FPGA IMPLEMENTATION OF BINARY INTEGER DECIMAL BASED FLOATING POINT MULTIPLIER

A Dissertation Submitted In Partial Fulfillment of the Required for the Degree of

MASTER OF TECHNOLOGY

In

VLSI Design

Submitted By

PRIYANKA RANI

Roll no. 601361019

Under the guidance of

Ms. Sakshi Bajaj

Assistant Professor, ECED

T.U. Patiala

Department of Electronics and Communication Engineering

THAPAR UNIVERSITY

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

PATIALA 147004 (PUNJAB)

July, 2015

DECLARATION

I hereby declare that the work which is being presented in dissertation titled **"FPGA IMPLEMENTATION OF BINARY INTEGER DECIMAL BASED FLOATING POINT 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 study carried out as under the guidance of **Ms. Sakshi Bajaj,** Assistant Professor, ECED during 2013-2015.

Priyanka

PRIYANKA RANI
Roll no.601361019

Date: 10-July-2015

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

Sakshi

(Sakshi Bajaj)
Assistant Professor, ECED,
Thapar University,
Patiala 147004, (Punjab)

Countersigned by:

(Dr. Sanjay Sharma)
Head
ECED, Thapar University,
Patiala, 147004

Dean of Academics Affairs
ECED, Thapar University,
Patiala, 147004

ACKNOWLEDGEMENT

To discover, analyze and to present something new is to venture on an untrodden path towards and unexplored destination is an arduous adventure unless one gets a true torch bearer to show the way. I would have never succeeded in completing my task without the cooperation, encouragement and help provided to me by various people. Words ar e 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 Bajaj, 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. Amit Kumar Kohli, 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.

(PRIYANKA RANI)

ABSTRACT

Decimal floating point (DFP) arithmetic is important in various applications such as currency conversion, billing, insurance, banking etc, as it is able to produce precise decimal fractions and minimize manual calculations that perform decimal rounding. But binary floating point arithmetic fails to provide correct decimal rounding and exact decimal fractions such as 0.10,0.0418. A multiplier is one of the main component in most digital and high performance systems such as digital signal processors, microprocessors and FIR filters etc. As technology advances, many scientists have tried and are trying to design multipliers which provide either of the following- high speed, low power consumption and hence less area or even combination of them in multiplier.

Multiplication is also an important operation in decimal application. This thesis aims to implement Binary Integer Decimal based floating point multiplier using the fastest adder. The maximum time of multiplication is consumed in accumulating the partial products and at the final stage of addition to get the significant product. So, the multiplication time can be reduced if the partial products are accumulated with the help of fast adders and tree multipliers. In this, the binary encoding for DFP numbers also known as BID format has also been implemented, where binary radix of 2 is used. Since BID encoding stores the significant as an unsigned binary integer for effective reuse of existing binary hardware. In this thesis, a hardware design is presented that multiplies BID encoded DFP numbers. An optimized technique that in parallel with significant multiplication is used to detect if rounding is needed and to find the number of product digits that are needs to be rounded. In this, both for significant multiplication and rounding, a single binary hardware along with carry save feedback is used. To design a BID multiplier, the partial products are generated using radix-8 algorithm, then Dadda and Wallace tree structures are used to accumulate partial products and different adders like ripple carry adder, carry select adder or carry look ahead adders are used at final stage to obtain the result. Then multiplier is synthesized and simulated using Xillinx ISE 14.5 targeting Spartan 6 FPGA device. Then their results in the terms of area and delay are compared for BID multiplier using different adders.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABBRIVEATIONS

FPGA	Field programmable gate array
ASIC	Application specific integrated circuit
DFP	Decimal Floating Point
BFP	Binary Floating Point
DSP	Digital signal processor
CPU	Central processing unit
IEEE	Institute of Electrical and Electronics Engineers
BID	Binary Integer Decimal Numbers
HDL	Hardware Description language
ISE	Integrated Software Environment
LSB	Least significant digit
MSB	Most significant digit
LUT	Look up table
RTL	Register transfer level
VLSI	Very large scale integration
RAM	Random access memory
HA	Half Adder
FA	Full Adder
PP	Partial Product
RCA	Ripple Carry Adder
CLA	Carry Look Ahead Adder
CSA	Carry Save Adder
XST	Xilinx synthesis technology

CHAPTER

INTRODUCTION

1.1 Introduction

Multipliers play a key role in many high performance systems. As the technology advances, the demand of high- speed digital systems increases or we can say the demand of high speed multiplier increases because the multiplier is a main component in every digital system. Multipliers are used as small blocks in large digital systems like FIR filters, microprocessors, digital signal processors, communication systems etc. So to find performance of a large digital system is measured by the performance of the multiplier because the multiplier is mainly the slowest element in the large systems; consequently multiplication dominates the execution time of most DSP algorithms, so a high speed multiplier is always needed. In comparison to other operations in an ALU, the multiplier also uses more power. Thus scientists have always been trying to design a multiplier which utilizes less power, time and area and increases speed. Multiplication is such a mathematical operation in which a number i.e. multiplicand is added to itself a given number of times as indicated by another number i.e. multiplier to form a result.

A digital circuit used to multiply two numbers is called a multiplier. As in mathematics, multiplication includes shifting and adding the partial products. The same approach is used here in digital multipliers. For unsigned number multiplication, AND gates are utilized for generating the partial products and full adders are utilized for adding those generated partial products. To multiply signed numbers, the negative numbers should be first converted into its 2's complement representation to make all the partial products positive. Digital multipliers can be classified into three type's i.e. parallel, serial and serial-parallel multiplier. Both the inputs are entered serially in a serial multiplier. Such an implementation will led to lesser area and lesser hardware cost and also lower power consumption. But the major drawback is its poor speed, because the inputs are entered serially. Now the speed can be increased by using

parallel multiplier implementation because operation is carried out in parallel. But it is more complex as compared to serial multipliers as it occupies larger area. Also its power consumption is higher. Parallel multipliers further can be classified into two types i.e. array multipliers and tree multipliers. Booth multiplier, Robertson multiplier, Braun multiplier, Baugh-Wooley multiplier etc. are most widely used array multipliers. For carrying out fast multiplication Tree multipliers are used like Dadda tree multiplier, Wallace tree multiplier etc. To take benefit of small area of serial multiplier and high speed operation of parallel multiplier, serial-parallel multipliers are used. In serial-parallel multipliers, one operand is entered serially while other is stored in parallel. This requires less area and also enhances the speed of the multiplier. The numbers can be represented in two ways fixed point and floating point representations. The representation in which digits after and before the decimal is fixed is called fixed point number representation. But the decimal point can float in floating point representation, hence given the name floating point. To represent extremely large and small values like distance between sun and earth or mass of electron, floating point numbers are used. To achieve better accuracy and larger range floating point representation is used although it is slower than fixed point representation.

Decimal floating point (DFP) number systems can be utilized to represent a wide variety of decimal numbers and do manual calculations that perform decimal rounding. However advanced PC's perform binary arithmetic, which have imperfections to represent and round decimal numbers i.e. it can neither give correct decimal rounding nor precise represent many decimal fractions. Due to which, errors from Binary floating point (BFP) arithmetic can combine to form a yearly billing error of over $6 million for a vast billing system. So a variety of business applications that can't endure errors because of BFP arithmetic have traditionally utilized software to perform DFP calculations. As the interest for decimal floating point arithmetic is growing, the IEEE P754 Draft Standard for floating point arithmetic incorporates details for decimal floating point arithmetic. For decimal floating point numbers, two encodings are specified by IEEE P754 Draft. One decimal floating point (DFP) encoding is specified in IEEE P754. It represents its mantissa as a binary integer and is called as binary integer decimal encoding. This encoding reuses the hardware of existing high speed binary arithmetic circuits. Second, densely packed decimal encoding is used. But it requires more expensive hardware. A floating point multiplier

has three main units: exponent unit, sign unit, mantissa unit, which works in parallel and a normalization unit.

As scientists always tried to achieve fast speed, low power, small equipment's in any equipment. To achieve fast speed, the delay from input can be decreased. To reduce area, sizes of transistors can be reduced. And power dissipation will automatically decrease with area. But there is a trade-off between these parameters, as we know we cannot improve all of them simultaneously. To estimate performance of a system, sometimes power delay product parameter is used.

1.2 Organization of Thesis Report

The organization of thesis report is as follows:

- Chapter 2- Literature Review
- Chapter 3- This chapter discusses the IEEE P754 standard, BID encoding and decoding technique, BID based floating point multiplication and its rounding algorithm, Booth Multipliers to generate the partial products and different tree multipliers to accumulate the partial products and the adders used in floating point multiplication.
- Chapter 4- In this chapter, FPGA flow using Xilinx is discussed.
- Chapter 5- Implementation results are described.
- Chapter 6- In this chapter, conclusion and future scope of the implemented design is discussed.

CHAPTER 2

LITERATURE REVIEW

S.G. Navarro et.al [1] presented the first complete design of a Binary Integer Decimal (BID)-based Decimal Floating Point (DFP) multiplier to achieve effectively adjusted results when two IEEE 754-2008 decimal64 numbers are multiplied. This multiplier works on binary integer decimal (BID) encoded decimal floating-point (DFP) numbers. This design showed that BID multiplication can be proficiently executed in hardware rather than a software implementation. This multiplier can likewise be shared to perform BFP multiplication. This design has variable latency to take advantage of the fact that multiplication results are not often adjusted. For significant multiplication and rounding, a single binary multiplier along with carry save feedback was utilized to reduce the area and critical path delay. Optimizations have decreased the BID multiplier's critical path delay and area.

C. Tsen et.al [2] proposed a hardware design for a rounding unit for 64-bit DFP numbers that utilizes the Binary integer decimal encoding (BID). This design evaluates area, critical path delay and latency for combinational and pipelined designs. A 55-bit by 54-bit binary multiplier consumed more than 86% of the rounding unit's area , which further could be shared with a double-precision binary floating-point multiplier. This design efficiently adjusts the coefficient of an IEEE P754 BID decimal64 number. To maintain the IEEE P754 operations of subtraction, quantize, multiplication, addition, roundToIntegral and for other operand sizes, the given design can be used. This hardware design has promised in terms of area, latency and potential hardware reuse. This design can be pipelined to achieve a cycle time equivalent to 13 FO4 inverter delays. This design showed that BID rounding can be effectively achieved. The results of the presented design verified that its delay is mainly because of the delay of multiplier.

L. H. Hiung et.al [3] presented a relative performance comparison of various 32-bits multiplier designs in the Area, Speed and Auto Optimized synthesis modes in

Leonardo Spectrum. These multiplier designs used Array, Wallace, Dadda, Reduced Area and Radix-4 Booth Encoding Technique. Comparative results showed that Radix-4 Booth Encoding multiplier gives best results in all the three Optimized modes. In the Speed-Optimized mode, Wallace multiplier showed the largest area performance instead of Dadda multiplier. Array multiplier exhibits the largest time delay performance while Dadda multiplier experienced the shortest time delay in terms of speed. Though, in the case of Speed-Optimized mode, results showed that the Array multiplier exhibits the longest delay while Wallace multiplier provides the fastest in terms of speed performance.

S.G. Navarro et.al [4] presented an IEEE P754-compliant multiplier which works on those values that utilize the binary encoding of DFP numbers which is also called as the Binary Integer Decimal (BID) encoding. A single high-speed binary hardware was utilized by this multiplier that has variable latency and enhanced for the basic case in which the product did not should be rounded. In this multiplier, an optimized method was utilized in parallel with the significant multiplication that finds if rounding is required and calculates the number of product digits that should have been adjusted. In this design, to multiply the significands and round the product, a single multiplier is used.

B.J. Hickmann et.al [5] introduced a parallel decimal floating-point multiplier which is compliant with the recent draft of the IEEE P754 Standard for Floating-point Arithmetic (IEEE P754). This parallel DFP multiplier provides low latency and high throughput. This design utilized alternate decimal digit encodings to decrease area and delay. Initially this encoding approach was used by fixed point designs. Then, fixed-point design was further extended to support floating-point multiplication by adding several components including rounding, shifting, exponent generation and exception handling. Results of delay and area were presented that showed a significant throughput and latency improvement at the expense of increase in area in comparison to the merely published IEEE P754 compliant sequential floating-point multiplier. The 11-stage pipelined version of this design essentially beats a successesive IEEE P754 floating point multiplier in terms of both throughput and latency, while keeping up a comparative clock frequency.

M.A. Erle et.al [6] proposed two plans for fixed-point decimal multiplication that utilized decimal carry-save addition to decrease the critical path delay. Initial, a

multiplier that utilizes decimal carry-save addition in the iterative portion of the configuration and stores a less number of multiplicand multiples was displayed. At that point, a second decimal multiplier configuration was presented with a few upgrades including quick generation of multiplicand multiples that do not needed to be stored, a common decimal carry-propagate addition to produce the final product and the use of decimal (4:2) compressors. During that time, when two n-digit operands are multiplied to generate a 2n-digit product, the improved multiplier design has an initiation interval of n + 1 cycles and a worst-case latency of n + 4 cycles.

Y. He et.al [7] proposed a new redundant booth encoding scheme, in which the idea was to polarize two adjacent booth encoded digits to directly form an RB partial product to avoid the hard multiple of high radix booth encoding without incurring any correction vector. Synthesis results showed that the RB multiplier designed with proposed booth encoding algorithm exhibit higher speed and less energy-delay product than the existing multiplication algorithms.

R.P.P. Singh et.al [8] presented performance analysis of various Fast Adders in terms of Speed, Area, and Power utilization. In this a design of modified hybrid carry lookahead/carry skip adders (CLSKAs) are presented. This modified carry skip adder is achieved by the use of both fix and variable block size. In usual carry skip adder, each block contains ripple carry adder and skip logic is utilized after each block to produce carry for next block. Its speed depends on carry propagation coming from previous block to next block. In CLSKAs, carry look ahead logic in each block is used to receive carry for next block. The enhanced carry skip adders gives improved speed and power utilization as compare to usual carry skip adder and other adders like carry look ahead adder, ripple carry adder, carry select adder, Ling adder. The improved carry skip adders with fix block needed few more CLB's because of Carry look ahead logic, while with variable block system, area optimization is accomplished.

C. Tsen et.al [9] proposed a hardware design of a joined decimal and binary floating-point multiplier complaint with IEEE P754-2008 Floating-point Standard. This design works on either (1) 64-bit binary floating-point numbers (BFP) or (2) 64-bit binary encoded decimal floating-point numbers (DFP). For the given rounding modes, this design returns appropriately adjusted results. Between above two floating-point data types, the given design shared many hardware resources. The synthesis results

established that hardware sharing has a sensible effect on delay, latency and area. The joined DFP and BFP multiplier design require just 58% of the total area needed by separate DFP and BFP units. Results of the presented design showed that sharing hardware is valuable in terms of area, with a little effect on the delay of BID multiplication and no effect on latency.

C. Tsen et.al [10] introduced a innovative algorithm and hardware design of a DFP adder. This adder performs subtraction and addition on 64-bit operands that utilized the IEEE P754-2008 binary encoding of decimal floating point numbers. This BID adder has utilized a hardware component for decimal digit counting and an improved version of formerly published BID rounding unit. They established that a BID-based DFP adder design can be accomplished with a little area increase in comparison to a single 2-stage pipelined fixed-point multiplier. More than 70% of the BID adder's area is used by 64-bit fixed-point multiplier, which can be shared with a binary floating-point multiplier. This adder gave effectively adjusted results for subtracting and adding IEEE P754-2008 decimal numbers. This design showed that BID subtraction and addition can be effectively achieved in hardware. The design has guaranteed in terms of area and potential hardware reuse.

