Dissertation on

Dadda Multiplier based hardware for convolution operation

Submitted in partial fulfilment of the requirements

for the award of the degree of

Master of Technology

in

VLSI Design and Embedded System

Submitted by:

VinodMeena

(Roll No. 2K14/VLS/21)

Under the Guidance of

Dr. S. Indu

(Associate Professor)

Department of Electronics and Communication Engineering

Delhi Technological University

Main Bawana Road Delhi-110042

CERTIFICATE

This is to certify that the dissertation titled "**Dadda multiplier based hardware for convolution operation**" is a bonafide record of work done by **Vinod Meena**, Roll No. **2K14/VLS/21** at Delhi Technological University in partial fulfillment of the requirements for the award of degree of Master of Technology in VLSI Design and Embedded System. This project was carried out under my supervision and has not been submitted elsewhere, either in part or full, for the award of any other degree or diploma to the best of my knowledge and belief.

Date: ……………………

Dr. S. Indu

Associate Professor

Department of Electronics & Communication Engineering

Delhi Technological University, Delhi

ACKNOWLEDGEMENT

I would like to express my deep sense of respect and gratitude to my project supervisor **Dr. S. Indu**, Associate Professor, Department of Electronics and Communication Engineering, Delhi Technological University Delhi for providing the opportunity of carrying out this project and being the guiding force behind this work. I am deeply indebted to her for the support, advice and encouragement she provided without which the project could not have been a success.

A special thanks to all my friends especially Mr. Shobhit Shrivastava and Mr. Ashwani Goel for their knowledge and investigation has helped me unconditionally to solve various problems.

I would also like to acknowledge Delhi Technological University for providing the right academic resources and environment for this work to be carried out. Last but not the least I would like to express sincere gratitude to my parents for constantly encouraging me during the course of work.

Vinod Meena

University Roll no: 2K14/VLS/21

M.Tech. (VLSI Design and Embedded System)

Department of Electronics & Communication Engineering

Delhi Technological University, Delhi

ABSTRACT

Multiplier is a central block in the Digital Signal Processor (DSP). In order to improve speed of processing, a hardware convolution unit is embedded in the design of multiplier. Convolution unit performs multiplication and addition process. Basic Convolution unit consists of multiplier, adder. Generally convolution unit is designed using different Multiplier and adder as Carry Save Adder (CSA). The proposed Convolution unit is designed using Dadda Multiplier (DM) and adder as Logically Optimized Full Adder (LOFA).However in the proposed model all traditional full adders are replaced by improved full adder. The performance analysis of Convolution unit models in terms of area, delay and power are compared. Various Convolution unit models are designed using Verilog HDL. Simulation and synthesis are done using Xilinx ISE 14.7 for Virtex-7 family 40nm technology device. The power is calculated using Lattice Diamond Design suite software.

TABLE OF CONTENTS

CERTIFICATE	II
ACKNOWLEDGEMENT	III
ABSTRACT	IV
TABLE OF CONTENTS	V
LIST OF TABLES	VIII
LIST OF FIGURES	IX
LIST OF ABBREVIATIONS	XI

CHAPTER 1 INTRODUCTION **1-2**

1.1 Introduction 1

1.2 Motivation 1

1.3 Applications 1

1.4 Outline of the Thesis 2

CHAPTER 2 LITERATURE SURVEY **3-17**

2.1 Digital System Design 3

2.1.1 Combinational and Sequential Circuits 3

2.2 Adders 4

2.2.1 Half adder 5

2.2.2 Full adder 6

2.2.3 Logically Optimized Full adder 7

2.2.4 Ripple Carry Adder 9

2.2.5 Carry Increment Adder 10

2.2.6 Carry Save Adder 10

2.3. Multipliers 11

2.3.1 Array Multiplier 12

2.3.2 Ripple Carry Array Multiplier with Row Bypassing Technique 13

2.3.3 Wallace Tree Multiplier 14

2.3.4 Dadda Multiplier 15

2.4 Conclusion 17

CHAPTER 3 PROPOSED MODELS 18-19

3.1 Proposed Convolution Model 18

CHAPTER 4 RESULTS 20-44

4.1 Tabular output for 16 different input combinations used 21

4.2 Waveform output for 16 different input combinations used 29

4.3 Simulation Results for Adders 37

4.3.1 Half Adder 38

4.3.2 Traditional Full Adder 39

4.3.3 Logically Optimized Full Adder 40

4.4 Simulation Results of Dadda Multipliers 40

4.5 Simulation results of convolution 42

4.6 Performance analysis of adders 43

4.7 Performance comparison of convolution unit 44

CHAPTER 5 CONCLUSION AND FUTURE WORK 45

5.1 CONCLUSION 45

5.2 FUTURE WORK 45

REFERENCES

APPANDIX A

LIST OF TABLES

2.1 Performance Comparison of Various Adders for 8 bit application 4-5

2.2 Performance Comparison of Carry Save Adder and Carry Increment adder for 16 bit application 5

2.3 Half Adder Truth Table 6

2.4 Full Adder Truth Table 7

2.5 Logical effort for inputs of static CMOS gates 9

2.6 Carry Save Adder Computation Flow 11

2.7 Performance Comparison of Various Multipliers 12

4.1 Device Utilization Summary for Half Adder 35

4.2 Device Utilization Summary for Traditional Full Adder 36

4.3 Device Utilization Summary for Logically Optimized Full Adder 37

4.4 Device Utilization Summary for Dadda Multiplier 39

4.5 Device Utilization Summary for Proposed Dadda Multiplier 41

4.6 Performance Analysis of Single Bit Adders 41

4.7 Advanced HDL Synthesis Report 43

4.8 FPGA Results of Proposed Convolution Model 43

4.9 Timing Summary of Proposed Convolution Model 43

LIST OF FIGURES

2.1 Block diagram for elementary combination circuits 3

2.2 Block diagram for elementary sequential circuits 4

2.3 Half Adder(HA) 6

2.4 Full Adder(FA) 7

2.5 Logically Optimized Full Adder(LOFA) 8

2.6 Ripple Carry Adder (RCA) 9

2.7 Carry Increment Adder (CIA) 10

2.8 Carry Save Adder (CSA) 11

2.9 Array Multiplier (AM) 13

2.10 Structure of 8x4 Ripple Carry Array Multiplier with Row Bypass 14

2.11 Structure of 8x8 Ripple Carry Array Multiplier with Row Bypass 14

2.12 Wallace Tree Multiplier (WT) 15

2.13 Dadda Multiplier Reduction 16

2.14 Dadda Multiplier (DM) Algorithm 17

3.1 Block diagram for the Proposed Convolution Model 18

4.1 Timing waveform for x(n)=h(n)=[0 0 0 0] 27

4.2 Timing waveform for x(n)=h(n)=[0 0 0 1] 27

4.3 Timing waveform for x(n)=h(n)=[0 0 1 0] 28

4.4 Timing waveform for x(n)=h(n)=[0 0 1 1] 28

4.5 Timing waveform for x(n)=h(n)=[0 1 0 0] 29

4.6 Timing waveform for x(n)=h(n)=[0 1 0 1] 29

4.7 Timing waveform for x(n)=h(n)=[0 1 1 0] 30

4.8 Timing waveform for x(n)=h(n)=[0 1 1 1] 30

4.9 Timing waveform for x(n)=h(n)=[1 0 0 0] 31

4.10 Timing waveform for x(n)=h(n)=[1 0 0 1] 31

4.11 Timing waveform for x(n)=h(n)=[1 0 1 0] 32

4.12 Timing waveform for x(n)=h(n)=[1 0 1 1] 32

4.13 Timing waveform for x(n)=h(n)=[1 1 0 0] 33
4.14 Timing waveform for x(n)=h(n)=[1 1 0 1] 33
4.15 Timing waveform for x(n)=h(n)=[1 1 1 0] 34
4.16 Timing waveform for x(n)=h(n)=[1 1 1 1] 34
4.17 Technology View and RTL View of Half Adder 35
4.18 Timing Waveform of Half Adder 35
4.19 Technology View and RTL View of Traditional Full Adder 36
4.20 Timing Waveform of Traditional Full Adder 36
4.21 Technology View and RTL View of Logically Optimized Full Adder 37
4.22 Timing Waveform of Logically Optimized Full Adder 37
4.23 Technology View and RTL View of Dadda Multiplier 38
4.24 Timing Waveform of Dadda Multiplier 38
4.25 Technology View and RTL View of Proposed Convolution 39
4.26 Hardware implemented output for x[n]=h[n]=[1 1 1 1] for convolution 40

ABBREVIATIONS

ADD	Adder
RCA	Ripple Carry Adder
CIA	Carry Increment Adder
CLAA	Carry Look Ahead Adder
CSA	Carry Save Adder
CS*l*A	Carry Select Adder
CBA	Carry Bypass Adder
AM	Array Multiplier
WTM	Wallace Tree Multiplier
RCAM RB	Ripple Carry Array Multiplier with Row Bypass Technique
DM	Dadda Multiplier
VM	Vedic Multiplier
HA	Half Adder
FA	Full Adder
LOFA	Logically Optimized Full Adder
ISE	Integrated System Environment
HDL	Hardware Description Language

CHAPTER 1

INTRODUCTION

1.1 INTRODUCTION

Convolution being an operation between two operands provides for mathematical framework in Digital Signal Processing (DSP). It is one the fundamental building block involved in DSP.

On receiving a signal, removal of less significant signal components is very important before further processing. Filtering, is a solution to remove unwanted insignificant signal components. This filtering is assisted by convolution. A large number of image processing options involve convolution filtering. Rotating or scaling an image is one such option.

1.2 MOTIVATION

Rapid industrialization and technological advances have paved way for more and more digital systems to occupy spaces in all spheres of life. Ranging from day to day work to rigorous scientific researches digital systems have evolved a lot.

This in turn developed need for low power consuming efficient basic elements to serve the purpose. Recent researches implementing the Dadda multiplier or the logically optimized full adder (LOFA) reducing the power consumption during convolution motivated this research work towards introducing and speculating the performance of the convolution operation using both Dadda multiplier and LOFA.

1.3 APPLICATIONS

. Convolution finds itself involved in various applications such as

- As a convolution filter in digital image processing
- In digital data processing, for example in analytical chemistry for smoothing filters, the moving average etc.
- In acoustics, electrical engineering and physics
- In radiotherapy

- Convolutional neural networks too are proving advantageous in artificial intelligence (AI)

Thus, due to varied uses, a method for convolution with higher efficiency and minimal power usage is in demand.

1.4 OUTLINE OF THE THESIS

In this thesis, we speculate the performance of the modified convolution scheme using LOFA and dada multiplier by implementing the same on the available Spartan kits. Preliminary knowledge and previously established information content about adders, multipliers and convolution are discussed in Chapter 2. Chapter 3 deals with the proposed modified convolution model as an introduction to it. Simulation and hardware implemented results are tabulated and shown in figures in Chapter 4. Chapter 5 discusses about the future possibilities and concludes the thesis.

CHAPTER 2
LITERATURE SURVEY

This chapter provides details about the previously established work in this domain. As concern to this research topic, during research session the references of many important involvements were very helpful during research.

2.1 DIGITAL SYSTEM DESIGN

Digital system design is the one which uses digital information in the form of alphabets, numbers and other information bits commonly termed as digital discrete inputs [2]. These inputs are then processed by a system designed to do a specific task or to multitask ranging from simple arithmetic to complex jobs. This results in output which is generally human interpretable. Such a system which processes this discrete information is a digital system.

Two common modes of circuits exist based on the use of memory, viz. combinational circuits and sequential circuits.

2.1.1 COMBINATIONAL AND SEQUENTIAL CIRCUITS

Combinational circuits are the ones which do not involve memory element thus removing from the picture the effect of previous input on the present output. Thus at any point of time the output of such a circuit depends upon the inputs involved at that instant of time. It has ‘m’ inputs and ‘n’ outputs.

Examples:- decoders, encoders, multiplexers etc.

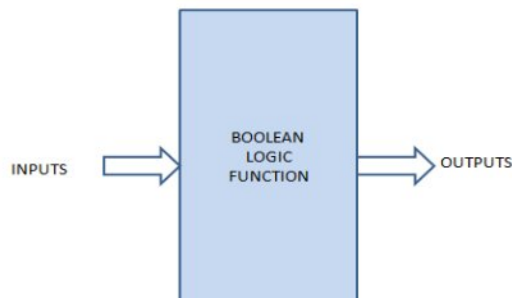

Fig. 2.1 Block diagram for elementary combinational circuits

Sequential Circuits are the ones which involves memory reflecting the effect of previous input on the output. In basic terms it is just a combinational circuit with a memory element involved. Examples- Flip Flops etc.

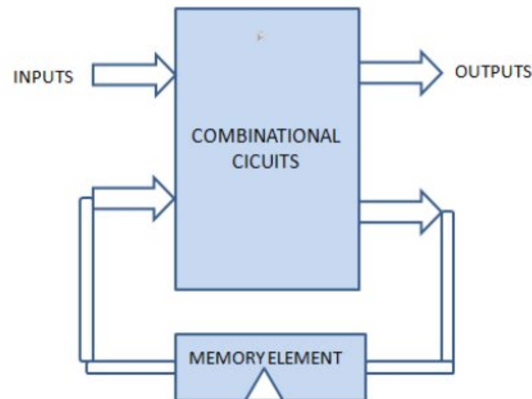

Fig. 2.2 Block diagram for elementary sequential circuits

2.2 ADDERS

Adders are the basic building blocks in Digital Design and are inseparable part of digital signal processing applications. Numerous advanced blocks like subtractor, multiplier, divider, and address calculator are obtained from Adders. As Addition is the base of all these operations. First three adders discussed here are single bit adders as they can perform addition of single bit numbers only therefore they are called single bit adders. After which only important parallel N bit adders namely Carry Save Adder (CSA), Carry Increment Adder (CIA), Ripple Carry Adder (RCA) are discussed in this section. Apart from this other parallel adder topologies are also available but we have not included them because their delay is higher than Ripple Carry Adder and there area is also same or higher [6].

S.No.	Design	Area (LUT's)	Area (Slices)	Delay (ns)
1.	Ripple Carry Adder (RCA)	8	5	2.191
2.	Carry Skip Adder (CSkA)	8	6	2.267
3.	Carry Increment Adder (CIA)	8	5	1.907
4.	Carry Look Ahead Adder (CLAA)	10	5	2.266
5.	Carry Save Adder (CSA)	13	9	1.433

6.	Carry Select Adder (CS*l*A)	8	5	2.588
7.	Carry Bypass Adder (CBA)	12	6	3.160

Table 2.1 Performance Comparison of Various Adders for 8 bit application

S.No.	Design	Area (Slices)	Delay (ns)
1	Carry Increment Adder (CIA)	22	14.32
2	Carry Save Adder (CSA)	23	19.8

Table 2.2 Performance Comparison of Carry Save Adder and Carry Increment adder for 16 bit application

It is found that for 8 bit addition applications Carry Save Adder provides the least delay at cost of increase in area by roughly 50% whereas Carry Increment adder provides good speed without compromising with area. Whereas for 16 bit addition applications Carry Increment adder is better than Carry Save Adder [4]. The same results can be verified from the tables given above. Table 2.1 shows performance comparison of various adders for 8 bit application [5]. While performance comparison of Carry Save Adder and Carry Increment adder for 16 bit application [4] is described in Table 2.2.

