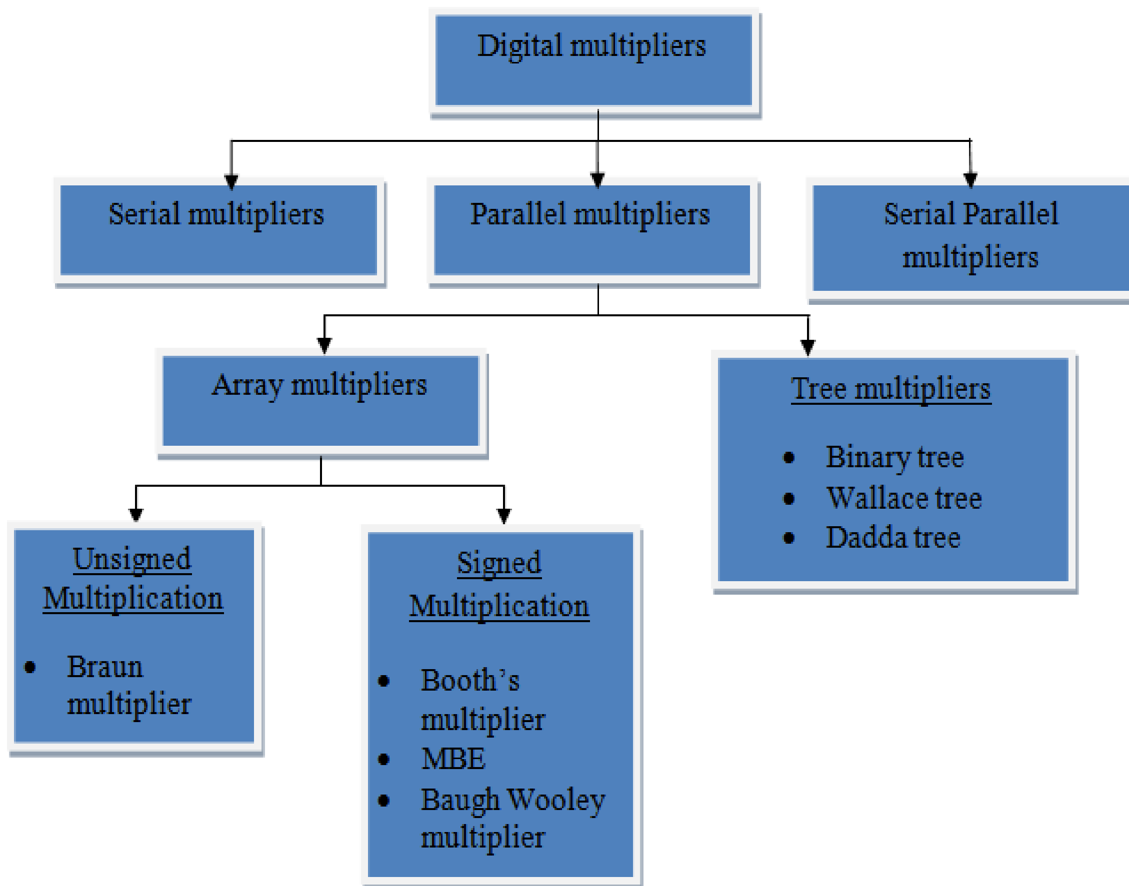


Fig. 1.1: Classification of Multipliers

The compactness of the design may allow to run a bit-serial multiplier at a high enough clock rate to make it competitive with much more complex designs with regard to speed. In a conventional parallel multiplier generation of partial products is done first by multiplying the multiplicand with each bit of the multiplier. Then these partial products are added together to generate the resultant product P. In short, we can break down multiplication process into two parts, namely partial product generation and partial product accumulation. The number of partial products to be added plays an important role in determining the performance of the Parallel multiplier.

The serial-parallel multiplier (SPM) operates on each bit of the multiplier serially, but uses a parallel adder for partial product accumulation. The serial parallel multiplier serves as a good trade-off between time consuming serial multiplier and area consuming parallel multiplier The SPM accumulates one partial product (the multiplicand times one bit of the multiplier) each time and shifts the temporary result to right for a simpler logic connection

(the output of the adder is always sent to the same place). The only reason one prefer parallel multiplier is that parallel multiplication algorithms often use combinational circuits and do not contain feedback structures [4].

In parallel multipliers, there are two main classifications. They are

1. Array multipliers
2. Tree multipliers.

C.S. Wallace proposed a tree multiplier architecture which performs high speed multiplication. But this has a high structural irregularity and is unsuitable for VLSI implementation as it demands regularity. Array multiplier is classified as signed and unsigned multipliers. Braun multiplier is used for unsigned multiplications whereas Baugh-Wooley multiplier, booth's multiplier and modified booth's multiplier can be used for signed multiplication.

1.2 Objective

The primary objective of this thesis is to design and implement multiplier focusing on methods to decrease the power consumption and minimizing the overall delay. No doubt these parameters are inversely proportional to each other and improving one comes at a cost of other. So a trade off is there between power, speed and area for an efficient multiplier because all these parameters are of utmost importance in today's time. Partial products are generated using radix-4 modified booth's algorithm and they are reduced using carry save adder and ripple carry adder respectively. Wallace and Dadda algorithms are implemented at the accumulation stage which uses half adders and full adders to sum up the partial product rows. Further compressors of different orders are used to sum up the partial product rows.

1.3 Thesis Organisation

Chapter 1: Introduces multipliers and its classification.

Chapter 2: Discusses Literature Review

Chapter 3: Starts with the introduction of multipliers and its block diagram. Discusses Dynamic range Detection and various Algorithms for partial product generation and accumulation.

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

Chapter 5: Shows the simulation and synthesis results of array multiplier using different algorithms.

Chapter 6: Derives the Conclusion and tells about future scope

CHAPTER 2

LITERATURE REVIEW

D. Garg et al.[7] presented a Dynamic Range Detection based multiplier which aimed at reducing the power consumption of multiplier. This technique basically chooses the operand which has a probability of making more partial products rows zeroes, after being taken as multiplier when it's Booth Recoding is done. The power consumption is further reduced by using truncation technique on the output product at the cost of output precision. . The proposed multiplier adopted due to its simple architecture outperforms over simple and conventional booth multiplier for these applications where truncation is performed. As a result, the proposed multiplier is very suitable for portable multimedia and DSP applications which require flexible processing ability, lesser switching activity and short design cycle.

S. R. Kuang et al.[8] proposed a configurable low power booth multiplier capable of single 16bit or twin parallel 8 bit multiplication operations. Dynamic Range detection technique is used in this paper which dynamically detect the effective dynamic ranges of two input operands. The detection result is used to not only pick the operand with smaller dynamic range for Booth encoding to increase the probability of partial products becoming zero but also deactivate the redundant switching activities in ineffective ranges as much as possible. In this multiplier truncation method is also used which truncates the output product to decrease the power consumption at the cost of a bit of output precision. Some additional components including a correcting vector generator, an adjustor, a sign-bit generator, a modified error compensation circuit, etc are also developed. The results show that the proposed multiplier is more complex but it certainly decreases the power consumption.

C.Y.H. Lee et al.[10] presented relative performance study of various multiplier designs namely Array, Wallace, Dadda and Reduced Area multipliers. All multiplier designs were modeled in Verilog HDL and synthesized based on the TSMC 0.35-micron ASIC Design Kit standard cell library. Logic synthesis report data for Area, Speed and Auto optimization mode indicates that Dadda multiplier may not always be faster than Wallace multiplier, it depends on the optimization mode used. Also it is found that Wallace multiplier is suited for high

speed applications independent of area constraints, while the Dadda and Reduced area designs deliver best speed when synthesized to minimize area or logic usage.

S. Shah et al.[11] shows a relative performance comparison of various 32-bit multiplier designs of Array, Dadda, Reduced Area and Radix-4 Booth Encoding multipliers in the Area Optimised, Speed Optimised and Auto Optimised modes. Different results are obtained with different modes. In all the modes, Radix-4 exhibits the best area performance. In speed optimised mode, Wallace multiplier exhibits largest area performance and is fastest in terms of speed performance and Array multiplier has the longest delay. In Area optimised and Auto optimised mode, Array multiplier shows the longest time delay performance and Dadda multiplier exhibits the shortest time delay in terms of speed.

L. H. Hiung et al.[13] shows a relative performance comparison of Radix-based Booth Encoding multiplier designs. Radix-2 Booth Encoding multiplier, Radix-4 Booth Encoding multiplier, Radix-8 Booth Encoding multiplier, Radix-16 Booth Encoding multiplier and Radix-32 Booth Encoding multiplier are compared. Synthesis data extracted from Area, Speed and Auto optimised modes indicate that largest area and longest timing delay is observed in Radix-2 Booth encoding multiplier design. As the number of Radix Based multiplier is increased, the number of partial product rows are reduced but at the cost of complexity of operations performed. Results indicate that Radix-4 Booth Encoding multiplier design is the best multiplier in terms of speed and area.