Liang-Kai Wang et.al [11] displayed novel designs for a decimal floating-point multifunction unit and a decimal floating-point adder. To decrease their delay, both the multifunction unit and the adder utilized decimal injection-based rounding, another type of decimal operand arrangement and a quick flag-based technique for adjusting and overflow detection. Synthesis results showed that the proposed adder is about 1.6% smaller and 21% quicker than a previous decimal floating-point adder design, when executed in the same technology. As compared to the decimal floating-point adder, the decimal floating-point multifunction unit gives six additional operations defined in IEEE P754, yet just has 9.7% more area and 2.8% more delay.

J. Harrison et.al [12] presented new algorithms and properties which are utilized as a part in a software implementation of the IEEE P754 decimal floating-point arithmetic, with emphasis on utilizing binary operations effectively. Since the binary floating-point arithmetic may introduce little but unacceptable errors. Utilizing binary floating-point computations to emulate decimal computations in order to correct this issue has led to the existence of numerous proprietary software packages, each with its own attributes and capabilities. In this outline, the attention is on rounding

methods for decimal values stored in binary format, but algorithms are laid out for the more essential or interesting operations of multiplication, division, and addition, including the cases of non homogeneous operands, as well as conversions between decimal and binary floating-point formats. Execution results were incorporated for a wider range of operations, indicated guarantee that this method is viable for applications that require decimal floating-point computation.

P. Gurjar et. al. [13] described the manufacturing of high speed adder circuit using Hardware Description Language (HDL). The purpose behind this is that an adder is a very critical building block of arithmetic and plays an important role in determining the performance of central processing unit (CPU). They have simulated and synthesised the various adders like full adder, ripple carry adder, carry look-ahead adder, carry skip adder, carry select adder and carry save adder. The simulated results are verified and functionality of high speed adders was compared on parameters like speed, area etc. They concluded that carry save adder is the best adder in terms of speed and area consumption.

M. Al-Ashrafy et. al. [14] presented a proficient implementation of an IEEE P754 single precision floating point multiplier. This multiplier handles both the overflow and underflow cases. To give more precision to the multiplier in a multiply and accumulate (MAC) unit rounding could not be implemented. With latency of three clock cycles, the presented design achieved 301 MFLOPs. The performance of the multiplier was improved using pipelining stages. These stages divide the critical path and increase the maximum operating frequency of the multiplier. These stages were used at various locations i.e. in the middle of significant multiplier, in the middle of exponent adder, after the significant multiplier and finally at the outputs. The synthesis tool “retiming” option was used so that the synthesizer uses its optimization logic to better place the pipelining registers across the critical path.

CHAPTER

3 FLOATING POINT MULTIPLIER

This chapter explained the IEEE P754-2008 standard for floating point arithmetic, binary integer decimal (BID) encoding and decoding technique, the different steps included in BID based decimal floating point multiplication and Booth multipliers for partial product generation and tree multipliers for partial product reduction and the adders which can be utilized at better places in BID based decimal floating point multiplier.

3.1 IEEE P754-2008 Standard for Floating-Point Arithmetic

This standard gives a control to implement floating-point computation that provides results independent of whether the processing has been done in software, hardware or a combination of the two. The IEEE P754 standard was developed by IEEE in 1985. The revised version of this standard was IEEE P754-2008 which was introduced in 2008.

This standard specifies:

- Formats for decimal and binary floating-point data, for computation and data interchange.
- Subtraction, Additions, division, multiplication, compare, fused multiply, add square root and other operations.
- Conversions between floating-point and integer formats.
- Conversions between different floating-point formats.
- Conversions between floating-point formats and external representations as character sequences.
- Floating-point exceptions and their handling that include the data which are not numbers (NaNs).

3.1.1 Format for floating point numbers

This section defines many floating-point formats, which can be used to represent the real numbers. These formats are different in terms of their precision, exponent range and radix and each format represents a unique set of floating-point data .The IEEE P754 standard defines five basic formats for floating point numbers. These formats have different base and the number of decimal digits to encode them. There are mainly three binary formats and two decimal formats. These three binary formats are called single, double and quadruple precision while the decimal formats are known as double and quad. These five formats in combination are also known as basic formats. Table 3.1 shows the basic formats used for floating point numbers.

Table 3.1: IEEE basic formats for floating point numbers

Basic formats	Base value	Digits or bits	e_{max}
Binary32	2	23 bits	+127
Binary64	2	52 bits	+1023
Binary128	2	112 bits	+16383
Decimal64	10	16 digits	+384
Decimal64	10	34 digits	+6144

Extended formats are also specified by this standard. These formats can be used to increase the precision and the range of the exponent of these basic formats. Using these formats one can specify the precision and exponent range. Interchange formats which are also specified by this standard can be used for the exchange of data. These five basic formats are discussed below:

3.1.1.1 Binary Formats: The three binary formats are discussed below:

1) Single precision

In single precision floating point format, 32 bits are used to represent a number. To store the sign of a number, the most significant bit is used. For exponent storage, next lower 8 bits are used and next 23 bits are used for storing significant of a number. Figure 3.1 shows the representation of single precision floating point number. To make negative numbers possible to avoid extra bit for sign, a bias value of 127 is added to the actual exponent and then it is represented. Many special numbers are

also defined for the single precision format. In IEEE P754-2008, the 32-bit having base 2 format is officially known as binary32.

So, the IEEE P754 standard defines a binary32 as below:

- Sign bit: 1 bit
- Exponent width: 8 bits
- Mantissa precision: 24 bits (23 bits explicitly stored)

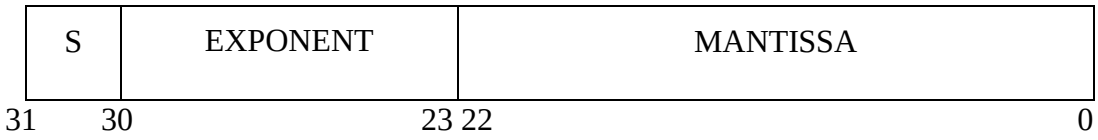

Figure 3.1: Representation of single precision floating point number

2) Double precision

Double-precision floating-point format occupies 8 bytes (64 bits) in computer memory. Computers having 32-bit storage locations utilize two memory locations for storage of 64-bit double-precision number. Double-precision floating-point format commonly known as binary64, as specified by the IEEE P754 standard. It is commonly called simply as double. In IEEE P754-2008, the 64-bit having base 2 formats is officially known as binary 64. The IEEE P754 standard defines a binary64 as below:

- Sign bit: 1 bit
- Exponent width: 11 bits
- Mantissa precision: 53 bits (52 bits explicitly stored)

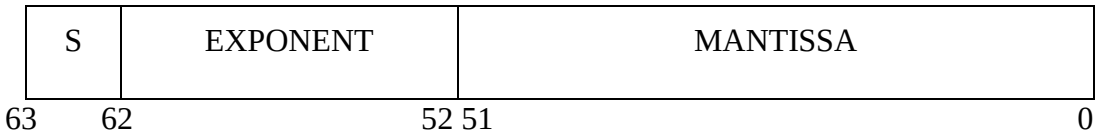

Figure 3.2: Representation of double precision floating point number

Double precision uses twice the size of single precision. It uses 1 bit for sign, 11 bits for exponent width and 53 bits for mantissa part. Figure 3.2 shows the representation of double precision floating point number. A little increase in exponent and large increase in mantissa has increased the accuracy and range for representing a number. This format can further be extended to improve accuracy. For extended format 80 bits are used. 12 bits are added at the end of mantissa and 4 bits are added after the

exponent. This 80 bit extended format is used in computer hardware's like Motorola 6800 series and Intel Pentium series.

3) Quadruple precision

Quadruple precision binary floating-point format occupies 16 bytes (128 bits) in computer memory and its precision is twice the 53-bit double precision. In IEEE P754-2008, the 128-bit base-2 format is commonly known as binary128. Thus only 112 bits of the Mantissa appear explicitly in the memory format, but its total precision is 113 bits. To store exponent, 15 bits are used and most significant 1 bit is used for sign. Figure 3.3 shows the representation of quadruple precision floating point number.

The IEEE 754 standard defines a binary128 as below:

- Sign bit: 1 bit
- Exponent width: 15 bits
- Mantissa precision: 113 bits (112 bits explicitly stored)

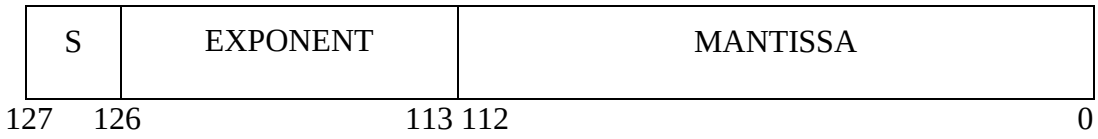

Figure 3.3: Representation of quadruple precision floating point number

3.1.1.2 Decimal Formats

IEEE P754-2008 mainly defines only two decimal formats: decimal64 and decimal128. But because of the significance of DFP arithmetic, the IEEE P754-2208 Standard for floating point arithmetic incorporates particulars for DFP formats and its operations. IEEE P754 defines DFP numbers in three interchange formats i.e. decimal32, decimal64, and decimal128 having 32, 64 and 128 bits representations respectively. The precision C, exponent range E and exponent bias differ for different interchange formats. For example in decimal 32, C has a precision of 7 decimal digits, E has a range from -95 to +96 and the exponent bias is 101. In all these formats, the mantissa is non-normalized, which means its most significant digits can be zero. For the result of each operation, the IEEE P754 Draft Standard defines a preferred exponent. IEEE P754 standard also defines five rounding modes: roundTiesToAway (RTA), roundTiesToEven (RTE), roundTowardNegative (RTN),

roundTowardZero (RTZ) and roundTowardPositve (RTP).

Table 3.2: Format Precisions, Exponent Ranges and Bias

	Decimal Digits In Significant	Exponent Range	Exponent Bias
Decimal32	7	-95 to +96	101
Decimal64	16	-383 to +384	398
Decimal128	34	-6143 to +6144	6176

3.1.2 Operations

The IEEE P754 standard includes all arithmetic operations like subtract, add, multiply, square root, divide etc. This standard also defines conversion between different formats, quantization and their scaling. It also specifies some functions related to sign like manipulating and copying the sign. Some other function like testing, comparisons and total ordering functions are also specified in it. Setting of flags can also be done. Some other miscellaneous operations are also included in this standard.

3.1.3 Rounding rules

IEEE P754 standard defines five rounding modes which are roundTiesToAway (RTA), roundTiesToEven (RTE), roundTowardNegative (RTN), roundTowardZero (RTZ) and (RTP). The first two rounding modes are round to nearest modes and rest three modes are direct rounding modes. In roundTiesToEven mode, if the number falls in the middle, then it will be rounded to nearest value with an even LSB. By default, this rounding mode is used for binary floating point numbers. In roundTiesToAway mode, if the number falls in the middle, then it is rounded to nearest value below for negative numbers and above for positive numbers. RoundTowardZero mode directly rounds the result towards zero while roundTowardPositve and roundTowardNegative modes rounds towards positive and negative infinity.

3.1.4 Exceptions

This standard defines five exceptions. These are divide by zero, invalid operation, underflow, overflow and inexact. When a operation is performed which is not defined

in the given standard, then invalid operation will occur. For instance, square root of a negative number will result in invalid operation. Divide by zero exception will occurs, a given number is divided by zero or result a infinite number. If a given number is too large to fit in exponent field, then overflow will occur. If a number is too small to fit in exponent field, then underflow will occur. If the obtained result is by default correct or inexact, then inexact exception will occur.

3.2 BID Encoding and Decoding of Decimal Floating Point Numbers

Encoding defines how a bit pattern can be derived corresponding to a non-numeric or numeric value. The subsequent bit patterns don't span all the possible bit patterns and the decoding defines how all possible bit patterns can be interpreted. So, encoding can be related to that of the output delivered by decimal arithmetic operations and decoding can be related to the translation of input operands to the decimal arithmetic operations.

3.2.1 Characteristics of BID Encoding

The BID encoding has the following main characteristics:

1. It utilizes an absolutely binary encoding for the whole decimal coefficient c. For e.g. in the 16-digit case, it encodes the coefficient c, as $0 \leq c < 10^{16}$, completely in binary format, and subsequently the name Binary-Integer Decimal (or Big-Integer Decimal), BID [6].
2. For a good portion of the numerical extent, a decimal encoding relates to an IEEE binary floating point encoding of another finite number. This encoding offers some interoperability with IEEE binary floating point. Basic mathematical relationships exist between the two numerical qualities. This property permits a decimal floating point operation be substituted by some successions of binary floating point operations.
3. The quality of BID is its potential for binary hardware reuse, because this allows for low overhead software and hardware solutions.
4. As in the situation for IEEE binary floating point encoding, the biased exponent field and the significand (coefficient) field are stored contiguously.

5. Similar to IEEE binary floating point encoding method, an "implicit bit" method is utilized to economize on the bits needed for the coefficient field. For instance, one does not require 54 bits to store a coefficient c in the range $0 \leq c < 10^{16}$.

3.2.1.1 Encoding to BID

Consider a numerical value is of the form

$$(-1)^s \times 10^{E-\beta_p} \times c \qquad (3.1)$$

Where, $0 \leq c \leq 10^p$, $0 \leq E \leq 3 \times 2^{w-2} - 1 \quad and \;\; p = 3J + 1.$

The parameters p (number of decimal digits), w (an exponent range specifier) and β_p (the bias value) are set according to a particular format. Variables that expresses the particular decimal value to be encoded are c (the coefficient), E (the biased exponent), and s (the sign bit). The parameters for the three fixed-precision formats are shown in Table 3.3.

The sign bit is clearly encoded as 1 binary bit with 1 indicates a negative number, and 0, positive. Because of range $0 \leq E \leq 3 \times 2^{w-2} - 1$, the binary encoding of E, always fit in w bits where the two msbs must be 01,00,10 but not 11.

Table 3.3: Ranges and Parameters for the 3 Decimal Formats

Format	w	J	$p = 3J + 1$	$\beta_p = 3 \times 2^{w-3} + p - 2$	$E \in [0, 3 \times 2^{w-2} - 1]$	c
Dec32	8	2	7	101	[0,191]	$0 \leq c \leq 10^7$
Dec64	10	5	16	398	[0,767]	$0 \leq c \leq 10^{16}$
Dec128	14	11	34	6176	[0,12287]	$0 \leq c \leq 10^{34}$

The binary encoding of the significant c can be represented in $10J + 4$ bits, for given value of $p = 3J + 1, where\ J = 2,5,11$. Additionally, at some point whenever $2^{10J} > 10^{3J}$, the binary encoding of c can't fit into $10J + 3$ bits. In this case, the 4 msbs can be as large as 1001. In other words, the three msbs should always be 100 in this case. (Otherwise, the value of c will be at least 10×2^{10J} which is larger than10^p.) So, the entire encoding is designed as given below. When the significant c fits into $10J + 3$ bits, the value $(-1)^s \times 10^{E-\beta_p} \times c$ is encoded into $w + 10J + 4$ (which is 32, 64, or 128) bits as in Figure 3.4.

