# "FPGA IMPLEMENTATION OF BOOTH WALLACE MULTIPLIER"

*Thesis report submitted towards the partial fulfillment of*

*the requirements for the award of the degree of*

**Master of Technology**

**In**

**VLSI Design & CAD**

Submitted by

**Prabhat Kumar**

**Roll No. 600861025**

Under the Guidance of

**Mrs. Manu Bansal**

**Assistant Professor, ECED**

**Department of Electronics and Communication Engineering**

**THAPAR UNIVERSITY**

**PATIALA-147004, INDIA**

**JULY - 2010**

# CERTIFICATE

I hereby certify that the work which is being presented in the thesis entitled, **"Fpga Implementation of Booth Wallace Multiplier"** in partial fulfillment of the requirement for the award of degree of M.Tech (VLSI Design & CAD) at Electronics and Communication Department of Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of Mrs. Manu Bansal, Assistant Professor, ECED.

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

Prabhat Kumar

**Date:** 22-06-2010

**Prabhat Kumar**

**Roll No. 600861025**

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

**Mrs. Manu Bansal**

Assistant Professor
ECED, Thapar University
Patiala-147004

Counter signed by:

**Dr. A. K. Chatterjee**

Professor and Head
Electronics and Communication Engg. Department,
Thapar University,
Patiala-147004

**Dr. R. K. Sharma**

Dean (Academic Affairs)
Thapar University,
Patiala-147004

## ACKNOWLEDGEMENT

To discover, analyze and to present something new is to venture on an untrodden path towards and unexplored destination is an arduous adventure unless one gets a true torchbearer to show the way. I would have never succeeded in completing my task without the cooperation, encouragement and help provided to me by various people. Words are often too less to reveals one's deep regards. I take this opportunity to express my profound sense of gratitude and respect to all those who helped me through the duration of this thesis. I acknowledge with gratitude and humility my indebtedness to **Mrs. Manu Bansal, Assistant Professor ,ECED,**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 **Dr.A.K.Chatterjee, Professor & Head of Electronics & Communication Department,**Thapar University, Patiala for his encouragement and cooperation.

I express my heartfelt gratitude towards **Mrs. Alpana Agarwal, Assistant Professor & PG Coordinator, ECED,** Thapar University, Patiala, for their valuable guidance, encouragement, inspiration and the enthusiasm with which she solved my difficulties in VLSI Design & CAD Lab.

I would also like to thank all staff members and my co-students who were always there at the need of the hour and provided with all the help and facilities, which I required for the completion of my thesis.

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.

Prabhat Kumar

**Prabhat Kumar**

## ABSTRACT

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

Since multiplication dominates the execution time of most DSP algorithms, so there is a need of high speed multiplier. So this thesis work is devoted for the design and comparison of different 8-bit and 16-bit Booth Wallace Tree Multipliers Comparison is based on the synthesis result obtained by synthesizing all multiplier architecture toward FPGA. A method has been proposed in the Booth Wallace tree multiplier to calculate 2's complement of multiplicand for final Partial Product Row (PPR) if using MBE technique so as to reduce an extra row of carry save adders which reduces area and delay of the multiplier. This method has been used in the design for speed enhancement of our multiplier.

The report throws light on the basic principles and methods of binary multiplication process. The new algorithm which reduces the last negative signal in the partial product row is discussed to develop the new architecture. The simulation and synthesis results show the values of both the multipliers regarding different parameters.

## TABLE OF CONTENTS

## LIST OF FIGURES

## LIST OF TABLES

## TERMINOLOGY

| | |
|---|---|
| **DSP** | Digital signal Processing |
| **ADSP** | Advanced digital signal processing |
| **XST** | Xilinx synthesis technology |
| **FPGA** | Field programming gate array |
| **ATE** | Automatic test equipment |
| **BIST** | Built in self test |
| **CUT** | Circuit under test |
| **ATPG** | Automatic test pattern generator |
| **TPG** | Test pattern generator |
| **DFT** | Design for test |
| **DFT** | Discrete Fourier transforms |
| **MAC** | Multiply and Accumulate |
| **FFT** | Fast Fourier transforms |
| **IFFT** | Inverse Fast Fourier transforms |
| **IC** | Integrated Circuits |
| **ROM** | Read only memory |
| **BILBO** | Built in Logic Block observer |
| **LFSR** | Linear feedback shift register |
| **MFSR** | Multiple feedback shift register |

| | |
|---|---|
| **PRPG** | Pseudo random pattern generator |
| **MISR** | Multiple input signature register |
| **ORA** | Output response analyzer |
| **PLA** | Programmable logic Arrays |
| **UCF** | User constraints file |
| **CLB** | Combinational logic blocks |
| **IOB** | Input output blocks |
| **PAR** | Place and Route |
| **ISE** | Integrated Software Environment |
| **IOP** | Input output processor |
| **CPLD** | Complex programmable logic device |
| **RTL** | Register transfer level |
| **JTAG** | Joint test action group |
| **RAM** | Random access memory |
| **FDR** | Fix data register |
| **SOPC** | System-on –a-Programmable-chip |
| **ASIC** | Application-specific integrated circuit |

CHAPTER

# 1 INTRODUCTION

Multiplication is an important fundamental function in arithmetic operations. Multiplication-based operations such as Multiply and Accumulate(MAC) and inner product are among some of the frequently used Computation- Intensive Arithmetic Functions(CIAF) currently implemented in many Digital Signal Processing (DSP) applications such as convolution, Fast Fourier Transform(FFT), filtering and in microprocessors in its arithmetic and logic unit [1]. Since multiplication dominates the execution time of most DSP algorithms, so there is a need of high speed multiplier. Currently, multiplication time is still the dominant factor in determining the instruction cycle time of a DSP chip.

The demand for high speed processing has been increasing as a result of expanding computer and signal processing applications. Higher throughput arithmetic operations are important to achieve the desired performance in many real-time signal and image processing applications [2]. One of the key arithmetic operations in such applications is multiplication and the development of fast multiplier circuit has been a subject of interest over decades. Reducing the time delay and power consumption are very essential requirements for many applications [2, 3]**.** This work presents different multiplier architectures. Multiplier based on Booth Wallace is one of the fast and low power multiplier.

Minimizing power consumption for digital systems involves optimization at all levels of the design. This optimization includes the technology used to implement the digital circuits, the circuit style and topology, the architecture for implementing the circuits and at the highest level the algorithms that are being implemented. Digital multipliers are the most commonly used components in any digital circuit design. They are fast, reliable and efficient components that are utilized to implement any operation.

Depending upon the arrangement of the components, there are different types of multipliers available. Particular multiplier architecture is chosen based on the application.

Digital multipliers are the core components of all the digital signal processors (DSPs) and the speed of the DSP is largely determined by the speed of its multipliers. Two most common multiplication algorithms followed in the digital hardware are array multiplication algorithm and Booth multiplication algorithm. The computation time taken by the array multiplier is comparatively less because the partial products are calculated independently in parallel. The delay associated with the array multiplier is the time taken by the signals to propagate through the gates that form the multiplication array. Booth multiplication is another important multiplication algorithm. Large booth arrays are required for high speed multiplication and exponential operations which in turn require large partial sum and partial carry registers. Multiplication of two $n$-bit operands using a radix-4 booth recording multiplier requires approximately $n / (2m)$ clock cycles to generate the least significant half of the final product, where $m$ is the number of Booth recorder adder stages. Thus, a large propagation delay is associated with this case. Due to the importance of digital multipliers in DSP, it has always been an active area of research and a number of interesting multiplication algorithms have been reported in the literature [4].

The multiplier is a fairly large block of a computing system. The amount of circuitry involved is directly proportional to the square of its resolution i.e. a multiplier of size n bits has O ($n^2$) gates. For multiplication algorithms performed in DSP applications latency and throughput are the two major concerns from delay perspective. Latency is the real delay of computing a function, a measure of how long the inputs to a device are stable is the final result available on outputs. Throughput is the measure of how many multiplications can be performed in a given period of time multiplier is not only a high delay block but also a major source of power dissipation. That's why if one also aims to minimize power consumption, it is of great interest to reduce the delay by using various delay optimizations.

There are different multipliers which can be compared on the basis of speed and area consideration. The multiplier architecture can be generally classified into three categories. First is the serial multiplier which emphasizes on hardware and minimum amount of chip area. Second is parallel multiplier (array and tree) which carries out high

speed mathematical operations. But the drawback is the relatively larger chip area consumption. Third is serial- parallel multiplier which serves as a good trade-off between the times consuming serial multiplier and the area consuming parallel multipliers.

## 1.1 Speed and Size

In this, when performance of circuits is compared, it is always done in terms of circuit speed and size. A good estimation of the circuit's size is to count the total number of gates used. The actual chip size of a circuit also depends on how the gates are placed on the chip – the circuit's layout. Since we do not deal with layout in this report, the only thing we can say about this is that regular circuits are usually smaller than non-regular ones (for the same number of gates), because regularity allows more compact layout. The physical delay of circuits originates from the small delays in single gates, and from the wiring between them. The delay of a wire depends on how long it is. Therefore, it is difficult to model the wiring delay; it requires knowledge about the circuit's layout on the chip. The gate delay, however, can easily be modeled by saying that the output is delayed a constant amount of time from the latest input. What we can say about the wiring delay is that larger circuits have longer wires, and hence more wiring delay. It follows that a circuit with a regular layout usually has shorter wires and hence less wiring delay than a non-regular circuit.

Therefore, if circuit delay is estimated as the total gate delay, one should also have in mind the circuit's size and amount of regularity, when comparing it to other circuits. "Delay" usually refers to the "worst-case delay". That is, if the delay of the output is dependent on the inputs given, it is always the largest possible output delay that sets the speed. Furthermore, if different bits in the output have different worst-case delays, it is always the slowest bit that sets the delay for the whole output. The slowest path between any input bit and any output bit is called the "critical path". If a circuit is to be speed up, it is always the critical path that should be attacked in the first place.

## 1.2 Objective

The main objective of this thesis is to design and implementation of a Booth Wallace multiplier. A multiplier which is a combination of Modified Booth and Wallace tree multiplier are designed taking into account the less area consumption of booth algorithm

because of less number of partial products and more speedy accumulation of partial products and more speedy accumulation of partial using tree approach (CSA).

## 1.3 Thesis Organization

Chapter2: Discusses about the basics of multiplication and how it works for binary numbers. Chapter3: Describes the Booth and modified Booth Algorithm. Chapter4: Describe the compressor and carry look Ahead adder. Chapter5: Tells the method of adding Partial Products using CSA with Example.Chapter6: Describes the flow of FPGA implementation. In Chapters 3-5, initially the multipliers i.e. Booth, Tree and Booth Recoded Wallace multiplier will be drafted describing the basic functionality in terms of blocks i.e. partial product generation logic and partial product reduction logic and are coded in Hardware Description language i.e. Verilog for both 8-bit and 16-bit numbers. The functionality of the design is verified by simulation software, Modelsim and ISE both. After getting the desired simulation results, verilog code is synthesized and timing & area results are obtained. But the timing results obtained are not actual as routing delays have not been added in these results. So, we apply timing constraints based on these results using Xilinx constraint editor. The design is then implemented to be targeted on FPGA (SPARTAN3E). Routing is followed by translation and mapping and this will take into account all the nets and wire delays. After post route simulation, we get the actual delay of the design. Chapter 7: Describes the new architecture which is implemented on Modelsim and verifying result results . Chapter 8: Gives the conclusion and future scope.

CHAPTER

# 2

# BASICS OF MULTIPLIER

Multiplication is a mathematical operation that at its simplest is an abbreviated process of adding an integer to itself a specified number of times. A number (multiplicand) is added to itself a number of times as specified by another number (multiplier) to form a result (product). In elementary school, students learn to multiply by placing the multiplicand on top of the multiplier. The multiplicand is then multiplied by each digit of the multiplier beginning with the rightmost, Least Significant Digit (LSD). Intermediate results (partial-products) are placed one atop the other, offset by one digit to align digits of the same weight. The final product is determined by summation of all the partial-products. Although most people think of multiplication only in base 10, this technique applies equally to any base, including binary. Figure 2.1 shows the data flow for the basic multiplication technique just described. Each black dot represents a single digit.