Y.H. Seo et al.[17] proposed a new VLSI architecture of parallel multiplier using Radix-2 Modified Booth algorithm for high speed arithmetic A hybrid type of carry save adder is devised by combining multiplication and accumulation. The proposed CSA tree uses 1's-complement-based radix-2 Booth's algorithm (MBA) and has the modified array for the sign extension in order to increase the bit density of the operands. The CSA propagates the carries to the least significant bits of the partial products and generates the least significant bits in advance to decrease the number of the input bits of the final adder.

S. Veeramachaneni et al.[18] presented novel architectures and designs of low power and high speed compressors used for addition in the partial product accumulation stage. The compressors 3:2, 4:2 and 5:2 are the basic components in many applications where addition is required specially in multiplication. The three basic parameters area, power and speed of

these compressor architectures are compared with existing and recently proposed compressor architectures and results show that recently proposed architectures gives a better performance. An emphasis is laid on the use of multiplexers in arithmetic circuits. These new compressors are a combination of low power, low transistor count and lesser delay.

V. Elakya et al.[19] proposed a power efficient multiplier which used low power Booth recoding technique to reduce the power consumption. This can be achieved by introducing as many zeroes as possible in partial product rows. This will reduce the switching activity and hence the reduction in power consumption is achieved. The switching activity is determined by the input bit coefficient and when this coefficient is zero corresponding rows or column of adder is deactivated. When multiplicand contains more number of zeroes the reduction in power consumption is higher.

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 in which an integer is added 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 basic method of multiplication, firstly the multiplicand is placed 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:

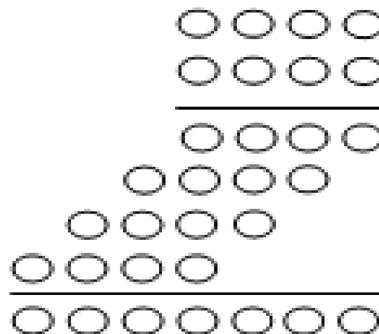


Fig. 3.1: Basic Multiplication

In the binary number system the digits, called bits, are denoted by the set [0,1]. The result of multiplying any binary number by a single binary bit is either 0 or the number itself. Hence forming the intermediate partial-products is 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 $x = x_{n-1}....x_1x_0$ and the multiplier $y = y_{n-1}....y_1y_0$ 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 parts:

1) Partial Product Generation.
2) Partial Product Reduction.
3) Final Adder.

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[4]. Block diagram of multiplier is shown in figure 3.2:

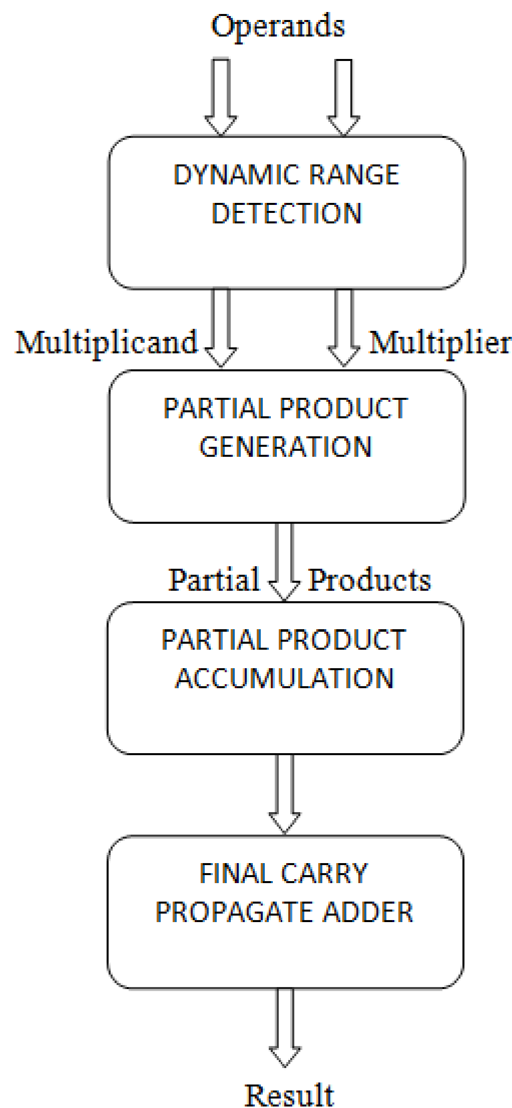


Fig. 3.2: Block Diagram of multiplier

3.2 Dynamic Range Detection Unit

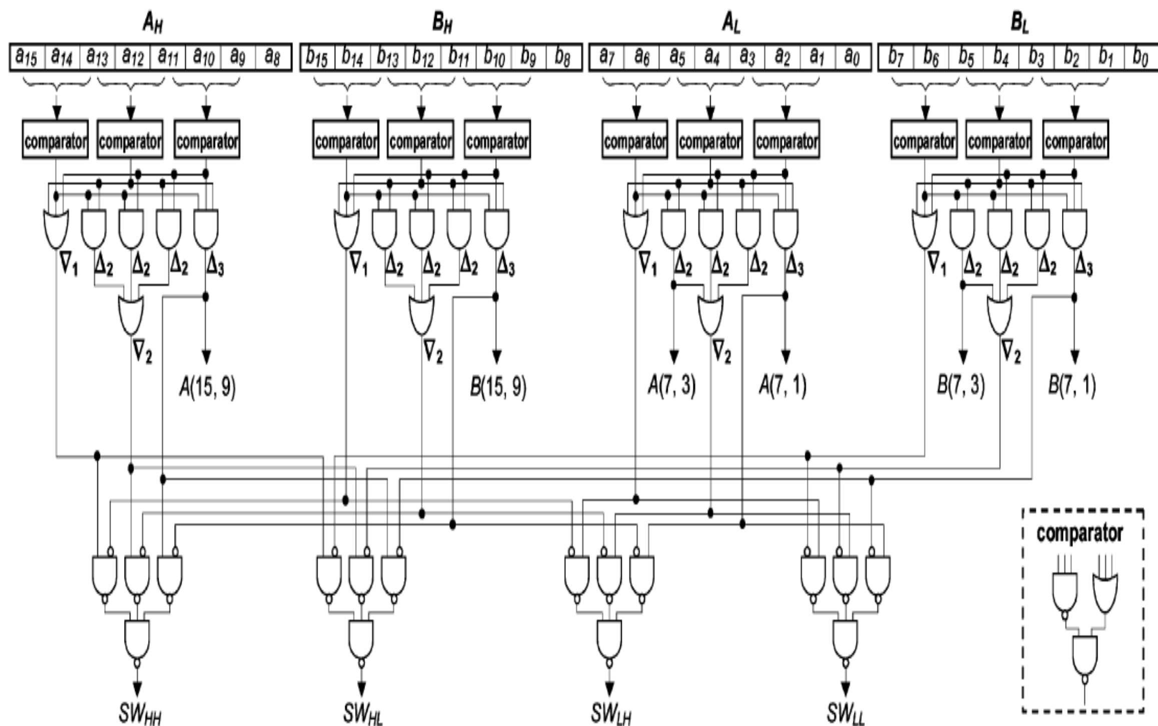


Fig. 3.3: Dynamic Range Detection Unit[8]

Basic multiplication operation involves adding the multiplicand, n number of times where n is the value of multiplier. So the number of addition operations depends on the value of multiplier. Suppose we need to multiply 5 and 3, adding 5 three times (5+5+5) would be faster than adding 3 five times (3+3+3+3+3). And also the first operation will consume less power. So wisely choosing the multiplicand and multiplier also contributes to the performance of multiplier. Dynamic Range Detection (DRD) technique aims at picking multiplier from the two operands taken for multiplication which can make the operation faster and will reduce the power consumption[6].

To efficiently reduce power consumption, a novel dynamic-range detector has been developed to dynamically detect the effective dynamic ranges of two input operands. The detection result is used to pick the operand with smaller dynamic range for Booth encoding to increase the probability of partial products becoming zero[7]. The proposed dynamic-range detector generates switching signals SW_{LL} and SW_{HH}, and for each 8 bit and 16 bit Booth multiplication to pick the operand that leads more partial products to zero for Booth encoding as shown in fig. 3.3.

The Dynamic Range Detector breaks Multiplicand A into A_L and A_H and Multiplier B into B_L and B_H where A_L, A_H, B_L and B_H are of 8bits each. Firstly the input operands are partitioned into 3-bit groups a_{2i+1}, a_{2i}, a_{2i-1} or b_{2i+1}, b_{2i}, b_{2i-1} and are fed into comparator. If the

output of a comparator is 1, it indicates that the input 3-bit group is successive zeroes or ones and hence it's booth encoded product will be zero. The number of booth encoded products with a zero value is denoted as $\Delta_{i.}$. These delta signals are further used to generate the switching logic signals[8].