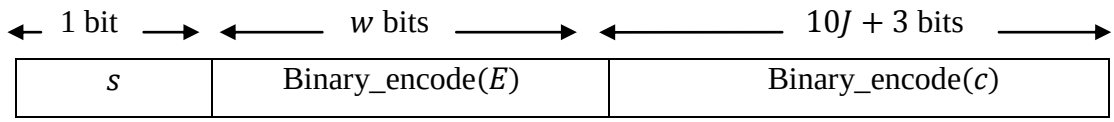

Figure 3.4: Encoding When Binary(c) Fits in 10J + 3 Bits

At the point, when value of c does not fit into $10J + 3$ bits, then its three msbs should always be 100 as discussed before. Consequently, like IEEE binary floating point, these bits are not stored explicitly; just the $10J + 1$ least significant bits (LSB) of c required to be stored. Obviously, certain markers must exist to mean that this is the situation. The value $(-1)^s \times 10^{E-\beta_p} \times c$ is encoded into $w + 10J + 4$ (which is 32,64 or 128) bits as in Figure 3.5.

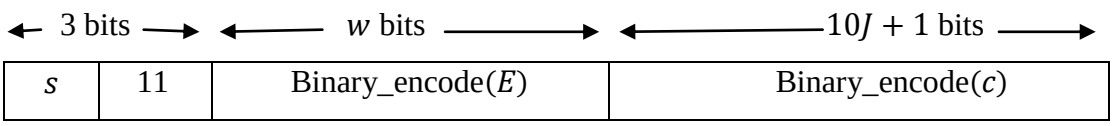

Figure 3.5: Encoding When Binary(c) Requires 10J + 4 Bits

As discussed previously, the 2 msbs of binary encode(E) can never be 11. Consequently the 2 bits immediately following the sign bit recognize the two encoding situations.

3.2.1.2 Decoding from BID

To decode exceptional values, it is recognized that the four bits taking after the sign bit being 1111.Otherwise, the bit pattern is decoded by first extracting several parts. Consider the bit pattern is

$$b_{L-1}b_{L-2}b_{L-3} \dots \dots \dots \dots b_2b_1b_0 \tag{3.2}$$

Where $L = 32,64$ or 128. b_{L-1} represents the sign bit. The parameters w and J are used according to Table 3.3. The definition of E is

$$E = \begin{cases} binary\_decode(b_{L-2}b_{L-3} \dots \dots b_{L-w-1}) & if \ \ (b_{L-2}b_{L-3}) \neq 11 \\ binary\_decode(b_{L-4}b_{L-5} \dots \dots b_{L-w-3}) & if \ \ (b_{L-2}b_{L-3}) = 11 \end{cases} \tag{3.3}$$

An integer value y is defined depending the situation whether $b_{L-2}b_{L-3}$ are 11 or not.

$$y = \begin{cases} binary\_decode\left(b_{10J+2}b_{10J+1} \dots \dots b_0\right) & if (b_{L-2}b_{L-3}) \neq 11 \\ binary\_decode\left(100b_{10J}b_{10J-1} \dots \dots b_0\right) & if (b_{L-2}b_{L-3}) = 11 \end{cases} \tag{3.4}$$

Using y, the decimal coefficient c can be defined by

$$c = \begin{cases} y & if \ 0 \leq y < 10^p, \\ 0 & otherwise. \end{cases} \tag{3.5}$$

According to the values defined above, the decoded numeric value is simply $(-1)^s \times 10^{E-p} \times c$. When y is 10^p or larger, then the definition of c is somewhat arbitrary. To make processing of this case trivial, pick zero rather than other reasonable choices such as 1 or $10^p - 1$.

3.3 BID Multiplication Technique

The basic algorithm to multiply BID numbers to understand the hardware implementation of BID multiplier is as follows.

A DFP number consists of a triple $(S, E - bias, C)$, where S is the sign bit, E is the biased exponent and bias is a constant value which makes E non-negative and C is the significant. For eg :- A and B being two IEEE P754-2008 input operands have (A_S, A_E, A_C) and (B_S, B_E, B_C) as their triples respectively.

To start the operation ,firstly to extract sign, biased exponent and significant ,the BID encoded operands are unpacked ,then A_C and B_C are multiplied using binary multiplication and intermediate product IP_C is obtained. Simultaneously the exponents are added and bias is removed and intermediate exponent is produced and two sign bits are xored together to produce the final sign bit. In the next step, IP_C is examined to determine if rounding is required and how many more digits are needed to be rounded, If the number of decimal digits in IP_C is less than the result precision p, rounding is not needed and multiplication is finished else IP_C must be rounded and accordingly IP_{EXP} should be adjusted .In the last stage the sign ,biased exponent and significant of the result are packed to calculate the BID encoded result of the multiplication. For e.g.-Consider the decimal 32 BID encoded operands of $A = 5\text{B08BBCB}_{16}$ and $B = 8\text{F000073}_{16}$ as inputs to the multiplication. As discussed above inputs are first decoded to obtain:

$A = (0{,}182 - 101{,}572363)$ and $B = (1{,}30 - 101{,}115)$

Then the significant are multiplied to produce $IP_C = 65821745$ .In parallel the biased exponents are added and bias is subtracted to obtain $IP_{EXP} = 182 + 30 - 101 = 111$. In this IP_C contains 8 digits and precision p=7 digits, so rounding off IP_C by one digit is necessary. Depending on the rounding mode, the rounded result is either

$Z = (1{,}112 - 101{,}6582174) = 6582174 \times 10^{11}$ or

$Z = (1{,}112 - 101{,}6582175) = 6582175 \times 10^{11}$

then, Z is encoded to give the BID encoded result of either $B8646F9E_{16}$ or $B8646F9F_{16}$.

3.3.1 Hardware Design of BID Multiplier

In this section, hardware design of decimal 32 BID multiplier is described. The multiplier has three inputs and one output. The inputs are two BID encoded operands A, B and the rounding mode information i.e. round_mode. The output represents BID encoded result of the multiplication through Z as shown in Figure.3.6.

The main blocks of BID multiplier design are two BID decoders, one BID encoder and a multiplier/rounder block. Decoders are used to extract the signs, exponents and significant from two input operands. The outputs from the decoders and third input round_mode are given to the multiplier/rounder block where multiplication and rounding operations are carried out. Output of multiplier/rounder block is the output significant, biased exponent and sign bit. Then these outputs are given to the BID encoder block that will pack these results and provide the final IEEE P754-2008 result.

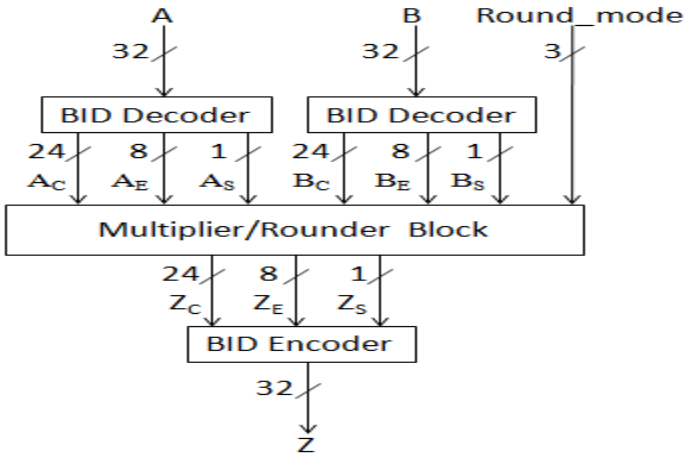

Figure 3.6: High-level block diagram of the 32-bit BID-based multiplier.

3.3.2 Rounding approach for BID multiplier

This section describes an efficient technique to find if rounding is needed. The number of digits to round off i.e. d can be calculated using sum of leading one position of input significant A_C and B_C, in parallel with the computation of IP_C[5].

Description of this technique as discussed in [1] is :-Let leading one position of A_C and B_C are A_{lop} and B_{lop} respectively and let $k = A_{lop} + B_{lop}$,where least significant bit is in position 0.

As $2^{A_{lop}} \leq A_C \leq 2^{A_{lop}+`1} - 1$ and $2^{B_{lop}} \leq B_C \leq 2^{B_{lop}+1} - 1$, then the following is also true:

$$2^{A_{lop}} \times 2^{B_{lop}} = 2^k \leq A_C \times B_C \leq \left(2^{A_{lop}+1} - 1\right) \times \left(2^{B_{lop}+1} - 1\right) \leq 2^{k+2} \quad (3.6)$$

According to this inequality, $A_C \times B_C$ is between 2^k and $2^{k+2} - 1$. The mapping of k to the number of decimal digits in IP_c is off by at most one digit as shown by table 3.4. This table shows lower and upper bound of IP_C for various values of k. The lower bound on the number of digits in IP_C is calculated as

$$n = ceil\left(k \times log_{10}(2)\right) \quad (3.7)$$

Since n is off by at most one digit, IP_C is compared with the smallest power of 10 greater than 2^k, to find the exact number of digits to round off i.e. d. Depending on the comparison, d is determined as follows:-

$$d = \begin{cases} n - p & if\ (IP_C < 10^n) \\ n - p + 1 & if\ (IP_C \geq 10^n) \end{cases} \quad (3.8)$$

Table 3.4: shows k versus no of decimal digits in IP_C and d' and 10^n LUT of decimal 32 multiplication

k	Lower Bound on value of $IP_C(2^k)$	Upper Bound on value of $IP_C\left(2^{k+2} - 1\right)$	Decimal Digits in IP_C (n is bold)	Minimum no. of digits to round off(d')	Boundary value $\left(10^n\right)$
23	8,388,608	33,554,431	7or 8	0	10^7
24	16,777,216	67,108,863	8	1	10^8
...	.....	.....	....	...	....
46	70,368,744,177,664	281,474,976,710,655	14	7	10^{14}

The technique to find if rounding is required uses the estimate k and the comparison of least significant bits of IP_C with constant 10^p. Two binary LOD's are used to determine A_{lop} and B_{lop}. The sum of A_{lop} and B_{lop} i.e. $'k'$ is indexed into a lookup table that stores the minimum number of decimal digits to round off $d' = n - p$ and

also stores boundary values which are precalculated powers of 10 used in the comparison to determine d. For $k < 23$, rounding is not required. Hence LUT index begins at $k = 23$. In parallel with calculation of IP_C, the calculation of A_{lop}, B_{lop}, k and the access to the LUT are performed and only the least significant bits of IP_C are compared with a constant from LUT and delay of BID multiplier is reduced.

Suppose $A_C = 3885$ and $B_C = 1A84C$. Thus,$A_{lop} = 13$, $B_{lop} = 16$ and $k = A_{lop} + B_{lop} = 29$.So IP_C will be in the range $\left[2^{29}, 2^{31} - 1\right]$. Here the number of decimal digits in IP_C is 9 or 10. Hence 2 or 3 digits will be rounded off. The values stored in the LUT indexed by $k = 29$ gives $d' = 2$ and boundary value of 10^9. Since$\left(10^9 > 2^{29}\right)$, which is compared to IP_C. On seeing the result of comparison, either $d = d' = 2$ or $d = d' + 1 = 3$, digits are rounded off. As $A_C \times B_C \geq 10^9$, so $d = 3$ digits are rounded off.

In this decimal rounding is carried out using reciprocal multiplication [2] so that binary multiplier that is being used for BID significant multiplication can be reused. The method to perform BID rounding is based on the method presented in [1], which extends reciprocal multiplication with a theorem presented in [2] to round precisely. The method consists of multiplying the number to round i.e. IP_C by a precalculated approximation of $w_d = 10^{-d}$ to effectively achieve division by $m = 10^d$ [2].In this algorithm five rounding modes are used i.e. RoundTowardNegative (RTN), roundTowardPositive (RTP), roundTowardZero (RTZ), roundTiesToAway (RTA), roundTiesToEven (RTE).

To truncate d digits, a straight forward BID rounding approach is used which performs division by 10^d Based on the value of the truncated digits, the rounding mode and the sign of the product, the truncated significant is accordingly incremented. But this approach is practically not preferable because this is costly in terms of latency or area.

So, decimal rounding using reciprocal multiplication has been done to remove a division circuit and to reuse the binary multiplier used for BID significant multiplication. Reciprocal multiplication can be seen as multiplication by a precalculated and scaled estimation of 10^{-d}.

The method to execute BID rounding comprises of multiplying the number to round by a precalculated estimation of $w_d = 10^{-d}$ to obtain effective division by $m = 10^d$.

Based on the theorem presented in [2], the product $P = IP_c - w_d$ is divided into three fields, TP, D and R, where R gives the rounding information required to find the correctly rounded result, TP provides the truncated product (i.e., floor ($IP_C \times 10^{-d}$)) and D is discarded. The sizes of above 3 fields are depicted by the parameters v and u. Maximum number of bits to represent the number to round off is indicated by parameter u. In this situation, number to round is IP_c and it has up to $2p$ digits, so $u = ceil(2p \times log_2(10))$. Number of bits required to represent $m = 10^d$ is indicated by parameter v. It is calculated as $v = ceil(d \times log_2(10))$. The important point to be noted is that v fluctuates with d, though u is fixed for a given format. The given multiplier has precalculated estimations of w_d which are stored in an LUT. Since IP_c can be maximum of $2p$ digits, which can be rounded to p digits, so the LUT will have $2p - p = p$ entries.

The $(2u + 1)$bit product, i.e. $P = IP_C \times w_d$, where P is divided into three fields: TP, D and R. D comprises of the u LSB's of IP_C, which contain no helpful data and are disposed of. The R and TP fields involve the remaining $(u + 1)$ bits and change in width based on v. The most significant $(u + 1 - v)$ bits represents the truncated product field i.e. TP. The remaining v bits relate to the R field and they are reviewed to find if the truncated decimal digits, when seen as a fraction, represent exactly one half, exactly zero, above one half or between zero and one half. To find this, a sticky bit, s^* and round bit, r^* are removed from the R field. The MSB of the R field represents the round bit, and the remaining bits of this field are OR'd together to create the sticky bit. Utilizing the r^* and s^* bits, the sign of the result and the rounding mode, the truncated product can be incremented, evaluating the correct result of the multiplication.

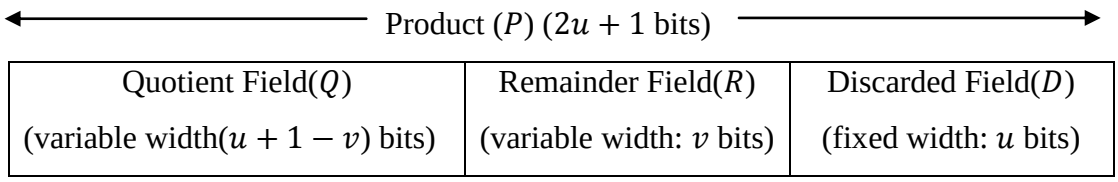

Figure 3.7: Product fields

To determine whether the pre-rounded result is exact, less than a midpoint, a midpoint, or greater than a midpoint, the properties described in Table 3.5 are used. As shown in Table 3.5, these two bits keep enough information to know if the pre-

rounded result is exact ($r^* = 0, s^* = 0$), less than a midpoint($r^* = 0\,, s^* = 1$), a midpoint ($r^* = 1\,, s^* = 0$) , or greater than a midpoint($r^* = 1\,, s^* = 1$).

Table 3.5: Midpoints and Exact Results

Conditions	Values of R	r^*	s^*
Pre-rounded result greater than midpoint	$R > 2^{v-1}$	1	1
Pre-rounded result exactly midpoint	$R == 2^{v-1}$	1	0
Pre-rounded result less than midpoint	$R < 2^{v-1}$	0	1
Pre-rounded result is exact	$R == 0$	0	0

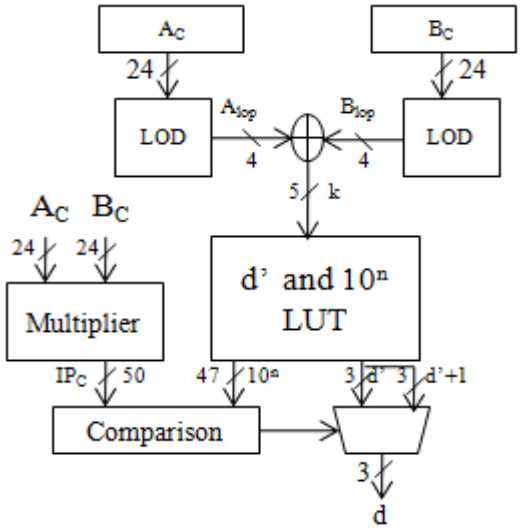

Figure 3.8: Hardware to determine d for BID decimal32 multiplication.