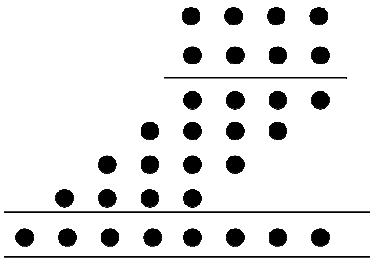


**Fig2. 1**: basic Multiplication

Here, we assume that MSB represent the sign of the digit. The operation of multiplication is rather simple in digital electronics. It has its origin from the classical algorithm for the product of two binary numbers. This algorithm uses addition and shift left operations to calculate the product of two numbers. Based upon the above procedure, we can deduce an algorithm for any kind of multiplication which is shown in figure 2.2. We can check at

the initial stage also that whether the product will be positive or negative or after getting the whole result, MSB of the results tells the sign of the product.

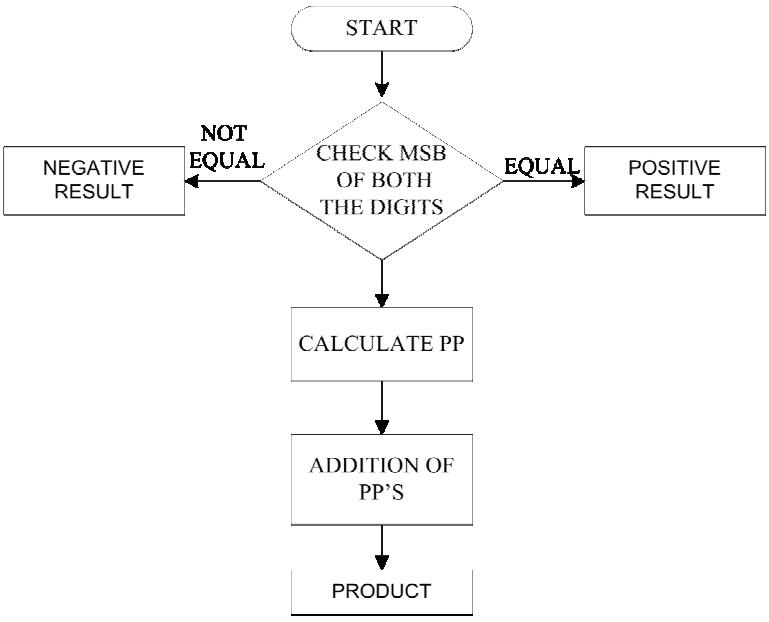



**Figure 2.2** Signed Multiplication Algorithm

## 2.1 Binary Multiplication

In the binary number system the digits, called bits, are limited to the set [0, 1]. The result of multiplying any binary number by a single binary bit is either 0, or the original number. This makes forming the intermediate partial-products simple and efficient. Summing these partial-products is the time consuming task for binary multipliers. One logical approach is to form the partial-products one at a time and sum them as they are generated. Often implemented by software on processors that do not have a hardware multiplier, this technique works fine, but is slow because at least one machine cycle is required to sum each additional partial-product. For applications where this approach does not provide enough performance, multipliers can be implemented directly in hardware. The two main categories of binary multiplication include signed and unsigned numbers. Digit multiplication is a series of bit shifts and series of bit additions, where the two numbers, the multiplicand and the multiplier are combined into the result. Considering the bit representation of the multiplicand $x = x_{n-1}.....x_1 x_0$ and the multiplier $y = y_{n-1}.....y_1 y_0$ in order to form the product up to n shifted copies of the multiplicand are to

be added for unsigned multiplication. The entire process consists of three steps, partial product generation, partial product reduction and final addition.

## 2.2 Multiplication Process

The simplest multiplication operation is to directly calculate the product of two numbers by hand. This procedure can be divided into three steps: partial product generation, partial product reduction and the final addition. To further specify the operation process, let us calculate the product of two two's complement numbers, for example, 1101two (−3ten) and 0101two (5ten), when computing the product by hand, which can be described according to figure 2.3.[5]

```
                                1   1   0   1        Multiplicand
                          ×     0   1   0   1        Multiplier
                          ---------------------------------------------------
          1   1   1   1   1   1   0   1            PP1
          0   0   0   0   0   0   0                PP2
          1   1   1   1   0   1                    PP3
  +           0   0   0   0   0                    PP4
-------------------------------------------------------------------------------
      1   1   1   1   1   0   0   0   1 = −15     Product
```

**Fig 3.3** Multiplication calculations by hand

The bold italic digits are the sign extension bits of the partial products. The first operand is called the multiplicand and the second the multiplier. The intermediate products are called partial products and the final result is called the product. However, the multiplication process, when this method is directly mapped to hardware, is shown in figure 2.2.

As can been seen in the figures, the multiplication operation in hardware consists of PP generation, PP reduction and final addition steps. The two rows before the product are called sum and carry bits. The operation of this method is to take one of the multiplier bits at a time from right to left, multiplying the multiplicand by the single bit of the multiplier and shifting the intermediate product one position to the left of the earlier intermediate products.

All the bits of the partial products in each column are added to obtain two bits: sum and carry. Finally, the sum and carry bits in each column have to be summed. Similarly, for the multiplication of an *n*-bit multiplicand and an *m*-bit multiplier, a product with $n + m$ bits long and *m* partial products can be generated. The method shown in figure 2.3 is also called a non-Booth encoding scheme.

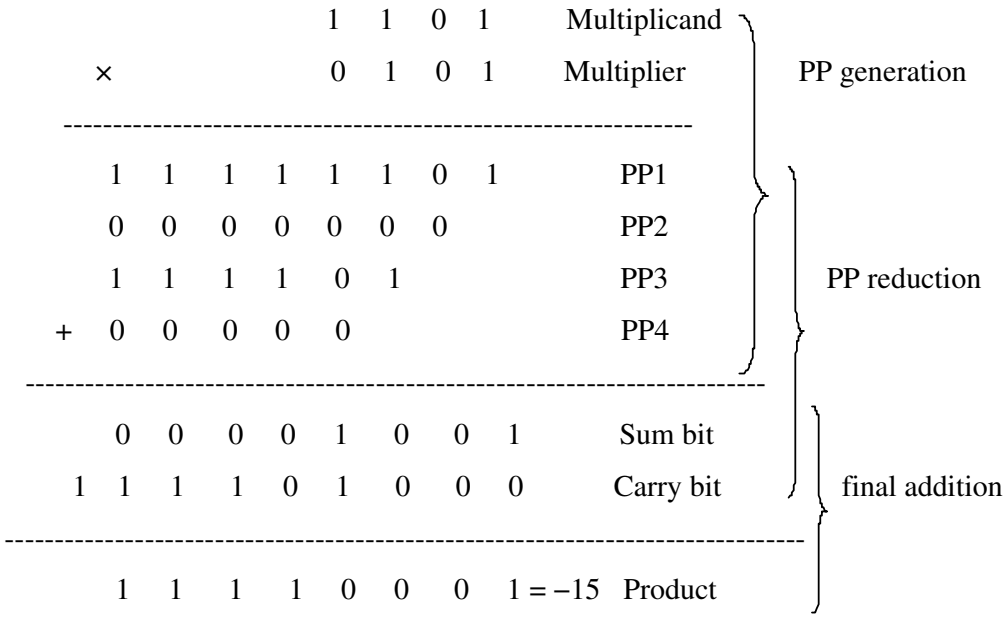



**Fig 4.4**: Multiplication 0peration in hardware

CHAPTER

# 3 MODIFIED BOOTH

**BOOTH MULTIPLIER**

Conventional array multipliers, like the Braun multiplier and Baugh Woolley multiplier achieve comparatively good performance but they require large area of silicon, unlike the add-shift algorithms, which require less hardware and exhibit poorer performance. The Booth multiplier makes use of Booth encoding algorithm in order to reduce the number of partial products by considering two bits of the multiplier at a time, thereby achieving a speed advantage over other multiplier architectures. This algorithm is valid for both signed and unsigned numbers. It accepts the number in 2's complement form, based on radix-2 computation. It can handle signed binary multiplication by using 2's complement representation. This increases the complexity of how signs of the operands get stored in auxiliary circuits.[6]

## 3.1 Non-Booth Recoding

Using the non-Booth encoding method for partial product generation, the multiplier bits are examined sequentially starting from LSB to MSB. If the multiplier bit is one, the partial product is simply the multiplicand. Otherwise, the partial product is zero. Each new partial product is shifted one bit position to the left. Each partial product can be produced by just using a row of two-input AND gates. The number of partial products generated equals the size of the multiplier bits.

## 3.2 BOOTH ALGORITHM

### 3.2.1 Booth Recoding

Booth's multiplication algorithm is a multiplication algorithm that multiplies two signed binary numbers in two's complement notation. The algorithm was invented by Andrew

Donald Booth in 1951 while doing research on crystallography at Birkbeck College in Bloomsbury, London. Booth used desk calculators that were faster at shifting than adding and created the algorithm to increase their speed. Booth's algorithm is of interest in the study of computer architecture. The Booth recoding, or Booth algorithm, was proposed by Andrew D.Booth in 1951 [7]. This method can be used to multiply two two's complement number without the sign bit extension. Booth recoding is a method of reducing the number of partial products to be summed. Booth observed that when strings of '1' bits occur in the multiplicand the number of partial products can be reduced by using subtraction. The operation of Booth recoding consists of two major steps [8]: the first one is to take one bit of the multiplier, and then to decide whether to add the multiplicand according to the current and previous bits of the multiplier. This encoding scheme is serial, which means that the different value of the 2-bits (current and previous bits) corresponds to the different operations.

**Table 3.1** Recoding in Booth Algorithm [9]

| $X_i$ | $X_{i-1}$ | Operations | Comments | $Y_i$ |
|---|---|---|---|---|
| 0 | 0 | Shift only | String of zeros | 0 |
| 1 | 1 | Shift only | String of zeros | 0 |
| 1 | 0 | Subtract and Shift | Beginning of string of ones | 1 |
| 0 | 1 | Add and Shift | End of string of ones | 1 |

The serial recoding scheme is usually applied in serial multipliers. The advantage of this method is that the partial product circuit is simple and easy to implement. Therefore, this scheme is suitable for the implementation of small multipliers. The drawback is that the method is not able to efficiently handle the sign extension and it generates a number of

partial products as many as the number of bits of the multiplier, which results in many adders needed so that the area and power consumption increase. This method is not applicable for large multipliers. An example of booth multiplier which multiply the following:

Multiplicand (a) = (-3)

Multiplier (b) = (-5)

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| a | 1 | 0 | 1 | 1 | (-3) | | | | Multiplicand |
| x | 1 | 1 | 0 | 1 | (-5) | | | | Multiplier |
| Y | 0 | -A | +A | -A | | | | | Booth Recorded |
| p (0) | 0 | 0 | 0 | 0 | | | | | |
| + Y0a | 0 | 1 | 0 | 1 | | | | | First Multiplier |
| 2p (1) | 0 | 1 | 0 | 1 | | | | | Addition |
| p (1) | 0 | 0 | 1 | 0 | 1 | | | | ARS |
| + Y1a | 1 | 0 | 1 | 1 | | | | | Second Multiplier |
| 2p (2) | 1 | 1 | 0 | 1 | 1 | | | | Addition |
| p (2) | 1 | 1 | 1 | 0 | 1 | 1 | | | ARS |
| + Y2a | 0 | 1 | 0 | 1 | | | | | Third Multiplier |
| 2p (3) | 0 | 0 | 1 | 1 | 1 | 1 | | | Addition |
| 2p (3) | 0 | 0 | 0 | 1 | 1 | 1 | | | ARS |
| + Y3a | 0 | 0 | 0 | 0 | | | | | Forth Multiplier |
| 2p (4) | 0 | 0 | 0 | 1 | 1 | 1 | 1 | | Addition |
| p (4) | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | Final Product |

**Fig 3.1** An example of booth multiplier [10]

## 3.3 Modified Booth Algorithm

The modified Booth encoding (MBE), or modified Booth's algorithm (MBA), was proposed by O. L. Macsorley in 1961 [11]. The recoding method is widely used to generate the partial products for implementation of large parallel multipliers, which adopts the parallel encoding scheme.