2.2.1 HALF ADDER

The half-adder adds two single binary digits A and B. It has two outputs, sum (S) and carry (C). The carry signal represents an overflow into the next digit of an addition. The basic half-adder design using an XOR gate for S and an AND gate for C. The half adder adds two input bits and produces a carry and sum. The truth table and logic diagram for the half adder are shown in figure 2.1. The characteristic equations for half adder are as follows:

$$\text{Sum} = A \oplus B$$

$$\text{Carry} = A.B$$

Fig. 2.3: Half Adder

Inputs		Outputs	
A	B	C	S
0	0	0	0
1	0	0	1
0	1	0	1
1	1	1	0

Table 2.3: Half Adder Truth Table

2.2.2 FULL ADDER

A full adder adds three one-bit numbers, usually written as A, B, and C_{in}; A and B are the operands, and C_{in} is a bit carried in from the previous less significant stage. The circuit produces a two-bit output, output carry and sum represented by the C_{out} and S. Here P and G are internal signals termed as propagate and generate signal respectively. The logic diagram and truth table for the full adder are shown in figure 2.2. The characteristic equations for traditional full adder are as follows:

Propagate Signal $P = A \oplus B;$

Generate Signal $G = A.B;$

Carry Out $C_{out} = A.B + (A \oplus B).(C_{in});$

Sum $S = A \oplus B \oplus C$

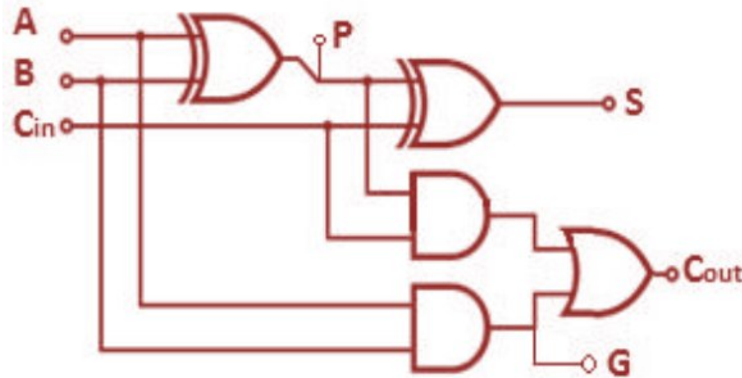

Fig. 2.4: Full Adder

Inputs			Outputs			
A	B	C_{in}	G	P	C_{out}	S
0	0	0	0	0	0	0
0	0	1	0	0	0	1
0	1	0	0	1	0	1
0	1	1	0	1	1	0
1	0	0	0	1	0	1
1	0	1	0	1	1	0
1	1	0	1	0	1	0
1	1	1	1	0	1	1

Table 2.4: Full Adder Truth Table

2.2.3 LOGICALLY OPTIMIZED FULL ADDER

In [4] R.Uma and P.Dhavachelvan proposed a logically optimized full adder. This adder incorporates two XOR gates and one 2X1 Multiplexer. They have simulated 20 different Boolean expressions for the full adder operation. The performance of all the full adders has been analysed in terms of delay, transistor count and power dissipation. It is observed that adder designed with XOR and MUX has the least delay, transistor count and power dissipation when compared to other combinations of gate. So the adder realized with MUX and XOR is considered to be the optimized adder in terms of delay, transistor count and power dissipation. The logic diagram for this full adder is shown in figure 2.3.The characteristic equations for Logically Optimized Full adder are as follows:

Sum $S = A \oplus B \oplus C;$

Carry Out $C_{out} = \overline{(A \oplus B)}.B + (A \oplus B).C$

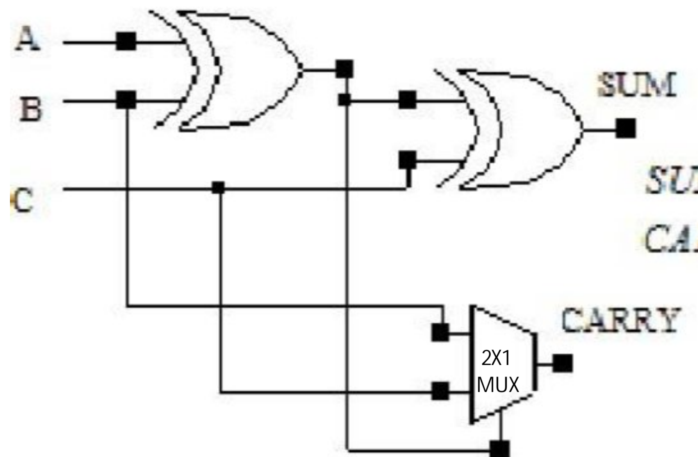

Fig. 2.5 Logically Optimized Full Adder

Reason: This Full Adder architecture uses 2X1 Multiplexer for carry computation instead of two AND and one OR gate which optimizes adder in terms of delay, transistor count and power dissipation. Because logical effort of Multiplexer is 2 while logical effort of replaced carry circuit is higher.

The logical effort of a logic gate tells how much worse it is at producing output current than is an inverter, given that each of its inputs may contain only the same input capacitance as the inverter. Reduced output current means slower operation, and thus the logical effort number for a logic gate tells how much more slowly it will drive a load than an inverter would. Equivalently, logical effort is how much more input capacitance a gate presents to deliver the same output current as an inverter [1].

Gate type	Number of inputs					
	1	2	3	4	5	n
inverter	1					
NAND		4/3	5/3	6/3	7/3	$(n+2)/3$
NOR		5/3	7/3	9/3	11/3	$(2n+1)/3$
multiplexer		2	2	2	2	2
XOR (parity)		4	12	32		

Table 2.5: Logical effort for inputs of static CMOS gates

It is interesting but not surprising to note from Table 1.1 that more complex logic functions have larger logical effort. Moreover, the logical effort of most logic gates grows with the number of inputs to the gate. Larger or more complex logic gates will thus exhibit greater delay.

2.2.4 RIPPLE CARRY ADDER (RCA)

Half Adders can be used to add two one bit binary numbers. It is also possible to create a logical circuit using multiple full adders to add N-bit binary numbers. The full adder inputs a **Cin**, which is the **Cout** of the previous adder. This type of adder is a **Ripple Carry Adder (RCA) [6]**, since each carry bit "ripples" to the next full adder. RCA contains series structure of Full Adders (FA); each FA is used to add two bits along with carry bit. Only the first full adder can be substituted by a half adder. The carry generated from each full adder is given to next full adder and so on. Hence, the carry is propagated in a serial computation. Hence, delay is more as the number of bits is increased in RCA. The 8bit RCA is shown in figure 2.4:

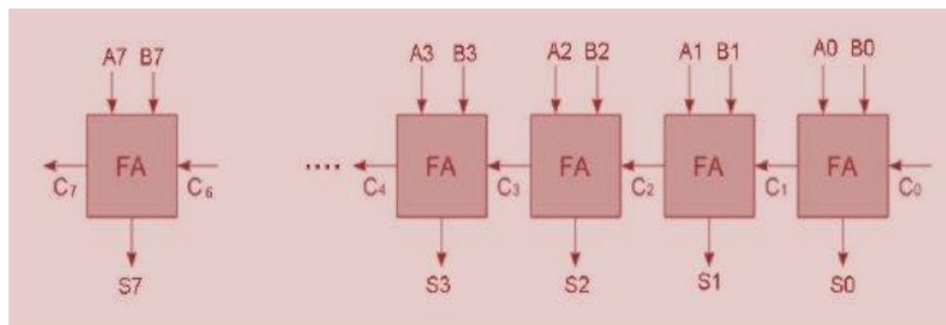

Fig.2.6: Ripple Carry Adder (RCA)

2.2.5 CARRY INCREMENT ADDER (CIA)

The design of Carry Increment Adder (CIA) consists of RCA's and incremental circuitry [7]. The incremental circuit can be designed using HA's in ripple carry chain with a sequential order. The addition operation is done by dividing total number of bits in to group of 4bits and addition operation is done using several 4bit RCA's. The architecture of CIA is shown in Fig 2.3.

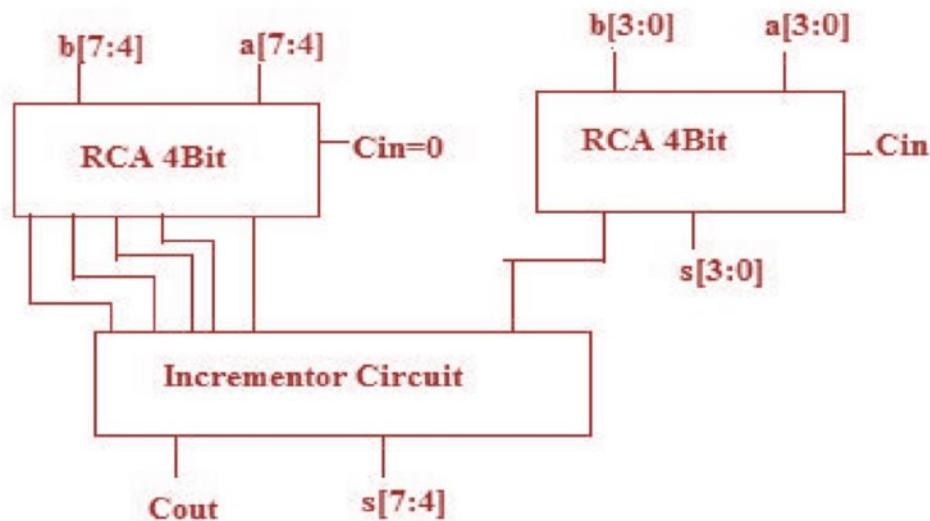

Fig.2.7: Carry Increment Adder (CIA)

2.2.6 CARRY SAVE ADDER (CSA)

The carry-save adder reduces the addition of 3 numbers to the addition of 2 numbers. The propagation delay is 3 gates regardless of the number of bits. The carry-save unit consists of n full adders, each of which computes a single sum and carries bit based solely on the corresponding bits of the three input numbers. The entire sum can then be computed by shifting the carry sequence left by one place and appending a 0 to the front (most significant bit) of the partial sum sequence and adding this sequence with RCA produces the resulting n + 1-bit value. This process can be continued indefinitely, adding an input for each stage of full adders, without any intermediate carry propagation. The main application of carry save algorithm is, well known for multiplier architecture is used for efficient CMOS implementation of much wider variety of

algorithms for high speed digital signal processing. In this scheme, the carry is not propagated through the stages. Instead, carry is stored in present stage, and updated as addend value in the next stage. Hence, the delay due to the carry is reduced in this scheme. The architecture of CSA is shown in Fig 2.4.

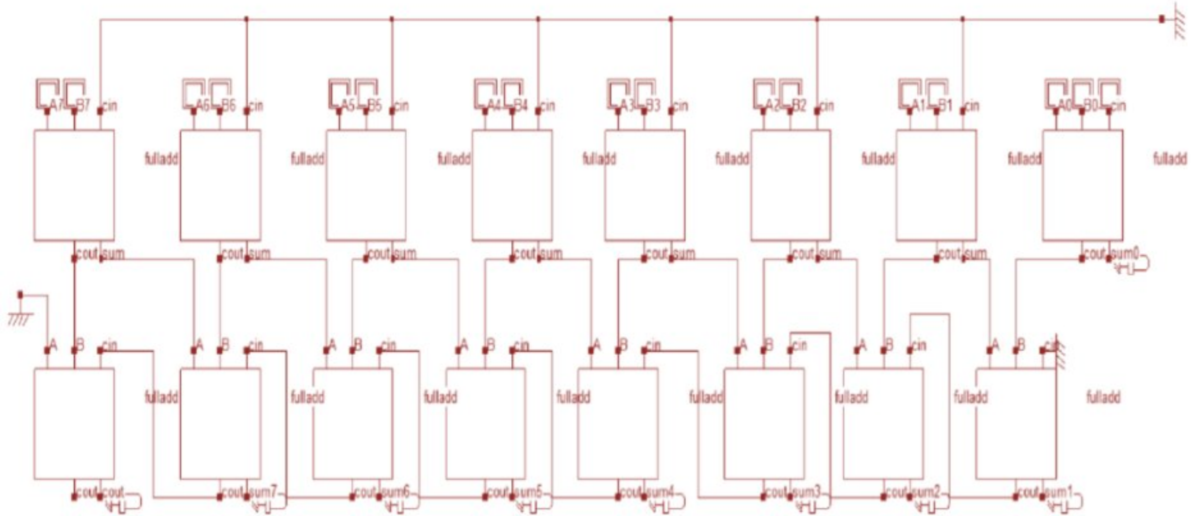

Fig.2.8: Carry Save Adder (CSA)

X		1	0	0	1	1
Y		1	1	0	0	1
S		0	1	0	1	0
C	1	0	0	0	1	
SUM	1	0	1	1	0	0

Table 2.6: Carry Save Adder Computation Flow

2.3 MULTIPLIERS

Multiplication is one of the simple functions which are used in digital signal processing applications (DSP). Multipliers requie more hardware resources and processing time compared to that of adders. In order to achieve the high speed and low power demand, the various multipliers has to design to meet requirements of current VLSI industry requirments. Multipliers are not only used in processor, but also used in other part of processor designs such as various data path

units. In general, two numbers such as multiplier and multiplicand are multiplied and generate a product value. All multipliers architectures are built with basic blocks such as Half Adders (HA), Full Adders (FA), and various complex adder architectures. In recent years, many researchers developed several multipliers for the current needs of VLSI industry. Here, a brief description of some traditional multipliers such as Array Multiplier (AM), Ripple Carry Array Multiplier using Row Bypass Technique (RCAM RB), Wallace Tree Multiplier (WTM), Dadda Multiplier (DM) and are discussed. Table 2.7 describes Performance Comparison of Various Multipliers [9] for 8 bit multiplication applications

S.No.	Multiplier (8 Bit)	Area (LUT's)	Delay(ns)
1.	Array Multiplier (AM)	79	8.369
2.	Ripple Carry Array Multiplier with Row Bypassing (RCAM RB)	74	6.417
3.	Wallace Tree Multiplier (WTM)	80	6.285
4.	Dadda Multiplier (DM)	86	3.862
5.	Vedic Multiplier (VM)	100	7.406
6.	Modified Radix-2 Booth Multiplier (MRBM)	108	7.627

Table 2.7 Performance Comparison of Various Multipliers

From above performance comparison table, it is observed that Dadda Multiplier (DM) has optimized performance in terms of Area and Delay.

2.3.1 ARRAY MULTIPLIERS (AM)

Array multiplier is one of the basic multiplier which comprises of partial products generated by AND Logic [10]. All partial products are added by the Half Adder (HA) and Full Adder (FA) [8] depending on the number of input bits. Architecture of array multiplier is shown in Fig.2.9.