For 32-bit BID multiplier, the maximum input coefficient supported is $A_C =$ 9,999,999. So, the maximum value of IP_C is the result of multiplying 9,999,999 by itself, which generates the following 47-bit number:

$$IP_C = 99{,}999{,}980{,}000{,}001$$

Subsequently, $u = 47$. By multiplying two decimal32 numbers, intermediate significant up to 14 digits will be generated. D ranges from 1 to 7, since any intermediate result having more than $p = 7$ digits should be adjusted to $p = 7$ digits. Accordingly, v varies from 4 to 24, w_d is 48 bits, and just 7 precalculated estimates of w_d are stored in a 7-entry by 48-bit LUT. In this decimal32 BID multiplier configuration, since $u = 47$, the TP field is of $48 - v$ bits in width. The lower 24 bits

in the TP field relate to the value of the 7-digit truncated product, since it is realized that the upper $(24 - v)$-bits of TP are ensured to be zero.

3.3.3 Multiplier/Rounder Block

Different radix-based algorithm was proposed by Booth, out of which radix-8 booth [7], [16] algorithm is used to reduces the number of partial products . So, in binary multiplier, booth radix 8 multiplier is used to generate the partial products and then Dadda and Wallace tree structures are used to accumulate the partial products [17], [18]. The binary multiplier produces the product in the form of carry save as $(PC, PS) = M.N + S + C$ [6]. Here PC is partial carry output and PS is partial sum output. The binary multiplier combines the sum of feedback information (C, S) and carry save partial product (PC, PS) using a 4:2 compressor which further decreases the area and delay of BID multiplication [1]. Figure2 represents whether rounding is required or not. As already discussed, rounding is needed if IP_C is greater than or equal to 10^7.A direct comparison of IP_C to 10^7 results in higher delay[1], so in order to reduce delay, this BID multiplier utilizes an optimized technique where A_{lop}, B_{lop} and a 25 bit adder that adds the 25 least significant bits of PC and PS are used. To add these bits any one of carry select adder, carry look ahead adder or ripple carry adder can be used[8], [21].

If $(A_{lop} + B_{lop} > 23)$ or if the output of the 25-bit adder is greater than 10^7, then rounding is done. The 25-bit adder forms the least significant 25 bits of IP_C. Then the most significant bits of IP_C are obtained in parallel with least significant bits of IP_C by using a compound adder. A compound adder adds the most significant 25 bits of PS and PC to produce $sum = PS[49:25] + PC[49 + 25]$ and $sum + 1$. Depending on the carry-out from the 25-bit adder, the multiplexer selects the correct sum for the most significant 25 bits of IP_C.To calculate precisely number of digits to round off, the BID multiplier does the comparison between 10^n and $IP_C = PS + PC$ using a 50-bit 3:2 compressor. $PC + 1$, PS and the bitwise negation of 10^n are the inputs to 3:2 compressor. This produces two vectors that are then added together, as shown in Fig. 3.10. After this the inspection of sign of the addition of the two vectors is done to determine if $IP_C > 10^n$. The LSB of PC is set to 1 so that $PC + 1$ can be obtained without any addition, as LSB of a carry vector is normally 0. This approach

has less delay than subtracting 10^n from IP_C. The exact number of decimal digits to round off is calculated, once it is determined if $(IP_C > 10^n)$, as described above.

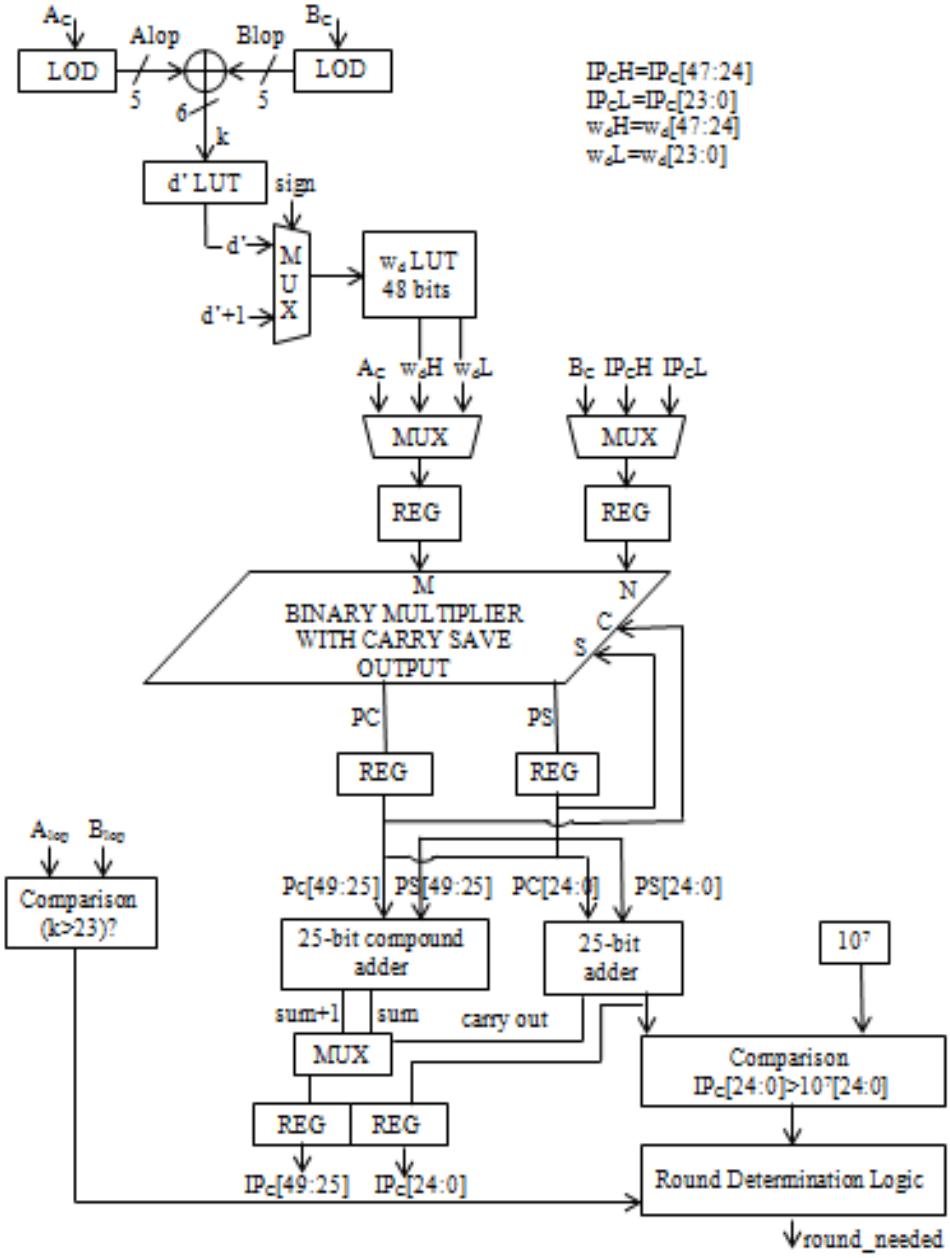

Figure 3.9: Block diagram of the multiplier/rounder for decimal32.

The rounding is carried out by splitting IP_C and w_d into upper and lower 24-bit halves and then obtaining four passes through the 24×24-bit multiplier [4]. By using this approach, both significant multiplication and rounding are performed by same multiplier, Output of multiplier/rounder block is the output significant, biased exponent and sign bit. Then these outputs are given to the BID encoder block that will pack these results and provide the final IEEE P754-2008 result.

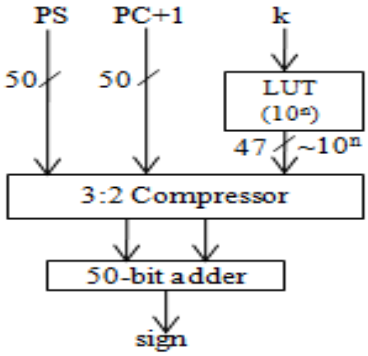

Figure 3.10: Hardware for comparing 10^n and $IP_C = PS + PC$ for decimal32 BID multiplication.

3.3.4 BID Multiplication Algorithm

Step1: Inputs A and B are decoded to find (A_S, A_E, A_C) and (B_S, B_E, B_C).

Step2: Compute sign: $Z_S = A_S \; XOR \; B_S$

Add Exponents: $IP_C = A_E \times B_E - bias$

Multiply Coefficients: $IP_C = A_C \times B_C$

Find leading one position of A_C: A_{lop}

Find leading one position of B_C: B_{lop}

Find leading one position of IP_C: $k = A_{lop} + B_{lop}$

Step3: Finding if rounding is required:

round_required= $(IP_C \geq 10^7)$

If (round_required) //Find d

$d^{'} = n - p$ //d' is stored in LUT that is indexed by k

$d = d^{'} + (IP_C \geq 10^7)$ //10^n is stored in LUT that is indexed by k

Step4: Execute rounding if required:

If (! round _required) //multiplication operation is finished

$Z_E = IP_E$

$Z_C = IP_C$

Else //rounding required

$TP_C = floor(IP_C \times 10^{-d})$ //from $IP_C \times w_d$

Step5: Obtain r^* and s^* information to get the final result

Optional increment of IP_C depending on sign, rnd_mode, r^* and s^*

$Z_{c\_tmp} = TP_C + 1 \; or \; Z_{c\_tmp} = TP_C$

Step6: Check whether adjustment is required:

If ($Z_{c\_tmp} == 10^7$) //adjustment required

$Z_C = 10^6$

$Z_E = IP_E + d + 1$

Else

$Z_C = Z_{c\_tmp}$ and $Z_E = IP_E + d$

Step7: Output is encoded in the result format

3.4 Booth Multiplier for Partial Product Generation

To start a digital multiplication, the first step is to generate n shifted copies of the multiplicand where n is the number of bits in a multiplier. Then these shifted partial products are added to produce the result. But there is no need to add all the partial products. To check which partial product needs to be added or not added depends on the bits of the multiplier. So, if the i^{th} bit of the multiplier is 1 then the generated partial product will be the entire multiplicand which needs to be added. But if the i^{th} bit of the multiplier is 0, then the generated partial product will be string of zeros which don't need to be added. AND gates can be used to generate and add partial products. But this approach takes large amount of area. Hence for high performance systems, this method of multiplication is not used. By reducing the generated partial products, multiplication can be fastened. There are Different algorithms available that reduce the number of partial products. The best algorithm used to reduce the number of partial products is Booth's algorithm. In 1951, Andrew Donald Booth devised a multiplication algorithm, which was named as Booth's Algorithm. The multiplier that uses this algorithm is called as Booth multiplier. It is used to multiply two signed binary numbers in which negative numbers are represented in two's complement form. This algorithm is applicable for both unsigned and signed numbers. In this algorithm, the partial products are reduced because to generate a partial product it consider certain number of bits of the multiplier instead of considering only 1 bit. So, this algorithm is used to reduce the multiplication time. Depending upon the multiplier input bit pattern, one can estimate the exact amount of time reduction. If a given multiplier has a stream of either 0's or 1's, the number of addition required can be minimized. Different versions of booth's algorithm are radix-2, radix-4, radix-8 and so forth. And these are explained below:

3.4.1 Radix-2 booth's algorithm

Radix 2 booth's algorithm[19] is that type of multiplication algorithm in which two signed binary numbers which are represented in two's compliment notations are multiplied. The Booth's algorithm has two main purposes:

1) Signed multiplication

2) Fast multiplication

Suppose Y is a multiplicand and X is a multiplier. The main function of Booth's algorithm is to examine the multiplier bits and then shifting the partial product. Before shifting, the multiplicand can be added to partial product or subtracted from partial product or remain same according to some rules as given below:

a) The first step is to append '0' to the LSB of the multiplier i.e. X_{-1} is 0.

b) Second step is to examine the bits X_i and X_{i-1} and perform operation as per the values stored in Table3.6.

c) The following step is to set two registers, which are named as p and q, to be zero. These will be the registers which are used to store the product during the working of the problem.

d) After addition, p and q are arithmetically shifted towards right and the last bit of q is dropped off and i is circularly shifted right, along with the copying LSB of i into $i-1$.

e) Now repeat above steps for the n number of times where n is the number of bits in multiplier and multiplicand. And finally result of $X \times Y$ will be obtained.

Drawbacks of Radix-2 Booth's Algorithm

a) There are a large number of add or subtract operations and of shift operations between two consecutive add or subtract operations.

b) It is not beneficial to use for synchronous multipliers.

c) Algorithm becomes inefficient for isolated 1's.

To solve the problems of radix-2 booths algorithm, other Booth algorithm of higher radix are used in which more than 2 bits are processed. As the number of processed bits increased, the number of partial products generated will decrease due to which the total number of cycles for obtaining the result gets reduced. Hence the speed of the multiplier will increase.

Table 3.6: Operations involved in radix-2 booth algorithm

X_i	X_{i-1}	Partial products	Operation on multiplicand
0	0	0	Only shift to right by 1 bit
0	1	$+Y$	Add multiplicand to current sum of partial products and then shift result right by 1 bit
1	0	$-Y$	Subtract multiplicand from existing sum of partial products and then shift right by 1 bit
1	1	0	Only right shift by 1 bit

3.4.2 Radix-4 Modified Booth's Algorithm

In 1961, O. L. Macsorley proposed the Modified Booth Encoding (MBE) also known as Modified Booth's Algorithm (MBA). It is extensively used to increment the speed and to decrement the area of multiplier. Radix-4 and Radix-8 modified booth's algorithm are explained as below:

a) Radix-4 Booth's Algorithm

In radix-4 booth algorithm[19], partial products are generated by considering three bits instead of two bits of a multiplier. To start the pairing operation, initially a zero bit is appended to the LSB of the multiplier. Now this appended zero bit will become the LSB of the multiplier. Now, pairing of bits start from LSB to MSB of the multiplier. Each pair of 3 bits is formed such that the lowest bit comes from the previous pair and other two bits are taken from next two bits of the multiplier as shown in figure 3.11. Pairing is needed until the MSB of the number is achieved. Different multiples of multiplicand are generated depending upon pairing of these three bits. These multiples may be any of $\{+Y, -Y, -2Y, +2Y, 0Y\}$ as shown in table 3.7.

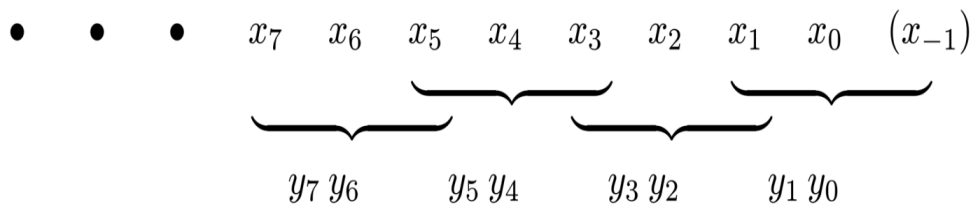

Figure 3.11: Recoding in Radix 4 [19]

$+Y$ is the multiplicand. On shifting the multiplicand by one bit to left, $+2Y$ will obtain. $-Y$ can be obtained by taking 2's complement of the multiplicand. While $-2Y$ is achieved by shifting the multiplicand and then taking its 2's complement. In this way, the multiplies will be generated and partial products will be reduced to half.