One of the solutions of realizing high speed multipliers is to enhance parallelism which helps to decrease the number of subsequent calculation stages. The original version of the Booth algorithm (Radix-2) had two drawbacks. They are: (i) The number of add subtract operations and the number of shift operations becomes variable and becomes inconvenient in designing parallel multipliers. (ii) The algorithm becomes inefficient when there are isolated 1's. These problems are overcome by using modified Radix4 Booth algorithm which scan strings of three bits with the algorithm given below:

**Table 3.2** modified booth algorithm[9]

| $Y_{i+1}$ | $Y_i$ | $Y_{i-1}$ | Recoded Digit | Operand multiplication |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 * multiplicand |
| 0 | 0 | 1 | +1 | +1 * multiplicand |
| 0 | 1 | 0 | +1 | +1 * multiplicand |
| 0 | 1 | 1 | +2 | +2 * multiplicand |
| 1 | 0 | 0 | -2 | -2 * multiplicand |
| 1 | 0 | 1 | -1 | -1 * multiplicand |
| 1 | 1 | 0 | -1 | -1 * multiplicand |
| 1 | 1 | 1 | 0 | 0 * multiplicand |

This module recodes the 16-bit multiplier using radix 4 Booth's algorithm. Radix 4 encoding reduces the total number of multiplier digits by a factor of two, which means in this case the number of multiplier digits will reduce from 16 to 8. This algorithm groups the original multiplier into groups of three consecutive digits where the outermost digit in each group is shared with the outermost digit of the adjacent group. Each of these groups of three binary digits then corresponds to one of the numbers from the set {2, 1, 0, -1, -2}. Each recoder produces a 3-bit output where the first bit represents the number 1 and

the second bit represents the number 2. The third and final bit indicates whether the number in the first or second bit is negative. Since there are 16 input bits, there will be a total of 8 Booth recoder modules in the overall multiplier architecture. The way the outputs are determined is shown in table 4 above.

| | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| a | 1 | 0 | 1 | 1 | 1 | 1 | (-17) | | | | | | | | | | Multiplicand |
| x | 0 | 0 | 1 | 0 | 0 | 1 | (9) | | | | | | | | | | Multiplier |
| Y | | | +a | | | -2a | | +a | | | | | | | | | Booth Recorded |
| P (0) | 0 | 0 | 0 | 0 | 0 | 0 | | | | | | | | | | | |
| +Y0a | 1 | 0 | 1 | 1 | 1 | 1 | | | | | | | | | | | First Multiplier |
| 4p (1) | 1 | 0 | 1 | 1 | 1 | 1 | | | | | | | | | | | Addition |
| p (1) | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | | | | | | | | | 2ARS |
| +Y1a | 1 | 0 | 0 | 0 | 1 | 0 | | | | | | | | | | | Second Multiplier |
| 4p (2) | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | | | | | | | | | Addition |
| p (2) | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | | | | | | | 2ARS |
| +Y2a | 1 | 0 | 1 | 1 | 1 | 1 | | | | | | | | | | | Third Multiplier |
| 4p (3) | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | | | | | | | (Final Product) |

**Fig 3.2** An example of Modified Booth Multiplier [10]

CHAPTER

# 4 COMPRESSOR

## COMPRESOR

Multiplier require high amount of power and delay during the partial products addition .At this stage, most of the multiplier are designed with different kind of adders that are capable to add two/three or most 4 bits by using 4-2 compressor [12, 13].For higher order multiplications, a huge number of adders or compressor are used to perform the partial product addition. We have minimized the number adder by introducing different compressors. Binary counter property has been merged with the compressor property to develop high order compressors such as 5-3, 6-3, and 7-3 compressor [14].

## 4.1 Conventional Compressor Algorithm

So far 4-2 and 5-2 compressors are reported .The conventional 4-2 compressor structure actually compress five partial product bit into three [12, 13].The architecture can be implemented with two stages of full adder (FA) connected in series as shown in fig.4.1. The output of compressor 4-2 compressor consists of one bit in position j and two bits in position (j+1).This straight forward approaches have four XORs gate delays [15].A compressor consists of five inputs and three outputs and can be implemented with two stages of full–adders (FA) connected in series as shown in fig.4.1. Various approaches have been proposed in the literature to improve their speed. In [18], a logic circuit optimization was used in shorten the critical path as shown in fig.4.2. The large number of inverter s used in their design increases the switching activity and hence the power consumption too. In [16], the full adders are implemented using MUX and XOR gates in complementary pass transistor logic (CPL) and connected in series to form the compressor as in fig.4.1.

Multiplier architecture with XORs as building blocks is presented in [17] which exhibit a 33% performance improvement over that of a Wallace tree multiplier.

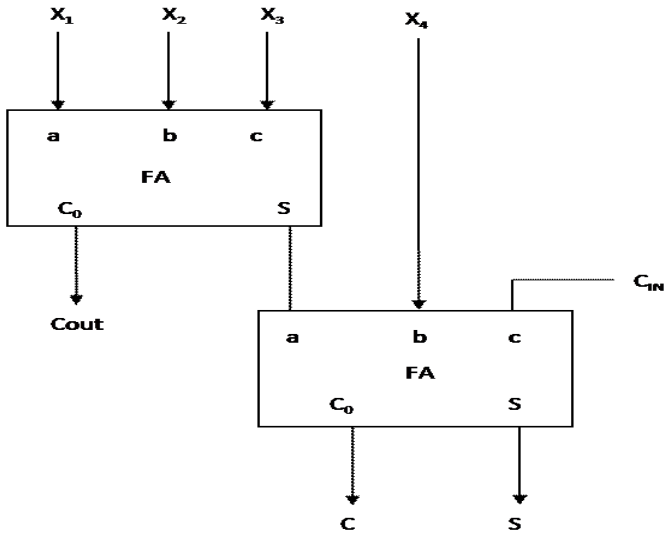



**Fig4.1:** 4-2 Compressor Composed of two FAs

They used a novel design of a 4-2 compressor based on a modified set of Equations for the sum and carry outputs of the compressor as:

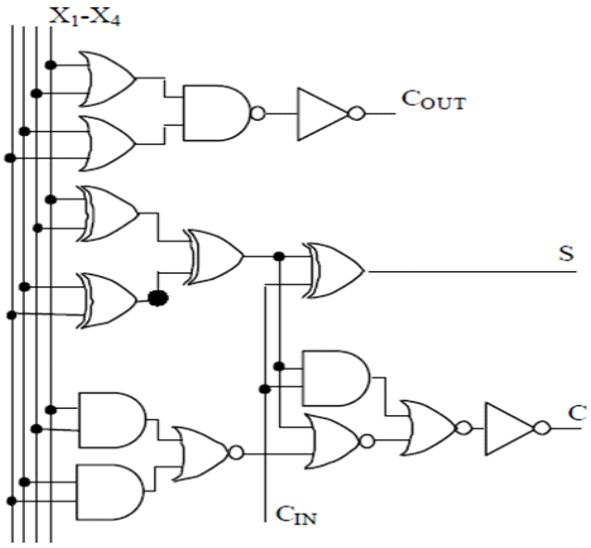



**Fig 4.2**: Compressor

$S = X_1 \oplus X2 \oplus X_3 \oplus X_4 \oplus Cin$

$C = (X_1 \oplus X_2 \oplus X_3 \oplus X_4)\, Cin + (\overline{X_1 \oplus X_2 \oplus X_3 \oplus X_4})\, X_4$

$C_{out} = (X_{\oplus} \quad X_2)\, X_3 + (X_{\oplus\text{—}} \quad X_2)\, X_1$

An equivalent gate logic realization using XORs and MUXs is shown in fig .4.3Eventhough their CPL implementation is very efficient for realizing MUXs and XORs; they need pull up circuits and inverters to minimize the reduced-swing switching as well as weak signal transmission

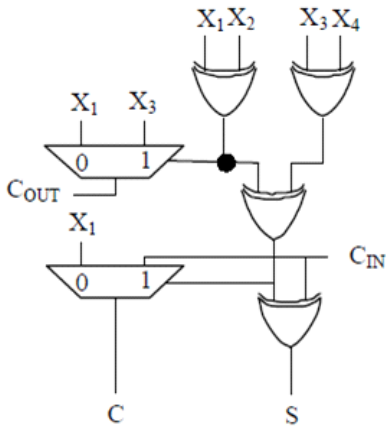



**Fig 4.3:** Compressor using XORs AND MUXs

## 4.2 Carry look ahead adder

Carry look-ahead logic uses the concepts of *generating* and *propagating* carries. Although in the context of a carry look-ahead adder, it is most natural to think of generating and propagating in the context of binary addition, the concepts can be used more generally than this.

For each bit in a binary sequence to be added, the Carry Look Ahead Logic will determine whether that bit pair will generate a carry or propagate a carry. This allows the circuit to "pre-process" the two numbers being added to determine the carry ahead of time. Then, when the actual addition is performed, there is no delay from waiting for the ripple carry effect (or time it takes for the carry from the first Full Adder to be passed down to the last Full Adder). Below is a simple 4-bit generalized Carry Look Ahead circuit that combines with the 4-bit Ripple Carry Adder we used above with some slight adjustments:

For the example provided, the logic for the generate (g) and propagate (p) values are given below. Note that the numeric value determines the signal from the circuit above, starting from 0 on the far left to 3 on the far right:

$$C_1 = G_0 + P_0.C_0$$

$$C_2 = G_1 + P_1.C_1$$

$$C_3 = G_2 + P2.C_2$$

$$C_4 = G_3 + P_3.C_3$$

Substituting $C_1$ into $C_2$, then $C_2$ into $C_3$, then $C_3$ into $C_4$ yields the expanded equations:

$$C_1 = G_0 + P_0 .C_0$$

$$C_2 = G_1 + G_0.P_1 + C_0.P_0.P_1$$

$$C_4 = G_2 + G_1.P_2 + G_0.P_1.P_2 + C_0.P_0.P_1.P_2$$

$$C_4 = G_3 + G_2.P_3 + G_1.P_2.P_3 + G_0.P_1.P_2.P_3 + C_0.P_0.P_1.P_2.P_3$$

To determine whether a bit pair will generate a carry, the following logic works:

$$G_i = A_i.B_i$$

To determine whether a bit pair will propagate a carry, either of the following logic statements work:

$$P_i = A_i \oplus B_i$$

$$P_i = A_i + B_i$$

The reason why this works is based on evaluation of $C1 = G_0+P_0.C_0$. The only difference in the truth tables between ($A \oplus B$) and ($A + B$) is when both $A$ and $B$ are 1. However, if both A and B are 1, then the $G_0$ term is 1 (since its equation is A.B), and the $P_0.C_0$ term becomes irrelevant. The XOR is used normally within a basic full adder circuit; the OR is

an alternate option (for a carry look-ahead only) which is far simpler in transistor-count terms.

The Carry Look Ahead 4-bit adder can also be used in a higher-level circuit by having each CLA Logic circuit produces propagate and generate signal to a higher-level CLA Logic circuit. The group propagates (*PG*) and group generates (*GG*) for a 4-bit CLA are:

$$PG = P_0.P_1.P_2.P_3$$

$$GG = G_3+G_2.P_3+G_1.P_3.P_2+G_0.P_3.P_2.P_1$$

Putting 4 4-bit CLAs together yields four groups propagates and four groups generates. A Look-ahead Carry Unit (LCU) takes these 8 values and uses identical logic to calculate $C_i$ in the CLAs. The LCU then generates the carry input for each of the 4 CLAs and a fifth equal to $C_{16}$.

The calculation of the gate delay of a 16-bit adder (using 4 CLAs and 1 LCU) is not as straight forward as the ripple carry adder. Starting at time of zero:

- calculation of $P_i$ and $G_i$ is done at time 1
- calculation of $C_i$ is done at time 3
- calculation of the *PG* is done at time 2
- calculation of the *GG* is done at time 3
- calculation of the inputs for the CLAs from the LCU are done at
  - time 0 for the first CLA
  - time 5 for the second CLA
  - time 5 for the third & fourth CLA
- calculation of the $S_i$ are done at
  - time 4 for the first CLA
  - time 8 for the second CLA
  - time 8 for the third & fourth CLA
- calculation of the final carry bit ($C_{16}$) is done at time 5

The maximum time is 8 gate delays (for $S_{[8\ -\ 15]}$). A standard 16-bit ripple carry adder would take 31 gate delays.

CHAPTER

# 5

# WALLACE TREE MULTIPLIER

## WALLACE TREE MULTIPLIER

Several popular and well-known schemes, with the objective of improving the speed of the parallel multiplier, have been developed in past. In 1964, C.S. Wallace observed that it is possible to find a structure, which performs the addition operations in parallel; thus resulting in less delay. Wallace introduced a different way of parallel addition of the partial product bits using a tree of carry save adders, which is known as “Wallace Tree”. In order to perform the multiplication of two numbers with the Wallace method, partial product matrix is reduced to a two-row matrix by using a carry save adder and the remaining two rows are summed using a fast carry-propagate adder to form the product. [20].