Switching signals SW_{HH} and SW_{LL} are generated by the circuit shown in fig. 3.3. If SW_{LL} ='1' then the lower bits of multiplicand and multiplier are interchanged else no operation is done. Similarly if SW_{HH} ='1' then the higher bits of multiplicand and multiplier are interchanged. So basically SW_{LL}='1' or SW_{HH} ='1' indicates that probability of more number of partial product rows becoming zero is higher, if the multiplicand and multiplier are interchanged. If multiplier bits ($C_{k)}$ are "000" or "111" then partial product row is 0. So operand which has more number of C_k as "000" or "111" is preferred to be taken as multiplier.

3.3 Generation of Partial Products

In the initial step of digital multiplication, n shifted copies of the multiplicand needs to be generated, which is then be added in the coming stage. The value of the multiplier bit is used to determine 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 else it is not added and ignored. 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 are better in speed and power but they require large area of silicon, unlike the add-shift algorithms, which require less hardware and exhibit relatively poor performance[5]. In 1951, Andrew Donald Booth devised a multiplication algorithm, which was named as Booth's Algorithm.

In the Booth multiplier, the number of partial products is reduced with the use of Booth encoding algorithm by considering certain number of bits of the multiplier instead of just one bit 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. If MSB of operand is 1, it is taken as a negative number and its actual value is 2's complement of the given number. For instance if the operand is 11111111 then its actual value is -1. The product obtained when one of the operand is a negative number is actually the 2's complement of obtained result and is a

negative number. The best part of Booth's algorithm is that it takes cares of the signed numbers by itself. Various algorithms of booth recoding are discussed as under:

3.3.1 Radix-2 Booth's Algorithm

First of all, to start with a '0' is appended to the right of the multiplier i.e LSB is now '0'. Subsequently, the current bit x_i and the previous bit x_{i-1} of the multiplier, $x = x_{n-1}....x_1x_0$ are examined in order to yield i^{th} bit, y_i of the recoded multiplier, $y = y_{n-1}....y_1y_0$. 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 algorithm[12].

Table 3.1: Recoding in Booth Radix 2 Algorithm[12]

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- $x_{n-1}x_{n-2}...x_1x_0$ in SD (sign digit) code

· Recoded multiplier- $y_{n-1}y_{n-2}...y_1y_0$

· x_ix_{i-1} of multiplier examined to generate y_i

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

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

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

Example in figure 3.4 shows the recoded version of the multiplier and figure 3.5 shows the multiplication using above technique.

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

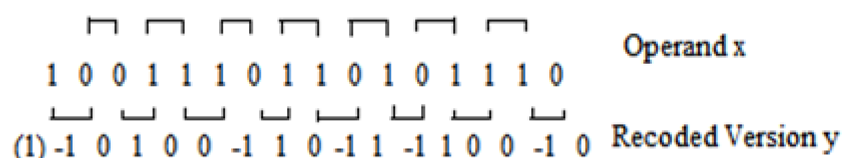


Figure 3.4: Recoded Version of the multiplier[12]

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 3.5: 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.

3.3.2 Radix-4 Booth's Algorithm

The booth encoding algorithm is a multiplier 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 $[Y(i+1), Y(i), Y(i-1)]$ 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 3.6.

- Separately: x_{i-1}, 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.

$$\bullet \quad \bullet \quad \bullet \quad x_7 \quad x_6 \quad x_5 \quad x_4 \quad x_3 \quad x_2 \quad x_1 \quad x_0 \quad (x_{-1})$$

$$y_7\, y_6 \qquad y_5\, y_4 \qquad y_3\, y_2 \qquad y_1\, y_0$$

Figure 3.6: Recoding in Radix 4

In the modified Booth's algorithm (radix-4 recoding) a zero is appended to the right of x_0 (multiplier LSB). Triplets are taken beginning from 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[12]. Recoding in Booth's Radix-4 is shown in table 3.2.

When this recoding scheme is applied to a parallel multiplier, the number of partial products is halved and hence the multiplication time decreases. Figure 3.7 shows the multiplication using radix-4 booth's algorithm.

Table 3.2: Recoding in Booth Radix-4 Algorithm [12]

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

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 3.7: Example of radix 4 Algorithm

3.3.3 Radix-8 Booth's Algorithm

In radix-8, overlapping groups of 4 bits each are taken at a time. Only n/3 partial products are generated as compared to n/2 in case of radix-4 where n is the number of bits of multiplier. In this, for a particular quartet of multiplier the operation involves generation of 3A as a partial product which increases the complexity. For example: recoding 010(1) yields $y_i y_{i-1} y_{i-2} = 011$. Technique for simplifying generation and accumulation of ±3A exists. The partial products multiplexer must choose one out of nine possibilities depending on the value of the corresponding signed-digit, as shown in figure 3.9.

Table 3.3: Recoding in Booth Radix-8 Algorithm [12]

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

Here we have an operation which is an odd multiple of the multiplicand i.e 3Y, which is not immediately available. We need to perform an additional add operation: 2Y+Y=3Y to generate it.

Example for Radix 8 is shown in figure 3.8:

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 3.8: Example of Radix 8 Algorithm

Radix-8 recoding applies the same algorithm as radix-4, but now quartets of bits are taken 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[12]. Each quartet is codified as a signed-digit using the table 3.3.

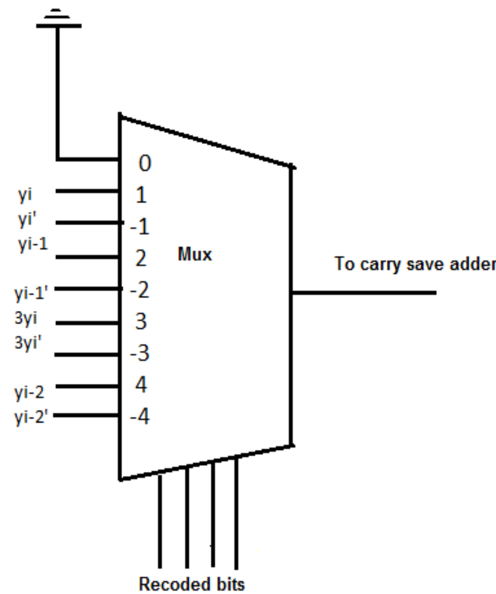


Fig 3.9: Partial products multiplexer

3.3.4 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. 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}
- With increase in radix-r, both the gate level and the delay performances will increase due to the complexity of the partial products encoded.
- The comparison of the 32-bit Radix-based Booth Encoding variants indicates that the Radix-4 Booth Encoding multiplier is the best multiplier in terms of high-speed applications and low area constraint[13].
- The other higher Radix-based Booth Encoding designs such as Radix-8, Radix-16 and Radix-32 shows a delay smaller than the Radix-2 but the delay increased when compared with the Radix-4 design[13].

3.4 Partial Product Reduction

Efficient implementation of a digital multiplier is dependent on the method used for the addition of partial product array bits. Since a delay proportional to the width of the multiplicand is given by each shifted version 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.

In the carry-save adder, carry propagation is avoided 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 $A(n-1)...A(0), B(n-1)...B(0), C_{in}(n-1)..C_{in}(0)$, generates two n-bit result values, $Sum(n-1)..Sum(0)$ and $C_{out}(n-1)..C_{out}(0)$ [n]. The most important application of a carry-save adder is to reduce the number of partial products in integer multiplication. Figure 3.10 presents 3: 2 carry save adder in dot notation. In half adder, only two dots are combined to form a sum-bit and a carry-bit.

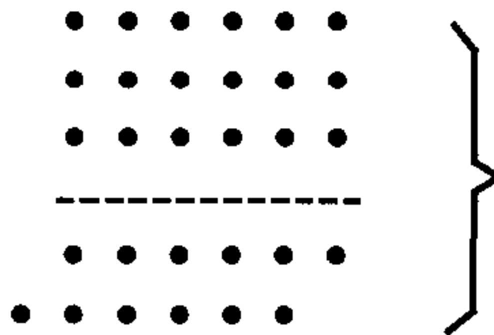


Figure 3.10: CSA function in dot notation[12]

3.4.1 3:2 Compressor

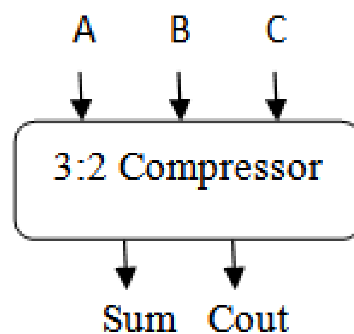


Fig. 3.11: 3:2 compressor I/O Diagram[15]

This is the basic compressor which is similar to full adder. It takes three input bits A, B and C_{in} of rank 0 (bit position 0) and produces sum of rank 0 and C_{out} of rank 1(bit position 1) as shown in fig. 3.11[16].