Table 3.7: Operations involved in radix-4 booths algorithm

X_{i+1}	X_i	X_{i-1}	Partial products PP_i	operation on multiplicand
0	0	0	0	Add 0
0	0	1	+Y	Add multiplicand
0	1	0	+Y	Add multiplicand
0	1	1	+2Y	Add 2*multiplicand
1	0	0	-2Y	Subtract 2*multiplicand
1	0	1	-Y	Subtract multiplicand
1	1	0	-Y	Subtract multiplicand
1	1	1	0	Subtract 0

3.4.3 Radix-8 booth's algorithm

Radix-8 algorithm[19] examines 4 bits simultaneously. It is same as radix-4 algorithm but here 4 bits are grouped instead of three bits at a time. Multiplier with this algorithm produces less number of partial products than radix-4 algorithm. If number of bits in the multiplier is n, then it generates $n/3$ partial products. But the generation of partial products is more complex. Because it is needed to perform a large number of operations to produce $+1Y, +2Y, +3Y, +4Y, -1Y, -2Y, -3Y, -4Y$. Since the full adder is required to generate $3Y$, hence it becomes more difficult. Initially $2Y$ is obtained by shifting operation, then Y is added to produce the correct result i.e. $3Y$. The numbers of partial product generated are less in radix-8 booth's algorithm but number of operations required are large due to which complexity increases. But the Radix-8 provides fastest speed among radix-4 and radix-2.

Table 3.8: Operations involved in radix-8 booths algorithm

X_{i+2}	X_{i+1}	X_i	X_{i-1}	Partial products
0	0	0	0	0
0	0	0	1	$+Y$
0	0	1	0	$+Y$
0	0	1	1	$+2Y$
0	1	0	0	$+2Y$
0	1	0	1	$+3Y$
0	1	1	0	$+3Y$
0	1	1	1	$+4Y$
1	0	0	0	$-4Y$
1	0	0	1	$-3Y$
1	0	1	0	$-3Y$
1	0	1	1	$-2Y$
1	1	0	0	$-2Y$
1	1	0	1	$-Y$
1	1	1	0	$-Y$
1	1	1	1	0

3.5 Tree Multipliers

Wallace and Dadda proposed two well known fast column compression multipliers. These both contain three steps. In the first step, the partial product matrix is created. In the next step, partial product matrix is compressed to a height of two. In the last step, these two rows are added using a propagate adder. The Wallace method reduces the partial products as soon as possible. On the other hand, Dadda method does the minimum reduction possible at each stage to obtain the reduction in the same number of levels as needed by a Wallace multiplier [3].

3.5.1 Dadda Tree Algorithm

Dadda multiplier uses less number of 3:2 and 2:2 compressors at each level during the compression to obtain required reduction. The following recursive algorithm gives the reduction procedure for the Dadda multipliers.

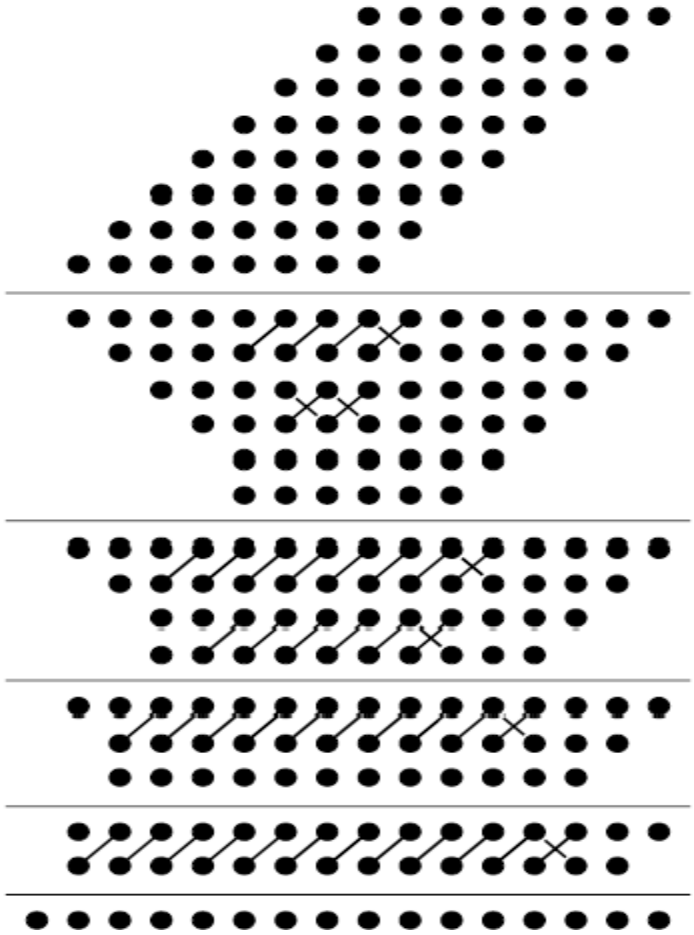

Figure 3.12: Dot Diagram for an 8 by 8 Dadda Multiplier [3]

1. Let $d_1 = 2$ and $d_{j+1} = [1.5.d_j]$ where d_j represents the height of the matrix for the j^{th} stage. Repeat this until the largest j^{th} stage is reached which has the original N height matrix that contains at least one column having more than d_j dots.

2. To achieve a reduced matrix in the j^{th} stage from the end, place 3:2 and 2:2 compressors as required. The columns with more than d_j dots are reduced when they receive carries from less significant 3:2 and 2:2 compressors.

3. Make $j = j - 1$ and repeat step 2 till a height of two is achieved. It is achieved when $j = 1$. The number of 3:2 and 2:2 compressors needed for a Dadda multiplier depends on N i.e. the number of bits of the operands and is calculated as follows:

$$(3,2)counters = N^2 - 4N + 3 \quad (3.9)$$

$$(2,2)counters = N - 1 \quad (3.10)$$

Dadda multipliers require lesser number of 3:2 and 2:2 compressors during the compression stage than the Wallace multipliers. When the matrix will reduce to a

height of two, then the carry propagating adder is used at the final stage to obtain the final product. The length of final carry propagate adder is given by:

$$CPA\ length = 2.N + 2 \quad (3.11)$$

The dot diagram shown in Fig. 3.12 shows the algorithm implemented for an 8×8 bit multiplier.

3.5.2 Wallace Tree Algorithm

Partial products for Wallace Tree multipliers are formed using N^2 AND gates in the same way as Dadda multipliers. Next the N rows of partial products are merged together in sets of three rows each. The additional rows which are not the member of a group of three are transported to the next level without being changed. In Wallace tree architecture, the compressors are used to add the bits of the partial products in each column parallelly without propagating any carries. Within each group of three rows, 3:2 compressors are applied to the columns. Columns having only a single bit is transferred to the next level without change. In the j^{th} reduction stage, the height of the matrix i.e. w_j is given by the recursive equations given below

$$w_0 = N \quad (3.12)$$

$$w_{j+1} = 2.\left\lfloor \frac{w_j}{3} \right\rfloor + w_j \ mod\ 3 \quad (3.13)$$

When the matrix will reduce to a height of two, a final carry propagate adder is used to obtain the product of the multiplication as in Dadda multipliers. For both Wallace and Dadda multipliers, the same number of levels is required to perform the reduction to a height of two. But, the heights of different levels can change between the two techniques. Although Wallace and Dadda multipliers consist of nearly equal number of full adders but during the reduction of the matrix, more of the Wallace full adders are applied. Due to these full adders and the additional half adders that are used in a Wallace reduction provides shorter final carry propagating adder.

A dot diagram for an 8 by 8 Wallace multiplier is shown in Fig. 3.13. The number of 3:2 compressors and the length of the final carry propagating adder required for a Wallace multiplier depend on N where N is the number of bits of the operands and S is the number of stages in the reduction and can be determined using equation (3.12), (3.13),(3.14),(3.15) and (3.16).

For $3 \leq N \leq 5$

(3,2) counters=$(N^2 - 4.N + 3 + S)$ (3.14)

CPA length=$2.N-2-S$ (3.15)

For $5 < N$

(3,2) counters=$N^2-4.N+2+S$ (3.16)

Or

(3,2) counters=$N^2-4.N+1+S$ (3.17)

CPA length=$2.N-1-S$ (3.18)

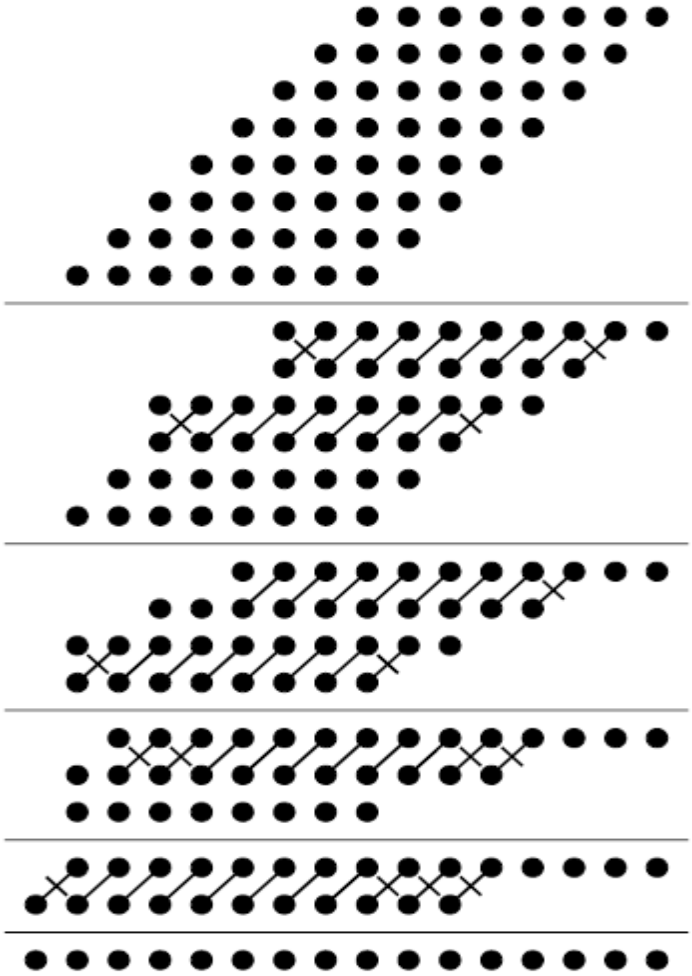

Figure 3.13: Dot Diagram for an 8 by 8 Wallace Multiplier [3]

3.6 Adders used in BID based floating point multiplication

Addition is the basic fundamental arithmetic operation and is most frequently used in general-purpose system and in application-specific processors. Every other arithmetic operation whether multiplication, subtraction or division, usually involve addition operation. In floating point multipliers, adders are used to add the exponents, to add the generated partial products and at the final stage to produce the result. To increase the speed of multiplication, compressors operations are used which also involve

adders. These adders affect the performance of the multiplier. The time consumed in adding the partial product's is a critical parameter to measure the performance of a multiplier. Various types of adders that can be used in floating point multiplier are explained below:

3.6.1 Half adder

The most basic adder used is half adder which adds two single binary bits. It has two outputs, sum and carry. The simplest half-adder design, incorporates an XOR gate and an AND gate. XOR gate is used to calculate the sum and AND gate is used to generate carry. The half adder can be used as a sub block in other digital circuits. Two half adder with an OR gate can be used to realize a full adder. Logic diagram of a half adder is shown in figure 3.14. The truth table for the half adder is given below in Table 3.9. which shows values for output sum and carry for each input combination. Boolean equation for sum and carry are:

$$S = A\ XOR\ B \tag{3.19}$$

$$C = A\ AND\ B \tag{3.20}$$

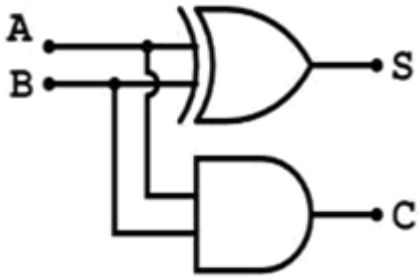

Figure 3.14: Half adder [19], [29]

Table 3.9: Truth table for half adder

INPUT		OUTPUT	
A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

3.6.2 Full adder

A 1-bit Full Adder (FA) is a digital circuit that adds three 1-bit binary numbers and produces two 1-bit binary numbers as output. The output of full adder is a sum and a carry value. Difference between full adder and half adder is that full adder accounts for values carried in as well as out. So, a 1-bit full adder adds three 1-bit numbers, commonly written as A, B and C_i, where A, B are the input operands and C_i is a bit carried in from previous addition. It produces two outputs sum and carry that are represented by the signals S (sum) and C_O (Carry). The truth table of a full adder is shown below in table 3.10. The implementation of a full adder circuit is shown below in Figure 3.15.The Boolean expressions to calculate the sum and carry bits for a full adder are given below.

$$S = A\ XOR\ B\ XOR\ Cin \tag{3.21}$$

$$Cout = (A\ AND\ B) OR\ (B\ AND\ Cin) OR\ (A\ AND\ Cin) \tag{3.22}$$

Table 3.10: Truth table for full adder

INPUT			OUTPUT	
A	B	C_i	S	C_O
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

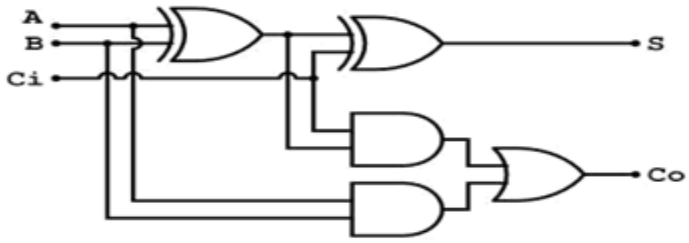

Figure 3.15: Full Adder [19], [29]

3.6.3 Ripple carry adder

Ripple carry adder is used to add two n-bit numbers. It is nothing but a cascading of n 1-bit full adders in series. In this the output is a n -bit sum and 1-bit carry. While

calculating the sum carry out from each full adder is rippled to the subsequent full adder that's why it is known as ripple carry adder. The first full adder used to calculate LSB of the sum can be replaced by a half adder since the carry in for it is zero. Benefit of this adder is its fast design time but its drawback is that it is relatively slower because the subsequent adders have to wait for carry bit from the previous full adders. As input bit stream increases, delay will increase. Therefore, to add large number bits, Ripple carry adder is not suitable. The worst case occurs when carry ripples from least significant bit to most significant bit.

The worse case delay is given by

$$T = (n-1)t_c + t_s \tag{3.23}$$

Where t_s is the delay to calculate sum of last stage and t_c is the delay of carry stage of full adder. Ripple carry adders are used in the situation in which minimum hardware is required. The circuit diagram for 4-bit RCA is shown below in figure 3.16.