## 5.1 Carry Save Adder

CSA performs the addition of m numbers in lesser duration compared to the simple addition. It takes three numbers (a+b+c) to add together and outputs two numbers, sum and carry (s+c).It is carried out in one time unit duration .In carry –save adder, the carry(c) is bought until the last step and the ordinary addition carried out in the very last step . The most important application of a carry-save adder is to calculate the partial products in integer multiplication. This allows for architectures, where a tree of carry-save adders (also called *Wallace tree*) is used to calculate the partial products very fast. One 'normal' adder is then used to add the last set of carry bits to the last partial products to give the final multiplication result. Usually, a very fast carry-look ahead or carry-select adder is used for this last stage, in order to obtain the optimal performance [10].

Let us illustrate this adder using an example. For simple addition of three numbers, we do as following:

| | | | | | |
|---|---|---|---|---|---|
| **Carry :** | **1** | **1** | **0** | **1** | |
| I: | 1 | 2 | 3 | 7 | 9 |
| J: | 3 | 3 | 9 | 0 | 1 |
| K: | 0 | 4 | 1 | 1 | 1 |

---

| | | | | | |
|---|---|---|---|---|---|
| **Sum:** | **5** | **0** | **3** | **9** | **1** |

The carry-save addition is carried out in three steps. In the first step, computes sum without considering carries as shown below.

| | | | | | |
|---|---|---|---|---|---|
| I: | 1 | 2 | 3 | 7 | 9 |
| J: | 3 | 3 | 9 | 0 | 1 |
| K: | 0 | 4 | 1 | 1 | 1 |

---

| | | | | | |
|---|---|---|---|---|---|
| **S:** | **4** | **9** | **3** | **8** | **1** |

In the second step, compute the carry on each column ignoring sum as shown below. The important thing to notice is that the carry calculated for one column is the carry obtained from the digits in previous column.

| | | | | | |
|---|---|---|---|---|---|
| I: | 1 | 2 | 3 | 7 | 9 |
| J: | 3 | 3 | 9 | 0 | 1 |
| K: | 0 | 4 | 1 | 1 | 1 |

---

| | | | | | |
|---|---|---|---|---|---|
| **C:** | **0** | **1** | **0** | **1** | |

In the final step, add both sum and carry to obtain the final result. In the last step, the ordinary addition is carried out as shown below.

**S: 4 9 3 8 1**

**C: 0 1 0 1**

---

**Sum: 5 0 3 9 1**

Since carry "c" and "sum" can be computed independently, this achieves the goal of Converting the duration required for the addition of three numbers into the duration required for the addition of two numbers duration .The same concept can be applied to addition of binary numbers. For example,

I: 1 0 1 1

J: 0 1 0 0

K: 0 0 0 1

---

**S: 0 0 0 0**

**C: 1 1 1 1**

**---**

**Sum: 1 1 1 1 0**

From the above example, it is easily concluded that "c"and "s" can be computed from a full adder implementation .thus, the CSA implementation can be achieved with a small changes in the full adder model as shown in fig5.1.

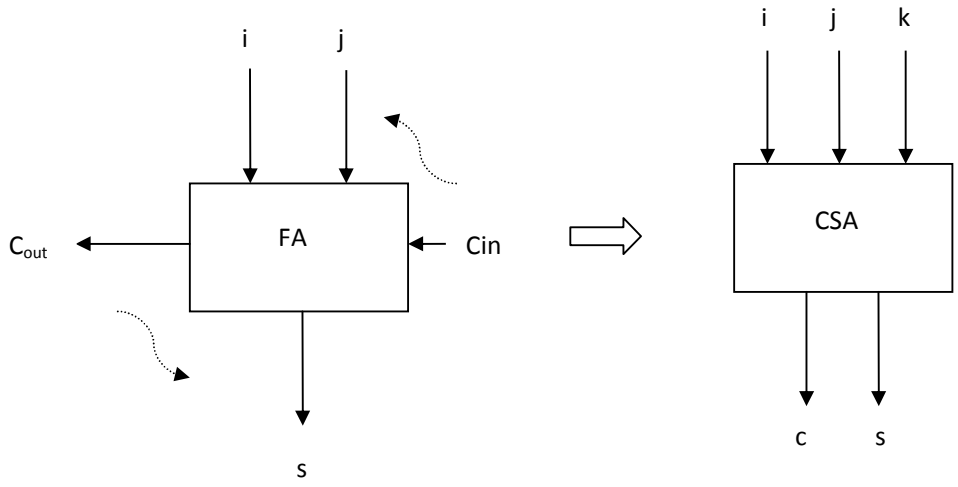



**Fig 5.1:** CSA block from full adder model

The above shown CSA block is used for each bit addition n-bit numbers. For example, three 4bit numbers are added as shown in fig5.2.

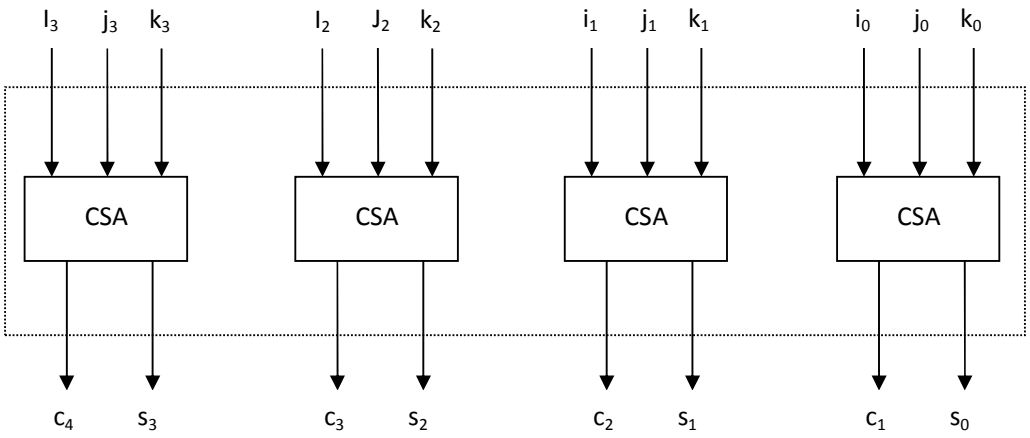



**Fig 5.2:** CSA Block with three 4-bit numbers

Now let us see how to carry out the addition of m n-bit numbers. It is shown in fig5.3. The m-2 CSA blocks shown in fig.5.2 are needed and the ordinary adder is needed in the last step. This is also explained with the example of addition of five 4-bit binary numbers. One of the important things to note is that every CSA block increase increases its size by one bit. Thus the numbers that go to the ordinary adder is n+m-2 bits long.

N0 =1001; N1=0010;  N2=1000; N3=0001; N4=0100;

N0  :  1  0  0  1

N1  :  0  0  1  0

N2  :  1  0  0  0

---

S0  :  0  0  1  1

C0  :  1  0  0  0

N3  :  0  0  0  1

---

S1  :  1  0  0  1  0

C1  :  0  0  0  0  1

N4  :  0  0  0  1

---

S2  :  0  1  0  1  0  0

C2  :  0  0  0  0  1  0

---

**Sum  : 0  0  1  1  0  0  0**

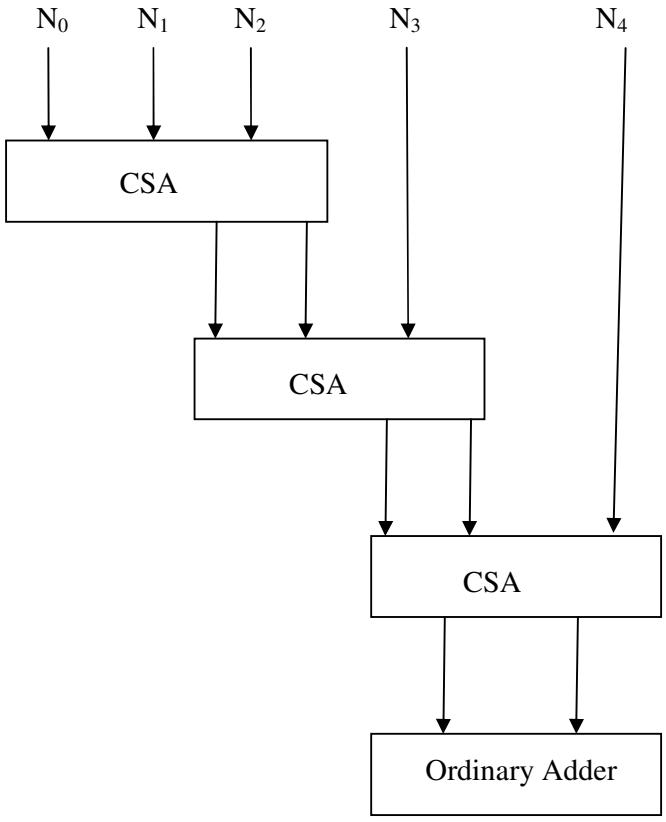



**Fig5.3:** Addition m n-bit numbers with CSA block arranged in chain fashion [19]

Instead of using CSA in a chain fashion, a tree formation can be used as shown in fig5. 4. This type or tree arrangement is called Wallace Tree. It is also explained with the example given below: $N_0$ = 1001; $N_1$=0010; $N_2$=1000; $N_3$=0001; $N_4$=0100; $N_5$=0100; $N_6$=0001; $N_7$=0100; $N_8$=0100;

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $N_0$: | 1 | 0 | 0 | 1 | $N_3$: | 0 | 0 | 0 | 1 | $N_6$: | 0 | 0 | 0 | 0 |
| $N_1$: | 0 | 0 | 1 | 0 | $N_4$: | 0 | 1 | 0 | 0 | $N_7$: | 0 | 1 | 0 | 0 |
| $N_2$: | 1 | 0 | 0 | 0 | $N_5$: | 0 | 1 | 0 | 0 | $N_8$: | 0 | 1 | 0 | 0 |

---

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **$S_{00}$:** | **0** | **0** | **1** | **1** | **$S_{01}$:** | **0** | **0** | **0** | **1** | **$S_{02}$:** | **0** | **0** | **0** | **1** |
| **$C_{00}$:** | **1** | **0** | **0** | **0** | **$C_{01}$:** | **0** | **1** | **0** | **0** | **$C_{02}$:** | **0** | **1** | **0** | **0** |

