FPGA based multiplication using MUX and Vedic Multiplier

Dessertation submitted in partial fulfillment of requirements
for the award of Degree of

Master of Technology
In
VLSI Design

Submitted By:

Rakhi Choudhary
Roll No. 601261022

Under the Guidance of:

Mr. Sukhwinder Kumar
Lecturer ECED

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

JULY, 2014

DECLARATION

It is hereby certified that the work embodied in this dissertation, entitled "**FPGA based multiplication using MUX and Vedic Multiplier**", was carried out in partial fulfilment of M.Tech (VLSI Design) at the Department of Electronics and Communication Engineering, Thapar University, Patiala, carried out under the supervision of **Mr. Sukhwinder Kumar, Lecturer ECED**, Thapar University. The matter has not been submitted, in part or full, for any other degree of this or any other University.

Date: 21/07/2014

Rakhi Choudhary

Roll No. 601261022

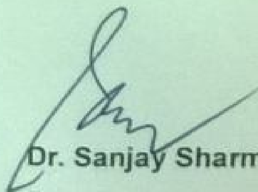
It is certified that the above statement made by the student is correct to the best of my knowledge and belief.

Mr. Sukhwinder Kumar

Lecturer ECED

Thapar University

Counter Signed By:



Dr. Sanjay Sharma

(Head of the Department)

ECED, Thapar University

Patiala-147004

Dr. S.K. Mohapatra

Dean Academic Affairs

Thapar University

Patiala-147004

ACKNOWLEDGEMENT

This dissertation work would not have been possible without the guidance and the help of several individuals who in one way or another contributed and extended their valuable assistance in the preparation and completion of this study.

Foremost, I would like to express my sincere gratitude to my Supervisor, **Mr. Sukhwinder Kumar**, **Lecturer ECED**, Thapar University, Patiala for his patience, motivation and continuous support during the course of this thesis. I also thank *Dr. Sanjay Sharma,* Head of the Department and *Dr. Kulbir Singh,* PG Coordinator for their guidance and support as well as all the faculty members and staff of the Department of Electronics and Communication Engineering for being very supportive to me. I would also like to thank all my colleagues for motivating me all the time whenever I needed them

Lastly, I am indebted to my family for their unending love, support and encouragement.

Rakhi Choudhary

ABSTRACT

Most of the algorithms which are used in DSP, image and video processing, computer graphics, vision and high performance supercomputing applications require multiplication and matrix operation as the kernel operation. In this dissertation, we propose Efficient FPGA based matrix multiplication using MUX and Vedic multiplier. The 2x2, 3x2 and 3x3 MUX based multipliers are designed. The basic lower order MUX based multipliers are used to design higher order MxN multipliers with a concept of UrdhvaTiryakbyham Vedic approach. The proposed multiplier is used for image processing applications. It is observed that the device utilization and combinational delay are less in the proposed architecture compared to existing architectures.

In this thesis we explore and analyse all the previous work on Floating Point Multiplication. The objective is to design a 32-bit single precision floating point multiplier operating on IEEE 754 standard floating point representations. Second objective is to model the behaviour of floating point multiplier design using VHDL.

.

TABLE OF CONTENTS

Declaration	I
Acknowledgement	II
Abstract	III
Table of contents	IV
List of Figures	VI
List of Tables	VII

CHAPTER 1 INTRODUCTION AND MOTIVATION………..1-6

1.1 Introduction	1
1.2 Motivation	2
1.3 Objectives	2
1.4 Tools Used	3
1.5 Modelsim Overview	
1.5.1 Creating a Working Library	3
1.5.2 Compiling the Design	4
1.5.3 Loading The Design into the Simulator	4
1.5.4 Run and Simulation	4
1.6 Dissertation Outline	6

CHAPTER 2 FLOATING POINT ARITHMATIC……………..7-15

2.2 Basic of Floating Point Numbers	7
2.2 Floating Point Addition	8
2.2.1 Floating Point Data Format	9
2.2.2 Exceptions	10
2.2.3 Rounding Modes	13
2.2.4 Normalisation	15

CHAPTER 3 FLOATING POINT MULTIPLIER……………..16-27

3.1 Floating Point Multiplication 17

3.1.1 Booth Algorithm 18

3.1.2 Vedic Multiplication Technique 20

3.1.3 Karastuba Multiplication 22

3.1.4 Modified Booth Encoding Algorithm (Radix-4 Algorithm) 24

CHAPTER 4 INTRODUCING MUX WITH VEDIC MULTIPIER……………………………………...28-37

4.1 Design of MUX Based 2x2 Multiplier 29

4.2 Design of MUX Based 3x3 Multiplier 30

4.3 Proposed 8x8 Multiplier Based on MUX and Vedic Concept 31

4.4 Look Ahead Adder 34

4.5 Proposed 16x16 Multiplier Using 8x8 Multiplier 37

CHAPTER 5 RESULTS AND VERIFICATION……………….38-40

5.1 Results 39

5.2 Verification 40

CHAPTER 6 CONCLUSION AND FUTURE SCOPE OF WORK ……………………………………………41-42

6.1 Conclusion 41

6.2 Future Scope 41

REFERENCES……………………………………………………42

LIST OF FIGURES

Figure No.	Title of Figure	Page No.
Figure 2a	Floating Point Data Format	8
Figure 2b	Rounding to nearest even	10
Figure 2c	Block Diagram of Floating Point adder	12
Figure 3a	Multiplication of two 4 bit numbers using Urdhava Triyakbhyam method	14
Figure 3b	Block diagram of 24x24 BIT Vedic multiplier	20
Figure 3c	Hardware architecture of 4 X 4 Urdhva Tiryak- -bhyam multiplier	21
Figure 3d	Architecture of Booth Multiplier	24
Figure 4a	Schematic of a 2-to-1 Multiplexer	28
Figure 4b	MUX based 2x2 multiplier	30
Figure 4c	MUX based 3x3 multiplier	31
Figure 4d	Line diagram of the proposed Vedic multiplier for 8x 8 bit multiplications.	32
Figure 4e	Architecture of 8X8 bit multiplier using 3X3, 3X2 and 2X2 bit multiplier blocks	33
Figure 4f	Logic schematic of 4-bit carry	35
Figure 4g	Block diagram of a 16- bit carry look-ahead adder	36
Figure 4h	16x16 bit multiplier using 8x8 bit multiplier block	37
Figure 5a	Output of single precision Floating Point number when above inputs were given	39

LIST OF TABLES

Table No.	Title of Table	Page No.
Table 2a	Layout for single and double precision	8
Table 2b	Details for Binary Floating Point Numbers	10
Table 4a	Truth Table of 2x2 MUX based Multiplier	29

CHAPTER 1

INTRODUCTION

1. INTRODUCTION AND MOTIVATION

1.1 INTRODUCTION

Floating point multiplication is considered an abstruse subject though it is found everywhere in computer systems. Day by day IC technology is getting more complex in terms of design and its performance analysis. A faster design with lower power consumption and smaller area is implicit to the modern electronic designs. Unceasing advancement in microelectronics design technology makes improved use of energy, encrypt data successfully, communicate information much more steadfastly, etc. Particularly, many of these technologies address low-power consumption to meet the requirements of various portable applications. In these application systems, a multiplier is a fundamental arithmetic unit and widely used in circuits, for which the multiplication process should be optimized properly.

Multipliers generally have extended latency, huge area and consume substantial amount of power. Hence low-power multiplier design has become an important part in VLSI system design. Everyday new approaches are being developed to design low-power multipliers at technological, physical, circuit and logic levels. Since the multiplier is generally the slowest element in a system, the system's performance is determined by performance of the multiplier. Also multipliers are the most area consuming entity in a design. Therefore, optimizing speed and area of a multiplier is a major design issue nowadays. However, area and

speed are usually conflicting constraints so that improving speed results in larger areas and vice-versa. Also area and power consumption of a circuit are linearly correlated. So a compromise has to be done in speed of the circuit for a greater improvement in reduction of area and power.

For implementing a digital multiplier a large variety of computer arithmetic algorithms could be used. Most techniques take into consideration generating a set of partial products, and then adding the partial products together once they have been shifted. In a multiplier to increase its speed, the number of partial product to be generated should be reduced. A higher representation radix effectively indicates to fewer digits. Thus, a single-digit multiplication algorithm necessitates fewer cycles as we start moving to much higher radices, which automatically leads to a lesser number of partial products. Several algorithms have been developed for this purpose like Booth's Algorithm, Wallace Tree method etc. For the summation process several adder architectures are available viz. Ripple Carry Addition, Carry Look-ahead Addition, Carry Save Addition etc. But to reduce the power consumption the summation architecture of the multiplier should be carefully chosen.

1.2 MOTIVATION

Constraints in representation of mathematical values using existing precision bring about the necessity for cores that can manipulate single precision floating point numbers. The single precision cores for multiplication discussed in this thesis.

Implementation of 32- bit multiplier is an important part of this thesis. The implementation uses ModelSim SE and same for simulation of VHDL source code synthesis.

1.3 OBJECTIVES

The objective is to design a 32 bit single precision floating point unit operating on the IEEE 754 standard floating point representations, supporting the three basic arithmetic operations: addition, subtraction and multiplication. Second objective is to model the behavior of the Floating point adder and multiplier design using VHDL.

The programming objective of the floating point applications fall into the following categories:

- Accuracy: the application produces the results that are close to the correct results.
- Performance: The application produces the most efficient code possible.
- Reproducibility and portability: The application produces the results that are consistent across different runs, different set of built options, different compilers, different platforms and different architecture.
- Latency: The application produces a single output with in less time.
- Throughput: The application produces more number of tasks that can be completed per unit time.
- Area: The application produces less number of flip flops and slices.

1.4 TOOLS USED

The tools used in the thesis are as follows:

Simulation Software:

ModelSim 13.2 is used for modelling and simulation.

Hardware : Xylinx Spartan 3E.

1.5 MODELSIM OVERVIEW

ModelSim is a tool that integrates with Xilinx ISE to provide simulation and testing. Two kinds of simulation are used for testing a design: functional simulation and timing simulation. Functional simulation is used to make sure that the logic of a design is correct. Timing simulation also takes into account the timing properties of the logic and the FPGA, so you can see how long signals take to propagate and make sure that your design will behave as expected when it is downloaded onto the FPGA.

1.5.1 CREATING A WORKING LIBRARY

Before one can simulate a design, they must first create a library and compile the source code into that library. “Work” is the library name that the compiler use as a default destination for compiled design units.

1.5.2 COMPILING THE DESIGN

Before the simulation of design, a library must be created and the source code must be compiled into that library.

1. Copy the design files for this lesson into a newly created directory and start by creating a new directory for this exercise.
2. Start ModelSim if necessary.
    - Type vsim at a UNIX shell prompt and for windows use the ModelSim icon.

- Select **File> Change** Directory and switch to the directory that was created in first step.

3. Create the Working Library

   Select **File> New> Library.** Specify the logical and physical names for the library here in the new opened dialog. You can also create a new directory to the existing library.

   Specify “work” in the library name and click **OK**

1.5.3 LOADING THE DESIGN INTO THE SIMULATOR

1. Load the test_counter module into the simulator.

   a. In the Library window, click the ‘+’ sign next to the work library to show the files contained there.

   b. Double-click test_counter to load the design.

You can also load the design by selecting **Simulate > Start Simulation** in the menu bar. This opens the Start Simulation dialog. With the Design tab selected, click the ’+’ sign next to the work library to see the counter and test_counter modules. Select the test_counter module and click **OK**.

When the design is loaded, a Structure window opens (labeled sim). This window displays the hierarchical structure of the design You can navigate within the design hierarchy in the Structure (sim) window by clicking on any line with a ’+’ (expand) or ’-’ (contract) icon.

1.5.4 RUN AND SIMULATION

We’re ready to run the simulation. But before we do, we’ll open the Wave window and add signals to it.

1. Open the Wave window.

a. Enter view wave at the command line.

The Wave window opens in the right side of the Main window. Resize it so it is visible.

You can also use the **View > Wave** menu selection to open a wave window. The Wave window is just one of several debugging windows available on the View menu.

2. Add signals to the Wave window.

a. In the Structure (sim) window, right-click test_counter to open a popup context menu.

b. Select **Add > To Wave > All items in region**.