3.4.2 4:2 Compressor

In the architecture of 4:2 compressor, two 3:2 compressors are used in series as shown in fig. 3.12. 4:2 compressors are also used as carry save adders. The higher order compressors such as 4:2 and 5:2 compressors have been widely employed in the high speed multipliers in order 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. In the 4:2 compressor structure, five partial products bits are actually compressed 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 neighbouring 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. The output C_{out}, being independent of the input C_{in} accelerates the carry save summation of the partial products[16].

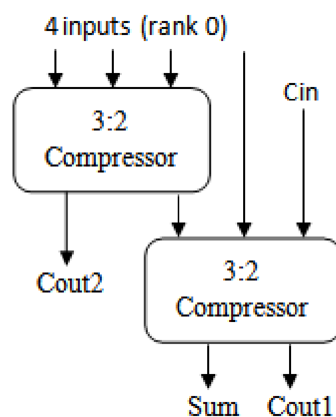


Fig. 3.12: 4:2 Compressor Architecture[15]

3.4.3. 5:2 Compressor

The 5:2 compressor takes 5 input bits and produces two output bits i.e Sum and C_{out1}. It also has carry-in bits (C_{in1} and C_{in2}) and carry-out bits (C_{out1},C_{out2} and C_{out3}). Thus the total number of input/output bits are 7/4[16]. All the input bits including C_{in1} and C_{in2} have rank 0

and the output bits C_{out1}, C_{out2} and C_{out3} have rank 1 and Sum bit has rank 0 as shown in fig. 3.13.

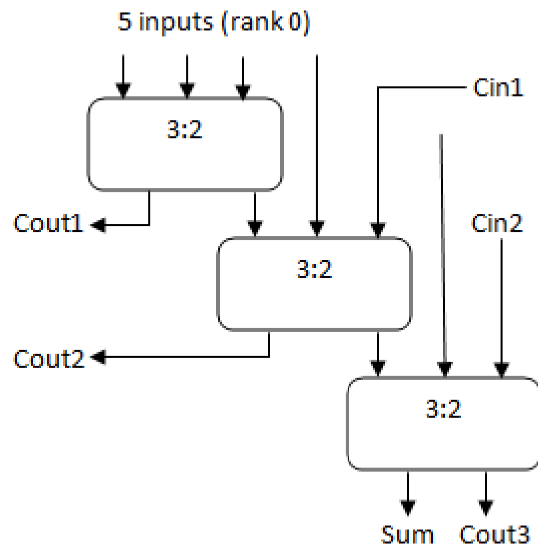


Fig. 3.13: 5:2 Compressor Architecture[15]

3.4.4 7:2 Compressor

The 7:2 compressor takes 7 input bits and produces 2 output bits (Sum and C_{out1}), it also has carry-in input bits (C_{in1}, C_{in2}) and carry-out (C_{out1}, C_{out2}) bits. Thus, the total numbers of input/output bits are 9 and 4[16]. All input bits, including C_{in1}, have rank 0 and C_{in2} has rank 1, the two output bits have ranks 0 and 1 respectively, while C_{out1} has rank 1 and C_{out2} has rank 2 as shown in fig. 3.14.

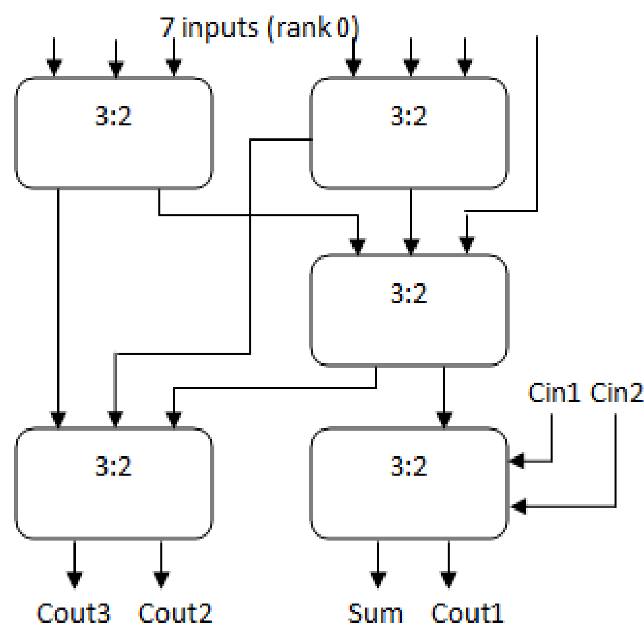


Fig. 3.14: 7:2 Compressor Architecture[15]

3.5 Final Stage Adder

The rise in the popularity of portable systems as well as the rapid growth of the circuit 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 and this largely determines the overall delay of the multiplier. So in this stage fast carry propagate adders like Ripple Carry Adder can be used as per our requirement and is discussed as under.

3.5.1 Ripple Carry Adder

It is used to obtain the final sum and the output carry by adding the final two rows obtained 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.

As shown in Fig. 3.15 a ripple carry adder is a chain of cascaded full adders and one half adder; each full adder has three inputs (A, B, C_{in}) and two outputs (S, C_{out}). The carry out (C_{out}) of each adder is fed to the next full adder[17].

The latency of k-bit ripple-carry adder can be derived by considering the worst-case signal propagation path.

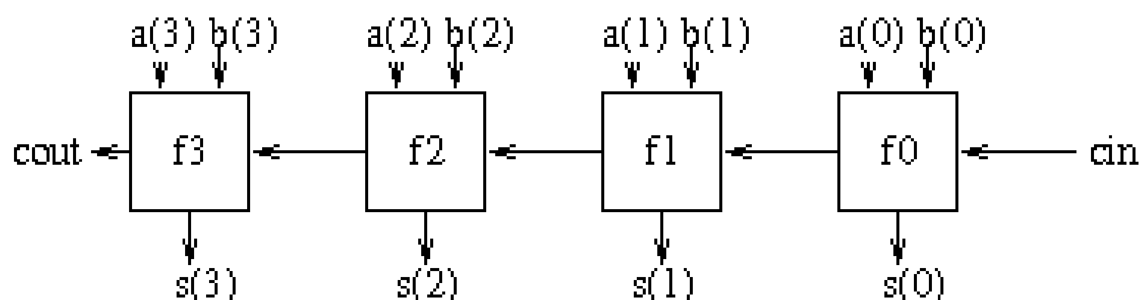


Figure 3.15: Ripple Carry Adder[17]

As shown in Figure 3.16 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.

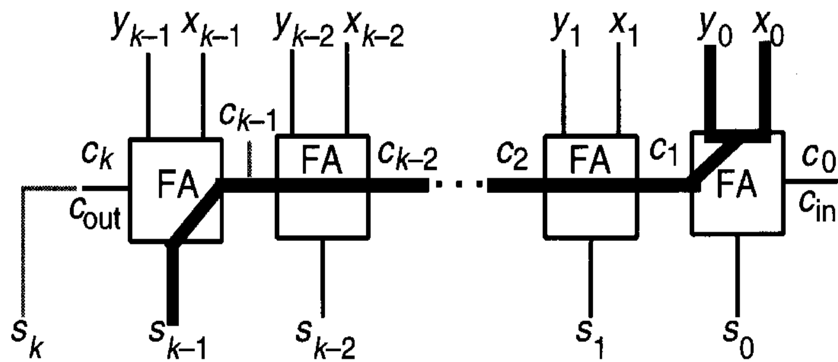


Figure 3.16: Critical paths in a k-bit RCA[9]

3.5.2 Carry Look Ahead Adder

Carry Look Ahead Adder can produce carries faster due to carry bits generated in parallel by an additional circuitry whenever inputs change. This technique uses carry by pass logic to speed up the carry propagation. Example of CLA logic is shown in figure 3.17

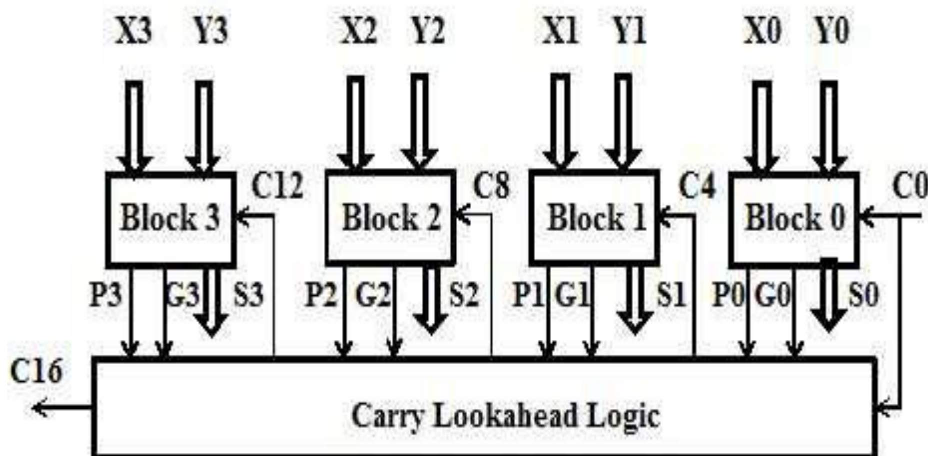


Figure 3.17: 4-BIT CLA Logic equations

Let a_i and b_i be the augends and addend inputs, c_i the carry input, s_i and c_{i+1}, the sum and carry-out to the i-th bit position. If the auxiliary functions, p_i and g_i called propagate and generate signals, the sum output respectively is defined as follows.