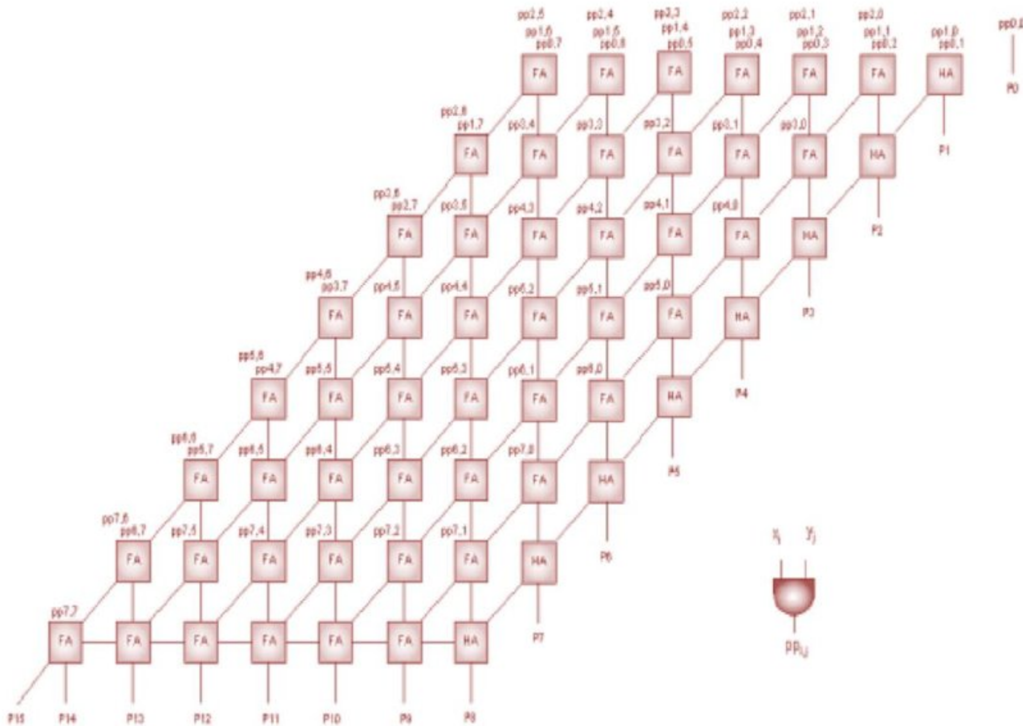

Fig.2.9.Array Multiplier

2.3.2 RIPPLE CARRY ARRAY MULTIPLIER WITH ROW BYPASSING TECHNIQUE (RCAM RB)

In ripple carry array multiplier with row bypassing technique [11], the multiplication method is similar to the array multiplier. But the partial product stages are bypassed from previous state to next state depending upon the carry value obtained in adder stage. An 8x8 Multiplier as shown in Fig.2.11 requires two 8x4 RCM multipliers and the architecture is shown in Fig.2.10.

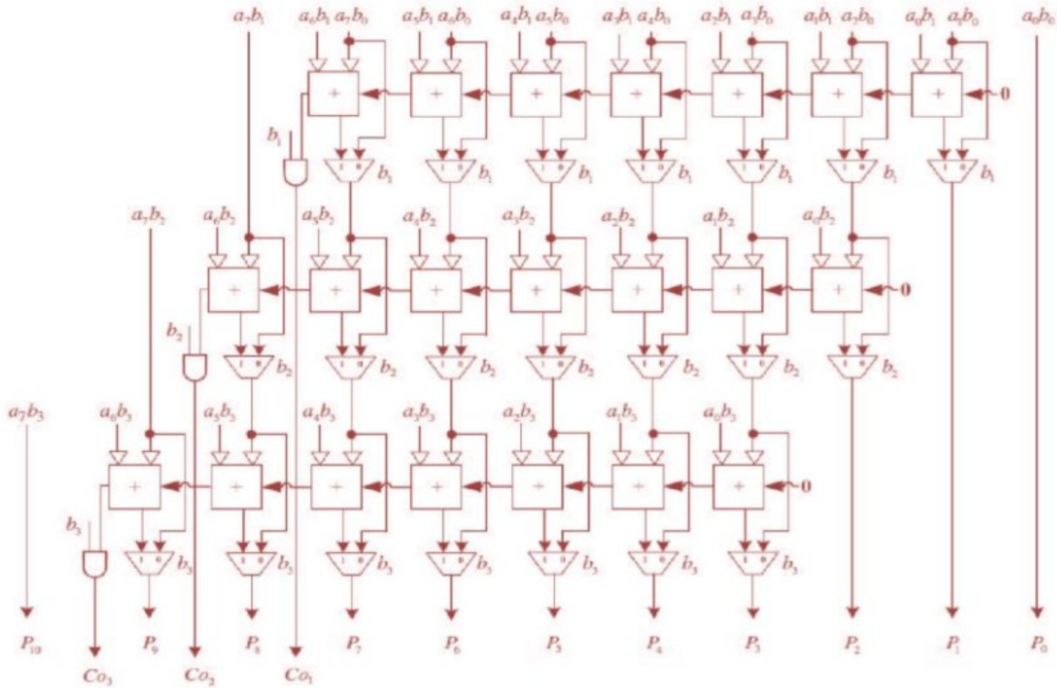

Fig.2.10.Structure of 8x4 Ripple Carry Array Multiplier with Row Bypassing

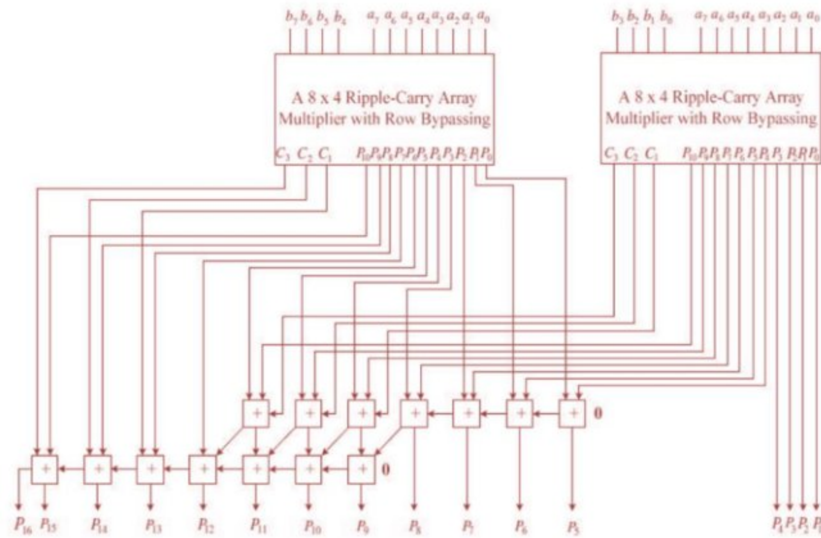

Fig.2.11.Structure of 8 bit Ripple Carry Array Multiplier (RCM) with Row By passing

2.3.3 WALLACE TREE MULTIPLIER (WTM)

In Wallace tree multiplier, the carry save adder scheme is used to add partial products generated in each stage [12]. Hence, carry generated in the present state is saved and added in the

next state. Hence the delay due to carry will be reduced in a greater extent. The design of Wallace Tree Multiplier [13] is shown in Fig 2.12.

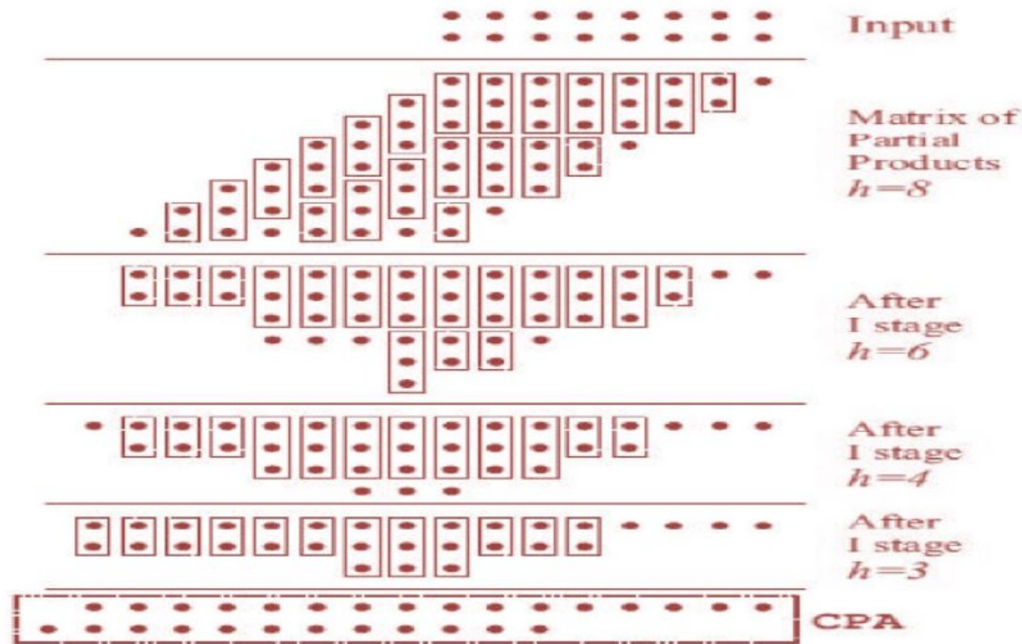

Fig.2.12.Wallace Tree Multiplier (WTM)

2.3.4 DADDA MULTIPLIER (DM)

Dadda multipliers are the refinement of parallel multipliers first presented by Wallace in 1964. In contrast to the Wallace reduction Dadda multiplier perform the least reduction at each stage [14]. The maximum height of each stage is determined by working back from final stage which consists of two rows of partial products. The height of each stage should be in the order 2, 3, 4, 6, 9, 13, 19, 28, 42, 63 etc. An 8 bit Dadda multiplier reduction is shown in Fig 2.11. For Dadda multipliers the required number of full adders and half adders are depend on the value of N. An 8 bit Dadda multiplier reduction is shown in Fig 5. For Dadda multipliers the required number of full adders and half adders are depend on the value of N.

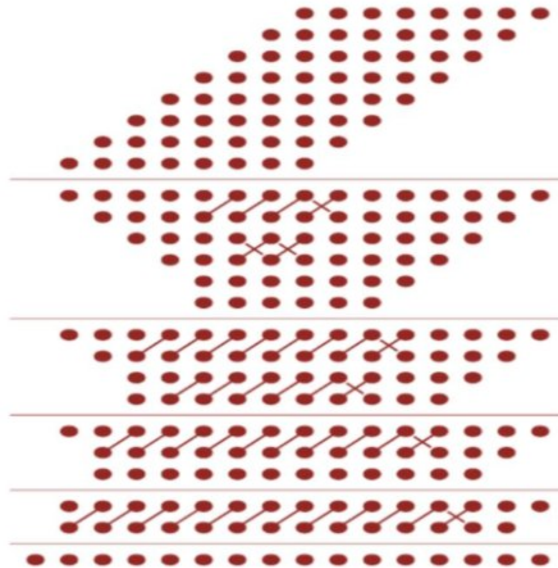

Fig.2.13.Dadda Multiplier Reduction

The principle behind Dadda Multiplier is discussed with the help of 4 bit multiplication example given below. Suppose we have to multiply tow 4 bit numbers A & B then the following algorithm is used in Dadda Multiplier. Figure 2.14 describes the algorithm used by Dadda Multiplier.

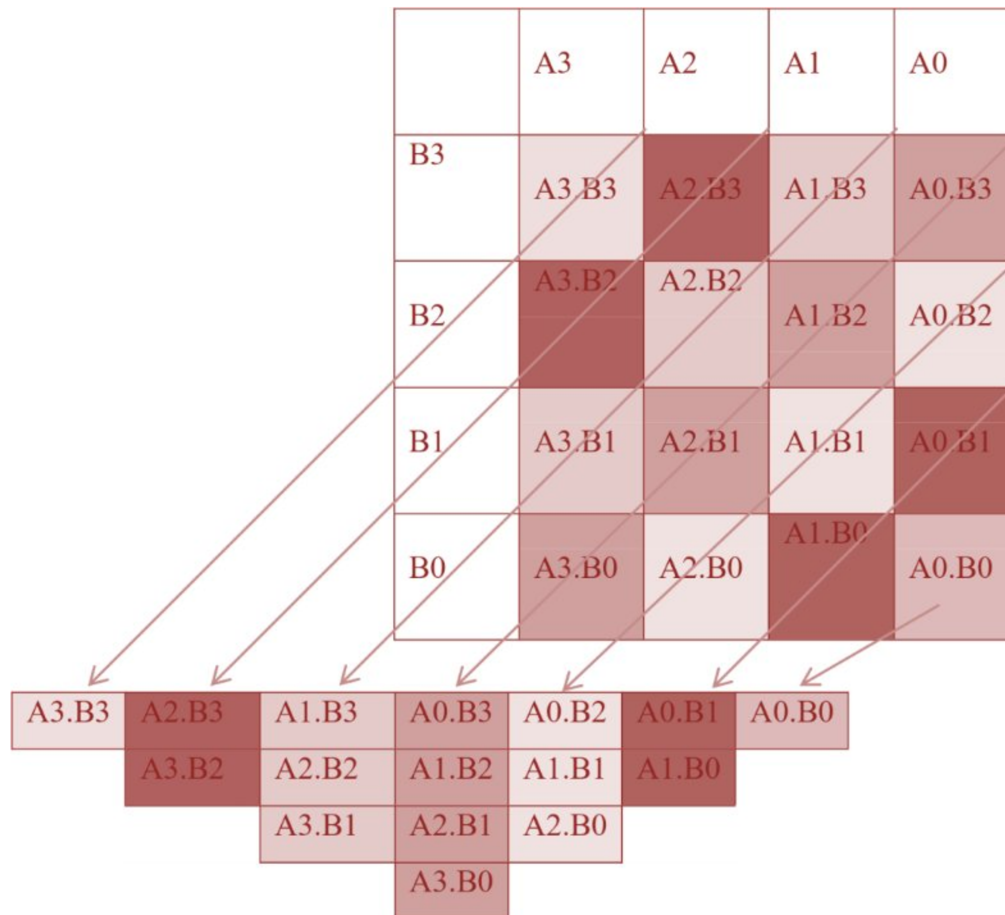

Fig.2.14.Dadda Multiplier Algorithm

2.4 CONCLUSION

After thoroughly studying various research work we have reached to following conclusion

- Use of Half adder at possible instances could minimize the area, delay & power.
- Use of logically optimized full adder instead of Traditional Full adders at all possible instances could minimize the area, delay & power as well [5].
- Dadda Multiplier is best suitable for 8-bit Convolution unit compared to other general purpose multiplier Architectures [4].

CHAPTER 3

PROPOSED MODEL FOR CONVOLUTION

3.1 PROPOSED CONVOLUTION MODEL

In this we are proposed a method to reduce to reduce the convolution processing time using hardware computing and implementation of discrete linear convolution of two finite length sequences (NXN). The proposed convolution model uses a modified hierarchical design approach, which is efficient and accurate to speed-up the computation, reduce power and hardware resources.

The circuit deals with two signals having N values each. We selected N=4 in this implementations which consider the two numbers like two arrays having four locations each to store values. The basic concept of convolution is to multiply and add. Now for two signals of four values each, we have to multiply and then add the values. Block diagram of convolution process is shown below.

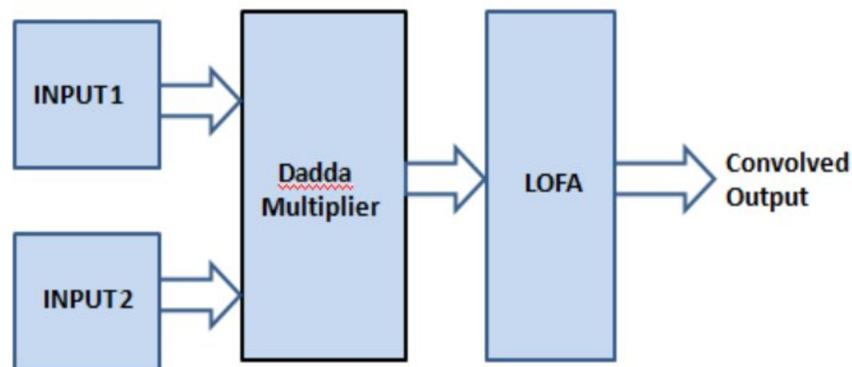

Fig 3.1: Block diagram for the proposed convolution model

In this we did the change in our multiplier which we are using Dadda Multiplier (DM) and for addition we are using Logically Optimized Full Adder (LOFA). We implement this on hardware and apply different inputs x[n] and h[n]. Here we apply 16 different input and got the result on FPGA LCD Display. We are showing one of those results for input x[n] = h[n] = [1 1 1 1] in result section.