All signals in the design are added to the Wave window.

3. Run the simulation.

a. Click the Run icon.

The simulation runs for 100 ns (the default simulation length) and waves are drawn in the Wave window.

b. Enter run 500 at the VSIM> prompt in the Transcript window.

The simulation advances another 500 ns for a total of 600 ns.

c. Click the Run -All icon on the Main or Wave window toolbar.

The simulation continues running until you execute a break command or it hits a statement in your code (e.g., a Verilog $stop statement) that halts the simulation.

1.1 DISSERTATION OUTLINE

Rest of the dissertation is organized as follows:

Chapter 2 includes the floating point arithmetic, basic architecture and functionality.

Chapter 3 includes the floating point multipliers' architecture and functionality and describes the different design modules of the block diagram of different multipliers. In this chapter the denormals, exceptions and rounding modes are also covered.

Chapter 4 The tool ModelSim SE used for verification and then design of single precision floating point multiplier is discussed.

Chapter 5 The design and implementation of 32 bit single presicion floating point multiplier using MUX and Vedic Multiplier is discussed.

Chapter 6 Results and their verification is discussed in this chapter.

CHAPTER 2

FLOATING POINT ARITHMATIC

2.1 BASIC OF FLOATING POINT NUMBERS

In computing, floating point describes a method of representing an approximation to real numbers in a way that can support a wide range of values. The numbers are, in general, represented approximately to a fixed number of significant digits (the mantissa) and scaled using an exponent. The typical number that can be represented exactly is of the form:

$$\text{Significant digits} \times \text{base}^{\text{exponent}}$$

The idea of floating-point representation over intrinsically integer fixed-point numbers, which consist purely of significant, is that expanding it with the exponent component achieves greater range. For instance, to represent large values, e.g. distances between galaxies, there is no need to keep all 39 decimal places down to femtometre -resolution and thus saving in computer data storage. However, since the 1990s, the most commonly encountered representation is that defined by the IEEE 754 Standard. [1]

2.1.1 IEEE-754 STANDARD FOR FLOATING POINT ARITHMETIC

The IEEE 754 standard for binary floating-point arithmetic provides a precise specification of floating-point number formats computation operations, and exceptions and their handling. The combination of a vast range of inputs, special cases, and rounding modes makes the hardware implementation of fully IEEE 754 standard compliant floating-point arithmetic a very challenging task. Ignoring certain aspects of the standard can lead to unexpected consequences in the context of numerical algorithms.

Hence, most floating-point hardware is IEEE-compliant or has an IEEE-compliant mode.

2.1.1.1 FLOATING POINT DATA FORMAT

IEEE floating point numbers have three basic components: the sign, the exponent, and the mantissa as shown in figure 1a. Depending on type single precision, double precision or quadruple precision the number bits of these fields are defined.

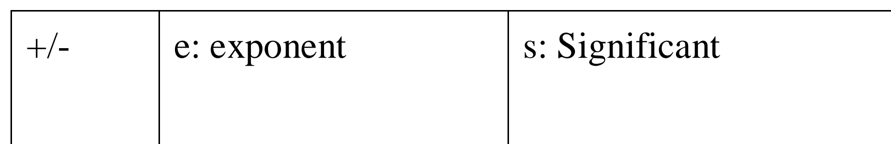


Figure 2a: Floating Point Data Format

The following figure shows the layout for single (32-bit) and double (64-bit) precision floating-point values. The number of bits for each field are shown (bit ranges are in square brackets):

	Sign	**Exponent**	**Fraction**	**Bias**
Single Precision	1 [31]	8 [30-23]	23 [22-00]	127
Double Precision	1 [63]	11 [62-52]	52 [51-00]	1023

Table 2a: Layout for single and double precision

The Sign Bit

The sign bit is as simple as it gets. 0 denotes a positive number and 1 denotes a negative number. Flipping the value of this bit flips the sign of the number.

The Exponent

The exponent field needs to represent both positive and negative exponents. To do this, a *bias* is added to the actual exponent in order to get the stored exponent. For IEEE single-precision floats, this value is 127. Thus, an exponent of zero means that 127 is stored in the exponent field. A stored value of 200 indicates an exponent of (200-127), or 73. For double precision, the exponent field is 11 bits, and has a bias of 1023.

The Mantissa

The mantissa, also known as the significant, represents the precision bits of the number. It is composed of an implicit leading bit and the fraction bits. To find out the value of the implicit leading bit, consider that any number can be expressed in scientific notation in many different ways. For example, the number five can be represented as any of these:

5.00×10^0

0.05×10^2

5000×10^{-3}

In order to maximize the quantity of representable numbers, floating-point numbers are typically stored in normalized form. This basically puts the radix point after the first non-zero digit. In normalized form, five is represented as 5.0×10^0. A nice little optimization is available to us in base two, since the only possible non-zero digit is 1. Thus, we can just assume a

leading digit of 1, and don't need to represent it explicitly. As a result, the mantissa has effectively 24 bits of resolution, by way of 23 fraction bits.

The precision, maximum and minimum value of exponent for a binary floating point numbers are shown in the following table 2.

PARAMETER	FORMAT	
	SINGLE PRECISION	DOUBLE PRECISION
Format width in bits	32	64
Precision(P)= fraction + hidden bits	23+1	52+1
Exponent width in bits	8	11
Maximum value of exponent	+127	+1023
Minimum value of exponent	-126	-1022

Table 2b : Details for Binary Floating Point Numbers

2.1.1.2 EXCEPTIONS

In IEEE 754 arithmetic standards an exception can be signalled along with the result of an operation. This can take the form of a status flag (which must be "sticky," so that the user does not need to check it immediately, but after some sequence of operations, for instance at the end of a function) and/or some trap mechanism.

Invalid: This exception is signalled when an input is invalid for the function. The result is a NaN (when supported). Examples: (+1) − (+1), 0/0, p −1.

Divide By Zero: This exception is signalled when an exact infinite result is defined for a
function on finite inputs, e.g., at a pole. Examples: 1/0.
Overflow: This exception is signalled when the rounded result with an unbounded exponent
range would have an exponent larger than Emax.

Underflow: This exception is signalled when a tiny (less than E_{min}) nonzero result is detected. Underflow handling can be different whether the exact result is exactly representable or not, which makes sense: the major interest in signalling the underflow exception is to warn the user that the obtained result might not be very accurate (in terms of relative error). Of course, this is not the case when the obtained result is exact. This is why, in the IEEE 754-2008 standard, if the result of an operation is exact, then the underflow flag is not raised.

Inexact: This exception is signalled when the exact result y is not exactly representable (y not being a NaN).

2.1.1.3 ROUNDING MODES

In general, the result of an operation (or function) on floating-point numbers is not exactly representable in the floating point system being used, so it has to be rounded. For this three bits are to be added internally and temporally to the actual fraction: guard, round and sticky bit. While the guard bit and round bits are normal storage holders, the sticky bit is turned ‘1’ whenever a ‘1’ is shifted out of range. The standard specifies four rounding modes:

Rounding to nearest (even), rounding towards zero, rounding to +∞, rounding to -∞.

• **Round to nearest even**: This is the stamdard default rounding. The value is rounded up or down to the nearest infinitely precise result. If the value is exactly the half way between two infinitely precise results, then it should be rounded up to the nearest infinitely precise even.

For example,

Unrounded	Rounded
3.4	3
5.6	6
3.5	4
2.5	2

Figure2c Rounding to nearest even

- **Round towards zero:** Basically in this mode number is not rounded. The excess bits are simply truncated. For example, 3.47 will betruncated to 3.4.
- **Round to** +∞ (Round up)**:** The number will be rounded up to +∞. For example, 3.2 will be rounded to 4.
- **Round to** -∞ (Round down): the opposite of round up, the number will be rounde towards -∞. For example, 3.2 will be rounded to 3 while -3.2 to -4.

2.2 FLOATING POINT ADDITION

Floating point addition is the operation most frequently used. Floating point adders are very important components in microprocessors and Digital Signal Processors. Due to the large sequentially dependent operations required for a

single addition, the design of Floating Point adders is considered to be the most difficult.

Standard Floating Point addition requires following steps:

- Exponent Difference.
- Pre-shift for mantissa alignment.
- Mantissa addition/ subtraction.
- Post shift for result normalization.
- Rounding.

In Floating Point additions, the exponent of the larger number is chosen as the tentative exponent of the result. Exponent equalisation of the smaller floating point number to that of larger number demands the shifting of the smaller number's significand through an appropriate number of bit positions. The absolute value of difference between the exponents of the numbers decides the magnitude of alignment shift. Addition of significands is essentially a signed a signed magnitude form. Signed magnitude addition of significand can lead to the generation of a carry out from the MSB position of the significand or the generation of leading zeros or even a zero result.

Normalization shifts are essential to restore the result of the signed magnitude significand addition into standard form. Rounding of normalised significands in the last step in the whole addition process. Rounding demands a conditional incrementing of the normalized significand. The operation of rounding, by itself can lead to the generation of carry out from the MSB position of the normalized significand. That means, the rounded significand needs to be subjected to a correction shifting in certain conditions.

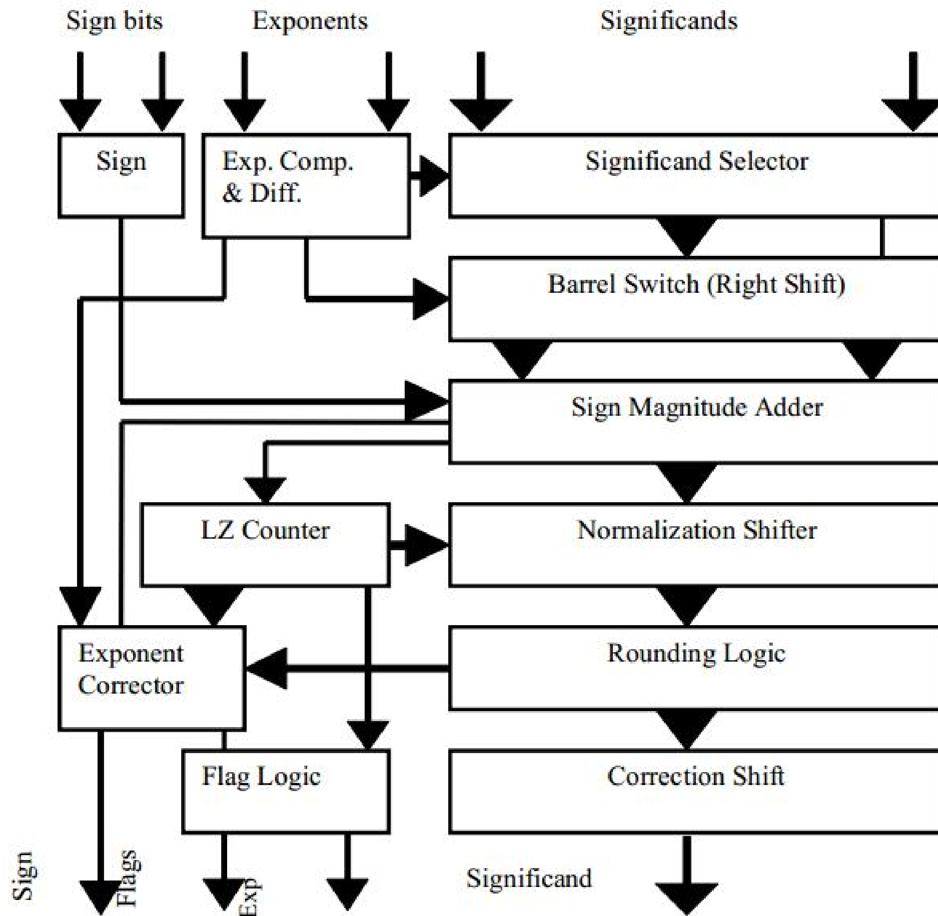


Figure 2c: Block Diagram of floating point adder

Figure 2.1 illustrates the block diagram of a floating point adder. The exponent comparator and differentiator block evaluates the relative magnitudes of the exponents as well as the difference between them. The significand selector block routes the significand of the larger number to the adder while routing the significand of the smaller number to the barrel switch. The barrel switch performs the requisite pre-alignment shift (right shift). The sign magnitude adder performs significant addition and subtraction, the result of which is again represented in sign magnitude form. The leading zero counter evaluates the number of leading zeros and encodes it into a binary number. This block, essentially controls normalization shifts. The rounding logic evaluates rounding conditions and performs significand correction whenever the rounding is necessary. The correction shift logic

performs a right shift of the rounded significand if the process of rounding has resulted in the generation of a carry out from the MSB position of the significand. The exponent corrector block evaluates the value of the exponent of the result. The flag logic asserts various status flags subject to the validity of various exception conditions.