$p_i = a_i + b_i$ and $g_i = a_i.b_i$

$s_i = a_i xor\ b_i xor c_i$ and $c_{i+1} = g_i + p_i.c_i$

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

3.6 Tree Multipliers

The two well known fast multipliers are those presented by Wallace and Dadda. Both consists of three stages. In the first stage, partial product matrix is formed. In the second stage, partial product matrix is reduced to a height of two. In the final stage, these two rows are combined using an adder. In the Wallace method, the partial products are reduced as soon as possible. In contrast, Dadda's method does the minimum reduction possible at each level to perform the reduction in the same number of levels as required by a Wallace multiplier[9].

3.6.1 Dadda Tree Algorithm

Dadda multiplier uses a minimal number of 3:2 and 2:2 compressors at each level during the compression to achieve the required reduction. The reduction procedure for Dadda compression trees is given by the following recursive algorithm.

1. Let d_1=2 and d_{j+1}=[1.5. d_j] where d_j is the height of the matrix for the jth stage. Repeat until the largest j^{th} stage is reached in which the original N height matrix contains at least one column which has more than d_j dots.
2. In the jth stage from the end, place 3:2 and 2:2 compressors as required to achieve a reduced matrix. Only columns with more than d_j dots as they receive carries from less significant 3:2 and 2:2 compressors are reduced.
3. Let j=j-1 and repeat step 2 until a height of two is generated. This should occur when j=1[10].

The number of 3:2 and 2:2 compressors required for a Dadda multiplier depends on N, the number of bits of the operands and is determined as follows:

(3,2) counters= $N^2 - 4N + 3$ …... (1)

(2,2) counters= $N - 1$ ...… (2)

CPA length= $2.N + 2$ …... (3)

Dadda multipliers require fewer 3:2 and 2:2 compressors during the compression stage than do the corresponding Wallace multipliers. Once the matrix has been reduced to a height of two, the final stage consists of using a carry propagating adder to produce the final product. The size of the final carry propagating adder is determined as follows

The dot diagram shown in Fig. 3.18 shows this algorithm implemented for an 8×8 bit multiplier. Four reduction levels are required with matrix heights of 6, 4, 3 and 2. Two dots joined by a diagonal line indicate that these dots are the outputs from a 3:2 compressor. Similarly two dots joined by a crossed diagonal indicate that these dots are the outputs from a

2:2 compressor. 64 AND gates, 35 3:2 compressors, 7 2:2 compressors and a 14-bit carry propagating adder are required to form the 16-bit product.

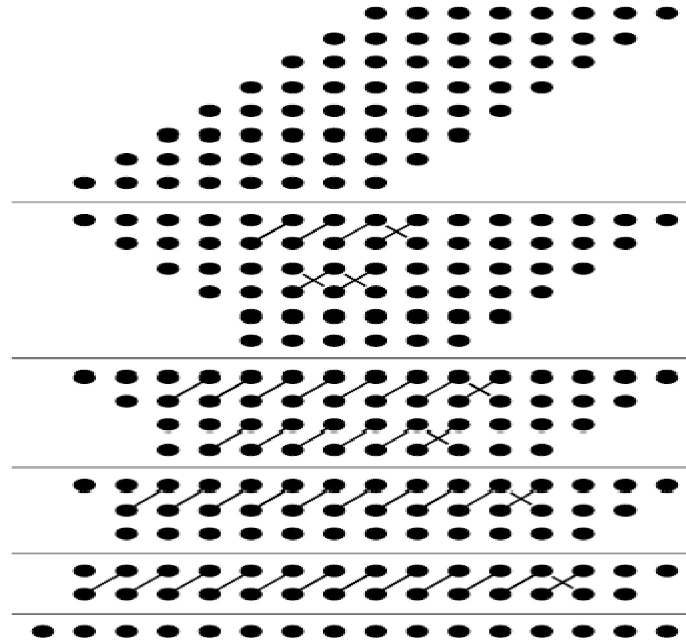


Fig. 3.18: Dot Diagram for an 8 by 8 Dadda Multiplier[10]

3.6.2 Wallace Tree Algorithm

For Wallace multipliers, the partial products are formed by n^2 AND gates in the same manner as for Dadda multipliers. Next the N rows of partial products are grouped together in sets of three rows each. Any additional rows that are not a member of a group of three are transferred to the next level without modification. In Wallace tree architecture, all the bits of all of the partial products in each column are added together by a set of compressors in parallel without propagating any carries[11].Within each group of three rows, 3:2 compressors are applied to the columns containing two bits. Columns containing only a single bit are transferred to the next level unchanged. The height of the matrix in the jth reduction stage, w_j is given by the following recursive equations

$W_0 = N$ .... (4)

$W_{j+1} = 2.[w_j/3] + w_j \bmod 3$ .... (5)

As for the Dadda multipliers, when the matrix has been reduced to a level with a height of two, a final adder is used to perform the final addition which gives the product of the multiplication. Wallace and Dadda multipliers each require the same number of levels to perform the reduction to a level with a height of two, however, the heights of different levels can vary between the two methodologies. Although Wallace and Dadda multipliers contain nearly identical numbers of full adders, more of the Wallace full adders are applied during the

reduction of the matrix. This and the additional half adders used in a Wallace reduction result in the shorter final carry propagating adder.

A dot diagram for an 8 by 8 Wallace multiplier is shown in Fig. 3.19. Four reduction stages are required with matrix heights 6, 4, 3 and 2. 64 AND gates, 1 OR gate, 38 3:2 compressors, 15 2:2 compressors, and a 10-bit carry propagating adder are required to form the 16-bit product[10].

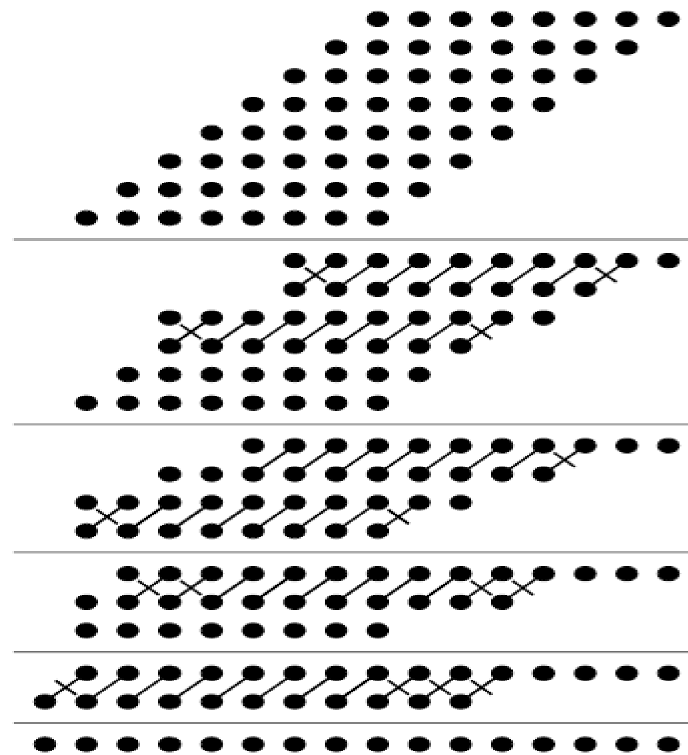


Fig. 3.19: Dot Diagram for an 8 by 8 Wallace Multiplier[10]

The number of 3:2 compressors and the size of the final carry propagating adder required for a Wallace multiplier depends on N, the number of bits of the operands, and S, the number of stages in the reduction, and can be determined using equation (1), (2) and (3).

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 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).

4.3 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.

4.3.1 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].