Proposed Convolution in which for multiplication, we used Dadda Multiplier (DM) and Logically Optimized Full Adder (LOFA) is given below.

	A0	A1	A2	A3
B0	A0xB0	A1xB0	A2xB0	A3xB0
B1	A0xB1	A1xB1	A2xB1	A3xB1
B2	A0xB2	A1xB2	A2xB2	A3xB2
B3	A0xB3	A1xB3	A2xB3	A3xB3

Y0= A0xB0

Y1=A0xB1+A1xB0

Y2=A0xB2+A1xB1+A2xB0

Y3=A0xB3+A2xB1+A2xB1+A3xB0

Y4=A1xB3+A2xB2+A3xB1

Y5=A2xB3+A3xB2

Y6=A3xB3

CHAPTER 4

RESULTS

In simulation results, technolgy view[15] is a schematic representation of a design in terms of logic elements optimized to the target xilinx device or technology in terms of Look Up Tables (LUTs)[16], carry logic, I/O buffers and other technology specific components. A Look Up Table is basically just a small memory. A four input and one output LUT can genrate any four input Bollena function(AND/OR/XOR/NOT/Combination of these/etc). When FPGA has to be configured, it is required to be configured the contents of the LUTs, and thus the will be implemented.

RTL view is te schematic reperesentation of te pre-optimized design in terms of the genraic symbols that are independent of te targeted xilinx devices in terms of adders, multipliers, counters, AND gates, OR gates. Timing Wavefrom[15] is genrated by writing test bench program in verilog design which contents the possible number of inputs test vectors applied to the design.

4.1TABULAR OUTPUTS FOR 16 DIFFERENT INPUT COMBINATIONS USED

Case 1 – when input x[n] = h[n] = [0 0 0 0]

Case(1)	x[0]=0	x[1]=0	x[2]=0	x[3]=0
h[0]=0	0	0	0	0
h[1]=0	0	0	0	0
h[2]=0	0	0	0	0
h[3]=0	0	0	0	0

Y[0]=x[0]*h[0]	0
Y[1]=x[0]*h[1]+x[1]*h[0]	0
Y[2]=x[0]*h[2]+x[1]*h[1]+x[2]*h[0]	0
Y[3]=x[0]*h[3]+x[1]*h[2]+x[2]*h[1]+x[3]*h[0]	0
Y[4]=x[1]*h[3]+x[2]*h[2]+x[3]*h[1]	0
Y[5]=x[2]*h[3]+x[3]*h[2]	0
Y[6]=x[4]*h[4]	0

Case 2 – when input x[n] = h[n] = [0 0 0 1]

Case(2)	x[0]=0	x[1]=0	x[2]=0	x[3]=1
h[0]=0	0	0	0	0
h[1]=0	0	0	0	0
h[2]=0	0	0	0	0
h[3]=1	0	0	0	1

Y[0]=x[0]*h[0]	0
Y[1]=x[0]*h[1]+x[1]*h[0]	0
Y[2]=x[0]*h[2]+x[1]*h[1]+x[2]*h[0]	0
Y[3]=x[0]*h[3]+x[1]*h[2]+x[2]*h[1]+x[3]*h[0]	0
Y[4]=x[1]*h[3]+x[2]*h[2]+x[3]*h[1]	0
Y[5]=x[2]*h[3]+x[3]*h[2]	0
Y[6]=x[4]*h[4]	1

Case 3 – when input x[n] = h[n] = [0 0 1 0]

Case(3)	x[0]=0	x[1]=0	x[2]=1	x[3]=0
h[0]=0	0	0	0	0
h[1]=0	0	0	0	0
h[2]=1	0	0	1	0
h[3]=0	0	0	0	0

Y[0]=x[0]*h[0]	0
Y[1]=x[0]*h[1]+x[1]*h[0]	0
Y[2]=x[0]*h[2]+x[1]*h[1]+x[2]*h[0]	0
Y[3]=x[0]*h[3]+x[1]*h[2]+x[2]*h[1]+x[3]*h[0]	0
Y[4]=x[1]*h[3]+x[2]*h[2]+x[3]*h[1]	1
Y[5]=x[2]*h[3]+x[3]*h[2]	0
Y[6]=x[4]*h[4]	0

Case 4 – when input x[n] = h[n] = [0 0 1 1]

Case(3)	x[0]=0	x[1]=0	x[2]=1	x[3]=1
h[0]=0	0	0	0	0
h[1]=0	0	1	0	0
h[2]=1	0	0	1	1
h[3]=1	0	0	1	1

Y[0]=x[0]*h[0]	0
Y[1]=x[0]*h[1]+x[1]*h[0]	0
Y[2]=x[0]*h[2]+x[1]*h[1]+x[2]*h[0]	0
Y[3]=x[0]*h[3]+x[1]*h[2]+x[2]*h[1]+x[3]*h[0]	0
Y[4]=x[1]*h[3]+x[2]*h[2]+x[3]*h[1]	1
Y[5]=x[2]*h[3]+x[3]*h[2]	2
Y[6]=x[4]*h[4]	1

Case 5 – when input x[n] = h[n] = [0 1 0 0]

Case(1)	x[0]=0	x[1]=1	x[2]=0	x[3]=0
h[0]=0	0	0	0	0
h[1]=1	0	1	0	0
h[2]=0	0	0	0	0
h[3]=0	0	0	0	0

Y[0]=x[0]*h[0]	0
Y[1]=x[0]*h[1]+x[1]*h[0]	0
Y[2]=x[0]*h[2]+x[1]*h[1]+x[2]*h[0]	1
Y[3]=x[0]*h[3]+x[1]*h[2]+x[2]*h[1]+x[3]*h[0]	0
Y[4]=x[1]*h[3]+x[2]*h[2]+x[3]*h[1]	0
Y[5]=x[2]*h[3]+x[3]*h[2]	0
Y[6]=x[4]*h[4]	0

Case 6 – when input x[n] = h[n] = [0 1 0 1]

Case(2)	x[0]=0	x[1]=1	x[2]=0	x[3]=1
h[0]=0	0	0	0	0
h[1]=1	0	1	0	1
h[2]=0	0	0	0	0
h[3]=1	0	1	0	1

Y[0]=x[0]*h[0]	0
Y[1]=x[0]*h[1]+x[1]*h[0]	0
Y[2]=x[0]*h[2]+x[1]*h[1]+x[2]*h[0]	1
Y[3]=x[0]*h[3]+x[1]*h[2]+x[2]*h[1]+x[3]*h[0]	0
Y[4]=x[1]*h[3]+x[2]*h[2]+x[3]*h[1]	2
Y[5]=x[2]*h[3]+x[3]*h[2]	0
Y[6]=x[4]*h[4]	1

Case 7 – when input x[n] = h[n] = [0 1 1 0]

Case(3)	x[0]=0	x[1]=1	x[2]=1	x[3]=0
h[0]=0	0	0	0	0
h[1]=1	0	1	1	0
h[2]=1	0	1	1	0
h[3]=0	0	0	0	0

Y[0]=x[0]*h[0]	0
Y[1]=x[0]*h[1]+x[1]*h[0]	0
Y[2]=x[0]*h[2]+x[1]*h[1]+x[2]*h[0]	1
Y[3]=x[0]*h[3]+x[1]*h[2]+x[2]*h[1]+x[3]*h[0]	2
Y[4]=x[1]*h[3]+x[2]*h[2]+x[3]*h[1]	1
Y[5]=x[2]*h[3]+x[3]*h[2]	0
Y[6]=x[4]*h[4]	0

Case 8 – when input x[n] = h[n] = [0 1 1 1]

Case(3)	x[0]=0	x[1]=1	x[2]=1	x[3]=1
h[0]=0	0	0	0	0
h[1]=1	0	1	1	1
h[2]=1	0	1	1	1
h[3]=1	0	1	1	1

Y[0]=x[0]*h[0]	0
Y[1]=x[0]*h[1]+x[1]*h[0]	0
Y[2]=x[0]*h[2]+x[1]*h[1]+x[2]*h[0]	1
Y[3]=x[0]*h[3]+x[1]*h[2]+x[2]*h[1]+x[3]*h[0]	2
Y[4]=x[1]*h[3]+x[2]*h[2]+x[3]*h[1]	3
Y[5]=x[2]*h[3]+x[3]*h[2]	2
Y[6]=x[4]*h[4]	1

Case 9 – when input x[n] = h[n] = [1 0 0 0]

Case(2)	x[0]=1	x[1]=0	x[2]=0	x[3]=0
h[0]=1	1	0	0	0
h[1]=0	0	0	0	0
h[2]=0	0	0	0	0
h[3]=0	0	0	0	0

Y[0]=x[0]*h[0]	1
Y[1]=x[0]*h[1]+x[1]*h[0]	0
Y[2]=x[0]*h[2]+x[1]*h[1]+x[2]*h[0]	0
Y[3]=x[0]*h[3]+x[1]*h[2]+x[2]*h[1]+x[3]*h[0]	0
Y[4]=x[1]*h[3]+x[2]*h[2]+x[3]*h[1]	0
Y[5]=x[2]*h[3]+x[3]*h[2]	0
Y[6]=x[4]*h[4]	0

Case 10 – when input x[n] = h[n] = [1 0 0 1]

Case(2)	x[0]=1	x[1]=0	x[2]=0	x[3]=1
h[0]=1	1	0	0	1
h[1]=0	0	0	0	0
h[2]=0	0	0	0	0
h[3]=1	1	0	0	1

Y[0]=x[0]*h[0]	1
Y[1]=x[0]*h[1]+x[1]*h[0]	0
Y[2]=x[0]*h[2]+x[1]*h[1]+x[2]*h[0]	0
Y[3]=x[0]*h[3]+x[1]*h[2]+x[2]*h[1]+x[3]*h[0]	1
Y[4]=x[1]*h[3]+x[2]*h[2]+x[3]*h[1]	0
Y[5]=x[2]*h[3]+x[3]*h[2]	0
Y[6]=x[4]*h[4]	1

Case 11 – when input x[n] = h[n] = [1 0 1 0]

Case(3)	x[0]=1	x[1]=0	x[2]=1	x[3]=0
h[0]=1	1	0	1	0
h[1]=0	0	0	0	0
h[2]=1	1	0	1	0
h[3]=0	0	0	0	0

Y[0]=x[0]*h[0]	0
Y[1]=x[0]*h[1]+x[1]*h[0]	0
Y[2]=x[0]*h[2]+x[1]*h[1]+x[2]*h[0]	0
Y[3]=x[0]*h[3]+x[1]*h[2]+x[2]*h[1]+x[3]*h[0]	0
Y[4]=x[1]*h[3]+x[2]*h[2]+x[3]*h[1]	1
Y[5]=x[2]*h[3]+x[3]*h[2]	0
Y[6]=x[4]*h[4]	0

Case 12 – when input x[n] = h[n] = [1 0 1 1]

Case(3)	x[0]=1	x[1]=0	x[2]=1	x[3]=1
h[0]=1	1	0	1	1
h[1]=0	0	0	0	0
h[2]=1	1	0	1	1
h[3]=1	1	0	1	1

Y[0]=x[0]*h[0]	1
Y[1]=x[0]*h[1]+x[1]*h[0]	0
Y[2]=x[0]*h[2]+x[1]*h[1]+x[2]*h[0]	2
Y[3]=x[0]*h[3]+x[1]*h[2]+x[2]*h[1]+x[3]*h[0]	2
Y[4]=x[1]*h[3]+x[2]*h[2]+x[3]*h[1]	1
Y[5]=x[2]*h[3]+x[3]*h[2]	2
Y[6]=x[4]*h[4]	1

Case 13 – when input x[n] = h[n] = [1 1 0 0]

Case(1)	x[0]=1	x[1]=1	x[2]=0	x[3]=0
h[0]=1	1	1	0	0
h[1]=1	1	1	0	0
h[2]=0	0	0	0	0
h[3]=0	0	0	0	0

Y[0]=x[0]*h[0]	1
Y[1]=x[0]*h[1]+x[1]*h[0]	2
Y[2]=x[0]*h[2]+x[1]*h[1]+x[2]*h[0]	1
Y[3]=x[0]*h[3]+x[1]*h[2]+x[2]*h[1]+x[3]*h[0]	0
Y[4]=x[1]*h[3]+x[2]*h[2]+x[3]*h[1]	0
Y[5]=x[2]*h[3]+x[3]*h[2]	0
Y[6]=x[4]*h[4]	0

Case 14 – when input x[n] = h[n] = [1 1 0 1]

Case(2)	x[0]=1	x[1]=1	x[2]=0	x[3]=1
h[0]=1	1	1	0	1
h[1]=1	1	1	0	1
h[2]=0	0	0	0	0
h[3]=1	1	1	0	1

Y[0]=x[0]*h[0]	1
Y[1]=x[0]*h[1]+x[1]*h[0]	2
Y[2]=x[0]*h[2]+x[1]*h[1]+x[2]*h[0]	1
Y[3]=x[0]*h[3]+x[1]*h[2]+x[2]*h[1]+x[3]*h[0]	2
Y[4]=x[1]*h[3]+x[2]*h[2]+x[3]*h[1]	2
Y[5]=x[2]*h[3]+x[3]*h[2]	0
Y[6]=x[4]*h[4]	1

Case 15 – when input x[n] = h[n] = [1 1 1 0]

Case(3)	x[0]=1	x[1]=1	x[2]=1	x[3]=0
h[0]=1	1	1	1	0
h[1]=1	1	1	1	0
h[2]=1	1	1	1	0
h[3]=0	0	0	0	0

Y[0]=x[0]*h[0]	1
Y[1]=x[0]*h[1]+x[1]*h[0]	2
Y[2]=x[0]*h[2]+x[1]*h[1]+x[2]*h[0]	3
Y[3]=x[0]*h[3]+x[1]*h[2]+x[2]*h[1]+x[3]*h[0]	2
Y[4]=x[1]*h[3]+x[2]*h[2]+x[3]*h[1]	1
Y[5]=x[2]*h[3]+x[3]*h[2]	0
Y[6]=x[4]*h[4]	0

Case 16 – when input x[n] = h[n] = [1 1 1 1]

Case(3)	x[0]=1	x[1]=1	x[2]=1	x[3]=1
h[0]=1	1	1	1	1
h[1]=1	1	1	1	1
h[2]=1	1	1	1	1
h[3]=1	1	1	1	1

Y[0]=x[0]*h[0]	1
Y[1]=x[0]*h[1]+x[1]*h[0]	2
Y[2]=x[0]*h[2]+x[1]*h[1]+x[2]*h[0]	3
Y[3]=x[0]*h[3]+x[1]*h[2]+x[2]*h[1]+x[3]*h[0]	4
Y[4]=x[1]*h[3]+x[2]*h[2]+x[3]*h[1]	3
Y[5]=x[2]*h[3]+x[3]*h[2]	2
Y[6]=x[4]*h[4]	1

4.2 WAVEFORM OUTPUTS FOR 16 DIFFERENT INPUT COMBINATIONS USED

Case 1 – when input x[n] = h[n] = [0 0 0 0]