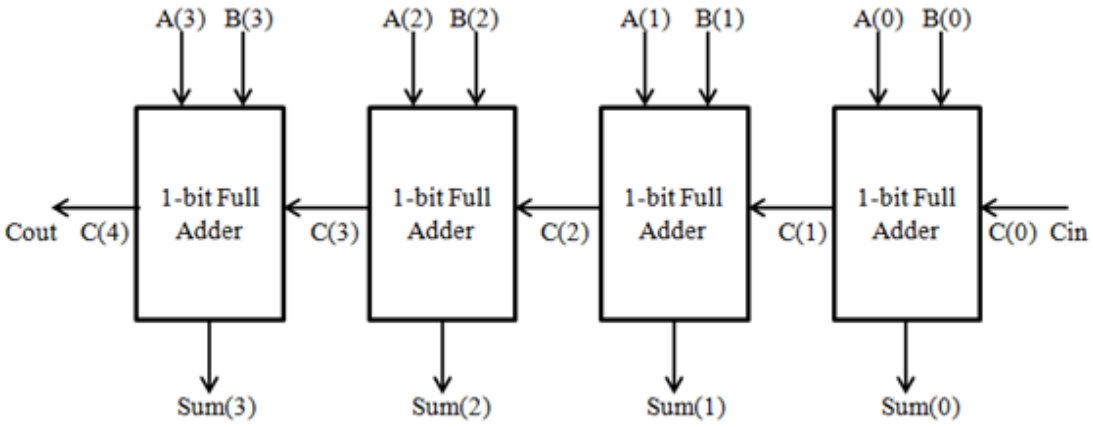

Figure 3.16: Ripple Carry adder [19], [29]

3.6.4 Carry look-ahead adder

Carry look-ahead adder (CLA) is a type of adder which is used in digital logic to speed up the operation of addition. This adder improves speed by generating the carry for next stages in advance based on input signals. This adder is faster than the ripple carry adder because in ripple carry adder, carry bit is calculated alongside the sum bit and each stage must wait until the previous carry bit has been generated to begin calculating its own result and carry bits. On the other hand, one or more carry bits are

calculated before the sum in carry look-ahead adder, which decreases the time to calculate the result of the larger value bits. The delay time of carry look-ahead adder depends logarithmically on size of adder due to which propagation delay of carry signal decreases. The carry look ahead adder has area requirement in O (n log n) and propagation time in O (log n). Carry look-ahead adder uses the concepts of generating and propagating signals. If A and B are two inputs, $C(i)$ is initial carry, then $S(i)$ and $C(i+1)$ are the output sum and carry respectively.

If $A(i) = B(i)$, then carry of 1 will generate if $A(i) = B(i) = 1$, or carry of 0 will generate if $A(i) = B(i) = 0$. If $A(i)$ is not equal to $B(i)$, then carry will be propagated. The structure of 4 bit CLA is shown below in Figure 3.17. This structure uses partial full adders to generate sum output and propagate and generate signals. Carry output is calculated using these propagate and generate signals.

The Boolean expressions to calculate next carry and addition is given by:

$$P(i) = A(i) XOR\ B(i) \qquad (Carry\ Propagation) \qquad (3.24)$$

$$G(i) = A(i) AND\ B(i) \qquad (Carry\ Generate) \qquad (3.25)$$

$$S(i) = P(i) XOR\ C(i) \qquad (Final\ Sum) \qquad (3.26)$$

$$C(i+1) = G(i) OR\ \big(P(i)\ AND\ C(i)\big) \qquad (Final\ Carry) \qquad (3.27)$$

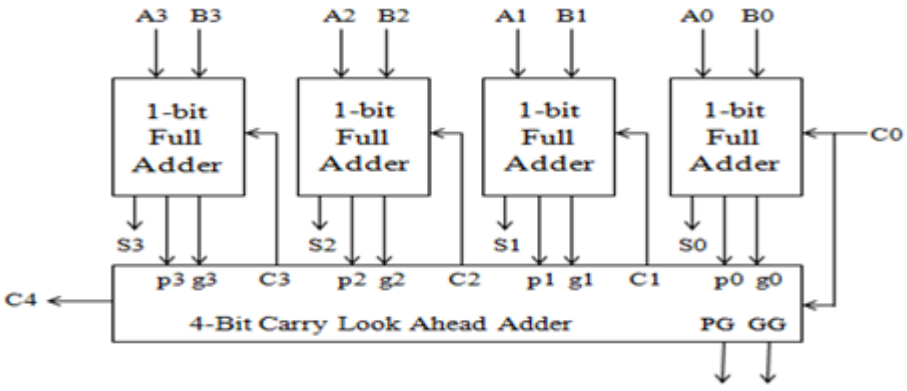

Figure 3.17: Carry Look Ahead Adder [19], [29]

3.6.5 Carry select adder

The carry-select adder adds two numbers by using redundancy to increase the speed of addition. This adder calculates both sum and carry bits for the two cases. In one case input carry assumed to be "0" and in another case it is assumed to be "1". Once the actual carry-in is delivered, the correct sum is chosen (using a MUX) to generate the desired output. Unlike ripple carry adder, Carry look-ahead adder does not need to

wait for carry-in to calculate the sum. Instead of this, the sum is correctly produced as soon as the carry-in bit gets there. So the carry select adder results in a good improvement in speed. But the speed of this adder is increased at the cost of increase in its area because two ripple carry adders are used one for carry in bit '1' and other for carry in bit '0'. Carry-select adders can be divided into uniform or non uniform sections. Figure 3.18 shows the diagram of 4 bits uniform carry-select adder with 4-bit sections. As shown in figure, the calculation of two sums is done using two 4-bit ripple-carry adders. One of these adders takes a carry-in value as '0' whereas the other takes it as '1'. Based on the real carryout from the previous section, the correct sum and carryout the result is selected using a multiplexer. This concept further can be expanded to any length.

For example a 16-bits carry-select adder can be implemented using four sections where each section computes twice using two 4-bit ripple carry adders, for both values of input carry. This is known as linear expansion. To increase the performance, sometimes carry select adder and carry look-ahead adder are combined to take advantages of both these adders.

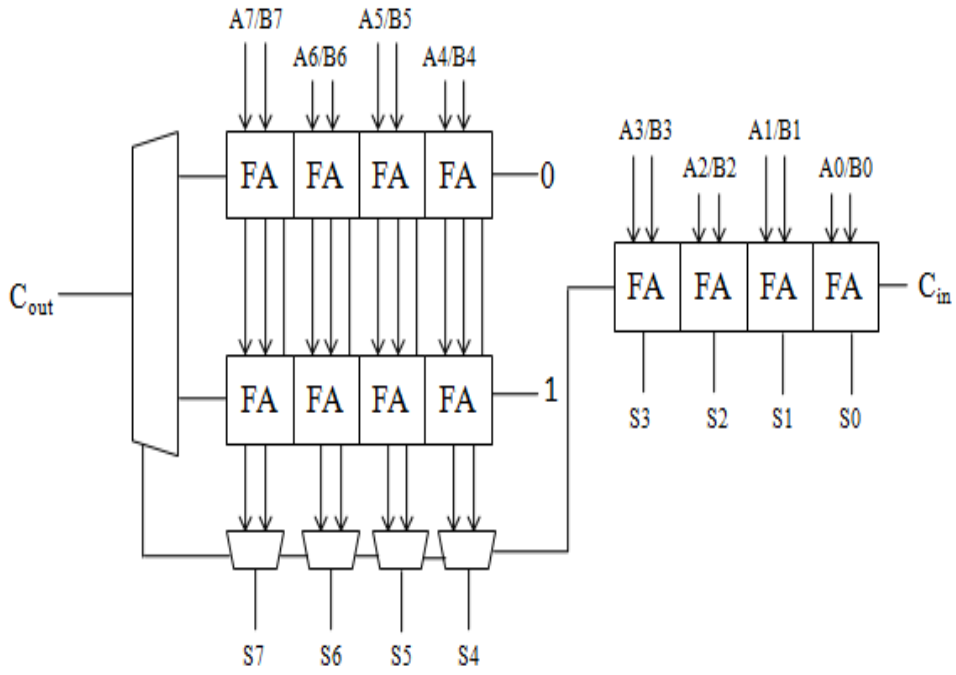

Figure 3.18: Carry Select Adder [19], [29]

3.6.6 Carry save adder

A carry-save adder is a digital adder that can be used in computer micro architecture to calculate the sum of three or more n-bit numbers in binary. It differs from other

adders in a way that it produces two outputs of the same dimensions as the inputs, one output is a sequence of partial sum bits and another output is a sequence of carry bits. A carry save adder can be used as 3:2 compressors. An n-bit carry save adder can be designed by using n separate 1-bit full adders. This adder save the carry out bits instead of using it immediately to compute the final sum, that's why it is known as carry save adder. For a large number of input bit stream, carry-save adders are useful. Since the design automatically removes the delay in carry out bits. Carry save adders can be connected in various configurations which are called trees. Different tree structures are Dadda tree, Wallace tree, overturned stairs tree, 4:2 compressor tree. Below Figure 3.19 show the Wallace tree configuration. To add partial products, carry save adders are used. Because accumulation of partial products become easy when carry save adders are used as the output for sum and carry strings is computed simultaneously. These strings are further given as input to other carry save adders. And at the end, only two bit strings are left. These two bit strings are added using one carry propagate adder at last stage to produce the final result.

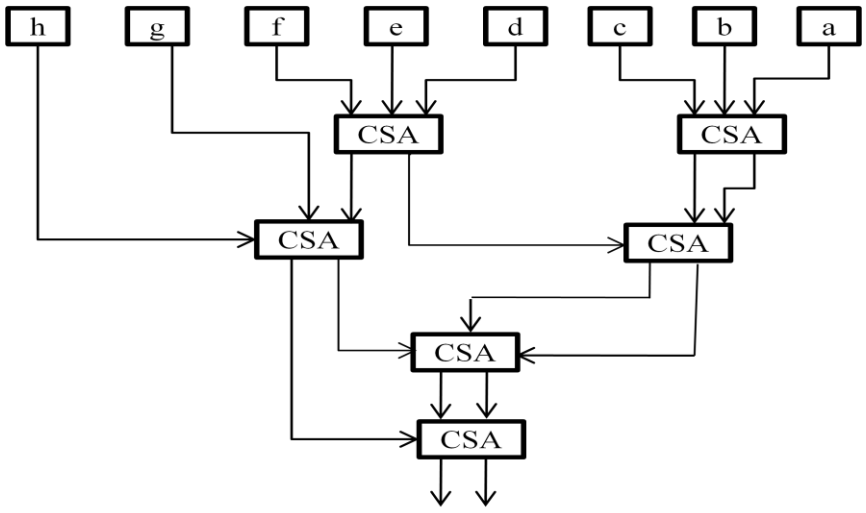

Figure 3.19: Tree structure using carry save adder [19], [29]

CHAPTER

4 FIELD PROGRAMMABLE GATE ARRAY

This chapter presents about the FPGA ideas and FPGA Synthesis Flow. An FPGA is a device that comprises of thousands or even large number of transistors connected to implement logic functions. They implement functions from simple addition and subtraction to complex digital filtering and error detection and its correction.

4.1 Introduction to FPGA

A field programmable gate array (FPGA) is a semiconductor device that can be designed by the designer or the customer after manufacturing, hence it is known as "field programmable". Field Programmable gate arrays (FPGAs) are truly innovatory devices that combine the benefits of both hardware and software. FPGAs are programmed with logic circuit diagram and the source code in Hardware Description Language (HDL) to determine how the chip will work. They may be used to implement any logical function that an Application Specific Integrated Circuit (ASIC) might perform but the ability to update the functionality after shipping offers benefits for many applications. FPGAs consists of programmable logic components also called "logic blocks", and a hierarchy of reconfigurable interconnects that permit the blocks to be "wired together" like a 1 chip programmable breadboard. Just like computer hardware, FPGAs perform calculations spatially and simultaneously computing a large number of operations in resources distributed across a silicon chip. These types of systems can be thousands of times faster than microprocessor-based designs. Though unlike in ASICs, these computations can be programmed into a chip, temporarily frozen by the manufacturing process. It means that an FPGA based design can be programmed and reprogrammed a large number of times.

4.2 FPGA Technology Trends

- Common trend is bigger and faster.

- ❖ This is achieved by increasing device density through even smaller fabrication process technology.
- ❖ New generations of FPGAs are geared towards performing entire systems on a single device.
- ❖ Features such as RAM, clock management, dedicated arithmetic hardware and transceivers are existed in addition to the main programmable logic.
- ❖ FPGAs are also available with the embedded processors.

4.3 XILINX Specifics

All Xilinx FPGAs consists of the following basic resources

1) Configurable logic blocks (CLBs).

2) Input/output blocks (IOBs).

3) RAM blocks.

4) Programmable Interconnections (PIs).

5) Other resources like clock buffers, three-state buffers and boundary scan logic and so on.

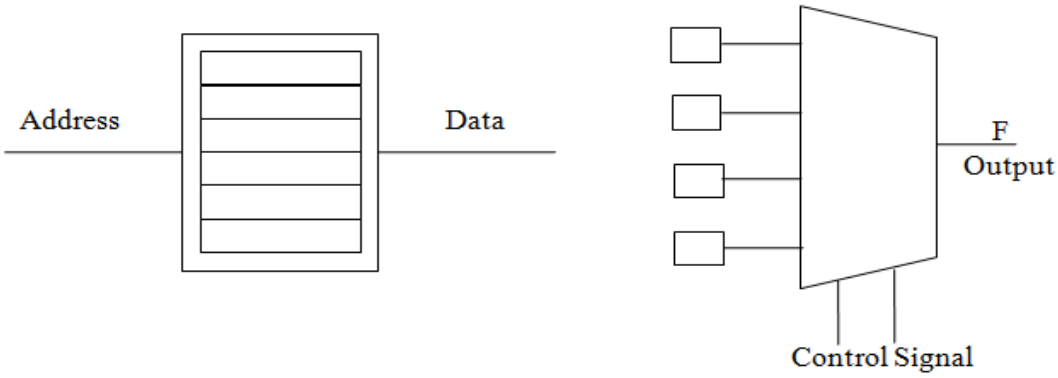

Figure 4.1: Look-up table implemented as (a) Memory (b) Multiplexers and Memory

4.3.1 Configurable Logic Blocks

The main building block of Xilinx CLBs is the slice. Spartan III holds four slices per CLB. Each slice contains two 4-input function generators (F/G), two storage elements and carry logic. Output of each function generator drives the CLB output as well as the D-input of a flip-flop. The look-up tables and storage elements of the CLB have the following characteristics:

1) Look-Up Tables

The way logic functions are implemented in a FPGA is another key feature. Logic blocks that carry out logical functions are look-up tables, implemented as memory, or multiplexer and memory. Figure 4.1 shows these alternatives, together with an example of memory contents for some basic operations. A 2n*1 ROM can implement any n-bit function. Typical sizes for n are 2, 3, 4, or 5. In figure 4.1(a), an n-bit LUT is executed as a 2n*1 memory and the input address chooses one out of 2n memory locations. In figure 4.1(b), the multiplexer control inputs are the LUT inputs. The result is a general-purpose "logic gate." An n-LUT can execute any n-bit function.

2) Storage Elements

The storage elements in a slice can be designed as either a edge-triggered D-type flip-flops or as level-sensitive latches. The D-input can be driven either by the function generators inside the slice or directly from the slice inputs, bypassing the function generators.

4.3.2 Input/output Blocks

The Xilinx IOB consists of inputs and outputs which maintain many I/O signaling standards. The IOB storage elements operate either as D-type flip-flops or as latches. For each flip-flop, the set/reset (SR) signals can be separately designed as synchronous set, asynchronous preset, synchronous reset or asynchronous clear. Pull-down and pull-up resistors can be joined to each pad. IOBs are programmable and can be categorized as follows:

1) Input Path

IOB input path contains a buffer that is used to route the input signals either directly to internal logic or through an optional input flip-flop.

2) Output Path

The output path contains a 3-state output buffer that drives the output signal onto the pad. The output signal can be routed to the buffer either from the internal logic or through an optional IOB output flip-flop. The 3-state control of the output can also be routed directly from the internal logic or through a flip-flop that provides synchronous enable and disable signals.

3) Bidirectional Block

This can be any combination of input and output configurations.