$S_{00}$: 0 0 1 1 $S_{01}$: 0 0 0 1

$C_{00}$: 1 0 0 0 $S_{02}$: 0 0 0 1

$C_{01}$: 0 1 0 0 $C_{02}$: 0 1 0 0

---

**$S_{10}$: 1 1 0 1 1** **$S_{11}$: 0 1 0 0 0**

**$C_{10}$: 0 0 0 0 0** **$C_{11}$: 0 0 0 0 1**

$S_{10}$: 1 1 0 1 1

$C_{10}$: 0 0 0 0 0

$C_{11}$: 0 0 0 0 1

---

**$S_{20}$: 0 1 1 0 0 1**

**$C_{20}$: 0 0 0 0 1 0**

$S_{20}$: 0 1 1 0 0 1

$C_{20}$: 0 0 0 0 1 0

$S_{11}$: 0 1 0 0 0

---

**$S_{30}$: 0 0 1 0 1 0 1**

**$C_{30}$: 0 0 0 1 0 0 0**

$S_{30}$: 0 0 1 0 1 0 1

$C_{30}$: 0 0 0 1 0 0 0

---

**Sum: 0 0 1 0 0 1 0 1**

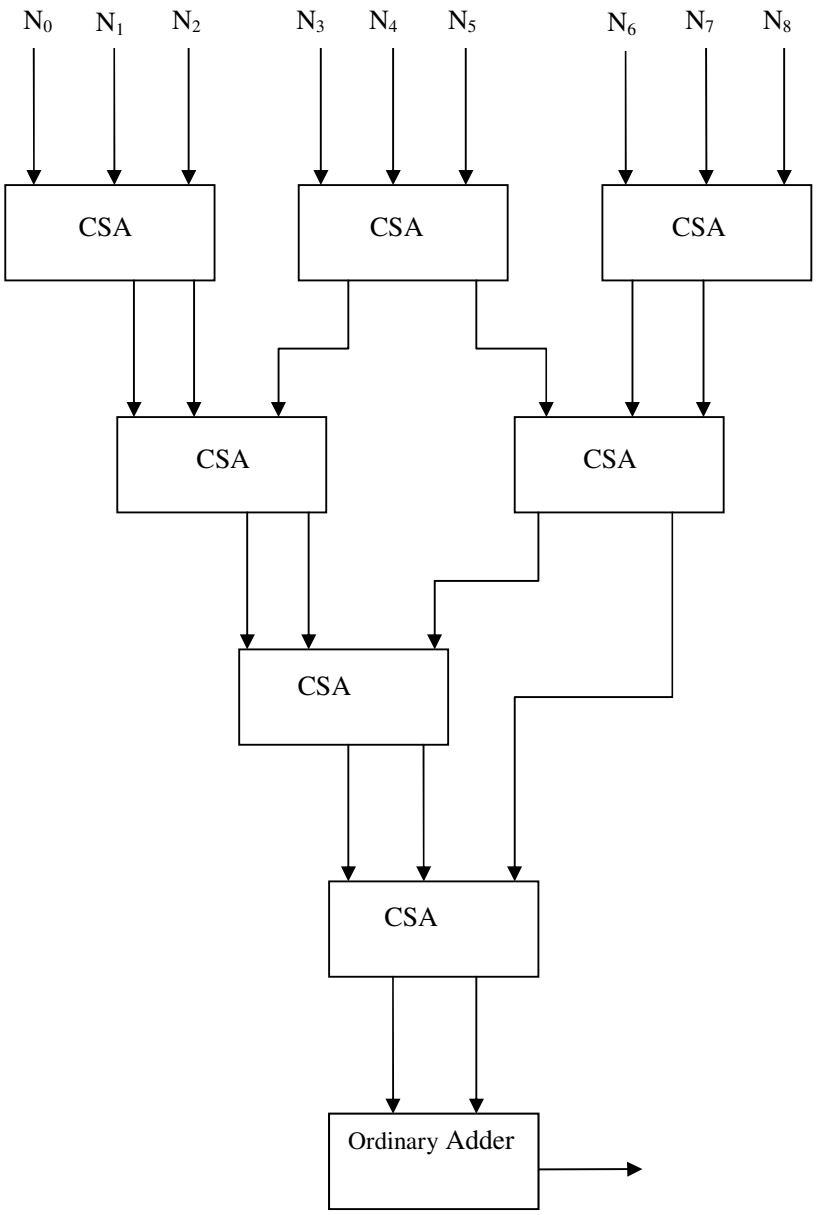



**Fig5.4:** Wallace Tree Structure using CSA [19]

## 5.2 Multiplication Algorithm

We have seen about the theory behind carry save adder (CSAs) and Wallace Tree structure. Now we will see how multiplier uses Wallace tree Structure.-The multiplier implementation involves three steps as shown in fig.55. They are Partial –Product generator, Wallace Tree structure and ordinary adder .The multiplication operation is explained in the example given below using Wallace Tree method of calculation .Multiply two 6-bit numbers (111001 and 110101) using Wallace tree Method.

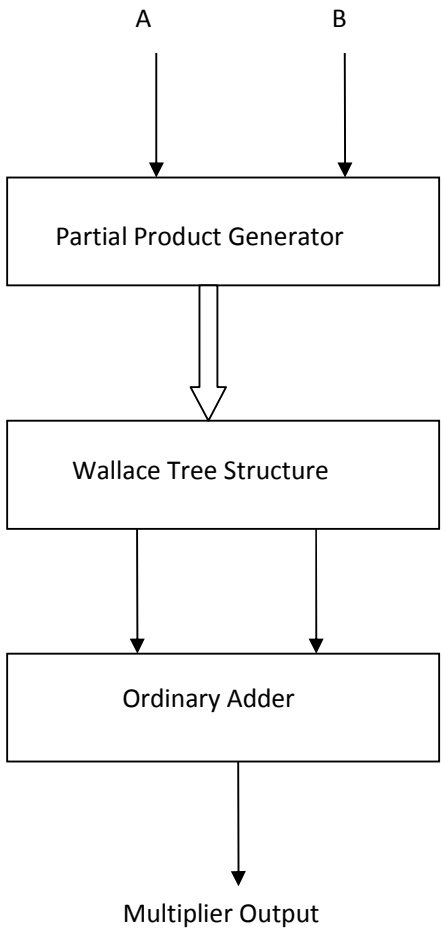



**Fig.5.5:** Multiplier Implementation –Block Diagram

### 5.2.1 Partial Product Generation

1 1 1 0 0 1 Multiplicand

1 1 0 1 0 1 Multiplier

---

1 1 1 0 0 1 (PP0)-partial product number

0 0 0 0 0 0 (PP1)

1 1 1 0 0 1 (PP2)

0 0 0 0 0 0 (PP3)

1 1 1 0 0 1 (PP4)

1 1 1 0 0 1 (PP5)

---

1 0 1 1 1 1 0 0 1 1 0 1 (Sum)

### 5.2.2 Wallace Tree Implementation

**CSA00:**

PP0: 1 1 1 0 0 1

PP1: 0 0 0 0 0 0

PP2: 1 1 1 0 0 1

S00: 1 1 0 1 1 1 0 1

C00: 0 1 0 0 0 0

**CSA01:**

PP3: 0 0 0 0 0 0

PP4: 1 1 1 0 0 1

PP5: 1 1 1 0 0 1

S01: 1 0 0 1 0 1 1 0

C01: 1 1 0 0 0 0

**CSA10:**

S00: 1 1 0 1 1 1 0 1

**C00**: 0 1 0 0 0 0

S01: 1 0 0 1 0 1 1 0

**CSA20:**

S10: 1 0 0 0 0 1 0 1 1 0 1

C10: 1 1 0 1 0 0

C01: 1 1 0 0 0 0

**Adder:**

S02: 0 1 1 1 0 0 0 1 1 0 1

C02: 1 0 0 0 0 1 0 0

---

Sum: 1 0 1 1 1 1 0 0 1 1 0 1

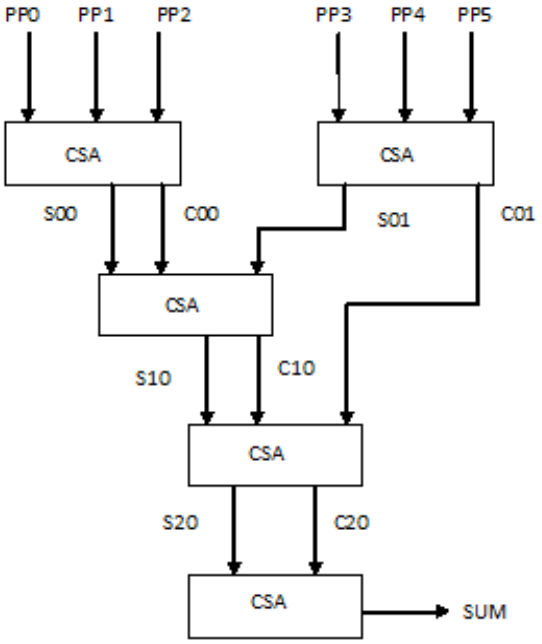



**Fig 5.6:** Wallace Tree Method

CHAPTER

# 6

# IMPLEMENTATION OF FPGA

## INTRODUCTION

FPGA stands for field programmable gate arrays that can be configured by the customer or designer after manufacturing. Field programmable gate arrays are called this because rather than having a structure similar to a PAL or other programmable device, they are structured very much like a gate array ASIC. This makes FPGAs very nice for use in prototyping ASICs, or in places where and ASIC will eventually be used [26].For example, an FPGA may be used in a design that need to get to market quickly regardless of cost. Later an ASIC can be used in place of the FPGA when the production volume increases, in order to reduce cost. FPGAs are programmed using a logic circuit diagram or a source code in a hardware description language (HDL) to specify how the chip will work. FPGAs contain programmable logic components called "logic blocks", and a hierarchy of reconfigurable interconnects that allow the blocks to be "wired together". The programmable logic blocks are called configurable logic blocks and reconfigurable interconnects are called switch boxes. Logic blocks (CLBs) can be configured to perform complex combinational functions, or merely simple logic gates like AND and XOR. In most FPGAs, the logic blocks also include memory elements, which may be simple flip-flops or more complete blocks of memory [21]

## 6.1 FPGA Implementation

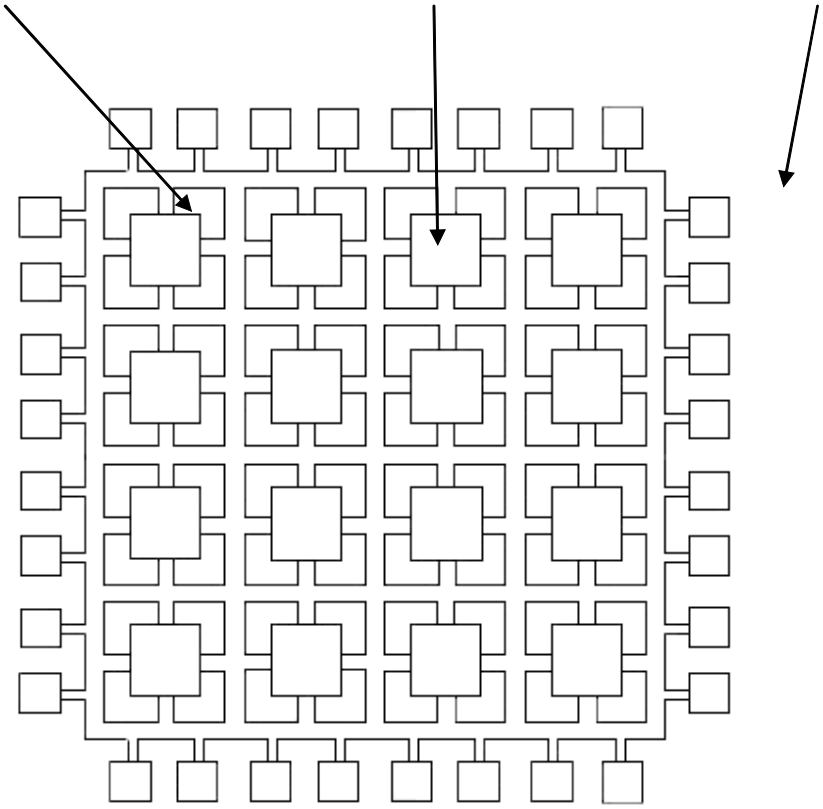
The FPGA that is used for the implementation of the circuit is the Xilinx Spartan 3E (Family), XC3S500 / XC3S1600 (Device), FG320 / FG484 (Package), -5 (Speed

Grade) FPGA device. The working environment/tool for the design is Xilinx ISE9.2i used for the FPGA implementation of Verilog code.

## 6.2 FPGA Architecture

Each FPGA vendor has its own FPGA architecture, but in general terms they are all a variation of that shown in Figure 6.1. The architecture consists of configurable logic blocks, configurable I/O blocks, and programmable interconnect. Also, there will be clock circuitry for driving the clock signals to each logic block, and additional logic resources such as ALUs, memory, and decoders may be available. The two basic types of

Logic Block Interconnection Resources I/O Cell



prog ication

circuit must be mapped into an FPGA with adequate resources. While the number of CLBs and I/Os required is easily determined from the design, the number of 3routing tracks needed may vary considerably even among designs with the same amount of logic. [22].

## 6.3 Configurable Logic Blocks

Configurable Logic Blocks contain the logic for the FPGA. In large grain architecture, these CLBs will contain enough logic to create a small state machine. In fine grain architecture, more like to a true gate array ASIC, the CLB will contain only very basic logic. The diagram in Figure 6.2 would be considered a large grain block. It contains RAM for creating arbitrary combinatorial logic functions. It also contains flip-flops for clocked storage elements, and multiplexers in order route the logic within the block and to and from external resources. The multiplexers also allow polarity selection and reset and clear input selection.