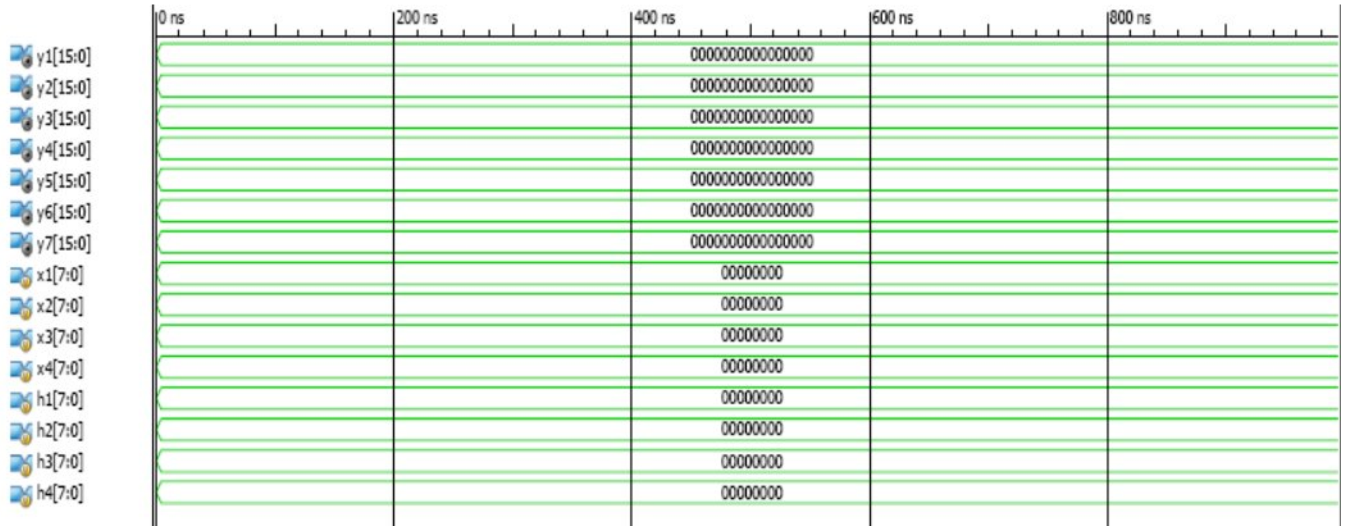

Fig. 4.1 Timing waveform for x[n] = h[n] = [0 0 0 0]

Case 2 – when input x[n] = h[n] = [0 0 0 1]

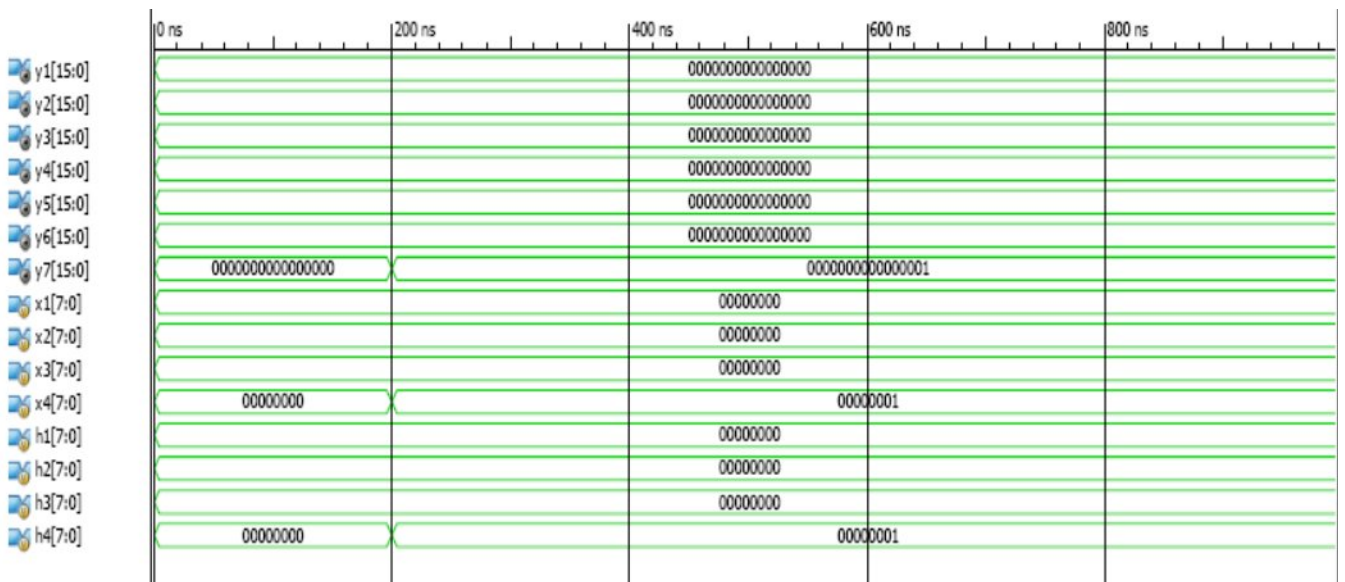

Fig. 4.2 Timing waveform for x[n] = h[n] = [0 0 0 1]

Case 3 – when input x[n] = h[n] = [0 0 1 0]

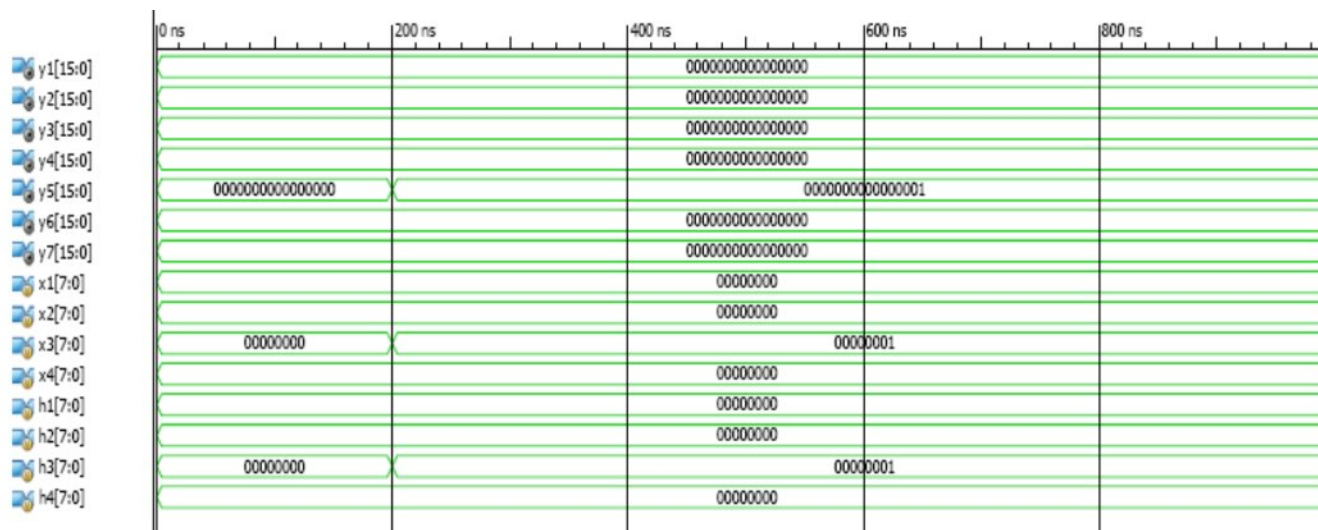

Fig. 4.3 Timing waveform for x[n] = h[n] = [0 0 1 0]

Case 4 – when input x[n] = h[n] = [0 0 1 1]

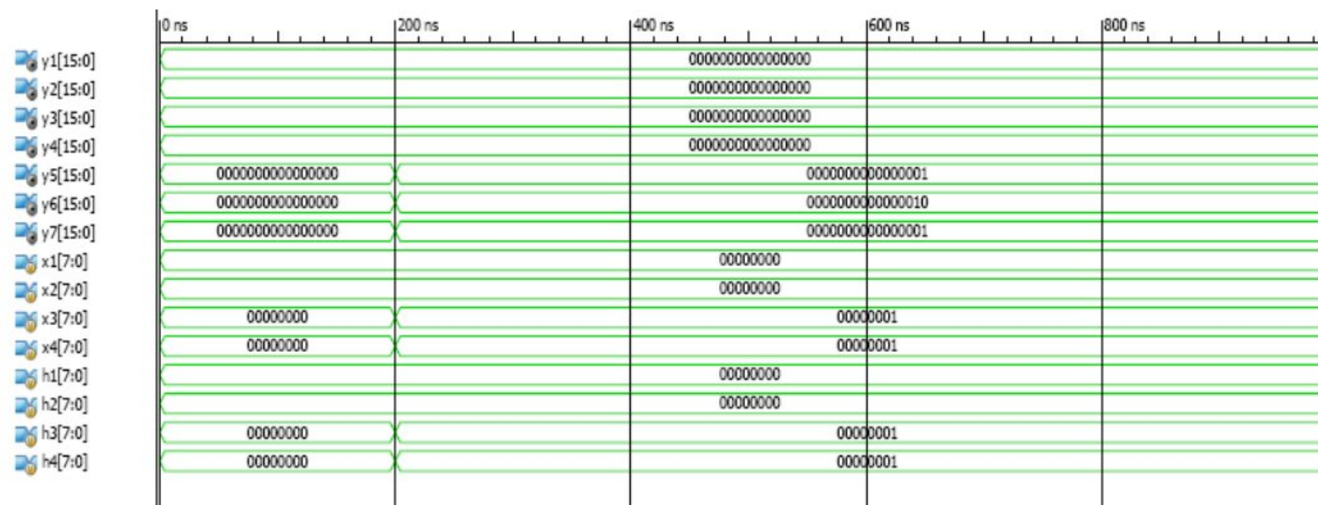

Fig. 4.4 Timing waveform for x[n] = h[n] = [0 0 1 1]

Case 5 – when input x[n] = h[n] = [0 1 0 0]

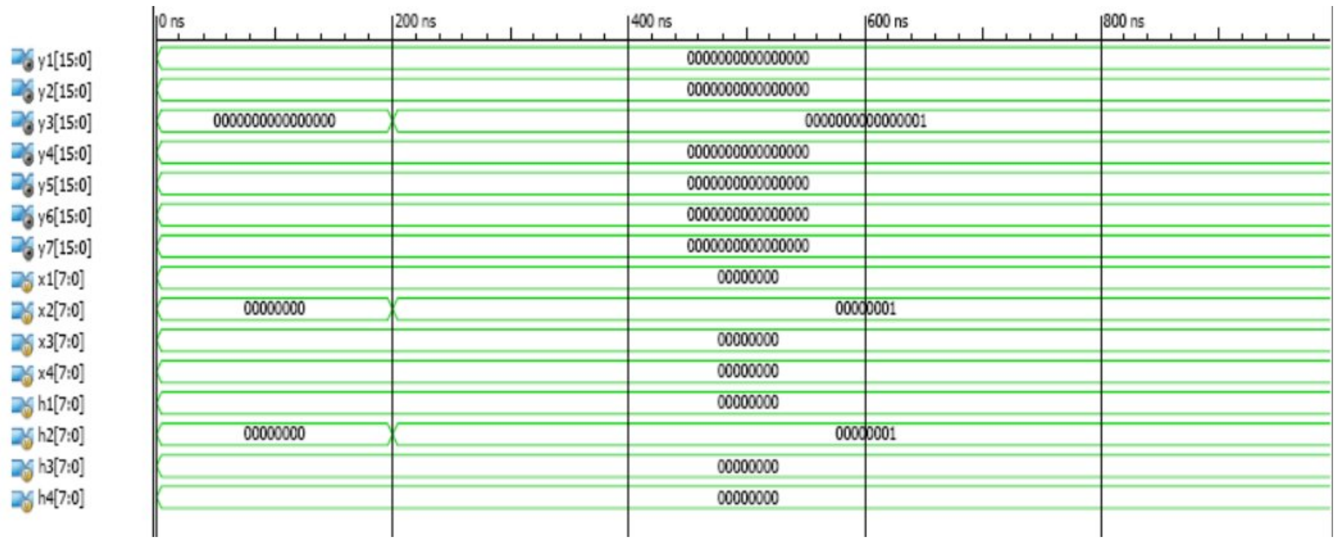

Fig. 4.5 Timing waveform for x[n] = h[n] = [0 1 0 0]

Case 6 – when input x[n] = h[n] = [0 1 0 1]

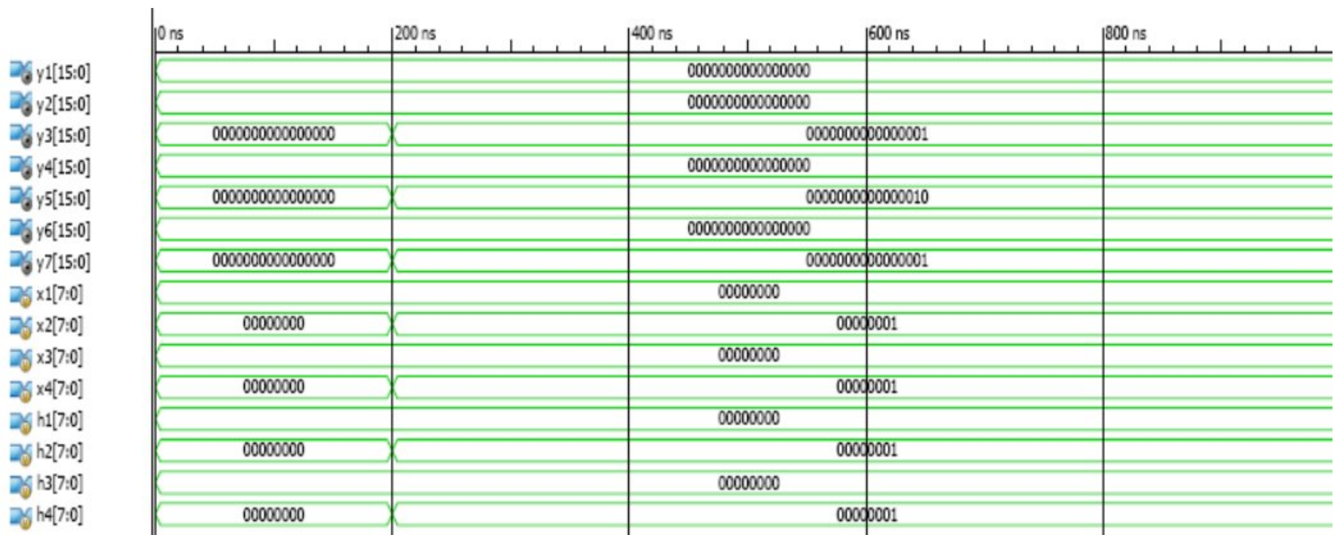

Fig. 4.6 Timing waveform for x[n] = h[n] = [0 1 0 1]

Case 7 – when input x[n] = h[n] = [0 1 1 0]