4.3.3 Ram Blocks

Xilinx FPGA incorporates several large RAM memories (block selects RAM). These RAM memories are prepared in columns along the chip. The number of blocks ranging from 8 up to more than 100, depending on the device size and family.

4.3.4 Programmable Routing

Adjacent to each CLB stands a general routing matrix (GRM). The GRM is a switch matrix throughout which resources are connected. Horizontal and vertical routing resources for each row or column include:

- Long Lines: Horizontal and Vertical long lines cover the full height and width of the device.
- Hex Lines: In all four directions, route signals are every third or sixth block away.
- Double Lines: In all four directions, route signals are every first or second block away.
- Direct Lines: Route signals to neighboring blocks vertically, horizontally & diagonally.
- Fast Lines: Internal CLB local connections from LUT outputs to LUT inputs.

4.4 FPGA Implementation Using XILINX

The FPGA that is used for the implementation of the circuit is the Xilinx Spartan 6E (Family), XC3S5000 (Device). The working environment/tool for the design is the Xilinx ISE 14.2i is used for FPGA Design flow of VHDL code.

4.4.1 Overview of FPGA Design Flow

Today, most FPGA vendors offers a quite absolute set of design tools that permits automatic synthesis and compilation from design specifications in hardware specification languages, such as VHDL or Verilog, down to a bit stream to program FPGA chips. A normal FPGA design flow consists the steps and components shown in Figure 4.2.

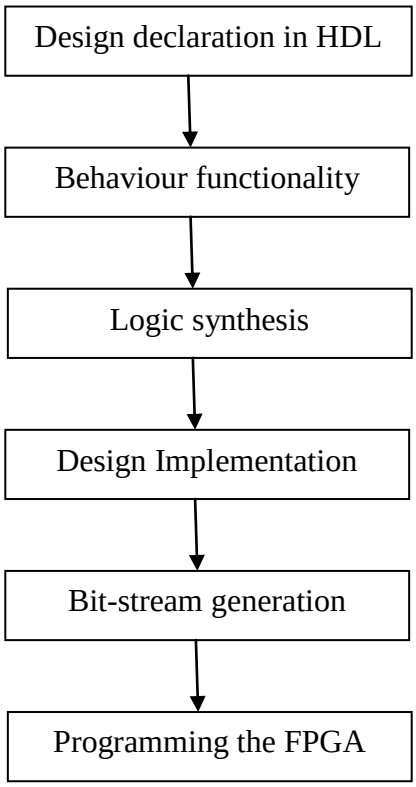

Figure 4.2: FPGA Design Flow

At times, delays between some particular sets of registers may be constrained. The second design input component is the selection of FPGA device. Each FPGA vendor commonly gives an extensive variety of FPGA devices, with different performancein terms of cost and power tradeoffs. The choice of target device may be an iterative procedure. The designer may begin with a little (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 [25]. The FPGA implementation of designs is done to check their functionality on actual hardware. The cost of implementation and design cycle time of ASICs are large therefore the bigger designs are first checked on FPGA and if they give satisfactory results then there ASIC implementation is done.

Figure 4.2 shows the different steps involved in the FPGA implementation. It includes design entry through HDL, behavior simulation, logic synthesis, design

implementation, bit stream generation and finally programming the FPGA. Xilinx provides all the functions necessary for implementing a design on FPGA. The Xilinx ISE suite includes ISIM simulator for functional and timing simulation, XST for synthesis applications, Xpower analyzer for estimating power and chip scope pro analyzer for debug and verification purposes. The different steps involved in FPGA implementation using Xilinx are described below:

4.4.1.1 Design Entry

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.

4.4.1.2 Behavioral Simulation

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

4.4.1.3 Design Synthesis

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

1) Design Hierarchy Analysis: analysis of the hierarchy of the design.

2)HDL Synthesis: The process which translates VHDL or Verilog code into a device netlist format, i.e. a complete circuit with logical elements such as Multiplexer, Adder/subtractions, counters, registers, flip flops Latches, Comparators, XORs, decoders, etc. for the design. If the design contains more than one sub designs, ex. to implement a processor, a CPU as one design element and RAM as another and so on are needed, 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)). Figure 4.3 shows the complete process of synthesis from HDL code to NGC file generation.

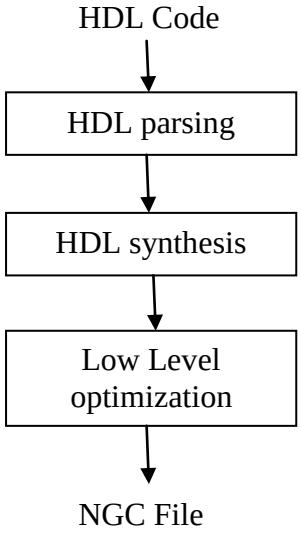

Figure 4.3: Steps in synthesis process

3)Advanced HDL Synthesis(Low Level synthesis): The blocks synthesized in the HDL synthesis are further defined in terms of the low level blocks such as buffers, lookup tables. 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.

4.4.1.4 Design Implementation

Design implementation process consists of the following sub processes

1) Translation: In this process all the input netlists and design constraints are combined and saved in a file called as native generic database file. The ports available in design are assigned to physical elements of the target device. Timing requirements of the design are also specified in translate process. Translate properties can also be changed by modifying them.

2) Mapping: After translate process mapping is done. In mapping the circuit is divided into sub-blocks. The sub-blocks are made so that they can fit into FPGA sub-blocks. A file is generated called as native circuit description file. This file contains our design mapped into components of FPGA.

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

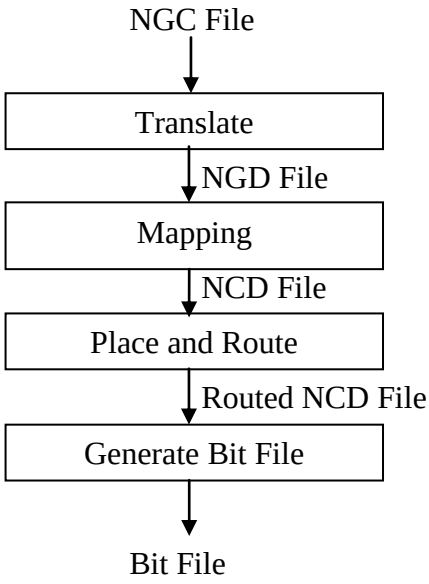

Figure 4.4: Different files generated in implementation process

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

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.

4.5 Analyzing Design Using Chip Scope Pro

The FPGA designs are becoming more complex, due to need of faster designs and shorter design times. Debugging and verification is important factor in determining the complete design time. It takes almost 50% of the design time. But the Xilinx chip scope pro software performs faster debugging and verification. It shrinks overall design by 25%. It is a powerful tool that is easy to use. It is used for debug, verification and inserting short signal sequences. Chip scope pro uses FPGA resources like block RAM for trigger and data storage, slice logic for trigger comparison. It uses three types of flows i.e. core generator, core inserter and plan-

ahead flow. The core inserter flow is similar to plan-ahead flow and provided in plan-ahead software. In core generator flow the core is instantiated in source HDL, while in core inserter flow the core is inserted into generated file after synthesis. Different types of cores are used by chip scope pro software like ICON core, ILA core, VIO core, IBA core. Some of them are explained below:

- ICON core: It is used to control up to 15 capture cores. It acts as interface between JTAG interface and capture cores. The main function of this core is to control the other cores. It can be used in both the core generator and core inserter flows.
- VIO core: It defines and generates virtual i/o s. It is used to apply stimulus and read outputs transition on the node that wants to be selected. This core is used to generate virtual input and outputs. It provides options for synchronous and asynchronous inputs and outputs, where each can have width of 256 bits. There is also option of clock. It can be system clock or JTAG clock The outputs of the design which are wanted to be implement are connected to the input of the VIO core and inputs of the design are connected to the output of the VIO core therefore the inputs are virtual LEDs and outputs are virtual DIP Switches. This core is controlled by the ICON core. So a control port is also provided. VIO core uses FPGA logic not RAM. It is only used in core generator flow.
- ILA core: It is a capture core. It can be used to create custom triggers when activated causes data to be stored during circuit operations. Signals can be stored depending on the condition specified by used. A design can contain up to 15 ILA cores.
- Agilent trace core: It used to store large amount of data off chip or when customer uses Agilent analyzer. ILA with Agilent trace is similar to ILA except data is captured off chip or by Agilent trace port analyzer.

CHAPTER 5

IMPLEMENTATION AND RESULTS

In this chapter, the implementation of BID based floating point multiplier and their synthesis and simulation results are described. Synthesis report describes about the utilization of actual hardware and the timing constraints of the design. Simulation results describe about the behavioral functionality of the design. Implementation of the design is done to transfer the design on to actual hardware. BID based multipliers are synthesized using Xilinx ISE 14.5 targeting Sparten-6E FPGA.
The working environment for all the designs is given below:

- Tool version : ISE 14.5
- Optimization Goal : Speed
- Design Strategy : Balanced
- Family : Spartan 6
- Device : XC6SLX45
- Speed : -3
- Package : CSG324
- Simulator : ISIM
- Total Slices : 27288
- Total LUTs : 54576

5.1 BID Based Floating Point Multiplier

In BID multiplier design, the partial products are generated using radix-8 algorithm, then Dadda and Wallace tree structures are used to accumulate partial products and different adders like ripple carry adder, carry select adder or carry look ahead adders are used at final stage to obtain the result. Using above approach, the BID multiplier is designed in Verilog HDL. Then multiplier is simulated and then synthesized using Xilinx ISE 14.5 targeting Spartan 6 FPGA device.

5.2 Simulation Results for BID based Floating Point Multiplier

After designing the BID based floating point multiplier in verilog HDL, the design is simulated using ISIM simulator. Figure 5.1 shows the simulation results for BID based floating point multiplier.

Considering the following are the inputs in BID format.

a = 01011011000010001011101111001011 (for 10ns) is the multiplicand

b = 10001111000000000000000001110011 (for 10ns) is the multiplier

CLK = changing 0 to 1 (for 10ns with period 1ns) is the clock

Depending upon the rounding mode used, the output obtained after calculating the sign, significant and the exponent and after rounding using the steps described above should be

c= 10111000011001000110111110011110 or 10111000011001000110111110011111

The obtained results on the ISIM simulator are similar to the desired results as shown in figure below figure 5.1.

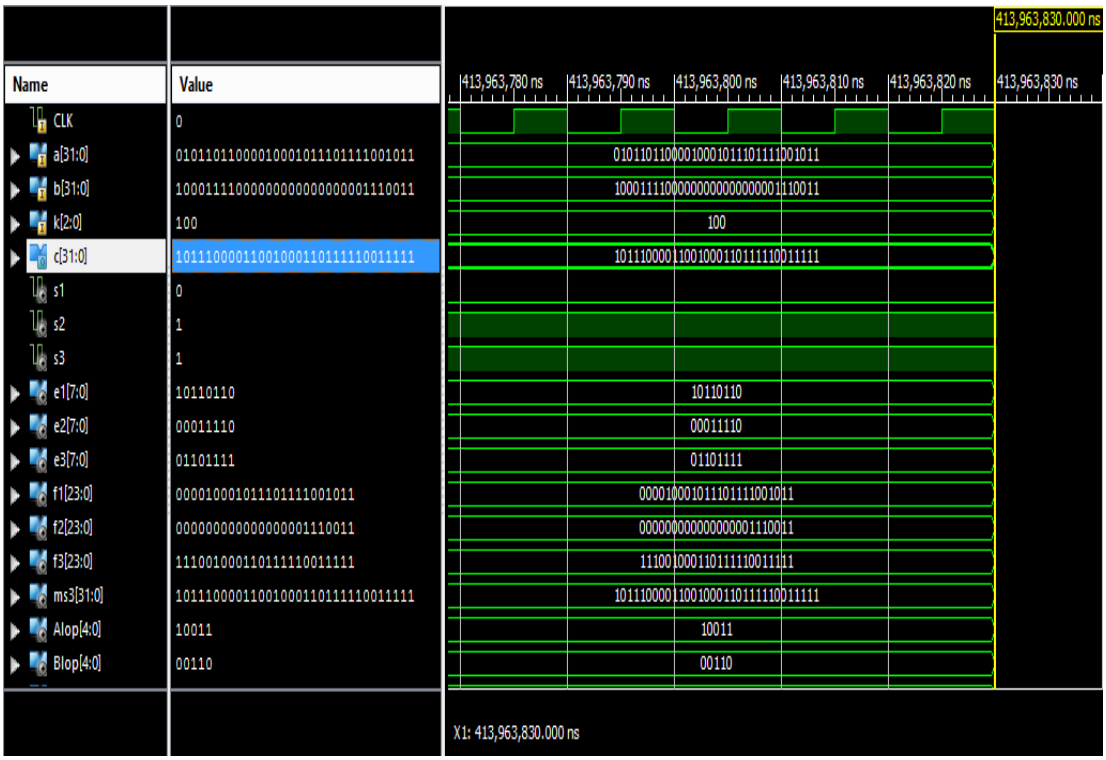

Figure 5.1: Simulation Results for BID based Floating Point Multiplier

5.3 Synthesis Results for BID based Floating Point Multiplier

Table 5.1,5.2 shows the total delay and area utilization summary in terms of number of slices for BID multiplier that uses ripple carry adder, carry look-ahead adder, carry select adder and Dadda and Wallace tree structures. Based upon synthesis reports of different multipliers available, the multiplier with Wallace or Dadda structure using carry select adder gives the best performance in terms of delay but consumes more area .And the multiplier with both tree structures using ripple carry adder is more efficient in terms of area but produces more delay. In terms of area, all multiplier's that uses Dadda structure gives better performance than all other multiplier's using Wallace structure. In terms of delay, all multiplier's that uses Wallace structure gives best performance than all other multiplier's using Dadda structure.

Table 5.1: Synthesis report of BID multiplier using Dadda structure with different adders

Parameter	last stage ripple carry adder	last stage carry select adder	Last stage carry look ahead adder
No of slice LUT(out of 27288)	8622 (31%)	8683 (31%)	9073 (33%)
Min period	44.028ns	27.340ns	28.116ns

Table 5.2: Synthesis report of BID multiplier using Wallace structure with different adders

Parameter	last stage ripple carry adder	last stage carry select adder	Last stage carry look ahead adder
No of slice LUT(out of 27288)	8918 (32%)	9029 (33%)	9340 (34%)
Min period	43.865ns	26.823ns	27.609ns

5.4 Power calculation using Xpower Estimator

The power of the BID multiplier design that uses different adders and tree structures can be determined by Xpower estimator software of Xilinx. The power is calculated by providing values for clock speed, toggle rate and then importing the mapped file of

the implemented design for the specific device. Power can also be calculated by modifying thermal parameters, worst case voltages, clock and resource information and connectivity parameters. Below table 5.3, 5.4 shows total power for the six implemented multipliers.