2.2.1 NORMALISATION

Normalisation step left shift the significand to obtain leading “1” in the MSB, and adjusts the exponent by the normalisation distance. For true subtraction,the position of the leading “1” of the significand result is predicted from the input significands to within 1-bit using an LZA [19,20] which is calculated concurrently with the significand addition. The normalization distance is passed to a normalization barrel shifter and subtracted from the exponent.

Example:

-100100101001.0012 (binary)

= -1.00100101001001 x 2^11 (normalized binary)

CHAPTER 3

FLOATING POINT MULTIPLIER

3.1 FLOATING POINT MULTIPLICATION

With the constant growth of computer applications such as signal processing and computer graphics, fast arithmetic units such as multipliers are increasingly required. Advances in VLSI technology have given designers the freedom to integrate many complex components, which was not possible in past. Multipliers are also required in many digital signal processing operations such as correlation, convolution, and filtering and frequency analysis, to perform multiplication. The most basic form of multiplication consists of forming the product of two positive binary numbers. This can be accomplished through traditional technique of successive additions and shift in which each addition is conditional on one of the multiplier bits. Thus the multiplication can be viewed as consisting of the following two steps:

1. Evaluation of partial products
2. Accumulation of shifted party products.

Various algorithms and multiplication techniques have been worked on for floating point multiplier. Following are some of those discussed here.

3.1.1 BOOTH ALGORITHM

There are various techniques used to perform multiplication. Booth multipliers allow the operation on signed operands in 2's complement. They derive from array multipliers where, for each bit in a partial product line, an encoding scheme is used to determine if this bit is positive, negative or zero.

The Modified Booth algorithm achieves a major performance improvement through radix-4 encoding. In this algorithm each partial product line operates on 2 bits at a time, thereby reducing the total number of the partial products. This is particularly true for operands using 16 bits or more.

The fastest types of multipliers are parallel multipliers. Among these, the Wallace multiplier is among the fastest. However, they suffer from a bad regularity. Hence, when regularity, high performance and low power are primary concerns, Booth multipliers tend to be the primary choice.

In 2007, Faycal Bensaali, Abbes Amira and Reza Sotudeh proposed a floating point multiplier that used a modified Booth's algorithm which is an effective algorithm to reduce the number of partial product to be added by factor of two and hence increasing the speed. [3]

Let M_1 and M_2 be the mantissas of the two floating point numbers to be multiplied. m = m1 £m2 is the mantissa of the multiplication result. Since the mantissas are positive numbers and have a constant word-length W, modified Booth encoding can be further improved. The unsigned representation of m2 is as follows:

$$m_2 = \sum_{x=0}^{W-1} m_{2,x} \times 2^x \qquad (1)$$

Equation (1) can be re-written as:

$$m_2 = \sum_{y=0}^{\frac{W}{2}-1} (m_{2,2y} + 2 \times m_{2,2y+1}) \times 2^{2y} \qquad (2)$$

or

$$m_2 = \sum_{y=0}^{\frac{W}{2}-1} (D_y) \times 4^y \qquad (3)$$

Where, Dy = (m2;2y + 2 £m2;2y+1) and Dy 2 f0; 1; 2; 3g

Using equation (3), the mantissas product m can be computed as follows:

$$m = \sum_{y=0}^{\frac{w}{2}-1} m_1 \times D_y \times 4^y = \sum_{y=0}^{\frac{w}{2}-1} PP_y \times 4^y \quad (4)$$

By using the unsigned representation for the mantissas, the three following points, which are part of the standard modified both algorithm, have been avoided:

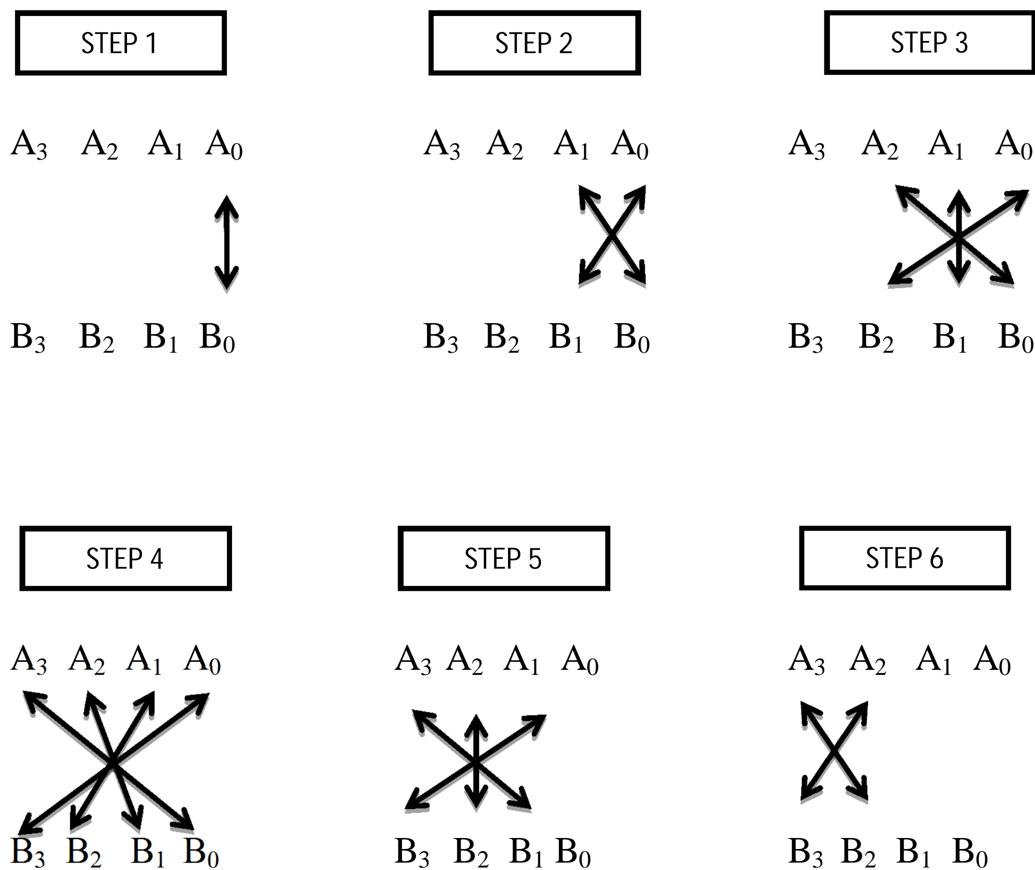
1. Extend the sign bit to ensure that the word-length of the multiplicand is even.
2. Append a 0 to the right of the LSB of the multiplicand.
3. Compute the two's complement of the multiplier.

3.1.2 VEDIC MULTIPLICATION TECHNIQUE

Vedic mathematics is based on 16 Sutras (or aphorisms) dealing with various branches of mathematics like arithmetic, algebra, geometry etc. one of the vedic sutra Urdhava Triyakbhyam deals with the multiplication of numbers.
Urdhava Tiryakbhyam [2] is a Sanskrit word which means vertically and crosswire in English. The method is a general multiplication formula applicable to all cases of multiplication. It is based on a novel concept through which all partial products are generated concurrently.
This type of multiplier is independent of the clock frequency of the processor because the partial products and their sums are calculated in parallel. The net advantage is that it reduces the need of microprocessors to operate at increasingly higher clock frequencies. As the operating frequency of a processor increases the number of switching instances also increases. This results in more power consumption and also dissipation in the form of heat which results in higher device operating temperatures.

Another advantage of Urdhva Tiryakbhyam multiplier is its scalability. The processing power can easily be increased by increasing the input and output data bus widths since it has a regular structure [3]. Due to its regular structure, it can be easily layout in a silicon chip and also consumes optimum area [2]. As the number of input bits increase, gate delay and area increase very slowly as compared to other multipliers. Therefore Urdhava Tiryakbhyam multiplier is time, space and power efficient.

The line diagram in fig. 2 illustrates the algorithm for multiplying two 4-bit binary numbers a3a2a1a0 and b3b2b1b0. The procedure is divided into 7 steps and each step generates partial products.



STEP 7

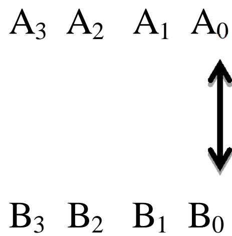


Figure 3.1.2a: Multiplication of two 4 bit numbers using Urdhava Triyakbhyam method [7].

Initially as shown in step 1 of fig. 2, the least significant bit (LSB) of the multiplier is multiplied with least significant bit of the multiplicand (vertical multiplication). This result forms the LSB of the product. In step 2 next higher bit of the multiplier is multiplied with the LSB of the multiplicand and the LSB of the multiplier is multiplied with the next higher bit of the multiplicand (crosswire multiplication). These two partial products are added and the LSB of the sum is the next higher bit of the final product and the remaining bits are carried to the next step.

For example, if in some intermediate step, we get the result as 1101, then 1 will act as the result bit(referred as rn) and 110 as the carry (referred as cn). Therefore cn may be a multi-bit number. Similarly other steps are carried out as indicated by the line diagram. The important feature is that all the partial products and their sums for every step can be calculated in parallel.

Thus every step in fig. 2 has a corresponding expression as follows:

$$r0=a0b0. \quad (1)$$

$$c_1r_1=a_1b_0+a_0b_1. \quad (2)$$

$$c_2r_2=c_1+a_2b_0+a_1b_1 + a_0b_2. \quad (3)$$

$$c_3r_3=c_2+a_3b_0+a_2b_1 + a_1b_2 + a_0b_3. \quad (4)$$

$c_4r_4=c_3+a_3b_1+a_2b_2 + a_1b_3$. (5)

$c_5r_5=c_4+a_3b_2+a_2b_3$. (6)

$c_6r_6=c_5+a_3b_3$ (7)

With $c_6r_6r_5r_4r_3r_2r_1r_0$ being the final product [5].