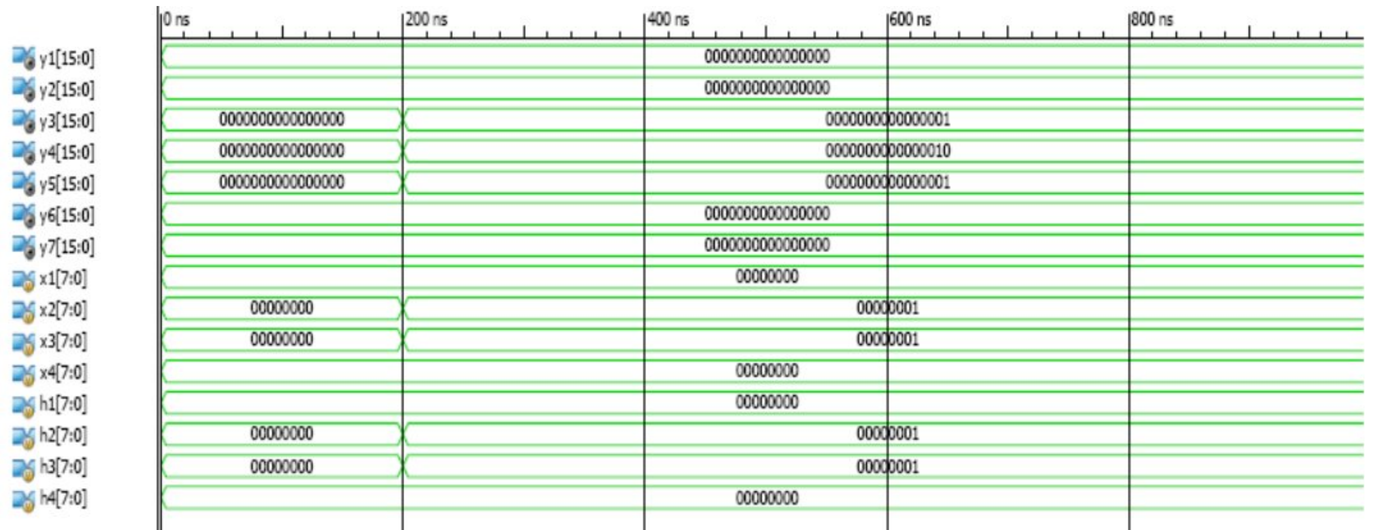

Fig. 4.7 Timing waveform for x[n] = h[n] = [0 1 1 0]

Case 8 – when input x[n] = h[n] = [0 1 1 1]

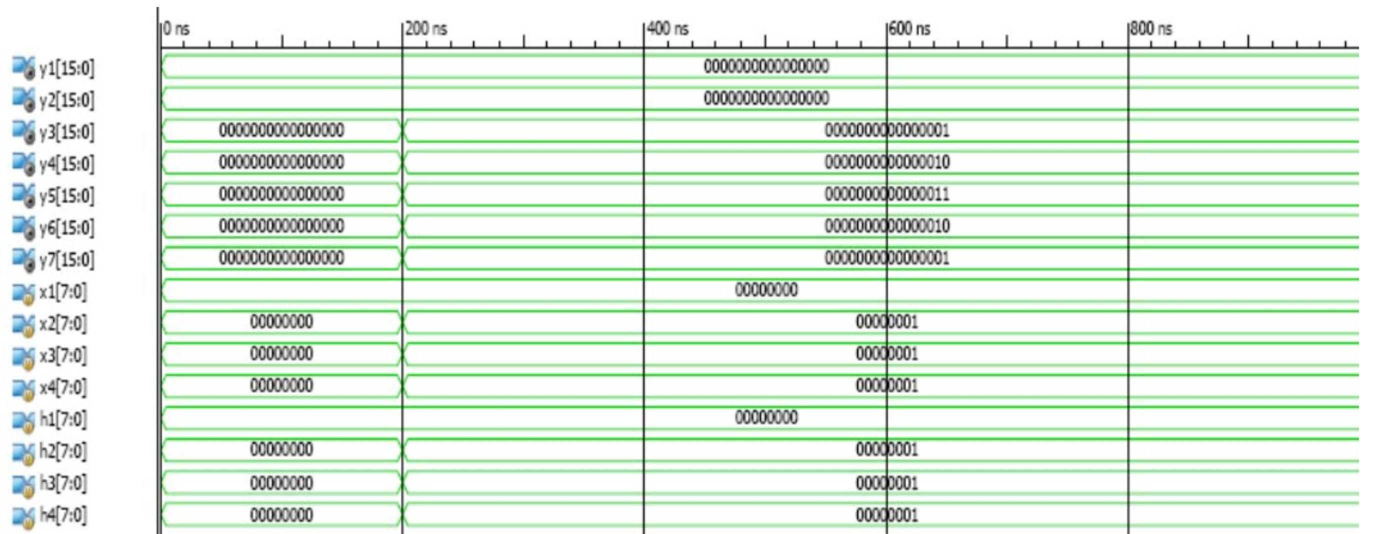

Fig. 4.8 Timing waveform for x[n] = h[n] = [0 1 1 1]

Case 9 – when input x[n] = h[n] = [1 0 0 0]

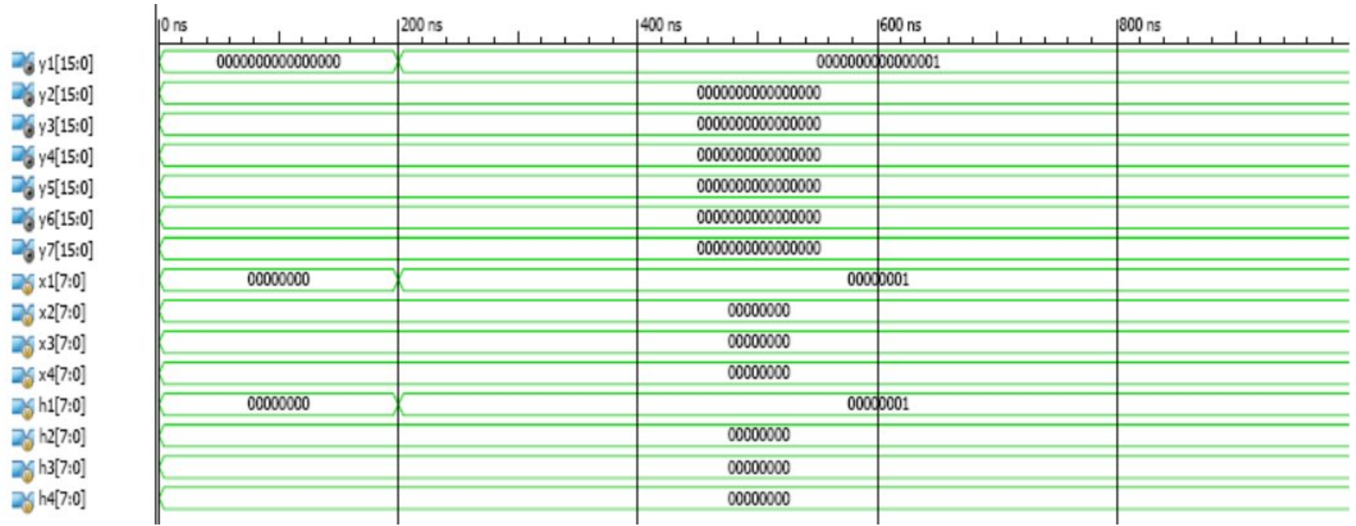

Fig. 4.9 Timing waveform for x[n] = h[n] = [1 0 0 0]

Case 10 – when input x[n] = h[n] = [1 0 0 1]

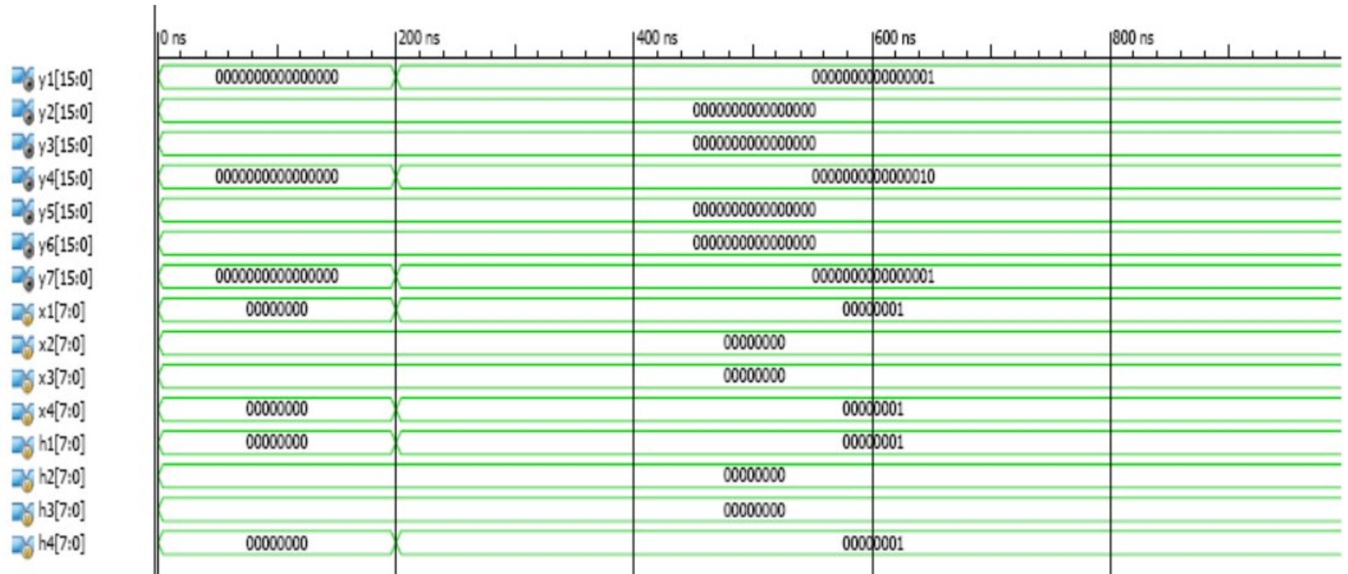

Fig. 4.10 Timing waveform for x[n] = h[n] = [1 0 0 1]

Case 11 – when input x[n] = h[n] = [1 0 1 0]

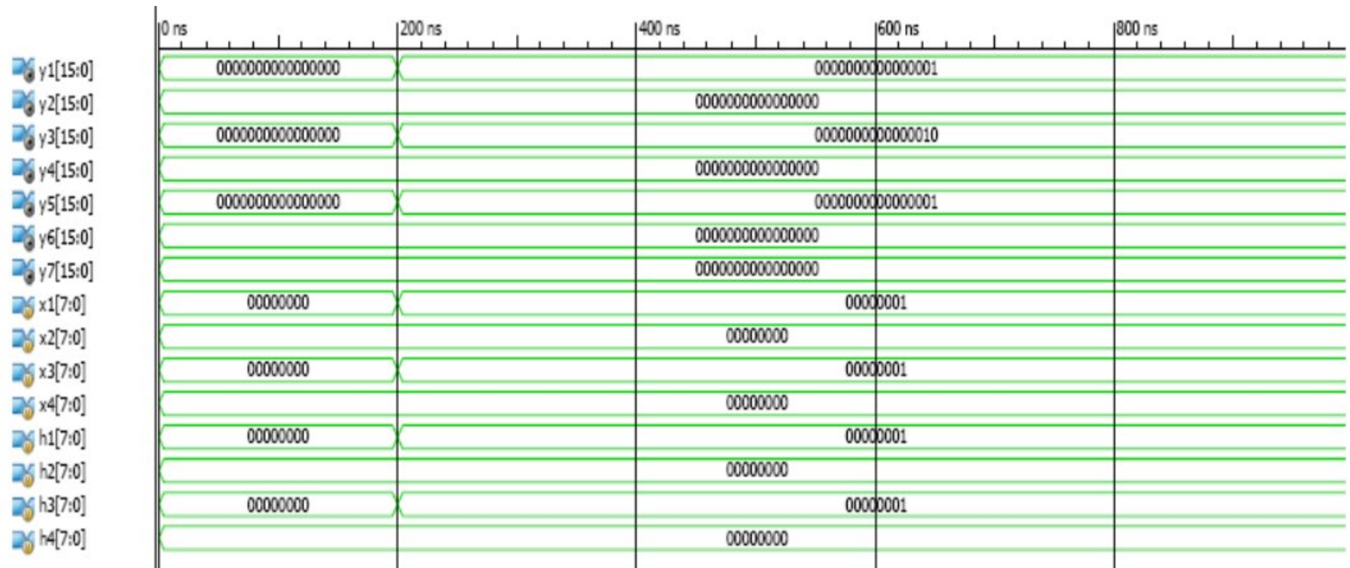

Fig. 4.11 Timing waveform for x[n] = h[n] = [1 0 1 0]

Case 12 – when input x[n] = h[n] = [1 0 1 1]

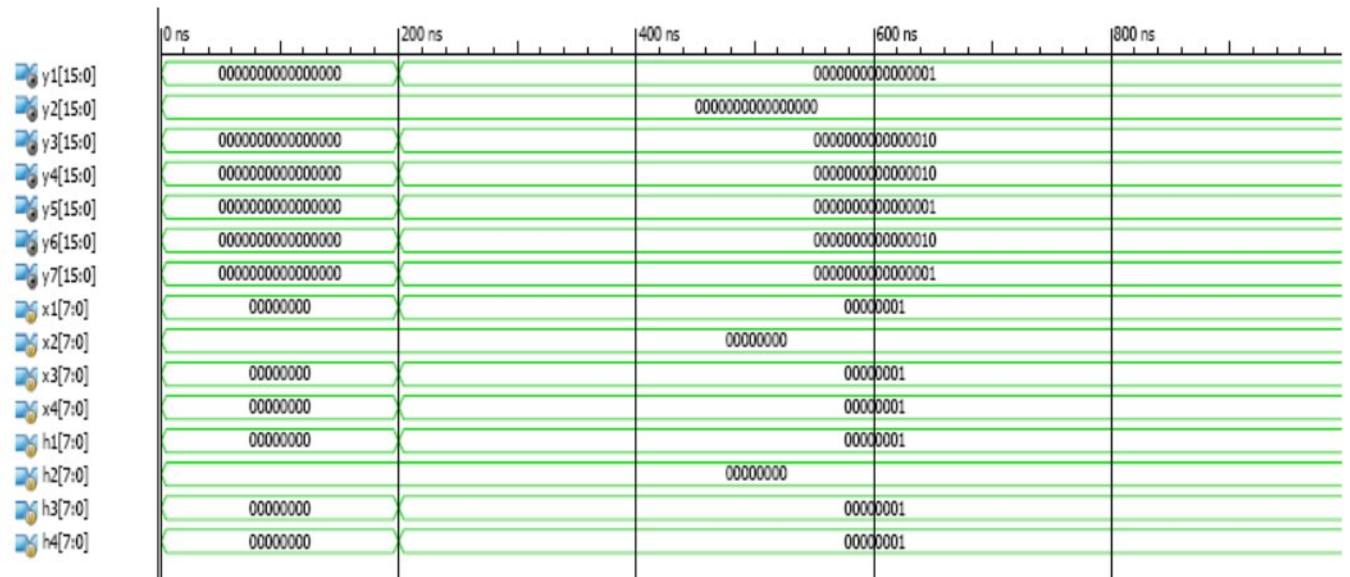

Fig. 4.12 Timing waveform for x[n] = h[n] = [1 0 1 1]

Case 13 – when input x[n] = h[n] = [1 1 0 0]