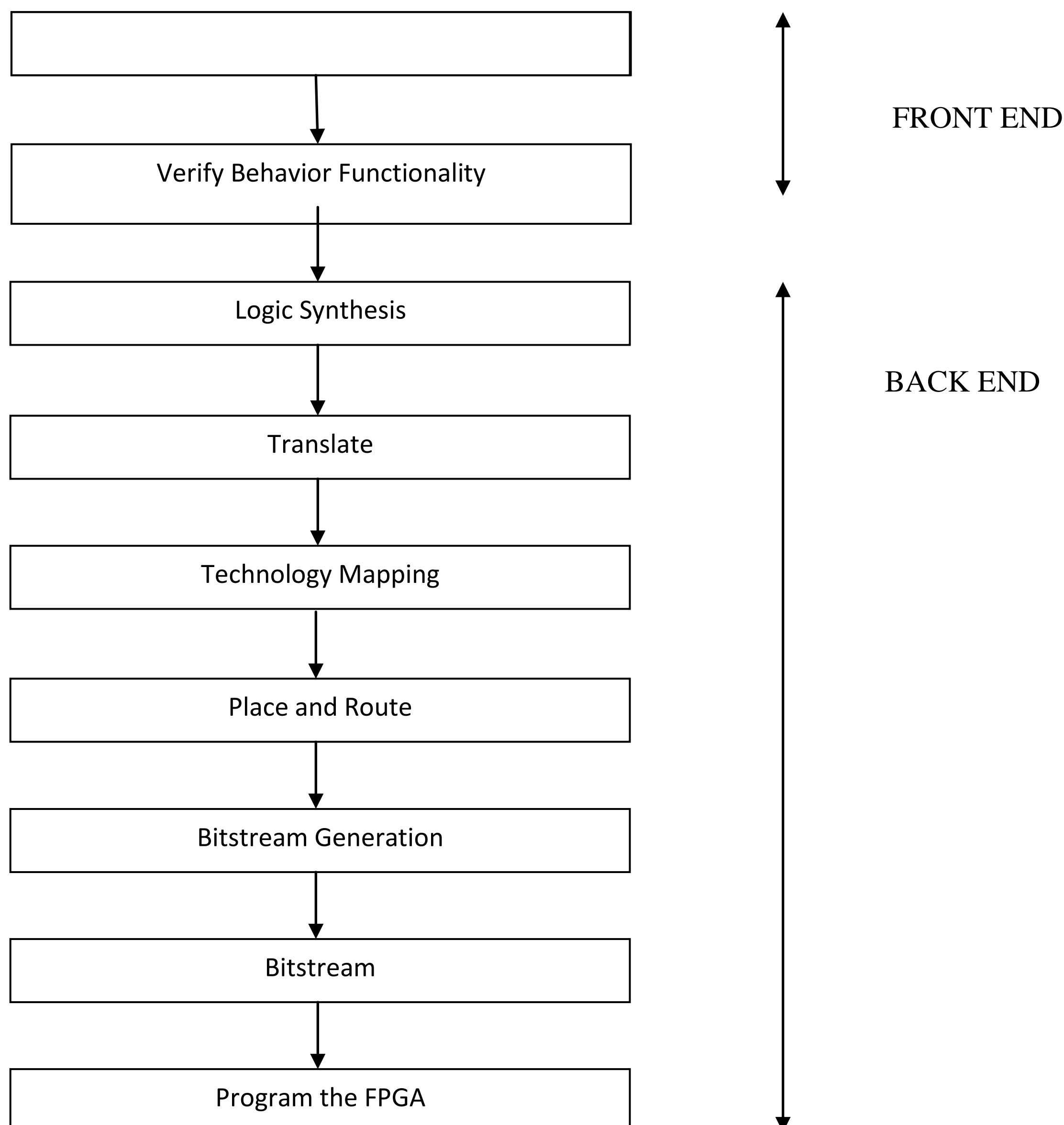


Figure 4.1: FPGA Design Flow

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 netlist 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. Bit stream 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 DYNAMIC RANGE DETECTION BASED PARALLEL MULTIPLIER

This chapter discusses the implementation results of DRD based Multiplier synthesized and implemented on Xilinx ISE 14.5.

The Working Environment for the design is:

- Target Device : XC3S500E-4FG320
- Tool Version : ISE Design suite 14.5
- Optimization Goal : Speed
- Design Strategy : Balanced
- Total Slices : 4656
- Total LUTs : 9312

5.1 Design of multipliers

Here, DRD based 8-bit multiplier and 16-bit multipliers have been coded in Verilog HDL and simulated and synthesized using Xilinx ISE design suite 14.5. Radix-4 Booth's recoding algorithm is used to generate the partial products as discussed in section 3.3.2. Number of partial product rows gets reduced to half with the use of radix-4 Booth's algorithm. Accumulation of these partial product rows is done using different compressors and also using tree compressions techniques. Last step in multiplication is the accumulation of remaining two rows which is done with the help of ripple carry adder. The various multipliers that have been designed and implemented are listed in Table 5.1. An example to demonstrate the multiplication of two 8 bit numbers is shown in Fig. 5.1.

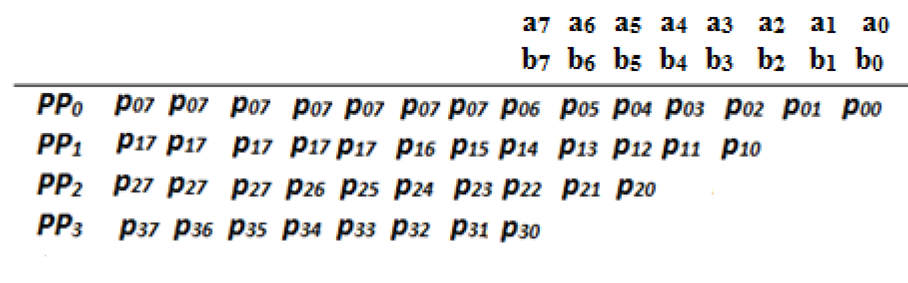


Fig 5.1: Multiplication of two 8 bit numbers

Table 5.1: Various multipliers designed and implemented

Multiplier	Compressor
8×8 bit with DRD	3:2
8×8 bit with DRD	4:2
8×8 bit without DRD	3:2
16×16 bit with and without DRD	3:2
16×16 bit with and without DRD	4:2
16×16 bit with and without DRD	5:2
16×16 bit with and without DRD	7:2
4×4 bit Wallace tree and Dadda tree multiplier	3:2 and 2:2
8×8 bit Wallace tree and Dadda tree multiplier	3:2 and 2:2

5.2 Results and Discussions

5.2.1 Simulation Results

Simulation results of 16×16 bit multiplier with DRD is shown in fig. Operands taken for Multiplication are Multiplicand(A)=1111111111111111(-1) and Multiplier(B)= 0000000000000011(3) and the resultant product is 11111111111111111111111111111101(-3).

Since operand A has more number of consecutive 1s, switching logic signal SW_{LL} will be 1. So operands will be interchanged and B will be taken as multiplier and used for booth recoding. As MSB of product is 1, the actual result is 2's complement of Product, hence the actual result is 00000000000000000000000000000011 i.e. -3. Fig. 5.3 and 5.4 shows the simulation waveforms of 8×8 bit Wallace tree and Dadda tree multiplier.