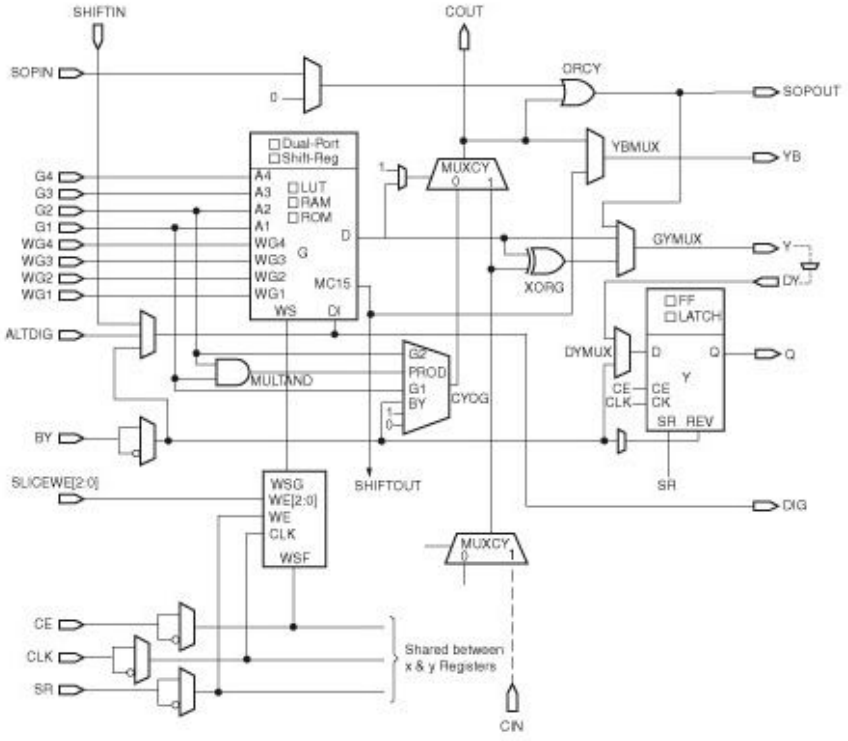



**Figure: 6.2** - The Configurable Logic Block [22]

## 6.4 The Design Flow

As the FPGA architecture evolves and its complexity increases, CAD software has become more mature as well. Today, most FPGA vendors provide a fairly complete set of design tools that allows automatic synthesis and compilation from design specifications in hardware specification languages, such as Verilog or VHDL, all the way down to a bit stream to program FPGA chips.

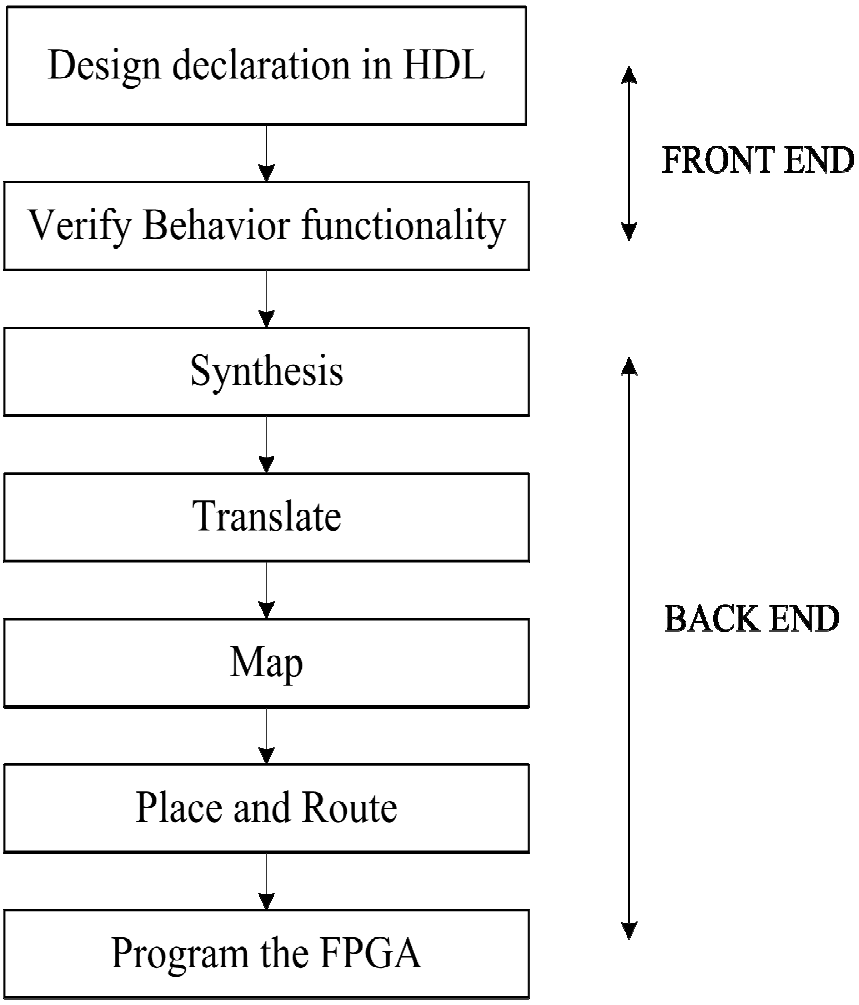



**Figure 6.3:** FPGA Design Flow

A typical FPGA design flow includes the steps and components shown in Fig. 6.1. Inputs to the design flow typically include the HDL specification of the design, design constraints, and specification of target FPGA devices. Design constraints typically include the expected operating frequencies of different clocks, the delay bounds of the signal path delays from input pads to output pads (I/O delay), from the input pads to registers (setup time), and from registers to output pads (clock-to-output delay). In some cases, delays between some specific pairs of registers may be constrained.

## 6.5 Design Implementation

The design implementation process consists of the following sub process:

(i) **Translation:** The translate process merges all of the input netlist and design constraint outputs a Xilinx NGD (Native information and Generic Database) file. The .ngd file describes the logical design reduced to the Xilinx device primitive cells. The output in the floor planner tool supplied with Xilinx ISE software. Here, defining constraints is nothing but, assigning the ports in the design to the physical elements (ex. pins, switches, buttons etc) of the targeted device and specifying time requirements of the design [23]. This information is stored in a file named UCF (User Constraints File). Tools used to create or modify the UCF are PACE, Constraint Editor Etc.

(ii) **Mapping:** The map process is run after the translate process is complete. Mapping maps the logical design described in the NGD file to the components/primitives (Slices/CLBs) present on the NCD file created by the Map process to place and route the design on the target FPGA design. In this process the whole circuit is divided into sub blocks so that they can fit into FPGA logic blocks. The logic defined in the input NGD file is mapped into targeted FPGA elements i.e. CLBs and IOBs and an output NCD (native circuit description) file is generated which depicts the design mapped into the FPGA.

(iii) **Place and Route:** After map the design is placed and routed. Place and Route (PAR) program is used for this process. The place and route process places the sub blocks from the map process into logic blocks according to the constraints and connects the logic blocks. Example if a sub block is placed in a logic block which is very near to IO pin, then it may save the time but it may affect some other constraint. So tradeoff between all the constraints is taken account by the place and route process .The PAR tool takes the mapped NCD file as input and produces a completely routed NCD file as output. Output NCD file consist the routing information. During the place process the sub blocks are placed according to the logic but these blocks do not have physical routing among them and with I/O pads also but there is only a logical connection

between them which can be clearly seen using the FPGA Editor just after the place process ends. These logical connections are shown by rats nets in FPGA Editor. Then the Route process is run which makes physical connections between the sub blocks placed on FPGA. The connections are made using the switch matrices.

**(iv)** **Bitstream Generation:** The collection of binary data used to program the reconfigurable logic device is most commonly referred to as a bit stream, although this is somewhat misleading because the data are no more bit oriented than that of an instruction set processor and there is generally no "streaming". While in an instruction set processor the configuration data are in fact continuously streamed into the internal units, they are typically loaded into the reconfigurable logic device only once during an initial setup phase.

**(v)** A programming file is generated by running the Generate Programming File process. This process can be run after the FPGA design has been completely routed. The Generate Programming File process runs BitGen, the Xilinx bitstream generation program, to produce a bitstream (.BIT) or (.ISC) file for Xilinx device configuration. The FPGA device is then configured with the .bit file using the JTAG boundary scan method. After the Spartan device is configured for the intended design, then its working is verified by applying different inputs.

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

**(vii)** **Static timing Analysis:** Three types of static timing analysis can be performed that are:

**(a)** **Post fit Static timing analysis:** The timing results of the Post fit process can be analyzed. The analyze Post fit Static Timing process opens the Timing Analyzer window, which lets you interactively select timing paths in your design for tracing.

**(b) Post map Static Timing Analyze:** You can analyze the timing results of the Map process. Post Map timing reports can be very useful in evaluating timing performance [25]. Although route delays are not accounted for the logic delay can provide valuable information about the design. If logic delay account for a significant portion (> 50 %) of the total allowable delay of a path, the path may not be able to meet your timing requirements when routing delays are added. Routing delay typically account for 45% to 65% of the total path delays. By identifying problem path, you can mitigate potential before investing time in place and route. You can redesign the logic paths to use fewer levels of logic, tag the paths for specialized routing resources, move to a faster device, or allocate more time for the path. If logic only delays account for much less (>35%) than the total allowable delay for a path or timing constraint, then the place-route software can use very low placement effort levels. In these cases, reducing effort levels allow you to decrease runtimes while still meeting performance requirements.

### (viii) Testing

For a programmable device, you simply program the device and immediately have your prototypes. You then have the responsibility to place these prototypes in your system and determine that the entire system actually works correctly. If you have followed the procedure up to this point, chances are very good that your system will perform correctly with only minor problems. These problems can often be worked around by modifying the system or changing the system software. These problems need to be tested and documented so that they can be fixed on the next revision of the chip. System integration and system testing is necessary at this point to insure that all parts of the system work correctly together. When the chips are put into production, it is necessary to have some sort of burn-in test of your system that continually tests your system over some long amount of time. If a chip has been designed correctly, it will only fail because of electrical or mechanical problems that will usually show up with this kind of stress testing.

## 6.6 Implementing the design on FPGA

The FPGA that is used for the implementation of the circuit is the Xilinx Spartan3E XC3S500 family FG320 (package) FPGA device. The working environment/tool for the design is the Xilinx ISE 9.1.Xilinx ISE simulator is used for the Behavioral, Post translate, Post map and Post place & route simulation of VHDL model

### 6.6.1 Spartan-3E FPGA specific features

- Parallel NOR Flash configuration
- MultiBoot FPGA configuration from Parallel NOR Flash PROM
- SPI serial Flash configuration

### 6.6.2 Embedded development

- Micro Blaze 32-bit embedded RISC processor
- Pico Blaze 8-bit embedded controller
- DDR memory interfaces

Implement the design and verify that it meets the timing constraints after check syntax and synthesis.

## 6.7 Procedure of Implement Design in FPGA

The Spartan-3E Starter Kit board highlights the unique features of the Spartan-3E FPGA family and provides a convenient development board for embedded processing applications. The board implements the design and verifies that it meets the timing constraints after check syntax and synthesis.

### 6.7.1 Implementing the Design

1. Select the lcd128 source file in the Sources window.
2. Open the Design Summary by double-clicking the View Design Summary process in the Processes tab.
3. Double-click the Implement Design process to view the Translate and Map Place & Route in the Processes tab.

4. Notice that after Implementation is complete, the Implementation processes have a Green check mark next to them indicating that they completed successfully without Errors or Warnings.

5. Locate the Performance Summary table near the bottom of the Design Summary.

6. Click the All Constraints Met link in the Timing Constraints field to view the Timing Constraints report. Verify that the design meets the specified timing requirements.

7. Close the Design Summary.

### 6.7.2 Assigning Pin Location Constraints

1. Verify that lcd128 is selected in the Sources window.

2. Double-click the Floor plan Area/IO/Logic - Post Synthesis process found in the User Constraints process group. The Xilinx Pin out and Area Constraints Editor (PACE) opens.

3. Select the Package View tab.

4. In the Design Object List window, enter a pin location for each pin in the Location column using the following information:

- CLK input port connects to FPGA pin C9
- A input port connects to FPGA pin L13
- B input port connects to FPGA pin L14
- SF_CE0 input port connects to FPGA pin D16
- LCD_RS input port connects to FPGA pin L18
- LCD_RW input port connects to FPGA pin L17
- LCD_E input port connects to FPGA pin M18
- LCD_4 input port connects to FPGA pin R15
- LCD_5 input port connects to FPGA pin R16
- LCD_6 input port connects to FPGA pin P17
- LCD_7 input port connects to FPGA pin M15

5. Select File to Save. You are prompted to select the bus delimiter type based on the Synthesis tool you are using. Select XST Default and click OK.

6. Close PACE.

Notice that the Implement Design processes have an orange question mark next to them, indicating they are out-of-date with one or more of the design files. This is because the UCF File has been modified.

### 6.7.3 Download Design to the Spartan™-3 Demo Board

This is the last step in the design verification process. This section provides simple Instructions for downloading the counter design to the Spartan-3 Starter Kit demo board.

1. Connect the 5V DC power cable to the power input on the demo board (J4).
2. Connect the download cable between the PC and demo board (J7).
3. Select Implementation from the drop-down list in the Sources window.
4. Select lcd128 in the Sources window.
5. In the Process window, double-click the Configure Target Device process.
6. The Xilinx web talk Dialog box may open during this process. Click Decline.
7. In the Welcome dialog box, select Configure devices using Boundary-Scan (JTAG).
8. Verify that automatically connect to a cable and identify Boundary-Scan chain is selected.
9. Click Finish.
10. If you get a message saying that there are two devices found, click OK to continue. The devices connected to the JTAG chain on the board will be detected and displayed in the impact window.
11. The Assign New Configuration File dialog box appears. To assign a configuration file To the XC3S500E device in the JTAG chain, select the counter.bit file and click Open.
12. If you get a Warning message, click OK.
13. Select Bypass to skip any remaining devices.
14. Right-click on the XC3S500E device image, and select Program. The Programming Properties dialog box opens.
15. Click OK to program the device.