Hence this is the general mathematical formula applicable to all cases of multiplication and its hardware architecture is shown in fig. 3

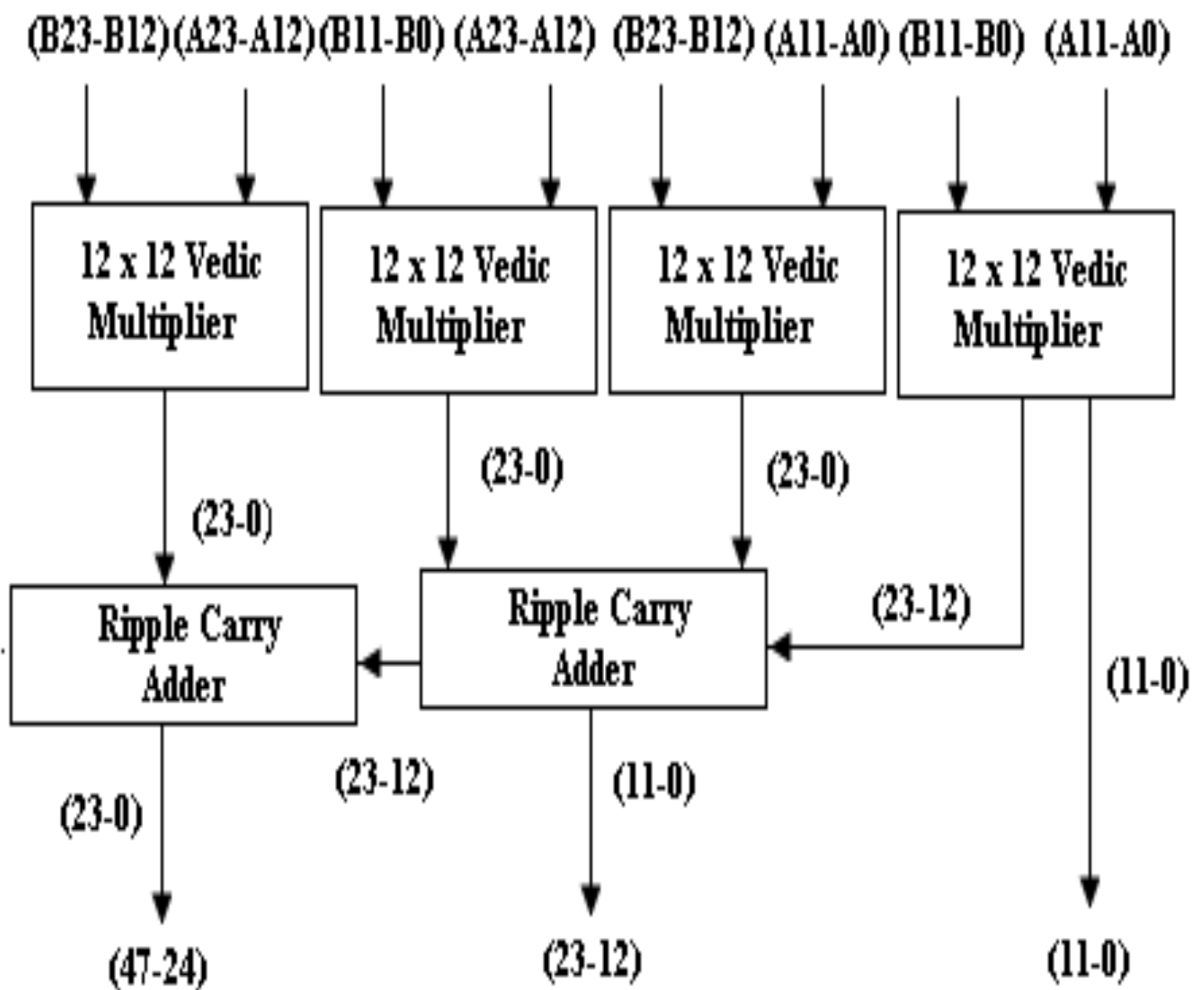


Figure3.1.2b: Block diagram of 24x24 BIT Vedic multiplier [5]

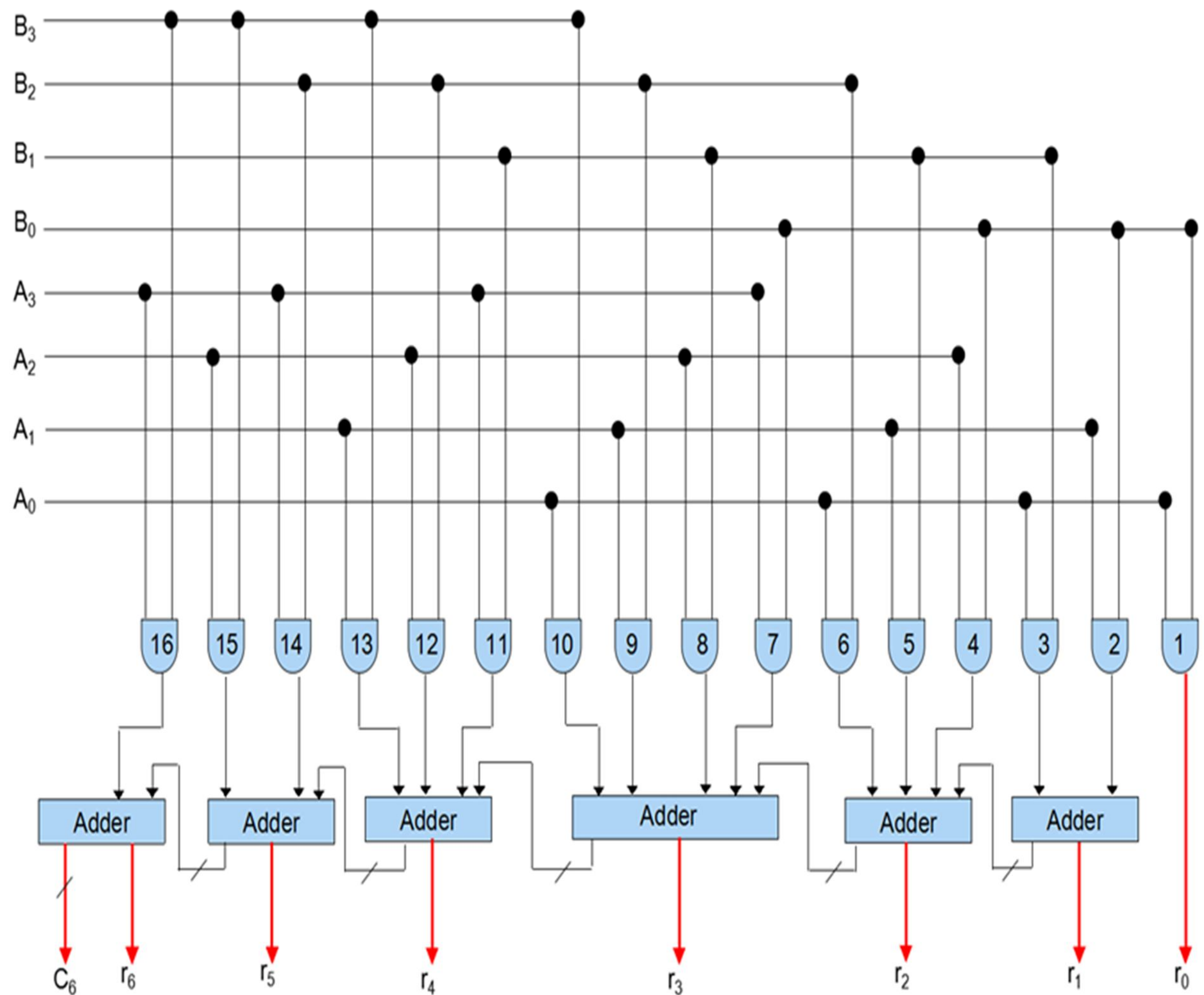


Figure 3.1.2c: Hardware architecture of 4 X 4 Urdhva Tiryakbhyam multiplier.

In order to multiply two 8-bit numbers using 4-bit multiplier we proceed as follows. Consider two 8 bit numbers denoted as AHAL and BHBL where AH and BH corresponds to the most significant 4 bits, AL and BL are the least significant 4 bits of an 8-bit number.

When the numbers are multiplied multiplied according to Urdhava Tiryakbhyam (vertically and crosswire) method, we get,

AH AL

BH BL

(AH x BH) + (AH x BL + BH x AL) + (AL x BL)

Thus we need four 4-bit multipliers and two adders to add the partial products and 4-bit intermediate carry generated. Since product of a 4 x 4 multiplier is 8 bits long, in every step the least significant 4 bits correspond to the product and the remaining 4 bits are carried to the next step. This process continues for 3 steps in this case. Similarly, 16 bit multiplier has four 8 x 8 multiplier and two 16 bit adders with 8 bit carry. Therefore we see that the multiplier is highly modular in nature. Hence it leads to regularity and scalability of the multiplier layout.

3.1.3 KARATSUBA MULTIPLICATION

The basic step of Karatsuba's algorithm is a formula that allows us to compute the product of two large numbers *x* and *y* using three multiplications of smaller numbers, each with about half as many digits as *x* or *y*, plus some additions and digit shifts.

Let *x* and *y* be represented as *n*-digit strings in some base *B*. For any positive integer *m* less than *n*, one can write the two given numbers as

$$x = x_1B^m + x_0$$

$$y = y_1B^m + y_0,$$

where x_0 and y_0 are less than B^m. The product is then

$$xy = (x_1B^m + x_0)(y_1B^m + y_0)$$

$$= z_2B^{2m} + z_1B^m + z_0$$

where

$$z_2 = x_1y_1$$

$$z_1 = x_1y_0 + x_0y_1$$

$$z_0 = x_0y_0.$$

These formulae require four multiplications, and were known to Charles Babbage. Karatsuba observed that xy can be computed in only three multiplications, at the cost of a few extra additions. With z_0 and z_2 as before we can calculate

$$z_1 = (x_1 + x_0)(y_1 + y_0) - z_2 - z_0$$

which holds since

$$z_1 = x_1y_0 + x_0y_1$$

$$z_1 = (x_1 + x_0)(y_1 + y_0) - x_1y_1 - x_0y_0.$$

A more efficient implementation of Karatsuba multiplication can be set as

$$x\,y = (b^2 + b)x_1y_1 - b(x_1 - x_0)(y_1 - y_0) + (b + 1)x_0y_0,$$

where b is the weight of x_1.

Karatsuba Multiplication is a fast-multiplication algorithm. It reduces the multiplication of two n-digit numbers from simple n^2 to at most $3n^{\log_2 3}$. The basic steps for this algorithm depends on divide-and conquer paradigm and proceeds in the following way. [6]

Let W and X be two n-digit numbers. By breaking these numbers for some base B, we can write them as below:

$$W = W1.B^m + W0$$

$$X = X1.B^m + X0$$

Where W0 and X0 are of m-digit. Now, we can write the product of W and X as

follows:

$$WX = [W1.B^m + W0][X1.B^m + X0]$$

$= W1.X1.B^{2m} + (W1.X0 + W0.X1)B^{m} + W0.X0$

$=\alpha.B^{2m} + \beta.B^{m} + \gamma$

Where α =W1.X1, β =W1.X0+W0.X1

and γ =W0.X0.

Thus, as a whole at this we need four multiplications to get the complete result.

But, as per Karatsuba, we need only three multiplications to get the complete result.

This happens as follows. We can modify the β as below:

β = (W1.X0+W0.X1) + (W1.X1+W0.X0) − (W1.X1+W0.X0)

= (W1+W0) (X1+X0) − W1.X1 − W0.X0

= (W1+W0) (X1+X0) − α − γ

This requires only one multiplication instead of two, with some extra overhead of Addition and subtraction. Thus to get a complete product of W and X we need three multiplications instead of four.

3.1.4 MODIFIED BOOTH ENCODING ALGORITHM (RADIX-4 ALGORITHM)

A modification of the Booth algorithm was proposed in which a triplet of bits is scanned instead of two bits. This technique has the advantage of reducing the number of partial products by one half regardless of the number of inputs. The recoding is done in two steps: encoding and selection. The purpose of encoding is to scan the triplet of bits of the multiplier and define the operation to be performed on the multiplicand.

For a $m\times n$- bits multiplication, the booth algorithm produces n/2[(n+1)/2, if n is odd] partial products, each has a length of (m+1) bits.

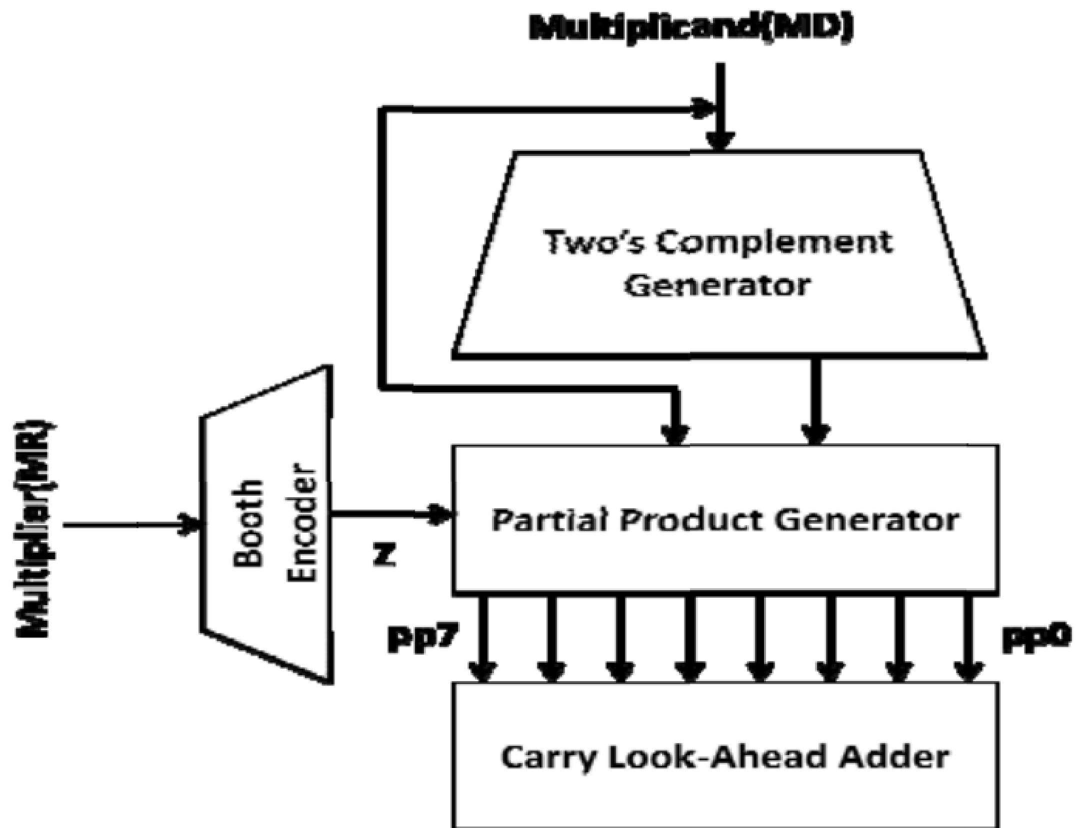


Figure 3.1.4a : Architecture of Booth Multiplier.