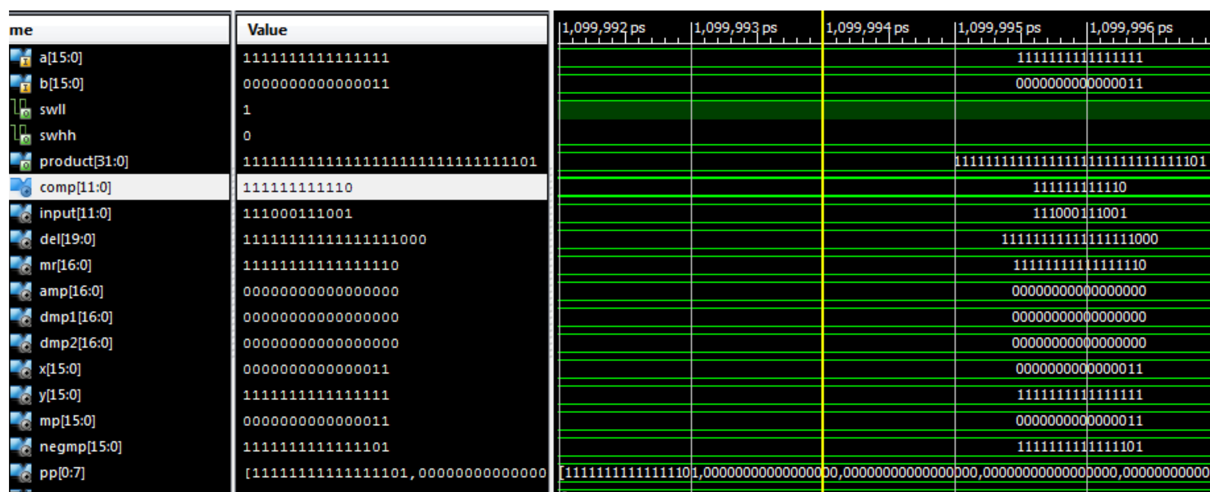


Fig. 5.2: Simulation Waveforms of 16×16 bit multiplier with DRD

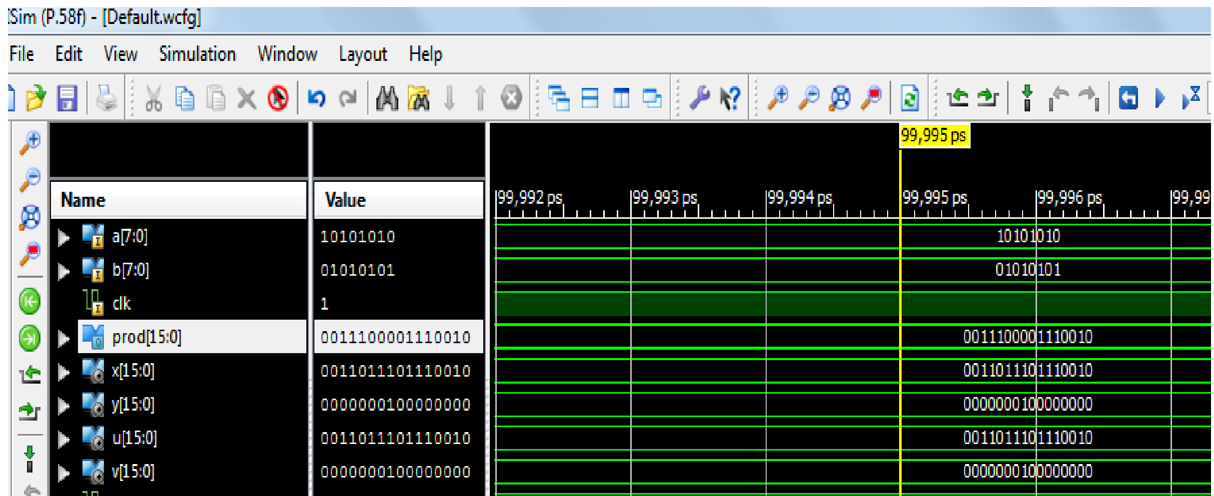


Fig. 5.3: Simulation waveforms of 8×8 bit Wallace tree multiplier

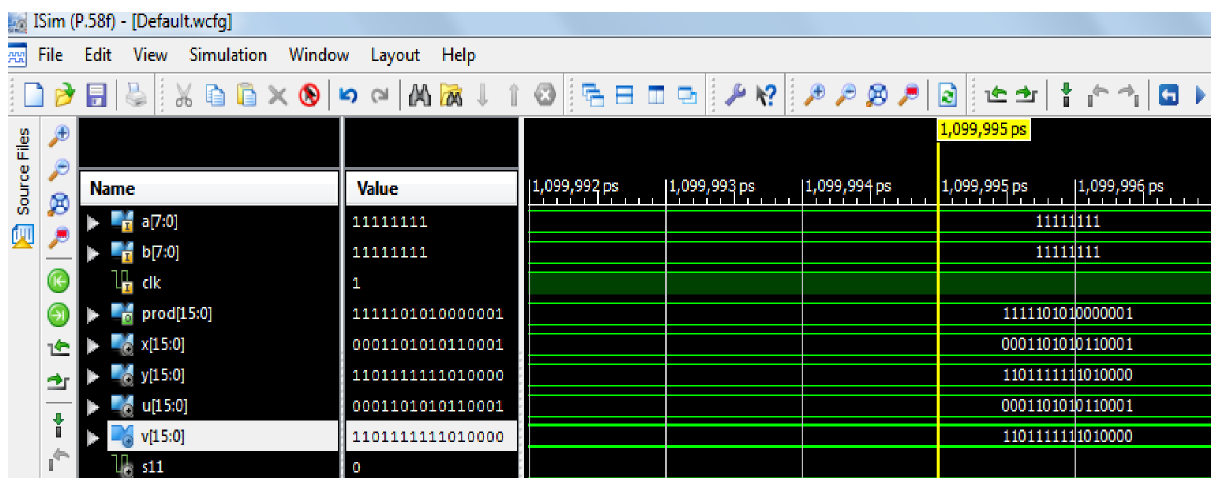


Fig. 5.4: Simulation waveforms of 8×8 bit Dadda tree multiplier

5.2.2 Synthesis Results

The synthesis results of various multiplier (with and without DRD unit) and applying compressors of different orders at the accumulation stage are shown in Table 5.2 and Table 5. 3. Power comparison of multipliers with DRD and without DRD is shown in Table 5.4.

Table 5.2: Delay and Area comparison of multipliers without using DRD

Multiplier (no DRD)	Compressor	No. of slices	Delay(ns)
8×8 bit	3:2	102/4656	4.365
16×16 bit	3:2	337/4656	5.673
16×16 bit	4:2	340/4656	5.516
16×16 bit	5:2	359/4656	5.326
16×16 bit	7:2	377/4656	5.032

Table 5.3: Area and delay comparison of multiplier designs without using DRD

Multiplier (DRD)	Compressor	No. of slices	Delay(ns)
8×8 bit	3:2	115/4656	4.876
8×8 bit	4:2	119/4656	4.778
16×16 bit	3:2	337/4656	6.514
16×16 bit	4:2	340/4656	6.216
16×16 bit	5:2	359/4656	6.117
16×16 bit	7:2	377/4656	5.089

Table 5.4: Power consumption of multiplier designs with and without using DRD

Power (mW)	Compressor	DRD	No DRD
16×16 bit	3:2	0.119	0.323
16×16 bit	4:2	0.107	0.312
16×16 bit	3:2	0.101	0.307
16×16 bit	4:2	0.996	0.299

Table 5.5: Area and Delay comparison of Wallace and Dadda multiplier

Multiplier	No. of slices	No. of 4 input LUTs	Delay (ns)
4×4 bit Wallace	20	28	4.014
4×4 bit Dadda	18	19	3.512
8×8 bit Wallace	103	175	6.663
8×8 bit Dadda	75	127	5.033

Table 5.5 shows the comparison of Wallace and Dadda tree multipliers in terms of area and delay. The multipliers taken are of 4 and 8 bit.

CHAPTER 6

CONCLUSION AND FUTURE SCOPE

6.1 Conclusion

In this thesis, multiplier is implemented using different compressors in the accumulation stage. Fig. 6.1 shows that the synthesis results of 16×16 bit multipliers using different compressors. From this graph, it has been concluded that with increase in the order of compressor, speed increases and power consumption decreases.

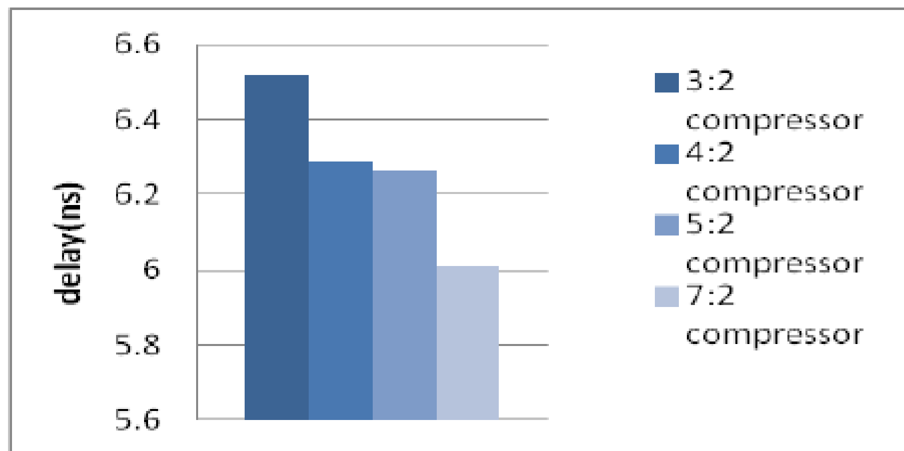


Fig. 6.1: Changes in Delay(ns) with increase in order of compressor(16×16 bit)

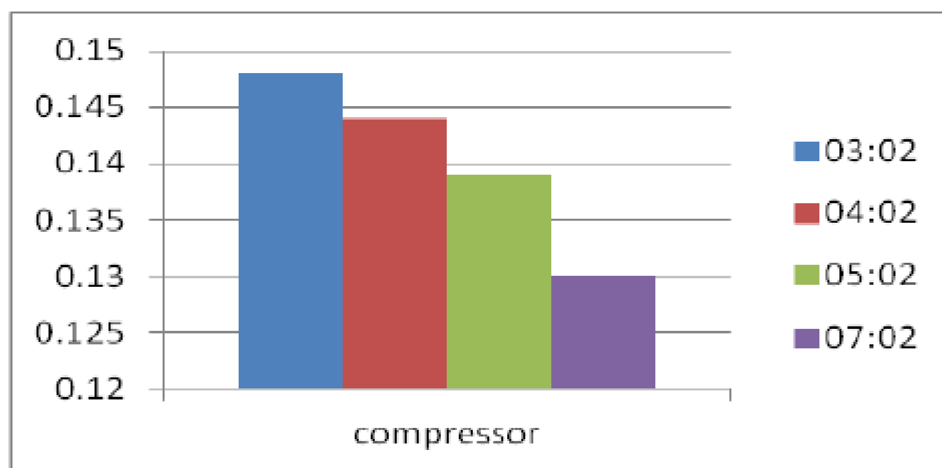


Fig. 6.2: Changes in Power(mW) with increase in order of compressor(16×16 bit)

From fig. 6.3 and fig. 6.4 it is clear that by using DRD power consumption of the multiplier reduces but at the the cost of speed.