Table 5.3: Total power for BID multiplier using Dadda structure with different adders

S.No.	BID multiplier using Dadda structure with different adders	Total power=dynamic + static (in mW)
1.	Ripple carry Adder	56.60=18.49+38.11
2.	Carry select adder	57.18=19.07+38.12
3.	Carry look ahead adder	58.83=20.69+38.15

Table 5.4 Total power for BID multiplier using Wallace structure with different adders

S.No.	BID multiplier using Dadda structure with different adders	Total power=dynamic + static (in mW)
1.	Ripple carry Adder	57.63=19.50+38.13
2.	Carry select adder	58.16=20.02+38.14
3.	Carry look ahead adder	58.13=19.99+38.13

5.5 Implementation using Chip Scope Pro Analyzer

The design is verified by chip scope pro tool of Xilinx. Chip scope pro analyzer is the in-built verification tool provided in the ISE pack of Xilinx that is used to verify the implemented multiplier. Since number of inputs is 67 and outputs are 32. So, direct implementation using on-board inputs outputs is very difficult on FPGA. Therefore chip scope pro analyzer is used. The code that is designed in Verilog HDL with necessary cores is instantiated in top module. For virtual input output operation and for controlling the generated VIO (virtual input/output) core, two IP cores are generated. VIO core is generated for providing virtual inputs and outputs. The outputs of the design are connected to the input of the VIO core and inputs of the design are connected to the output of the VIO core. This core is also provided with clock input to synchronize the operation. This same clock is also connected to the design block also. ICON core is generated for controlling the VIO core. The main function of the ICON

core is to control the other cores. After generating the VIO and ICON core a top module is designed which instantiates both these cores and the design module. Then synthesis of top module is done. The RTL schematic for the top module is shown in Figure 5.2. The figure shows that the connections are according to our desire. After that the top module is implemented on FPGA. The generated bit file is burned on the FPGA. The results can be verified by using chip scope analyzer window. The output will be generated on the port sync_in when inputs are given on async_out port. The value for the sync_in port is obtained after applying values to async_out is written below.

async_out=
100010110110000100010111011110010111000111100000000000000000001110011
sync_in = 10111000011001000110111110011111

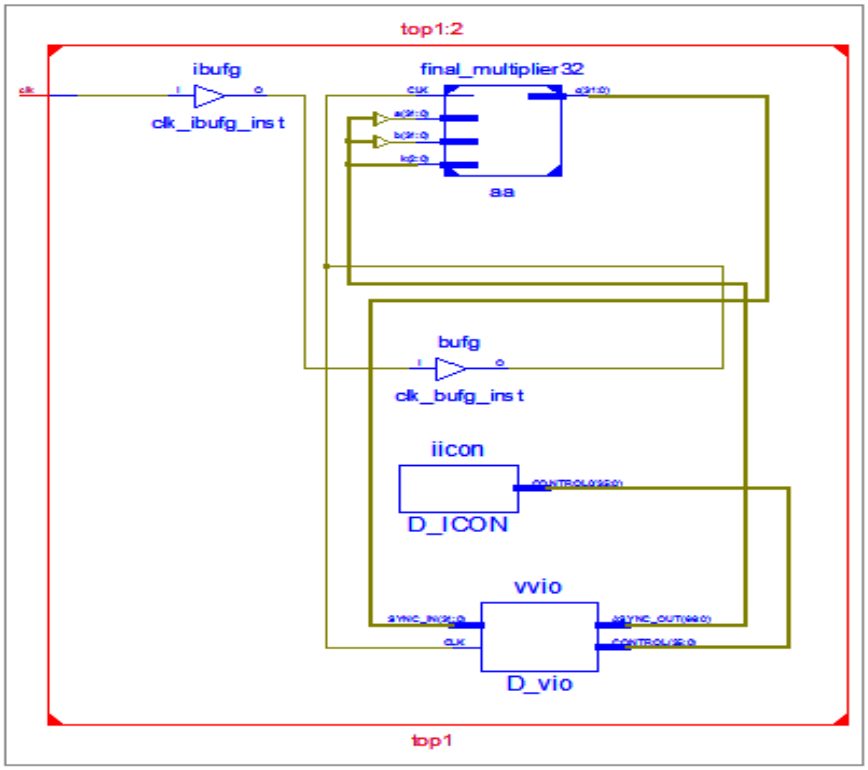

Figure 5.2: Block diagram for FPGA implementation

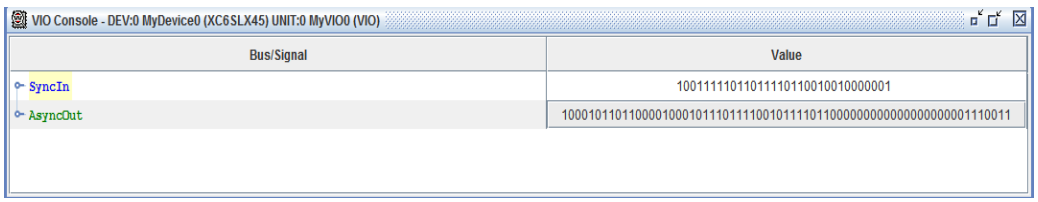

Figure 5.3: Output results on chip scope pro analyzer

CHAPTER

6 CONCLUSION AND FUTURE SCOPE

6.1 Conclusion

In this thesis, comparison with respect to area and delay is done for BID multiplier using different adders. Dadda and Wallace tree structures were used along with different adders to calculate which one of the combinations give best performance in terms of area and delay. Figure 6.1 and Figure 6.2 shows the comparison, in terms of delay and area, for different BID multiplier's that uses ripple carry adder, carry look-ahead adder, carry select adder and Dadda and Wallace tree structures. Comparison has been done using the synthesis results obtained from Xilinx ISE 14.5 targeting Spartan 6 FPGA device as shown in Table 5.1 and Table 5.2.

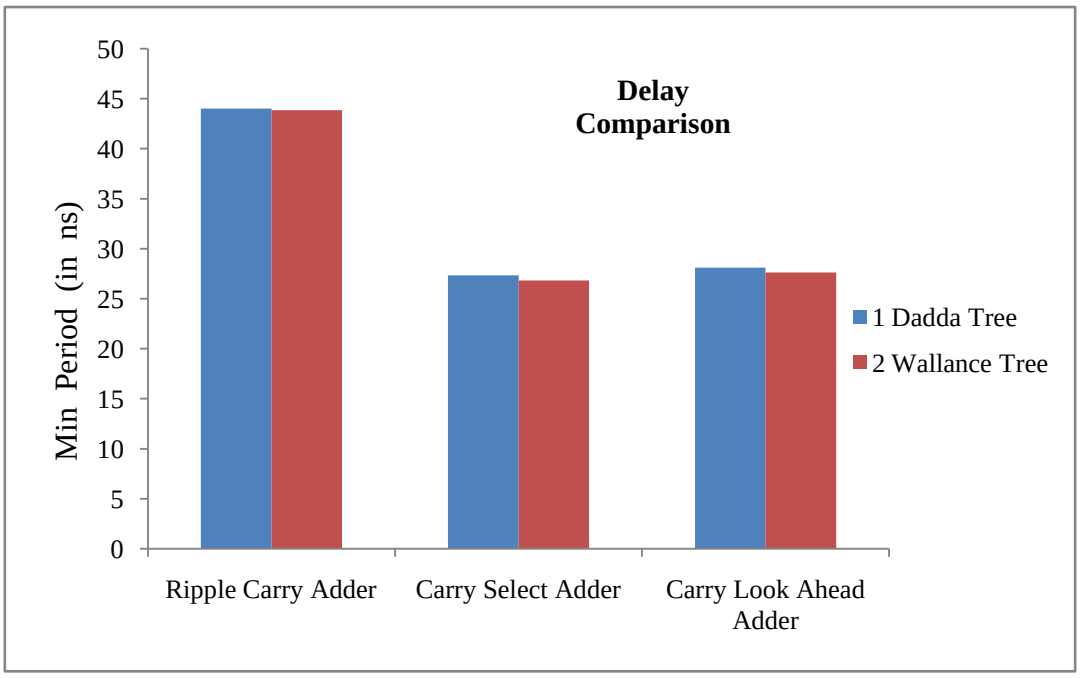

Figure 6.1: Delay Comparison of BID multiplier using Wallace and Dadda tree with different adders

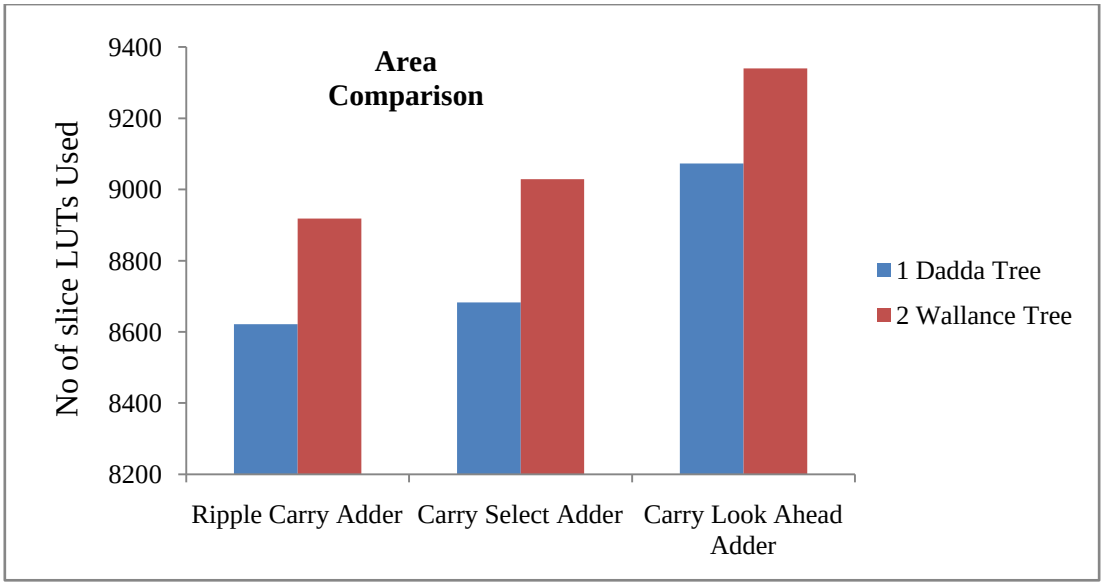

Figure 6.2: Area Comparison of BID multiplier using Wallace and Dadda tree with different adders

From the comparison, it is found that the multiplier with Wallace or Dadda structure using **carry select adder** is **best** in terms of **delay** and the multiplier with both tree structure using **ripple carry adder** is **best** in terms of **area**. In terms of area, all multiplier's that uses Dadda structure gives better performance than all other multiplier's using Wallace structure. In terms of delay, all multiplier's that uses Wallace structure gives performance than all other multiplier's using Dadda structure.

6.2 Future Scope

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

1) Hybrid adders can be used to increase the speed of floating point multiplier.
2) Different rounding methods can be analyzed to optimize the speed and area.
3) Different compressors can be used for accumulation of partial products so that delay can be further reduced.

REFERENCES

[1] S.G. Navarro, C. Tsen and M.J. Schulte, "Binary Integer Decimal-Based Floating-Point Multiplication," in *IEEE Transactions on computers*, vol.62, no.7, pp.1460-1466, 2013.

[2] C. Tsen, M.J. Schulte and S.G. Navarro, "Hardware Design of a Binary Integer Decimal Based IEEE P754 Rounding Unit*," Proc. IEEE 18th Int'l Conf. Application-Specific Systems, Architectures and Processors*, pp. 115-121, 2007.

[3] J. W. Townsend, A. J. Abraham, E. Swartzlander, "A comparison of Wallace and Dadda multiplier delays," *International Society for Optical Engineering (SPIE)*, vol. 52, no. 5, pp. 836-841, 2003.

[4] S.G. Navarro, C. Tsen and M.J. Schulte, "A Binary Integer Decimal-Based Multiplier for Decimal Floating-Point Arithmetic" *Proc. 41st Asilomar Conf. Signals, Systems, and Computers*, pp. 164-170, 2007.

[5] B.J. Hickmann, A. Krioukov, M.J. Schulte, and M.A. Erle, "A Parallel IEEE P754 Decimal Floating-Point Multiplier*," Proc. IEEE 25th Int'l Conf. Computer Design*, pp. 296-303, 2007.

[6] M.A. Erle and M.J. Schulte, "Decimal Multiplication via Carry-Save Addition*," Proc. IEEE Int'l Conf. Application-Specific Systems, Architectures, and Processors*, pp. 348-358, 2003.

[7] Y. He and C. Chang, "A New Binary Booth Encoding for Fast 2^n-Bit Multiplier Design", in *IEEE Transactions on Circuits and Systems-I*, vol.56, no.6, pp. 1192 – 1201, 2009.

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

[9] C. Tsen, S.G. Navarro, M.J. Schulte, B. Hickmann, and K. Compton, "A Combined Decimal and Binary Floating-Point Multiplier," *Proc. IEEE 20th Int'l Conf. Application-Specific Systems, Architectures, and Processors*, pp. 8-15, 2009.

[10] C. Tsen, M.J. Schulte, and S.G. Navarro, "Hardware Design of a Binary Integer Decimal Based IEEE P754 Rounding Unit*," Proc. IEEE 18th Int'l Conf.*

Application-Specific Systems, Architectures and Processors, pp. 115-121, 2007.

[11] Liang-Kai Wang, M.J Schulte, "Decimal Floating-Point Adder and Multifunction Unit with Injection-Based Rounding", *Proc. IEEE 18th Symp. Computer Arithmetic*, pp.56-68, 2007.

[12] M.A. Erle, E.M. Schwarz, and M.J. Schulte, "Decimal Multiplication with Efficient Partial Product Generation," *Proc. IEEE 17th Symp. Computer Arithmetic (ARITH)*, pp. 21-28, 2005.

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

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

[15] P.T.P. Tang, "BID- Binary-Integer Decimal Encoding for Decimal Floating Point," Intel Corporation, technical report, http://754r.ucbtest.org/issues/ decimal/bid_rationale.pdf, 2005.

[16] P.K. Patil and K. laxmi, "High Speed-Low Power Radix-8 Booth Decoded Multiplier" *Proc. International Journal of Computer Applications*, vol.73, no.14, pp. 42-45, 2013.

[17] B. Jeevan, S. Narender, C.V.K. Reddy and K. Sivani, "A High Speed Binary Floating Point Multiplier Using Dadda Algorithm," *Proc. International Multi-Conference on Automation, Computing, Communication, Control and Compressed Sensing (iMac4s)* .pp.455-460, 2013.

[18] R. S. Waters and E.E. Swartzlander, "A Reduced Complexity Wallace Multiplier Reduction," in *IEEE Transactions on computers.* vol.59, no.8, pp.1134-1137, 2010.

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

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

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

[22] M.A. Erle, B.J. Hickmann, M.J. Schulte, “Decimal Floating-Point Multiplication," in *IEEE Transactions on computers,* vol. 58, no. 7, pp.902-916, 2009.

[23] M. Cornea, J. Harrison, C. Anderson, P.T.P Tang, E. Schneider and E. Gvozdev, "A Software Implementation of the IEEE 754R Decimal Floating-Point Arithmetic Using the Binary Encoding Format", in *IEEE Transactions on computers,* vol. 58, no. 2, pp.148-162, 2009.

[24] Liang-Kai Wang, M.J. Schulte, J.D. Thompson and N. Jairam, "Hardware Designs for Decimal Floating-Point Addition and Related Operations”, in *IEEE Transactions on computers,* vol. 58, no. 3, pp.322-335, 2009.

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

[26] N. Brisebarre, M. Mezzarobba, J.-M. Muller, C. Lauter, "Comparison between Binary64 and Decimal64 Floating-Point Numbers", *Proc. IEEE 21th Symp. Computer Arithmetic*, pp. 145- 152, 2013.

[27] ANSI/IEEE 754-1985, “Standard for Binary Floating-Point Arithmetic,” 1985.

[28] IEEE 754-2008, “IEEE Standard for Floating-Point Arithmetic,” 2008.

[29] C. Nagenndra, M.J. Irwin and R.M. Owens, “Area-Time-Power Tradeoffs in Parallel Adders,” in *IEEE Transactions on Circuits and Systems-II,* vol.43, no.10, pp. 689-702, 1996.

LIST OF PUBLICATIONS

[1]	"FPGA implementation of binary integer decimal based floating point multiplier" will be published in the Journal of VLSI Design Tools and Technology (JoVDTT).