This can half the number of partial products. It reduces the number of adders by 50% which which results in a higher speed, a lower power dissipation, and a smaller area than a conventional multiplication array.

One more method of increasing speed is to use pipelined or parallel multiplier. Malte Baesler and Thomas Teufel presented a parallel decimal fixed-point multiplier, designed to exploit the features of FPGAs. Their multiplier was based on BCD recoding schemes, fast partial product generation and a BCD-4221 Carry Save Adder reduction tree. [7]

Manish Kumar Jaiswal and Nitin Chandra choodan have presented an efficient architecture based on the idea of partial block multiplication, for implementation of double precision floating point multiplication on FPGAs.

The proposed module is taking less latency and achieves high performance compared to other modules previously reported in the literature.
In addition, it also requires less hardware resources (MULT18x18 block and slices) compared to others. [8]

Numerical problems are usually formulated in decimal notation. Therefore, in order to avoid conversion errorsduring input and output, the problems should be solved with a decimal floating-point arithmetic. Because of the increasing importance, specifications for decimal floatingpointarithmetic have been added to theIEEE 754-2008 Standard for Floating-Point Arithmetic. [1]

A hardwired algorithm for computing the variable precision multiplication was presented in a paper Hardware Algorithm for Variable PrecisionMultiplication on FPGA. The computation method was based on the use of a parallel multiplier of size m to compute the multiplication of two numbers of n×m bits. These numbers are represented in the variable precision floating point format, but in this work only the mantissas are considered; the exponents are easily obtained by adding the exponents of the two operands to be multiplied. In this computing method of multiplication, the partial products are added as soon as they are computed, resulting in the use of the lowest memory for intermediate results storage, (i.e. the size of the result is of 2n×m bits).
The hardware implementation of a high speed floating point multiplier with pipeline architecture based on FPGA is presented in the paper. In the design of the floating point multiplier, the utilization of a new Radix-4 Booth"s encoding algorithm, the improved 4:2 compression structure and summation circuit is made to implement the compression of the partial products, and the sum and carry vectors are added by a final carry look-ahead adder to obtain

the product. The timing simulation results show that the floating point multiplier can be steadily run at the frequency of 80 MHz.[9]

In [10] VHDL is used to implement a technology-independent pipelined design of a floating point multiplier which handles the overflow and underflow cases without supporting rounding. L.Louca, T.A.Cook, W.H. Johnson [11] implemented a single precision floating point multiplier by using a digit-serial multiplier and Altera FLEX 8000. The design achieved 2.3 MFlops and doesn't support rounding modes.

In [12], a parameterizable floating point multiplier is implemented using five stages pipeline, Handel-C software and Xilinx XCYIOOO FPGA. The design achieved the operating frequency of 28MFlops.

The floating point unit [13] is implemented using the primitives of Xilinx Yirtex IT FPGA. The design achieved the operating frequency of 100 MHz with a latency of 4 clock cycles.

Mohamed AI-Ashraf, Ashraf Salem, and Wagdy Anis [14] implemented an efficient IEEE- 754 single precision floating point multiplier and targeted for Xilinx Yirtex-5 FPGA. The multiplier handles the overflow and underflow cases but rounding is not implemented. The design achieves 30 I MFLOPs with latency of three clock cycles. The multiplier was verified against Xilinx floating point multiplier core.

CHAPTER 4

INTRODUCING MUX WITH VEDIC MULTIPIER

In this section, we have introduced new concept of MxN bit multiplication based on multiplexer and UrdhvaTiryakbyham sutra (vertically and crosswise)Vedic concept. The disadvantage of direct multiplication in using UrdhvaTiryakbyham Vedic concept for higher order bits require more number of carry propagation results in more delay. The multiplexer based multiplier with UrdhvaTiryakbhyam Sutra concept eliminated delay and minimizes IC package count.

4.1 DESIGN OF MUX BASED 2X2 MULTIPLIER

A multiplexer (or mux) is a device that selects one of several analog and digital input signals and forwards the selected input into a single line. A multiplexer of 2^n inputs has n select lines, which are used to select which input line to send to the output. Multiplexers are mainly used to increase the amount of data that can be sent over the network within a certain amount of time and bandwidth. A multiplexer is also called a data selector.

An electronic multiplexer makes it possible for several signals to share one device or resource, for example one A/D convertor one communication line, instead of having one device per input signal.

The following figure shows the schematic of a 2-to-1 Multiplexer. When the select signal is low the output follows the first input and if the high then the output follows the second input.

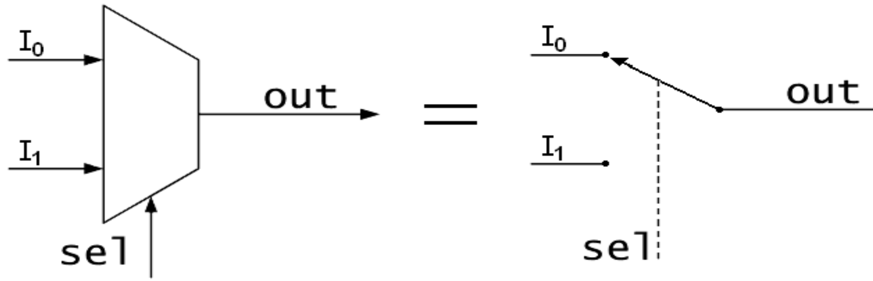


Figure 4.1a : Schematic of a 2-to-1 Multiplexer

The truth table of 2x2 mux based multiplier is given in Table 4.1a.

- The four input lines for multiplier B with two bits for 2x2 multiplier and three bits for 3x2 multiplier are considered with A multiplicand having two controls S0 and S1.
- The first line input of multiplier B is always s 00 or 000. The second, third and fourth line input values are any combinations of two or three bits based on either 2x2 or 3x2 multiplier .
- The third line input is connected to shift left by one shifter and fourth input line is connected to shift left by one alone with adder to get proper multiplication results in the output.

A Multiplicand		B Multiplier		Output Y			
S1	S0	B1	B0	Y3	Y2	Y1	Y0
0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0
0	1	0	1	0	0	0	1
0	1	1	0	0	0	1	0
0	1	1	1	0	0	1	1
1	0	0	0	0	0	0	0
1	0	0	1	0	0	1	0
1	0	1	0	0	1	0	0

1	0	1	1	0	1	1	0
1	1	0	0	0	0	0	0
1	1	0	1	0	0	1	1
1	1	1	0	0	1	1	0
1	1	1	1	1	0	0	1

Table 4.1a Truth Table of 2x2 MUX based Multiplier.

The 2x2 MUX based multiplier using 4:1 multiplexer as shown in Figure 4.1b A as multiplicand and B as multiplier.

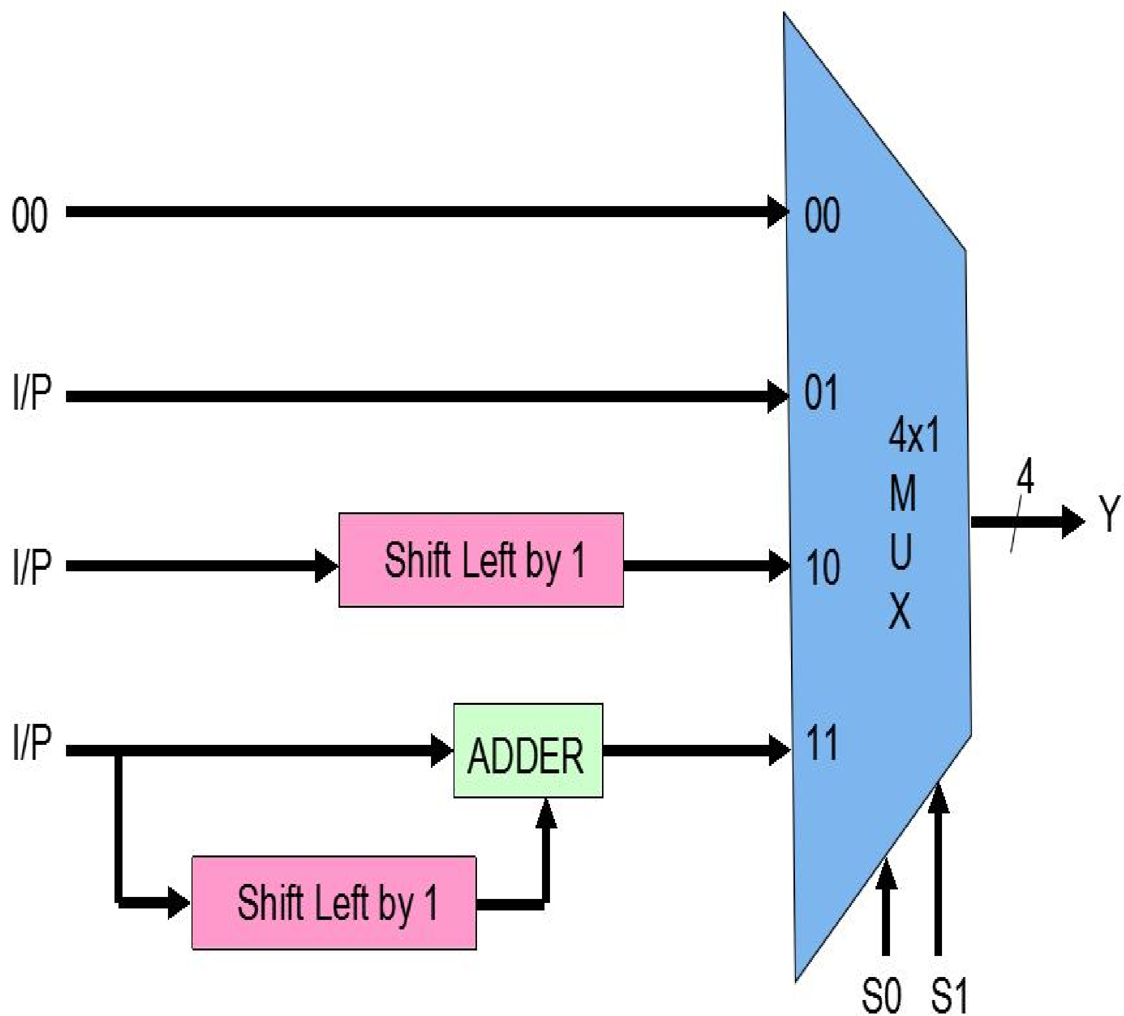


Figure 4.1b : MUX based 2x2 multiplier.

4.2 DESIGN OF MUX BASED 3X3 MULTIPLIER

The 3x3 multiplier using multiplexer using 8:1 multiplexer is shown in Fig. 2. The multiplicand A has three bits S0, S1, S2. The multiplier B has eight input lines with first line input is always 000.

The second line input values are any combinations of three bits and values are passed to output for control value 001.

The third input line and forth input line are connected to shift left by one and shift left by one and shift left by two respectively and correspond input are passed to output for control values 010 and 100 respectively.