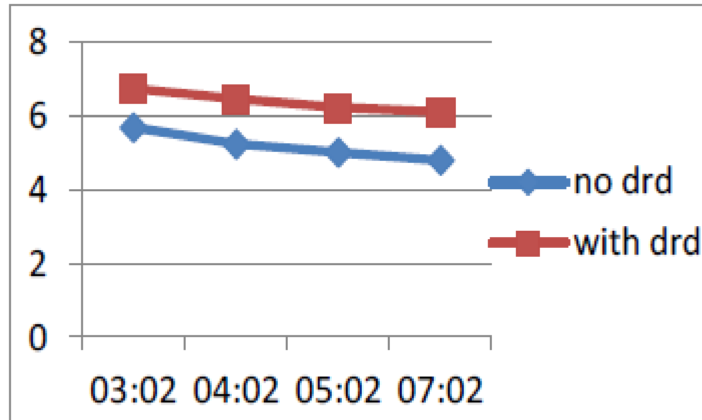


Fig. 6.3: Delay(ns) comparison of 16bit multiplier with and without DRD

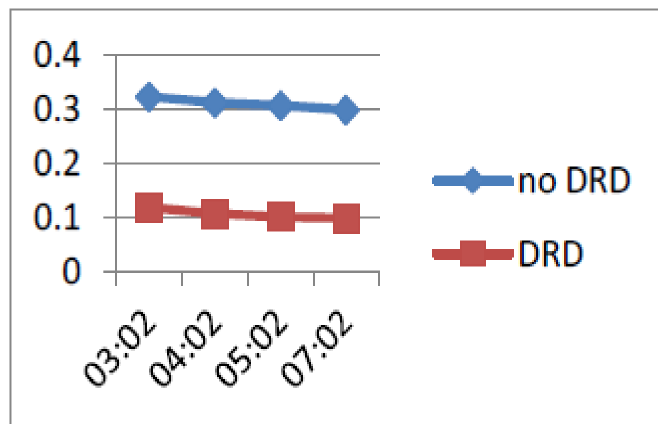


Fig.6.4: Power(mW) comparison of 16bit multiplier with and without DRD

Wallace tree and Dadda tree algorithms have been applied both on 4×4 bit and 8×8 bit multipliers. It is observed that Dadda algorithm is faster than Wallace. Also the number of slices occupied by Dadda is less as compared to Wallace. Fig. 6.5 and fig. 6.6 shows the comparison of Wallace and Dadda algorithms in terms of delay and area respectively.

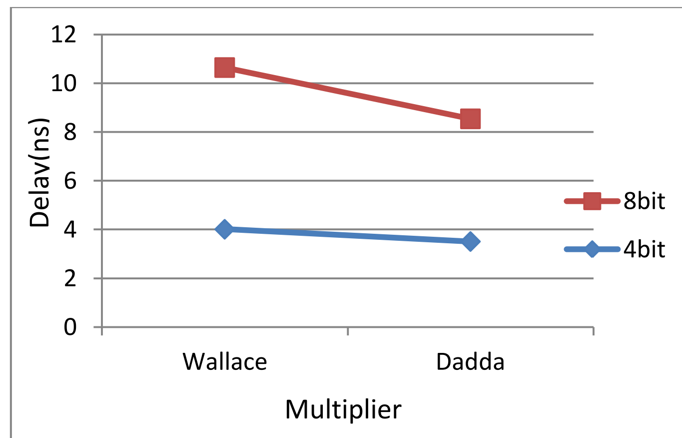


Fig. 6.5: Delay comparison of Wallace and Dadda multiplier

6.2 Future scope

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

1. Truncation of output product can be done to decrease the power consumption further.
2. An option of choosing 8-bit or 16-bit multiplication can be introduced. If first 8 bits of 16 bit operands are zero, then only 8 bit multiplication will be performed which will reduce power consumption, delay and area.
3. Higher order radix algorithm can be implemented for further reducing partial products in order to enhance the speed of the multiplier.
4. In order to enhance the performance higher order compressors like 9:2 and 11:2 can be used to accumulate the partial products.
5. In place of ripple carry adder, other adders such as carry select adder and carry look ahead adder can be used to increase the performance.

References

[1]. J. M. Rudagi, V. Amblr, V. Munavalli, R. Patil, V. Sajjan, "Design and implementation of efficient multiplier using Vedic Mathematics," in *proc. IEEE International Conference on Advances in Recent Technologies in Communication and Computing*, November 2011, pp. 162-166.

[2]. T. Arunachalam, S. Kirubaveni, "Analysis of High Speed Multipliers," in *proc. IEEE International Conference on Communication and Signal Processing*, April 2013, pp. 211-214.

[3]. A. S. Swee, V. Elakya, "Design of modified low power Booth multiplier," in *proc. IEEE International Conference on Computing, Communication and Applications*, June 2011, pp. 854-859.

[4]. K. S. Yeo, K. Roy, Low Voltage, Low Power VLSI, Tata Mcgraw-Hill Education, 2nd edition.

[5]. J. Sharma, S. Kumar, "Digital Multipliers-A Review," *International Journal of Engineering and Management (IJEMR)*, June 2014, Vol. 4, No. 3.

[6]. Y. Mareswara Rao, A. Madhusudan, "Radix4 Configurable Booth Multiplier for Low Power and High Speed Applications" *IOSR Journal of Electrical and Electronics Engineering (IOSR-JEEE)*, November 2012, Vol. 4, No. 2.

[7]. D. Garg, S. Arya, "Design of Configurable Booth Multiplier Using Dynamic Range Detector," *International Journal of Electronics Engineering(IJEER)*, March 2012, Vol. 4, No. 3.

[8]. S. R. Kuang, J. P. Wang, "Design of power efficient configurable booth multiplier", *IEEE International Transaction on Circuits and Systems,* March 2010, Vol. 57, No. 3.

[9]. J. W. Townsend, A. J. Abraham, E. Swartzlander, "A comparison of Wallace and dadda multiplier delays," *International Society for Optical Engineering (SPIE)*, December 2003, Vol. 52, No. 5.

[10]. C. Y. H. Lee, L. H. Hiung, S. W. F. Lee, N. H. Hamid, “A performance comparison study on multiplier designs”, in *proc. IEEE International Conference on Embedded Systems,* June 2010, pp. 1 – 6.

[11]. S. Shah, A. J. Khabb, D. A. Khabb, “Comparison of 32-bit Multipliers for Various Performance Measures,” in *proc. IEEE International Conference on Microelectronics*, November 2000, pp. 75-80.

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

[13]. K. L. S. Swee, L. H. Hiung, “Performance Comparison Review of 32 bit Multiplier Designs”, in *proc. IEEE International Conference on Intelligent and Advanced Systems*, June 2011, pp. 854-859.

[14]. G.J. Lakshmi Devi, M. Ramesh Kumar, “Implementation of Energy Efficient and Low Complex Fir Filters with Reconfigurability Usage,” *International Journal of Applied Research & Studies (IJARS)*, June 2012, Vol. 1, No.1.

[15]. P. Shukla, N.K. Gahlan, J. Kaur, “Techniques on FPGA Implementation of 8-bit Multipliers,” *International Journal of Computer Science And Technology (IJCST)*, June 2012, Vol. 3 , No. 2.

[16]. J. Kaur, N.K. Gahlan, P. Shukla, “Delay Power Performance Comparison of Array Multiplier in VLSI Design,” *International Journal of Advanced Research in Computer Science and Electronics Engineering,* May 2012, Vol. 1, No.3.

[17]. Y. H. Seo, D. W. Kim, “A New VLSI Architecture of Parallel Multiplier–Accumulator Based on Radix-2 Modified Booth Algorithm,” *IEEE Transactions on Very Large Scale Integration Systems*, February 2010, Vol. 18, No. 2.

[18]. S. Veeramachaneni, “Novel Architectures for High-Speed and Low-Power 3-2, 4-2 and 5-2 Compressors,” in *proc. IEEE International Conference on VLSI Design and Embedded Systems,* January 2007, pp. 1063-9667

[19]. V. Elakya, A. S. Prabhu, "Design of modified low power booth multiplier," in *proc. IEEE International Conference on Computing, Communication and Applications*, February 2012, pp. 1-6.

[20]. http://www.xilinx.com/support/documentation/boards_and_kits/ug230.pdf

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

[22]. M. Morris Mano, "Computer System Architecture," Pearson Education India, 3rd edition.