When programming is complete, the Program Succeeded message is displayed. On the board, LCD Display the output of lcd128.

CHAPTER

7

# BOOTH WALLACE MULTIPLIER

**BOOTH WALLACE MULTIPLIER**

Now, we can put the suggestions by Booth and Wallace together. The partial products are generated with the booth recoder. The partial products are added with the Wallace tree adders, similar to the carry adder approach. The last row of carry and sum output is added together with the carry skewed to the left by position .Figure 1 shows a 16816 multiplier, using Booth recoders. Each recorder takes 3bits from the multiplier with '0' append the right end. The recoder has 17-bit output. Each recoder output is shifted to its correct position, sign extended, and zero filled in the right end. There are three rows of unit adder(CLA) .Each row has 32 bit unit adder .The carry and sum outputs of the last rows are added with the carry output bits shifted left one bit position to add with sum bits. The output of the adder forms the product.

## 7.1 New Multiplication Architecture

The new architecture of multiplier removing the problem of an extra CSA stage is given shown in figure 6.2. It consists of a Recoder block, Wallace tree and a carry look-ahead adder. The first block is the new modified booth algorithm and Booth is one such algorithm which is based on the fact that fewer partial products have to be generated for groups of consecutive zero's and one's. For consecutive groups of zero's and one's, there is no need to generate any new partial product. We only need to shift the previously accumulated partial product one bit position to the right for every zero in the multiplier. In this, three consecutive bits of the multiplier are examined to find the PP's. The recoding of the multiplier bits need not to be done in predetermined order from the most significant bit to the least significant bit or vice versa) and can be done in parallel in all bit positions. After the encoder block, we must add the PP's. Using ripple carry adder is

quiet slow because of carry propagation. But Wallace tree is fast because carries are saved and not propagated. We use Wallace tree with 4:2 carry save adder. An adder tree that uses 4:2 compressors will have a more regular structure, so area of Wallace tree using 4:2 carry save adder is smaller than that of using 3:2 carry save adders.

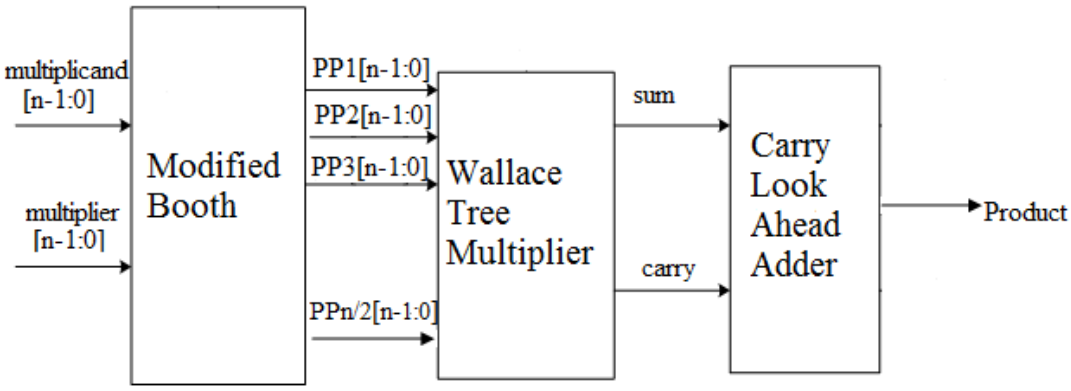



**Fig.7. 1 :** Block diagram of Booth Encoded Wallace Tree Multiplier

Here, we make Wallace tree using half adder, full adder and 4:2 carry save adders (CSA) in same stage working at the same time. After the final outputs from the Wallace tree sum, carry of Wallace tree are selected in the look ahead carry adder and then the final results of multiplication is come out.

## 7.2 SIMULATION RESULTS

### 7.2.1 Synthesis Results of 8 *8 booth Wallace multiplier:

**Device utilization summary:**

Selected Device: 3s500efg320-4

| | |
|---|---|
| Number of Slices: | 53 out of 4656 11% |
| Number of 4 input LUTs: | 102 out of 9312 13% |
| Number of IOs: | 64 |
| Number of bonded IOBs: | 32 out of 232 15% |
| Minimum period: | 8.909ns |
| Maximum Frequency: | 112.246MHz |

The simulation results and RTL schematic of the two multipliers are shown below.

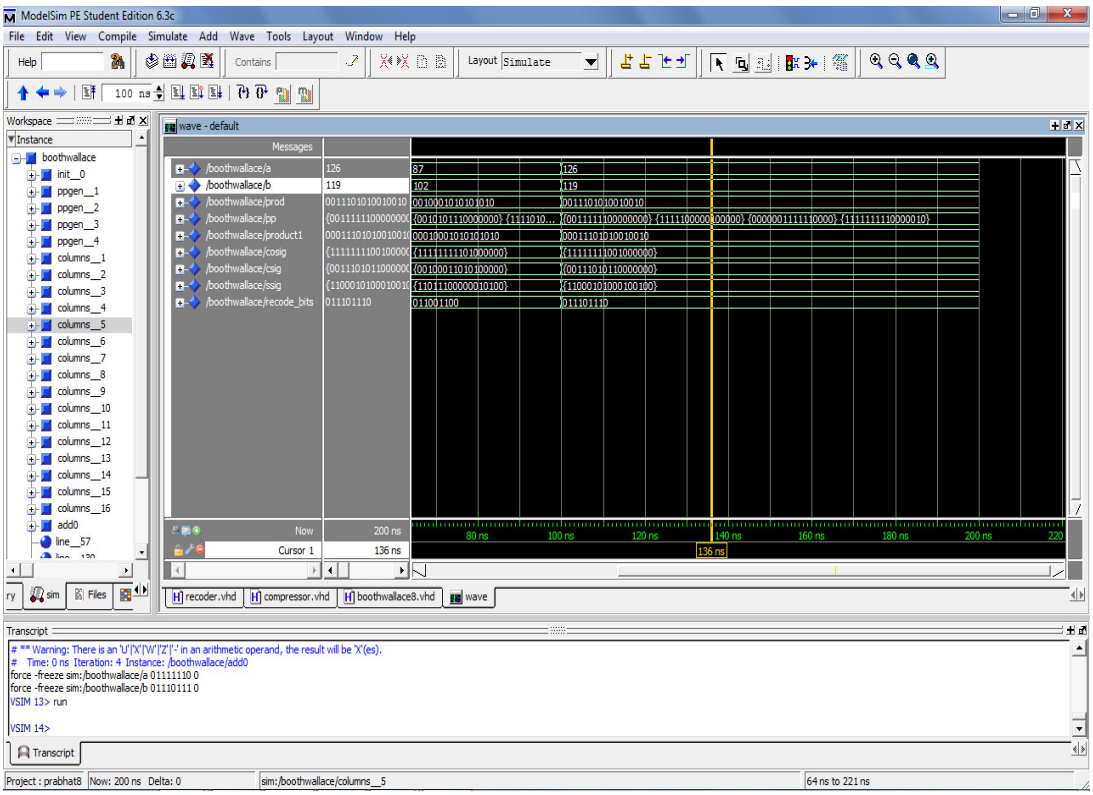



**Fig.7.2:** Simulation results of 8*8 bit Booth Wallace multiplier

## Description:

a :Input data 8-bit

b : Input data 8-bit

Product : output data 32-bit

Input a = 01111100

Input b =011101110

Output Product = 0010001010101010

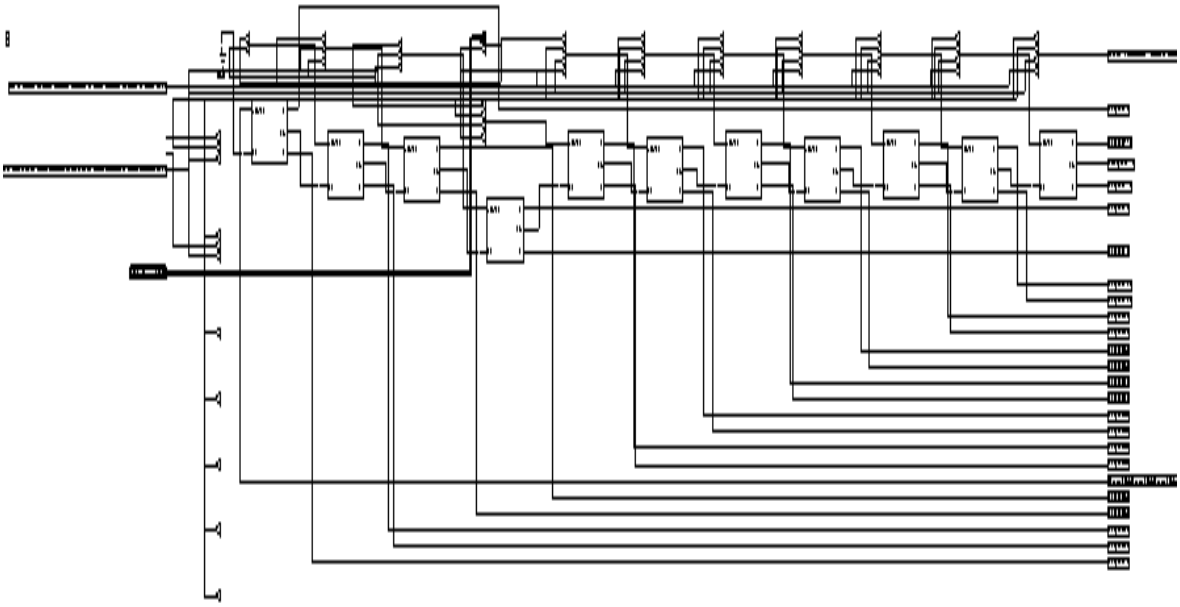


**Fig. 7.3**: RTL Schematic of 8*8 Booth Wallace multiplier

### 7.2.2 Synthesis Results of 16 * 16 booth Wallace multiplier:

**Device utilization summary:**

Selected Device: 3s500efg320-4

| | |
|---|---|
| Number of Slices: | 298 out of 4656 6% |
| Number of 4 input LUTs: | 534 out of 9312 13% |
| Number of IOs: | 64 |
| Number of bonded IOBs: | 64 out of 232 27% |
| Minimum period: | 11.358ns |
| Maximum Frequency: | 88.043MHz |

The simulation results and RTL schematic of the 16 * 16 Booth Wallace multipliers are shown below.