The fourth and sixth line input are connected to shift left by one with adder and shift left by two with adder respectively and the corresponding modified inputs are transferred to output for control values 011 and101.

The seventh line input is connected to shift left by one and two with adder and modified input is transferred to output for control value 110. The eight line input is connected to shift left by one and two.

The modified input of eight lines is an addition of shift left by one, shift left by two and direct eight line input. The modified eight line input is transferred to output for control value 111.

The above explained output vales for various combinations is shown in below figure.

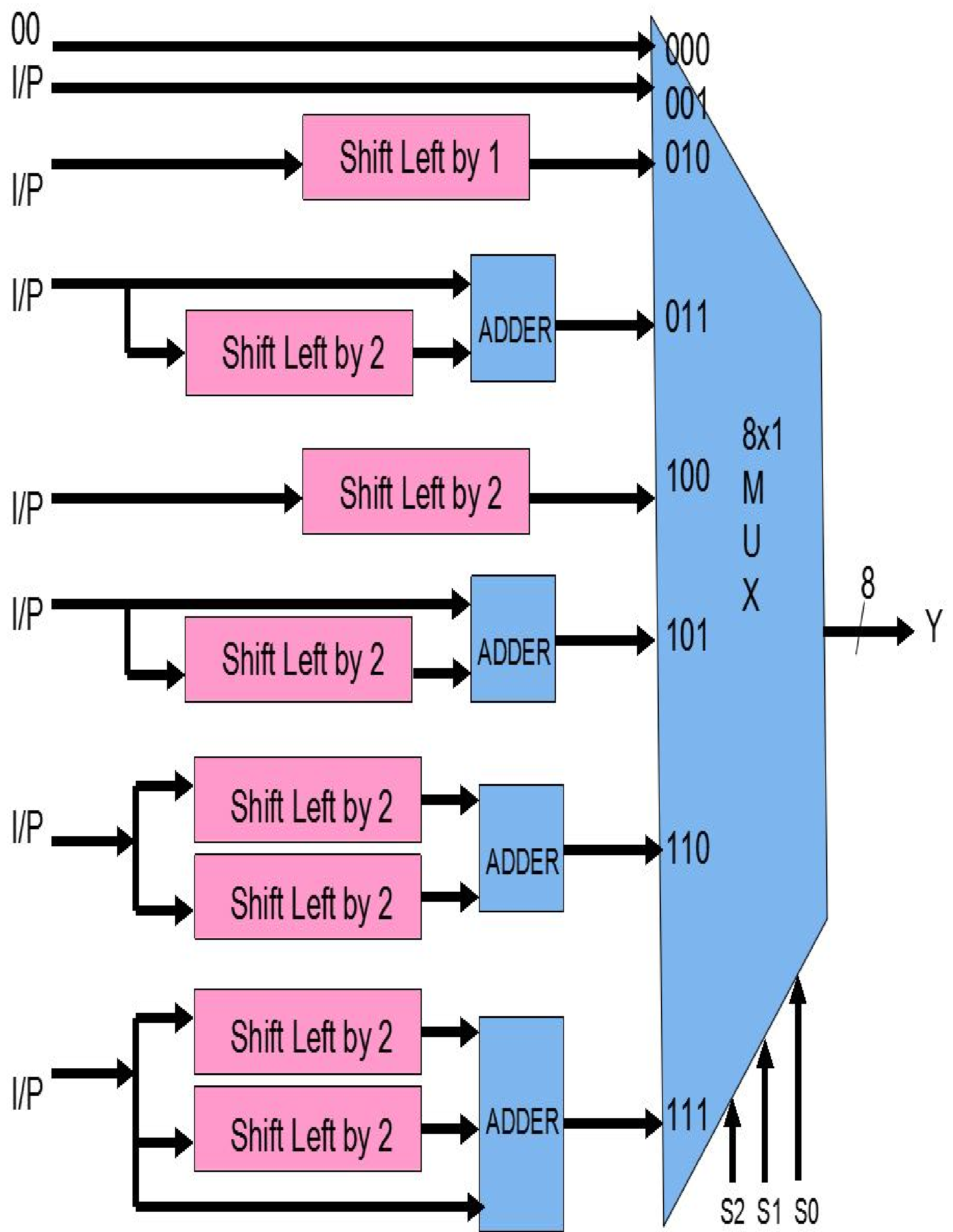


Figure 4.2a : MUX based 3x3 multiplier.

4.3 PROPOSED 8X8 MULTIPLIER BASED ON MUX AND VEDIC CONCEPT

Multiplier using 3x3, 3x2, 2x2 mux based multiplier is shown in Fig3. The two eight bit number (a7 a6 a5 a4 a3 a2 a1 a0) and (b7 b6 b5 b4 b3 b2 b1 b0) are considered for multiplication. The bits (a2 a1 a0) and (a5 a4 a3) and (a7 a6) are group as C,B and A respectively. The bits (b2b1 b0), (b5b4b3) and (b7 b6) are grouped as F, E and D respectively.

The following steps are used for multiplication:

Step1: The group C and F are multiplied using 3x3 mux based multiplier.

Step2: (B*F) + (E*C) + carry of step1.

Step3: (F*A) + (D*C) + (B*E) + carry of step2.

Step4: (E*A) + (B*D) + Carry of step 3. step5: (A*D) + carry of step4.

The one block of 2x2, four block of 3x2 and four block of 3x3 mux based multiplier along with four adders are used in 8x8 multiplier using Vedic concept as shown in Fig 5.3a

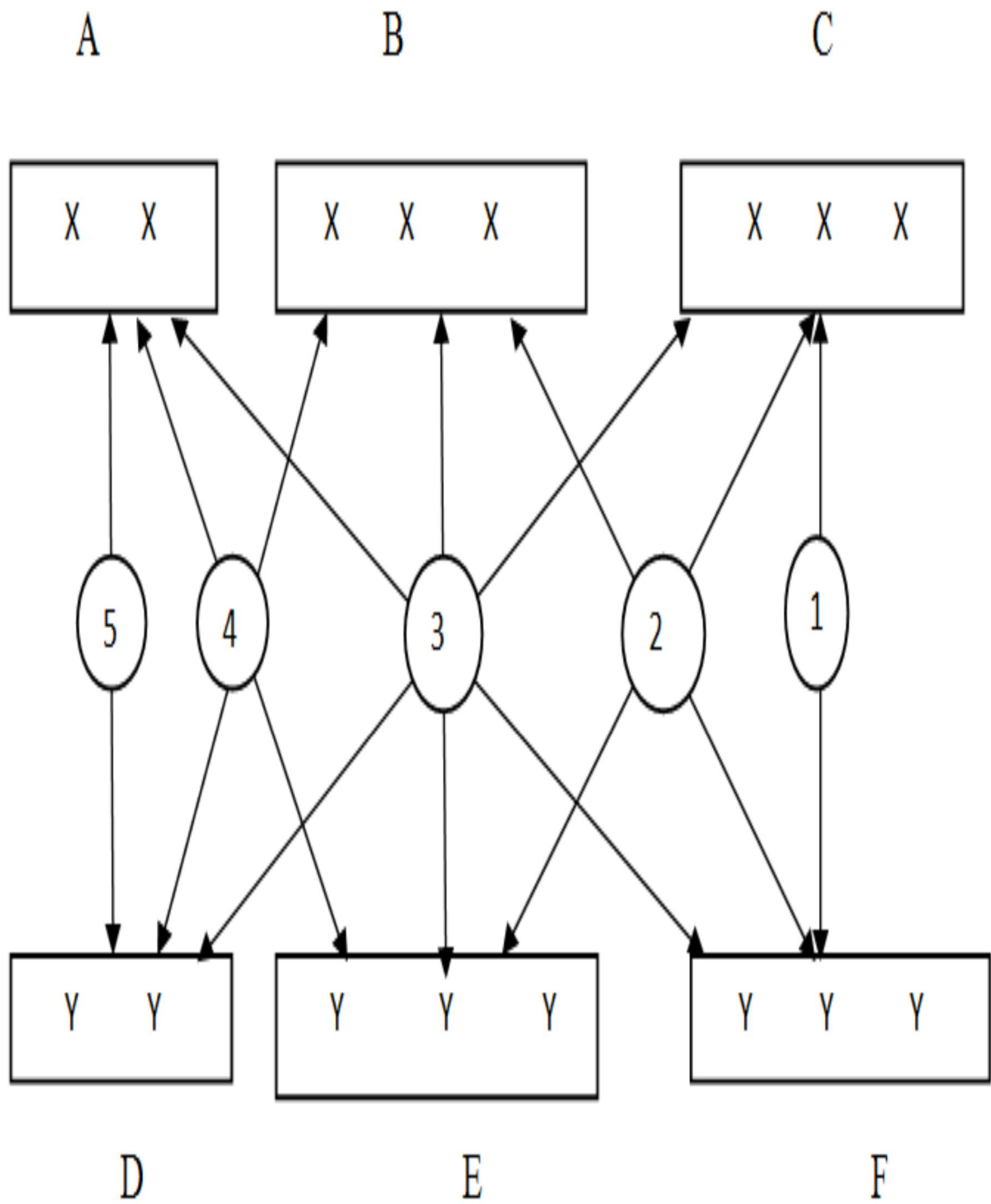


Figure 4.3a : Line diagram of the proposed Vedic multiplier for 8x 8 bit multiplications.

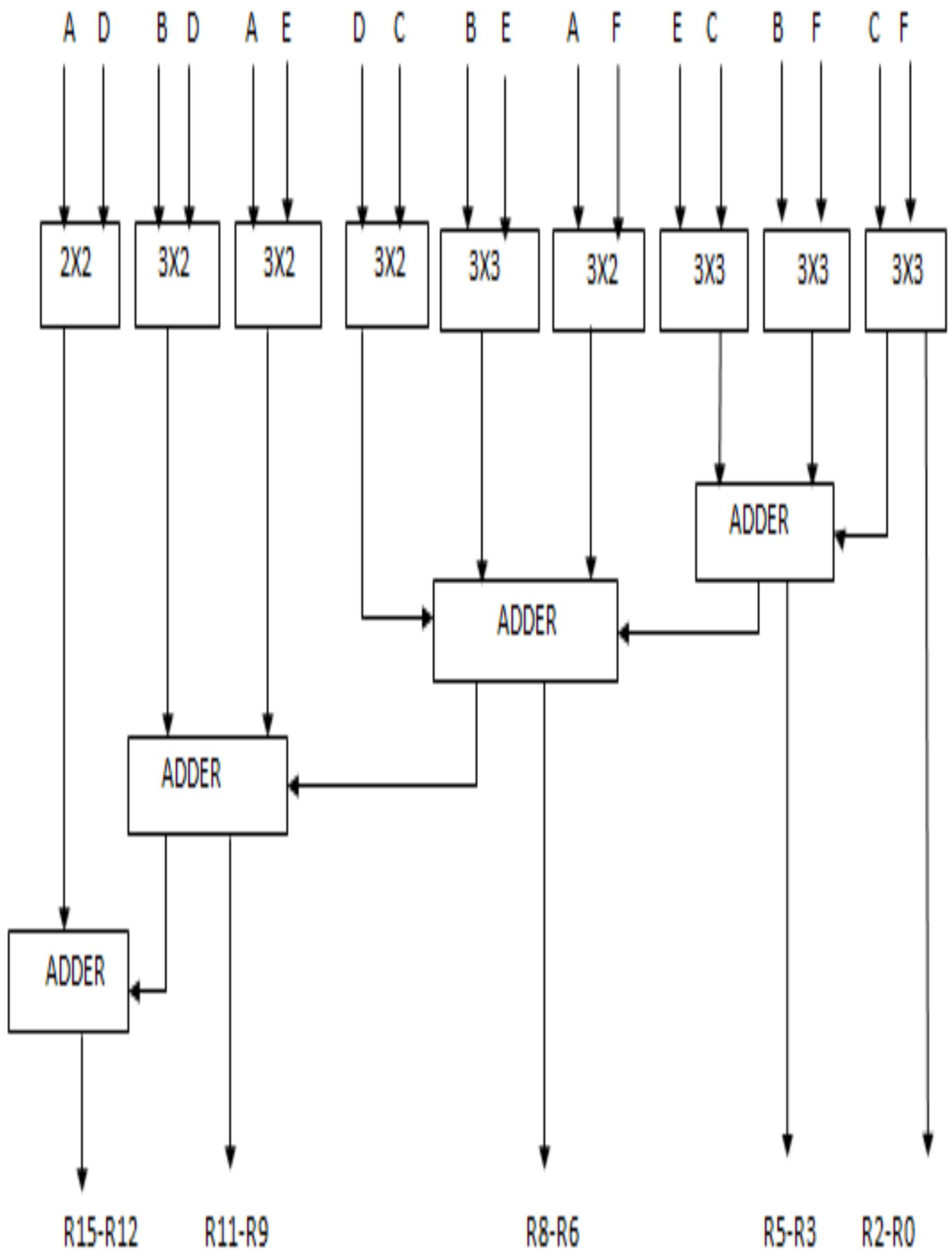


Figure 4.3b : Architecture of 8X8 bit multiplier using 3X3, 3X2 and 2X2 bit multiplier blocks.