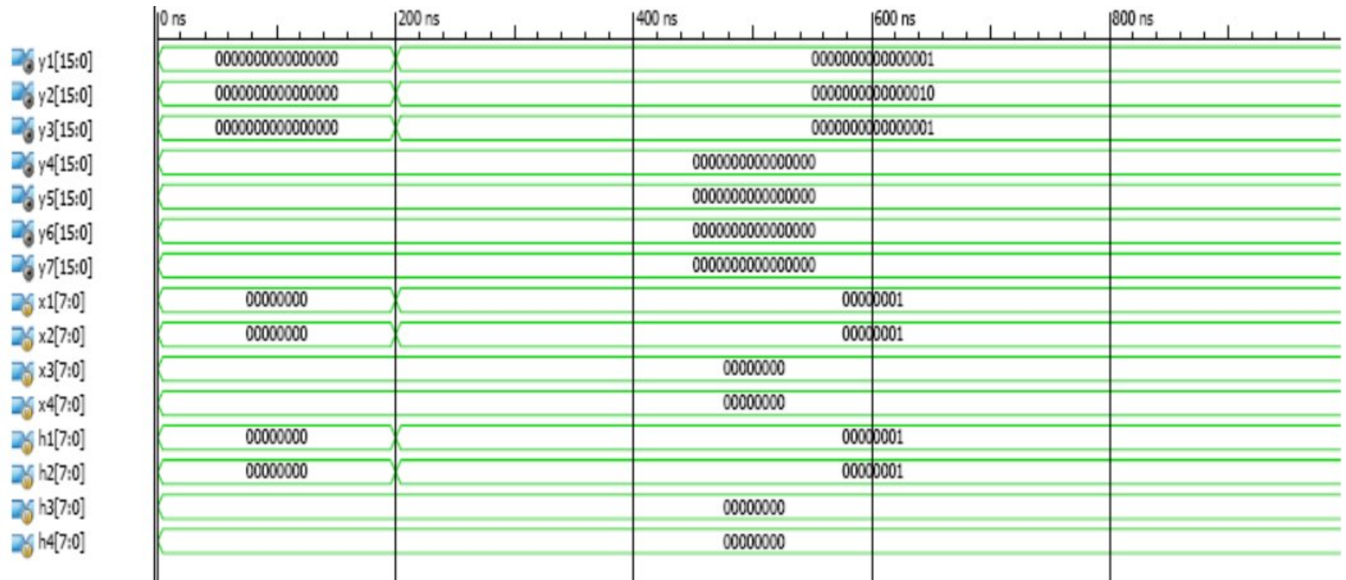

Fig. 4.13 Timing waveform for x[n] = h[n] = [1 1 0 0]

Case 14 – when input x[n] = h[n] = [1 1 0 1]

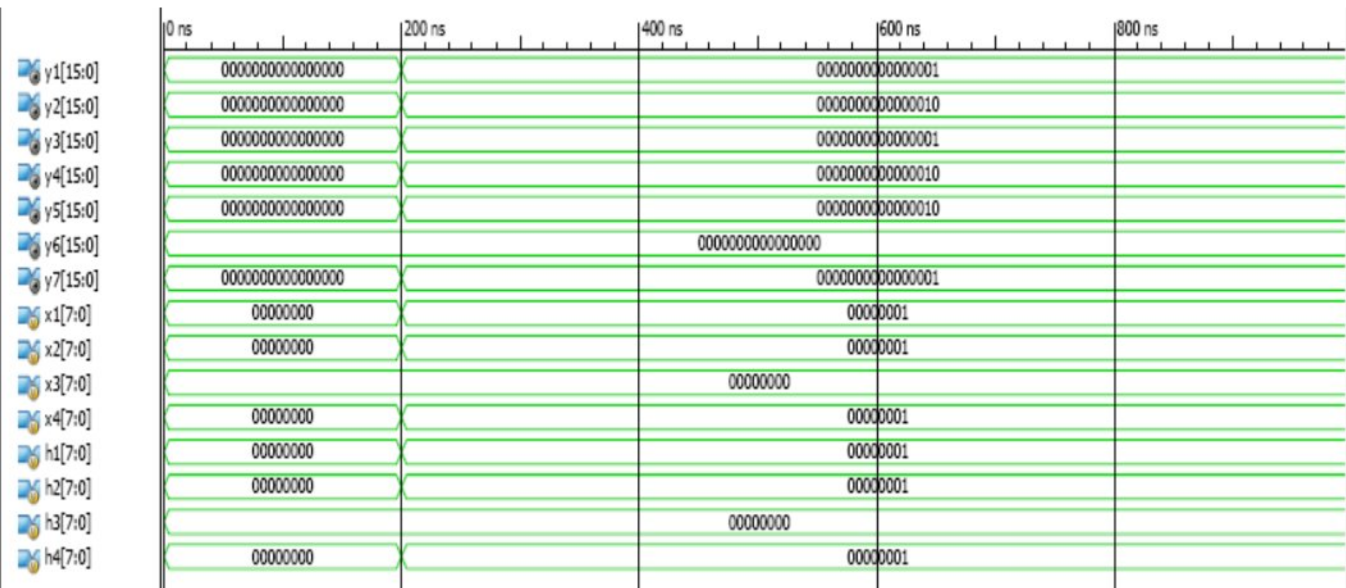

Fig. 4.14 Timing waveform for x[n] = h[n] = [1 1 0 1]

Case 15 – when input x[n] = h[n] = [1 1 1 0]

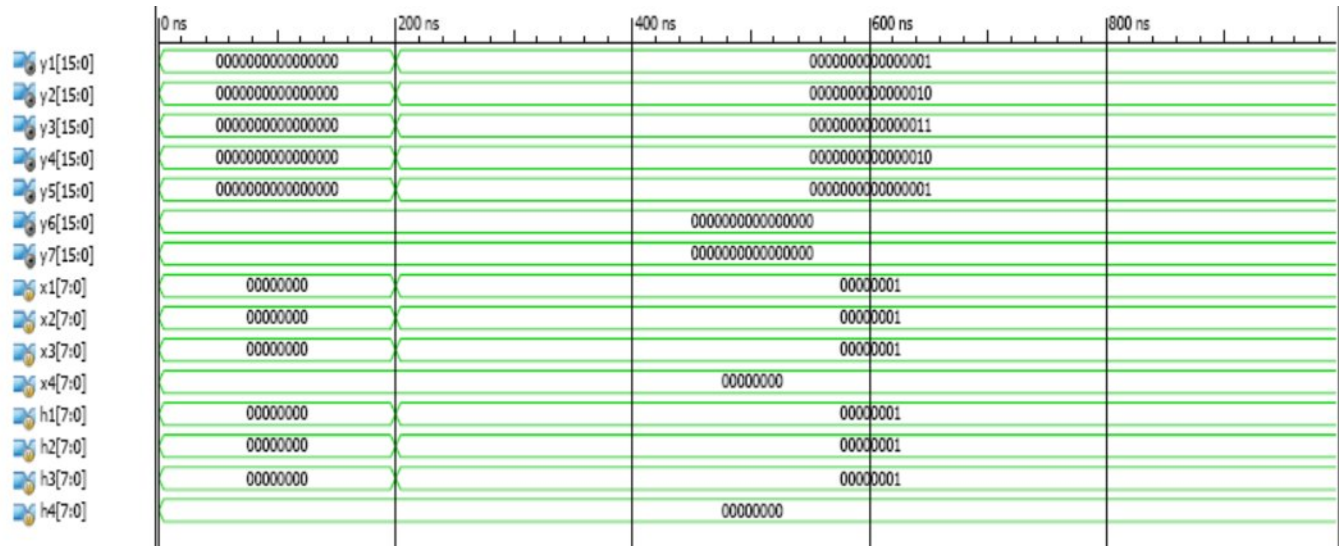

Fig. 4.15 Timing waveform for x[n] = h[n] = [1 1 1 0]

Case 16 – when input x[n] = h[n] = [1 1 1 1]

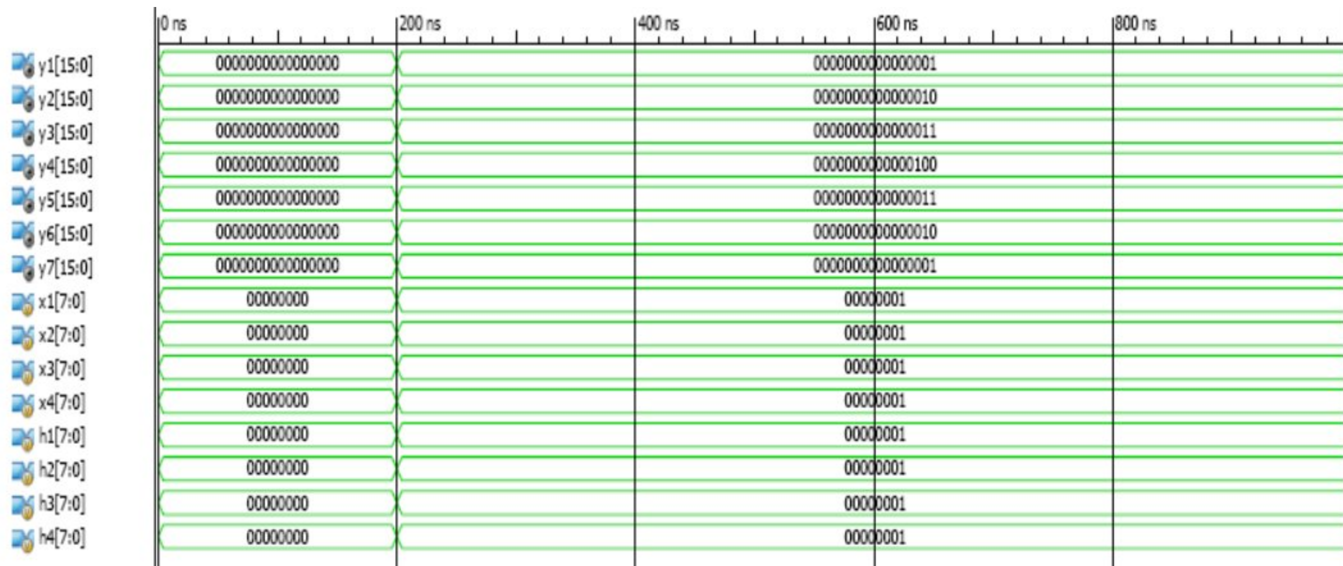

Fig. 4.16 Timing waveform for x[n] = h[n] = [1 1 1 1]

4.3 SIMULATION RESULTS FOR ADDERS

4.3.1 HALF ADDER

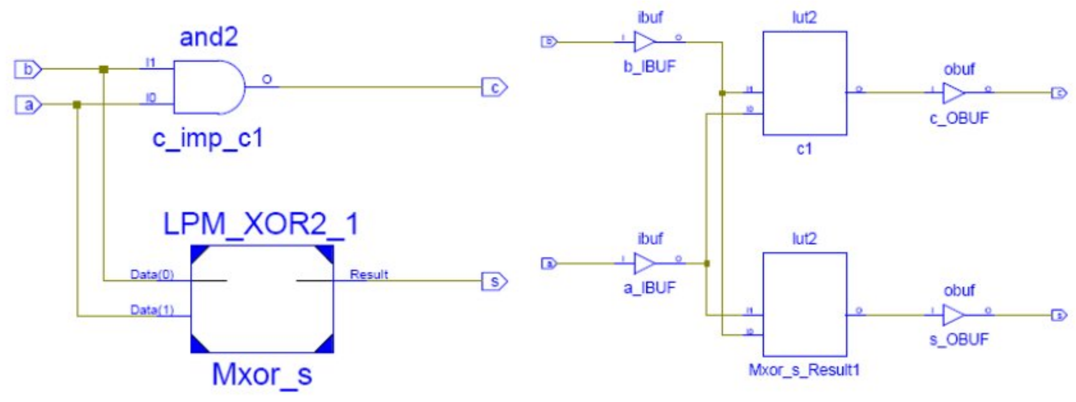

Fig. 4.17 RTL and Technology view of Half Adder

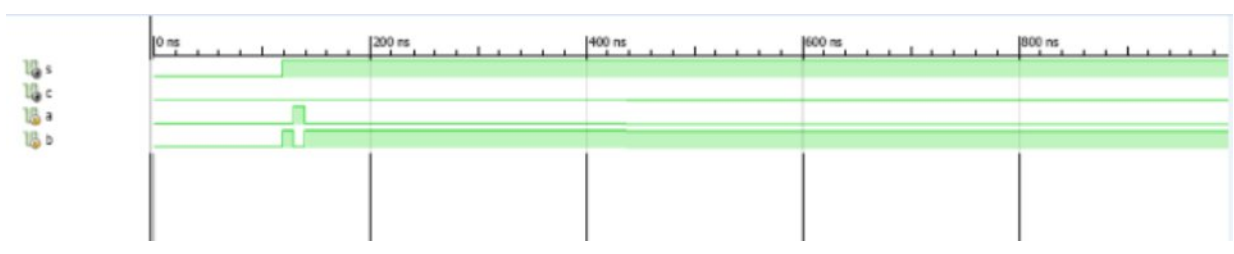

Fig. 4.18Timing waveform for Half Adder

Logic Utilization	Used
Number of Slice LUTs	1
Number of occupied Slices	1
Number of bonded IOBs	4
Average Fanout of Non-Clock Nets	4.00
Maximum combinational path delay in nanosecond	0.770
Power in milliwatts (mW)	90.6

Table 4.1 Device Utilization Summary for Half Adder

4.3.2 Full Adder

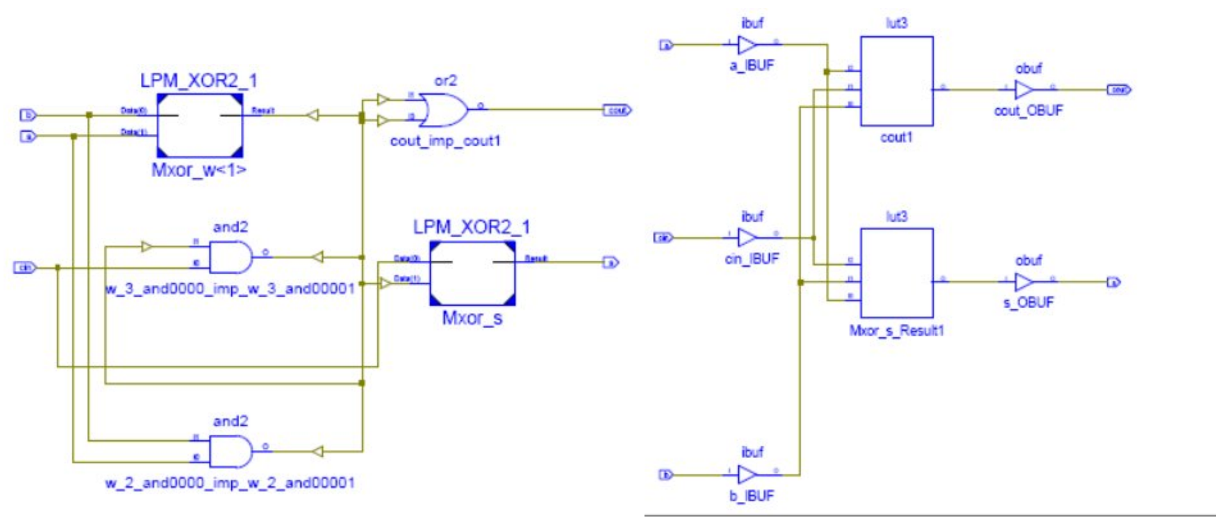

Fig. 4.19 RTL and Technology view of Full Adder

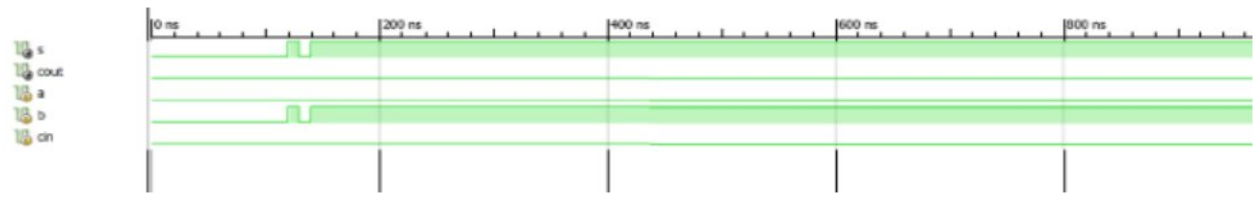

Fig. 4.20Timing waveform for Full Adder

Logic Utilization	Used
Number of Slice LUTs	1
Number of occupied Slices	1
Number of bonded IOBs	5
Average Fanout of Non-Clock Nets	3.50
Maximum combinational path delay in Nanosecond	0.923
Power in milliwatts (mW)	90.6