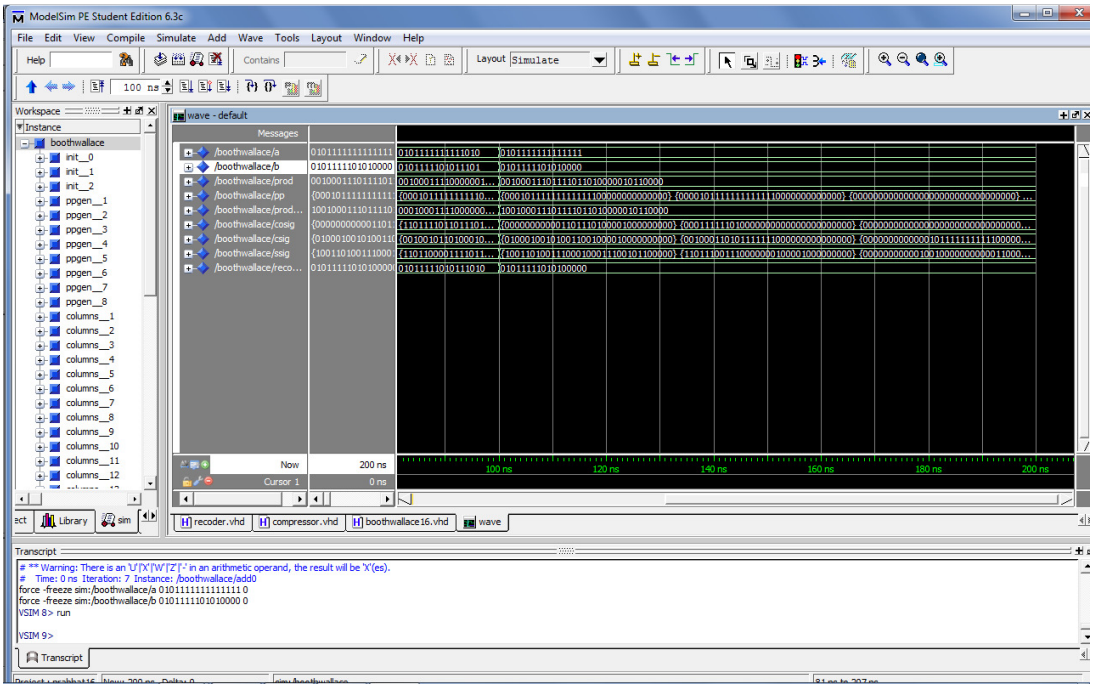



**Fig 7.4**: Simulation results of 16*16 bit Booth Wallace multiplier

**Description:**

a :Input data 16-bit

B : Input data 16-bit

Product : output data 32-bit

Input a = 0101111111111111

Input b =0101111101010000

Output Product = 00100011101111011010000010110000

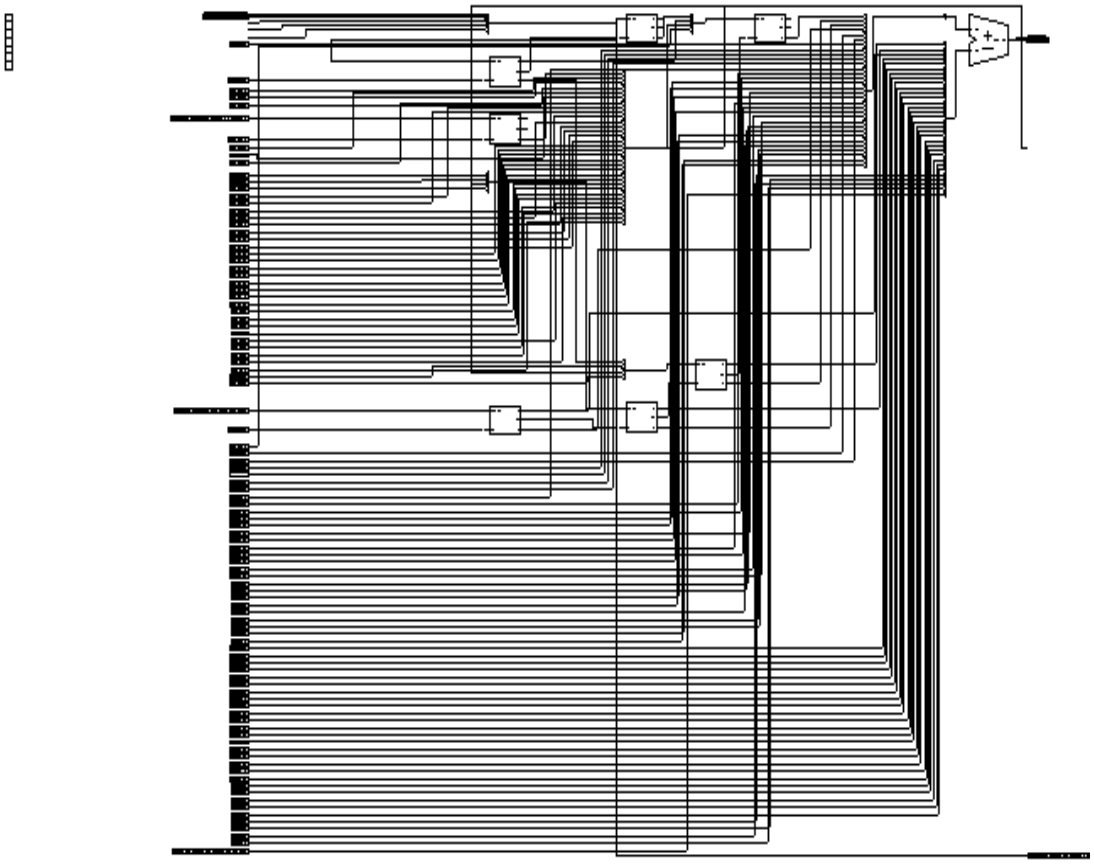


**Fig 7.5:** RTL Schematic of 16*16bit Booth Wallace multiplier

CHAPTER

8

# CONCLUSION AND FUTURE SCOPE

## 8.1 Conclusion

In this thesis report, two different multipliers have been discussed. These are 8*8 bit and 16*16 bit Booth Wallace Tree multiplier. The entire process of multiplication consists of two steps, partial product generation and partial product accumulation. Various algorithms have been discussed for partial product generation and their reduction.

In this, performance measures of these two multipliers discussed so far are summarized and compared. These results were obtained after synthesizing each architecture targeting towards Xilinx FPGA Spartan3E (family). All comparisons are based on the synthesis reports keeping one common base for comparison. That means we target same FPGA device (part number and speed grade) with same design constraints implied for the synthesis of each multiplier. We summarize area as number of LUT's, slices, Bonded I/Os, Time delay and memory Usage.

All these results are for processing 8-bit and 16- bit data and are shown in table 8.1. In table we can see that we can use 16*16 bit multiplier has some disadvantages such as number of LUT's and Time Delay with respect to 8*8 bit Booth Wallace Tree Multiplier but on the other hand we have the advantage of processing 16bits rather than 8bits. From the table, we can conclude that all the multipliers have their own advantages and disadvantages for different performance measure and choice of a particular multiplier depends upon the nature of application and constraints on area, delay and power or combination of them for that particular application.

One can select a multiplier from these tables depending on which performance criteria is most critical for a particular application. Multiplier can be optimized individually by either changing the verilog coding style or adding constraints specific to an architecture during synthesis, changing target device, part number, speed grade etc. Individual optimization is not considered the main purpose here is comparison of two multipliers with some common base for comparison.

**Table 8.1:** Comparison of 8 bit and 16 bit multipliers

| **Factors** | 8*8 Bit Booth Wallace multiplier | 16*16 Bit Booth Wallace multiplier |
|---|---|---|
| No. of Slices | 53 | 298 |
| No. of 4 I/P LUTs | 102 | 534 |
| No. of IOs | 32 | 64 |
| No. of Bonded IOs | 32/232 | 64 |
| Delay(ns) | 8.909ns | 11.234ns |
| Frequency | 112.246MHz | 88.043MHz |
| Memory Usage | 124632Kilobytes | 127312Kilobytes |

## 8.2 Future Scope

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

1. In order to enhance the performance, higher order compressors like 7:2, 9:2 can be used to accumulate the partial products.
2. In order to increase the performance of ALU's, Booth Multipliers can be used.
3. Possible measures can be taken to decrease the power consumption of the multiplier. Because there is always a trade-off between power and delay. If we try to optimize one, the other gets worse.

The ability to construct very small high performance multipliers provides many other interesting possibilities. A double precision IEEE multiplier could be placed on the same chip with an existing RISC or CISC processor. Multiplication intensive applications, such as DSP or graphics, could benefit significantly from several high performance multipliers on the same chip. A single very high throughput multiplier, or several multipliers working in parallel on the same chip, could open up new possibilities such as single chip video signal processors.

# REFERENCES


[1] Purushottam D. Chidgupkar and Mangesh T. Karad, "The Implementation of Vedic Algorithms in Digital Signal Processing", Global J. of Engng. Educ., Vol.8, No.2 © 2004 UICEE Published in Australia.

[2] Himanshu Thapliyal and Hamid R. Arabnia, "A Time-Area- Power Efficient Multiplier and Square Architecture Based On Ancient Indian Vedic Mathematics", Department of Computer Science, The University of Georgia, 415 Graduate Studies Research Center Athens, Georgia 30602-7404, U.S.A.

[3] E. Abu-Shama, M. B. Maaz, M. A. Bayoumi, "A Fast and Low Power Multiplier Architecture", The Center for Advanced Computer Studies, The University of Southwestern Louisiana Lafayette, LA 70504.

[4] Harpreet Singh Dhillon and Abhijit Mitra, "A Reduced- Bit Multiplication Algorithm for Digital Arithmetics", International Journal of Computational and Mathematical Sciences 2;2 © www.waset.org Spring 2008.

[5] Kaihong Sun," Design And implementation of a Module Generator for Low power Multiplier "Division of Electronics Systems Department of Electrical Engineering Linkoping Institute Of Technology, sweden.

[6] Kaushik Roy and Kiet-seng Yeo" "Low voltage, Low-power VLSI Subsystems", McGraw-Hill pp.124-141.

**[7]** **A**.D.Booth,"A signed binary multiplication technique, "Quarterly J.Mechan Appl.Math., vol.IV, 1951

[8] J.L Hennessy and D.A.patterson,"Computer organization and Design.The hardware /software Interface,"Second Edition,Morgan Kaufmann,998.

[9] David A. Patterson and John L.hennessy, "Computer Organization and Design: The Hardware/ Software Interface", an imprint of Elsevier, pp.250-264

[10] Behrooz parahmi "Computer Arithmetic: Algorithms and Hardware Designs" Oxford University Press 2000.

[11] O. L. MacSORLEY, “High-Speed Arithmetic in Binary Computers,” Proc. IRE, Jan. 1961.

[12] V.G. Oklobdzija, D. Villager, S.S. Liu. “A method for speed optimized partial product reduction and generation of fast parallel multipliers using an algorithmic approach”. *IEEE Trans. on Computers*, vol. 45, No. 3, March 1996.

[13] V. Oklobdzija. High speed VLSI arithmetic unit: Adders and Multipliers. in “Design of high performance microprocessor circuits”,Ed. A. Chandrakasan, *IEEE Press*, 2000.

[14] A. Dandapat**,** P. Bose, Sayan Ghosh, Pikul Sarkar, and D.Mukhopadhyay**,** “Design of an Application Specific Low-Power High Performance Carry Save 4-2 Compressor” in IEEE VLSI Design and Test Symposium 2007, VDAT-07, pp.360, 2007.

[15] S. F. Hsiao, M.R. Jiang and J.S Yeh. “Design of high speed low power 3-2 counter and 4-2 compressor for fast multipliers”. *Electronics Letters*, vol. 34, No. 4, pp 341-343, 1998.

[16] K. Yano, T. Yamanaka, T. Nishida, M. Saito, H. Shimohigashi and A. Shimizu, "A 3.8-ns CMOS 16X16-b Multiplier Using Complementary Pass-Transistor Logic," IEEE J. Solid-State Circuits, vol. 25, pp. 388-395, April 1990.

[17] D. Ghosh, S.K. Nandy and K. Parthasarathy, "TWTXBB: A Low Latency, High Throughput Multiplier Architecture Using a New 4-2 Compressor," 7th Intl. Conf. on VLSI Design, Calcutta, India, pp. 77-82, Jan. 1994.

[18] M. Nagamatsu, S. Tanaka, J. Mori, T. Noguchi and K. Hatanaka, "A 15nS 32X32-bit CMOS Multiplier with an Improved Parallel Structure," Proc. IEEE Custom Integrated Circuits Conf., pp. 10.3.1-10.3.4, 1989.

[19] M.J. Flynn “Advanced Computer Arithmetic EE486” Feb, 2003, Fact Sheet. Winter Quarter 02-03

[20] Moises E.Robinson and Earl Switzerland, Jr “A Reduction Schem to optimize the Wallace Multiplier “Department of Electrical and computer Engineering, University of Texas at Austin, USA.

[21] Deming Chen, Jason Cong, and Peichan Pan, “FPGA Design Automation: A Survey”, Foundations and Trends in Electronic Design Automation Volume 1 Issue 3, November 2006.

[22] Rao, P.R. Chakrabarti “High performance compensation technique for radix-4 CORDIC algorithm”, IEEE September 2002.

[23] Y. H. Hu and S. Naganathan, “An angle recoding method for CORDIC algorithm implementation,” IEEE Trans. on Computers, vol. 42, pp. 99-102, Jan. 1993.

[24] Y. H. Hu, “CORDIC-based VLSI architectures for digital signal processing” ,IEEE Signal processing Magazine, pp. 16-35, July 1992.

[25] Y. C. Lim, R. Yang, D. Li, and J. Song, “Signed power-of-two term allocation scheme for the design of digital filters,” IEEE Trans. Circuits Syst. II, vol. 46, pp. 577-584, May 1999 Lecture , IEEE, june 2001.

[25] Alireza Sarvi, Dr. San Jose and Jenny Fan, “Automated BIST-based Diagnostic Solution for SOPC”, *ieeexplore.ieee.org/iel5/11202/36064/01708692.*

[26] Hamill R. Mccanny “Online CORDIC algorithm and VLSI architecture for implementating OR–array processor”, IEEE February 2000.