4.4 LOOK AHEAD ADDER

In Carry Look-Ahead adder (CLA), carries are generated concurrently by means of look-ahead logic. It is an adder with time propagation duration in O(log n) and whose area size requirement is in O(n log n). The delay time of the CLA architecture therefore exhibits logarithmic dependency on the size of the adder, which allows the propagation delay of the carry signal to be minimized.

Because forming carries in the RCA sequentially makes it necessary to specify Ci+1.

As a specific function of Ci, limitations arise according to the number of Cout signals. In the CLA, however, a carry does not depend explicitly on the preceding one. It can, however be expressed as a function of the relevant propagate and generate signals, Pi and Gi as well as the initial carry-in, Cin. Therefore, the CLA comes in handy for better delay-reduction performance.

For a particular combination of inputs Ai and Bi, the propagate signal Pi determines whether the carry-in to the ith block would propagate to the output, whereas the generate signal Gi determines if a carry-out would be set from inside the block independently from the inputs.

The expressions for Gi and Pi with two input binary operands, Ai and Bi are as follows:

Gi= Ai . Bi

Pi= Ai xor Bi

The sum output and carry recurrence for the ith stage are given by:

$Si = Pi \text{ xor } Ci$

$Ci+1 = Gi + Pi. Ci$

The logic schematic of 4-bit carry is shown in figure 5.4a. As seen in figure, the carry generation requires only two gate delays. This makes the addition of two n-bit operands extremely fast as compared to the ripple carry adder. However, it costs more gates to implement this logic circuit, because for large values of n, a huge number og gates and very big fan-in gates are required. Therefore, to reduce the span of the carry look-ahead, the n-bit operands are divided into equal sized groups, which are then interconnected by the incorporating the concept of RCA.

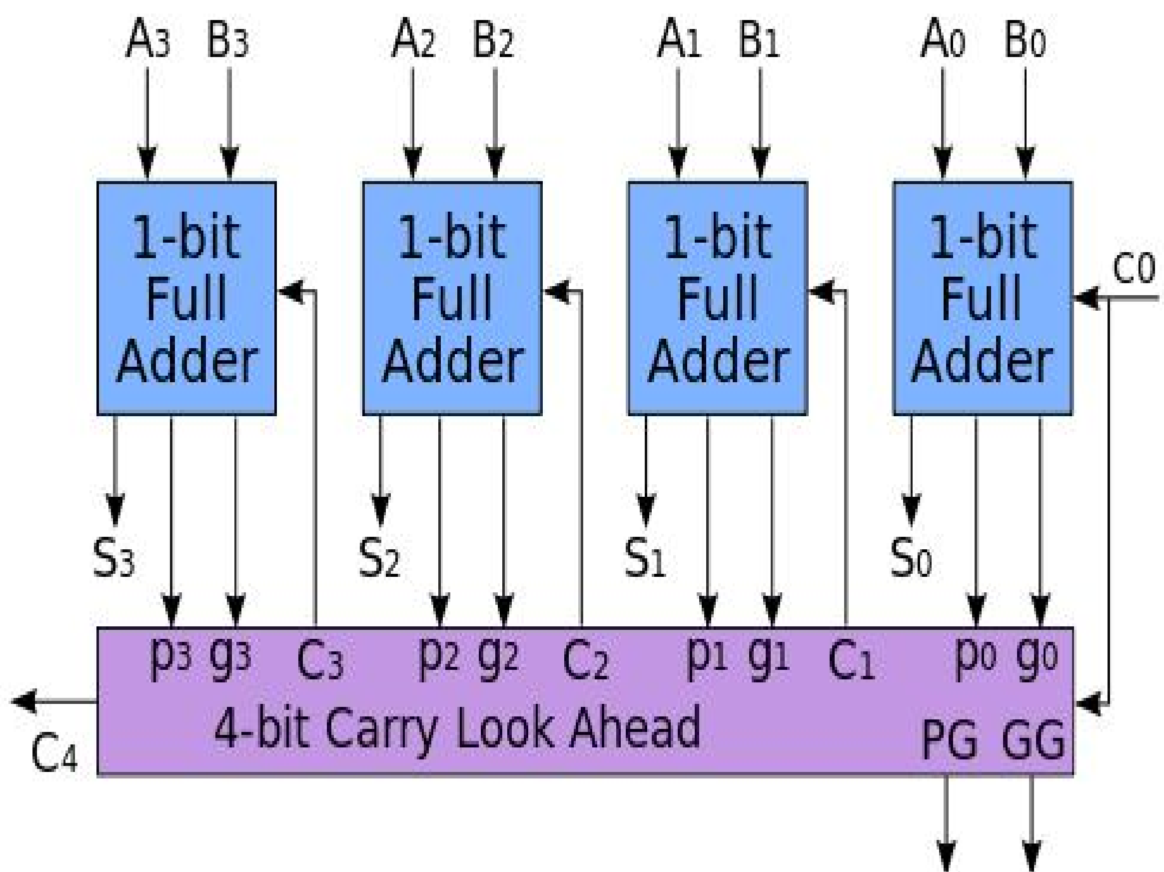


Figure 4.4a : Logic schematic of 4-bit carry.[5]

The block diagram of a 16 bit CLA is shown in Figure 4.4b [9,10] It is divided into four 4-bit groups and comprises a combinatorial circuit, namely the look ahead carry generator.

The term G^*_k denotes the group- generated carry, whereas P^*_k denotes the group- propagated carry. The Boolean equations for these carries are

$G^*_k = G_k + G_{k-1} . P_k + G_{k-2} . P_{k-1} . P_k + G_{k-3} . P_{k-2} . P_{k-1} . P_k$

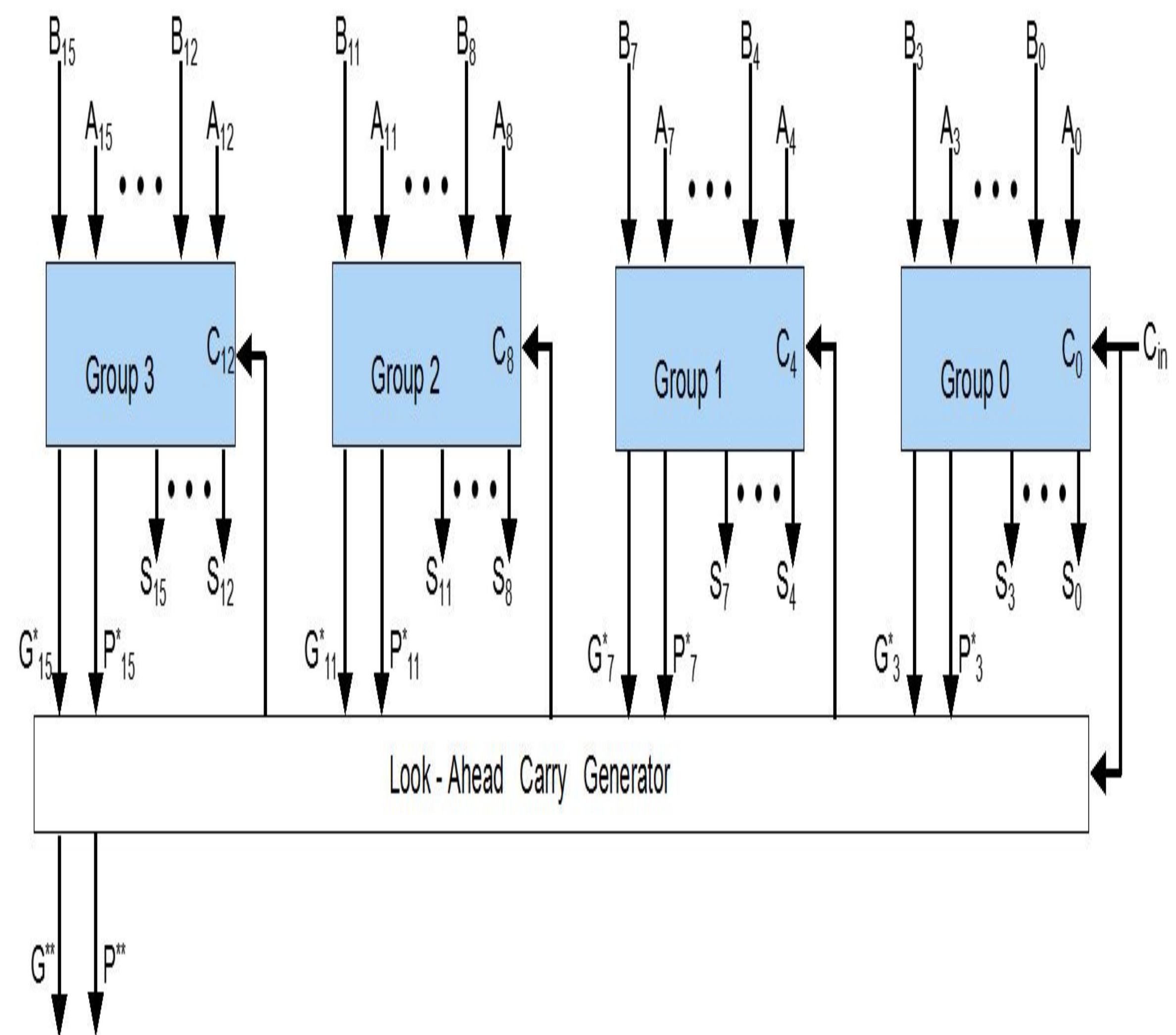


Figure 4.4b : Block diagram of a 16- bit carry look-ahead adder [9,10].

4.5 PROPOSED 16X16 MULTIPLIER USING 8X8 MULTIPLIER

The block diagram of 16 x16 multiplier using 8x8 multiplier is shown in Figure 4.5a. The four 8x8 multiplier along with two adders are used to implement 16x16 multiplier. The two numbers of 16 bits are a0 to a15 and b0 to b15 are considered. The proposed multiplier can be extended to any value of MxN.

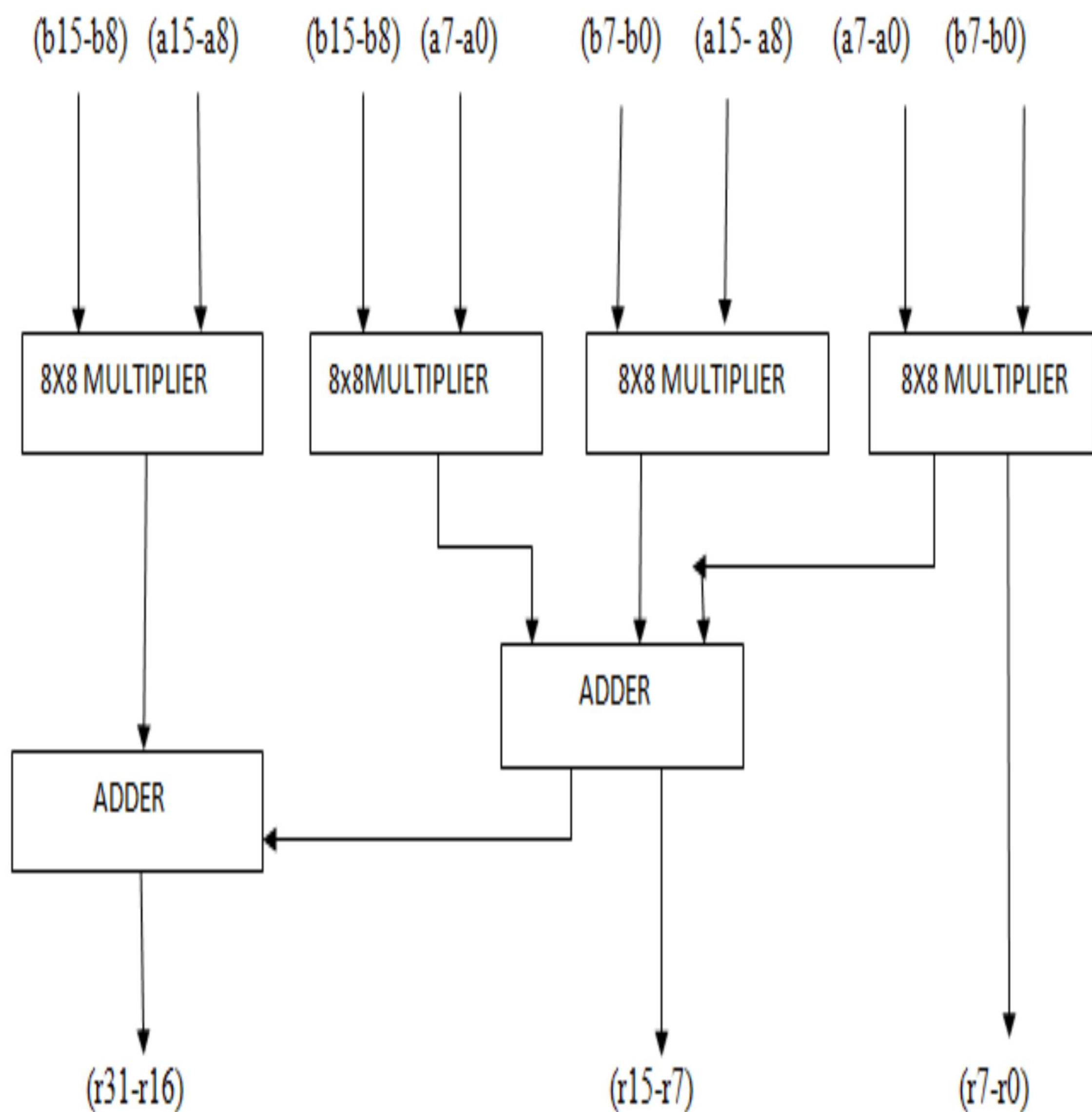


Figure 4.5a : Architecture of 16x16 bit multiplier using 8x8 bit multiplier block.

CHAPTER 5

RESULTS AND VERIFICATION

5.1 RESULTS

The design proposed in chapter 5 has been implemented, simulated on ModelSim and synthesized for VHDL. The HDL code uses VHDL 2001 constructs that provide certain benefits over the VHDL 95 standard in terms of scalability and code reusability. Simulation based verification is one of the methods for functional verification of a design. In this method, test inputs are provided using standard test benches. The test bench forms the top module that instantiates the other modules. Simulation based verification ensures that the design is functionally correct when tested with a given set of inputs. Though it is not fully complete, by picking a random set of inputs as well as corner cases, simulation based verification can still yield reasonably good results.

The following snapshots were taken from ModelSim after the timing simulation of the floating point multiplier core:

Consider the inputs to the floating point multiplier are:

A: 1 10000010 000000000000000000000000

B: 0 10000011 000000000000000000000000

The output of multiplier should be:

1 1000010 0000000000000000000000

And flags output of this multiplier are:

Ine=0; inf=0; overflow=0; qnan= 0; snan= 0; underflow=0; zero=0;

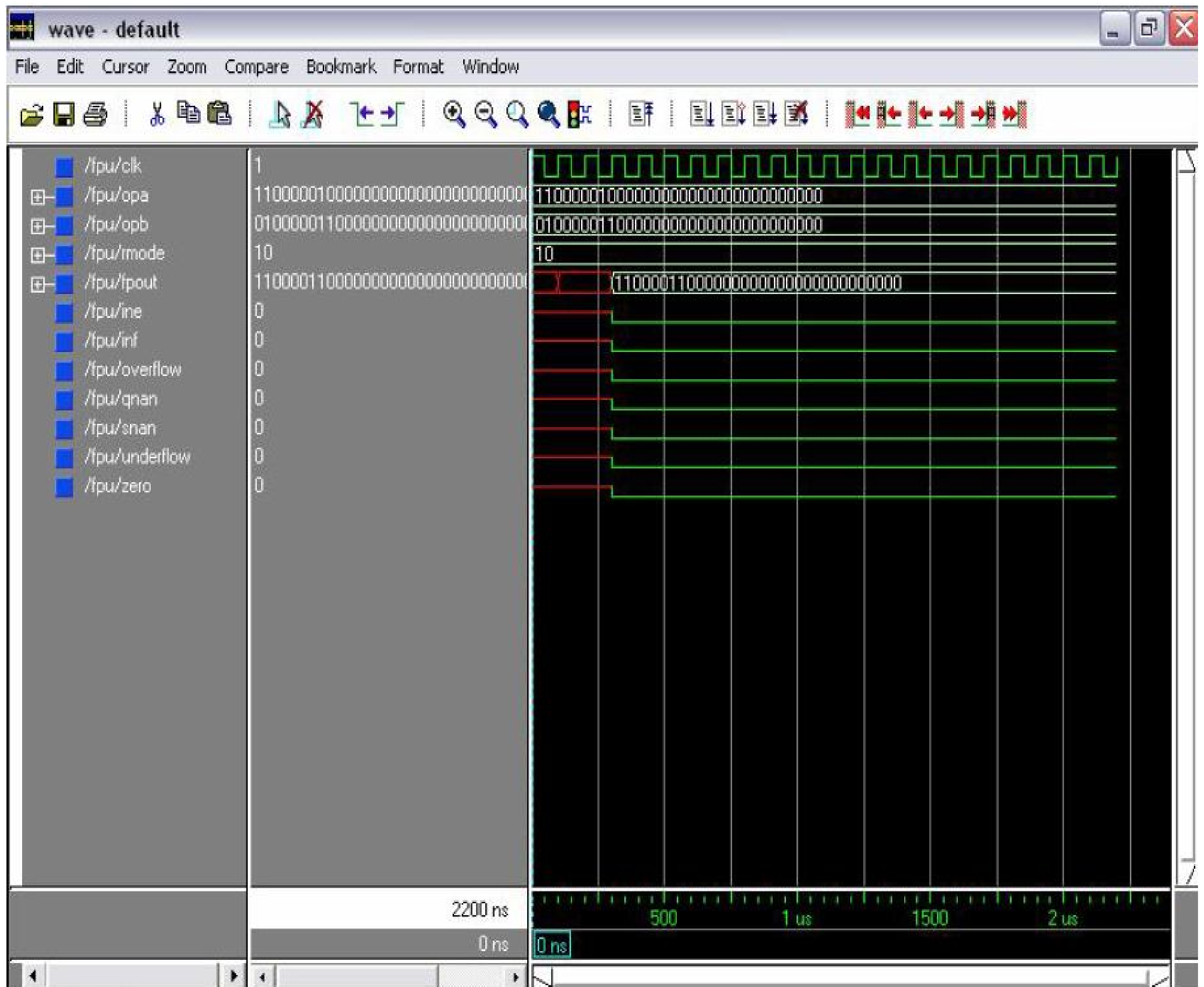


Figure 5.1a : Output of single precision Floating Point number when above inputs were given.

5.2 VERIFICATION

Design verification is defined as the reverse process of design. Design verification includes functional verification, timing verification, layout verification and electrical verification but functional verification by default is termed as design verification. [13]

Simulation based approach and the formal verification approach are two popular forms of verification. The major difference between these two approaches is that the input vectors are needed in simulation based approach but in not in formal verification approach. In former the input vectors are generated and reference outputs are derived.

The desirable output behaviour is predetermined in formal verification approach and the formal checker is used to see if it agrees or disagrees with the desired behaviour. This shows that simulation based approach is input driven but the formal approach is output driven.

Simulation based approach takes selected points in input space at a particular time and thus samples few points only. While, fomal verification methodology instead of choosing vectors operates on input space and causes the extensive use of memory and long runtime. Moreover during memory overflow the tools are not able to show what are the right problems and their fix.

This proposed design is verified using simulation based approach. In order to verify the functionality of cores, it is necessary to have inputs with signal combination. As the design includes a 2's compliment unit, the negative and positive numbers must be distinguished by the core.

We needed to verify if the design terminates gracefully with the zero output if such inputs are given.

Since the module needed to be completely verified therefore inputs were designed such that every case of Multiplexer is exercised.

The overflow and underflow cases were also tested. A value of exponent greater than 127 should set the result to infinity and when the exponent and mantissa both are zero then result is set to zero.

CHAPTER 6

CONCLUSION AND FUTURE SCOPE OF WORK

6.1 CONCLUSION

Higher order multipliers are required in image processing applications. 32 bit Single precision floating point multiplier is implemented using Multiplexer with vedic multiplication concept using ModelSim in this thesis. The designed multiplier conforms to IEEE 754 single precision floating point standard. In this implementation the exceptions like invalid, inexact, infinity and zero are considered. The design is verified using test bench.

6.2 FUTURE SCOPE

In future,

- Multiplier can be designed using higher order MUX based multipliers.
- The proposed multiplier of any size can be used in multiplication of two matrices of any size for image processing applications.
- Double precision Floating point multiplier can be implemented by this proposed method.

REFERENCES

[1] IEEE Standard for Binary Floating-Point Arithmatic, ANSI /IEEE Standard 754,1985.

[2] Jagadguru Swami Sri Bharati Krisna Tirthaji Maharaja, “Vedic Mathematics: Sixteen Simple Mathematical Formulae from the Veda,” Motilal Banarasidas Publishers, Delhi, 2009, pp. 5-45.

[3] H. Thapliyal and M. B. Shrinivas and H. Arbania, “Design and Analysis of a VLSI Based High Performance Low Power Parallel Square Architecture,” Int. Conf. Algo.Math.Comp. Sc., Las Vegas, June 2005, pp. 72-76.

[4] Himanshu Thapliyal and M. B. Srinivas,“An efficient method of elliptic curve encryption using Ancient Indian Vedic Mathematics,”48th IEEE International Midwest Symposium on Circuits and Systems, 2005, vol. 1, pp. 826-828.

[5] M. Pradhan and R. Panda, “Design and Implementation of Vedic Multiplier,” A.M.S.E Journal, Computer Science and Statistics, France vol. 15, July 2010, pp. 1-19.

[6] N. Shirazi, A. Walters and P. Athanas, “Quantitative Analysis of Floating Point Arithmetic on FPGA Based Custom Computing Machines,” Proceedings of the IEEE Symposium on FPGA for Custom Computing Machines (FCCM”95), pp.155-162, 1995.

[7] G. Renxil, Z. Shangjun2, Z. Hainan1, M. Xiaobil, G. Wenying1, X. Lingling1, H. Yang1, “Hardware Implementation of a Hign Speed Floating Point Multiplier Based on FPGA”, 2009.

[8] I. koren. “Computer Arithmetic Algorithms”, Englewood cliffs, New jersey: Prentice hall, 1993.

[9] S. Waser and M.J. Flynn, “Introduction to Arithmetic for Digital Systems Designers”, New York: CBS College Publishing, 1982.”

[10] Mohamed Al- Ashrafy, A. Salem and W. Anis, “An Efficient Implementation of Floating Point Multiplier”, 978-1-4577-0069, 2011.

[11] L. Louca, T. A. Cook, and W. H. Johnson, "Implementation of IEEE Single Precision Floating Point Addition and Multiplication on FPGAs," Proceedings of 83rd IEEE Symposium on FPGAs for Custom Computing Machines (FCCM"96), pp. 107-116, 1996.

[12] L. Louca, T.A. Cook, W.H. Johnson, implementation of IEEE Single Precision Floatin Point addition and multiplication on FPGA”, Proc. Of IEEE Symposium on FPGA, IEEE Computer Society Press, 1995. pp 195-162.