Table 4.2 Device Utilization Summary for Full Adder

4.3.3 Logically Optimized Full Adder

Technology view and RTL view of Logically Optimized Full Adder are given in fig

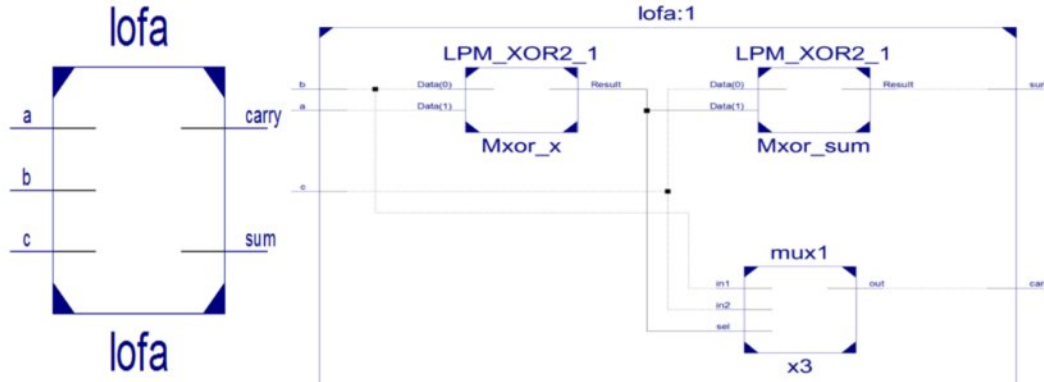

Fig. 4.21 RTL and Technology view of Logically Optimized Full Adder

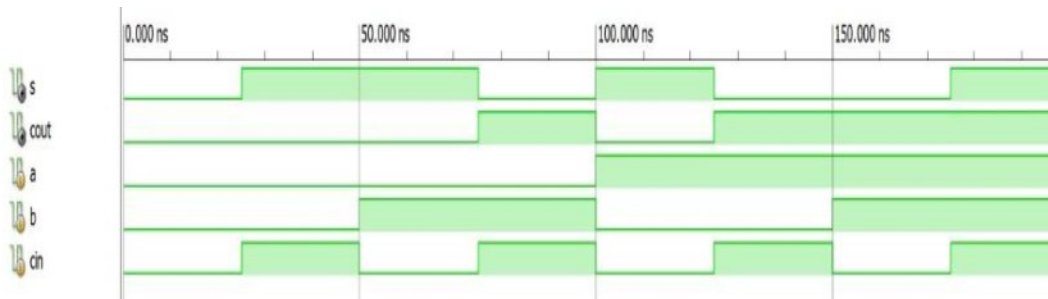

Fig. 4.22 Timing waveform of Logically Optimized Full Adder

Logic Utilization	Used
Number of Slice LUTs	1
Number of occupied Slices	1
Number of bonded IOBs	5
Average Fanout of Non-Clock Nets	3.50
Maximum combinational path delay in Nanosecond	0.776
Power in milliwatts (mW)	90.6

Table 4.3 Device Utilization Summary for Logically Optimized Full Adder

4.4 SIMULATION RESULTS FOR DADDA MULTIPLIER

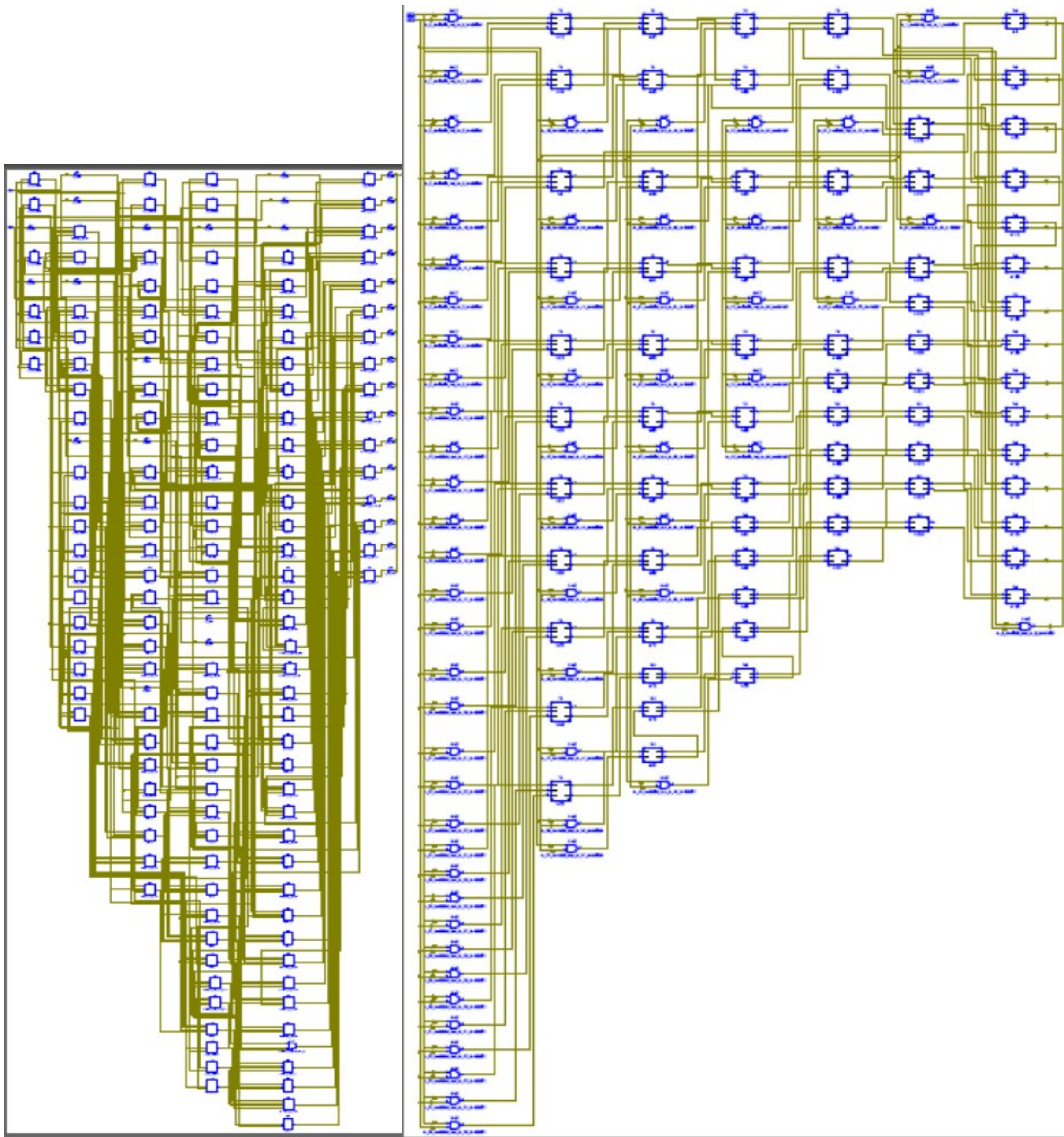

Fig. 4.23 RTL and Technology view of Dadda Multiplier

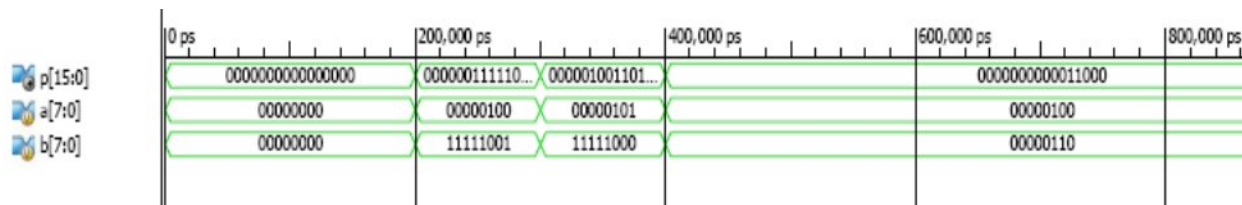

Fig. 4.24 Timing waveform for Dadda Multiplier

Logic Utilization	**Used**
Number of Slice LUTs	86
Number of occupied Slices	34
Number of bonded IOBs	32
Average Fanout of Non-Clock Nets	3.93
Maximum combinational path delay in Nanosecond	4.395
Power in milliwatts (mW)	96.8

Table 4.4 Device Utilization Summary for Dadda8 Multiplier

4.5 SIMULATION RESULTS FOR CONVOLUTION

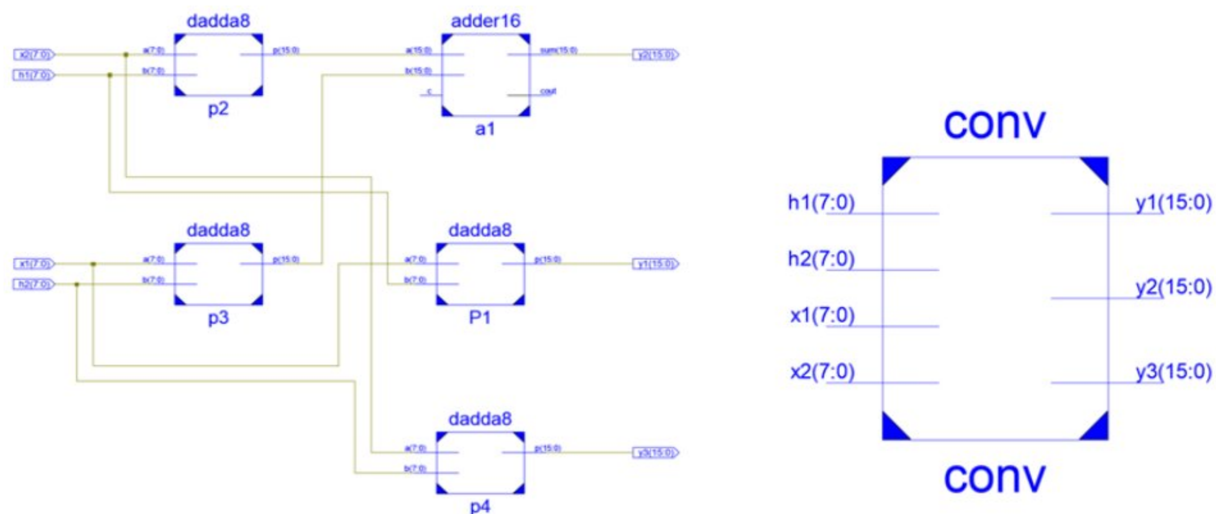

Fig. 4.25 RTL and Technology view for Convolution

Fig. 4.26 Figure showing hardware implemented output for x[n]=h[n]=[1 1 1 1] for convolution

Logic Utilization	Used
Number of Slice Flip Flops	39
Number of 4 input LUTs	47
Number of occupied Slices	44
Number of Slices containing only related logic	44
Number of Slices containing unrelated logic	0
Total Number of 4 input LUTs	73
Number used as logic	45
Number used as a route-thru	26
Number used as Shift registers	2
Number of bonded IOBs	13
Number of BUFGMUXs	1
Average Fanout of Non-Clock Nets	2.33

Table 4.5 Device Utilization Summary for Convolution

4.6 PERFORMANCE ANALYSIS OF ADDERS

The comparison of performance of different adders with respect to the performance metrics such as area, delay and power are given in table 4.6.

S. No.	Adders	Area (LUT's)	Area (Slices)	Delay (ns)	Power
1.	Half Adder	1	1	0.770	90.6
2.	Full Adder	1	1	0.923	90.6
3.	Logically Optimized Full Adder	1	1	0.776	90.6

Table 4.6 Performance Analysis of single bit Adders

From above performance analysis table, it is observed that, Logically Optimized Full Adder having better performance in terms of area (LUT's and Slices, delay and Power).

4.7 PERFORMANCE COMPARISION OF CONVOLUTION

Table 4.7 shows Advanced HDL synthesis report in which no of XOR's, Multiplexor's and Registers are shown.

#Registers	14
FlipFlop	14
#ROMs	4
4 to 6 bit ROM	4
#Counters	1
27 bit UP Counter	1

Table 4.7 Advanced HDL Synthesis report

Table 4.8 shows FPGA results of this Proposed Convolution Model.

Logic Cells	No.
LUT2	6
LUT3	8
LUT4	21
FlipFlop/Latches	39
Clock/Buffers	1
BUFGP	1
IO Buffers	12
IBUF	4
OBUF	8

Table 4.8 FPGA Results Convolution

Table 4.9 is showing the Timing Summary of this Proposed Convolution Model

Minimum Period	5.359ns(Maximum Frequency186.618MHz)
Minimum input arrival time before clock	6.199ns
Maximum output required time after clock	4.04ns

Table 4.9 Timing Summary

Simulation Comparison of different design approaches show that this circuit is faster than what is implemented in [1] and [3]. In addition the Proposed Convolution Model uses less power consumption and has a delay approximated 5.4ns from input to output.

CHAPTER 5

CONCLUSION AND FUTURE SCOPE

5.1 CONCLUSION

In this, we presented an optimized implementation of discrete linear convolution. This particular model has advantage to speed up convolution process. This hardware implementation of has the advantage of being optimized based on operation, speed, power and area. To accurately analyze our proposed system, we have coded our design using the VERILOG and simulate on FPGA using ISE, Modelsim and DC compiler for other processor usage. We implemented an example 4x4 Convolution. Similarly, this concept can be extended on an NxN case. The functionality of this convolution was tested and verified successfully on a XILINX ISE FPGA. The delay come through this is lesser than other convolution.

5.2 FUTURE SCOPE

In future work, it is required to design Convolution unit architecture with low area, delay and power in order to meet the needs of current VLSI Industry. Further, these models can be designed using ASIC technology for the specific application purpose.

By using analog to digital convertor (ADC) and Digital to Analog Convertor (DAC) we perform the Linear Convolution and it will be used in Amplitude Modulation (AM) Technique.

REFERENCES

[1] John W. Pierre, "A Novel Method for calculating the Convolution Sum of Two Finite Lentgh Sequences" , IEEE Transaction on education, VOL. 39, NO.1, 1996.

[2] Samir Palnitkar "Verilog HDL A Guide to Digital Design and Synthesis" Published by Prentice Hall, March 1996.

[3] Asmita Haveliya / International Journal of Engineering Research and Applications (IJERA) ISSN: 2248-9622 www.ijera.com Vol. 2, Issue 1,Jan-Feb 2012, pp.678-684

[4] R.Uma and P.Dhavachelvan "Logic optimization using technology independent mux based adders in FPGA" International Journal of VLSI design & Communication Systems (VLSICS) Vol.3, No.4, pp.135-149, August 2012

[5] MarojuSaiKumar, Dr. P. Samundiswary "Design and Performance Analysis of Various Adders using Verilog" IJCSMC, Vol. 2, Issue. 9, September 2013, pp.128 – 138

[6] Padma Devi, AshimaGirdher, and Balwinder Singh, "Improved Carry Select Adder with Reduced Area and Low Power Consumption", International Journal of Computer Applications, vol.3, no.4, pp.14-18, June 2010.

[7] R.Uma, VidyaVijayan, M.Mohanapriya, and Sharon Paul, "Area, Delay and Power Comparison of Adder Topologies", International Journal of VLSI Design & Communication Systems, vol.3, no.1, pp.153-168, February 2012.

[8] PrathibadeviTapashetti, A.S. Umesh, AshalathaKulshrestha, "Design and Simulation of Energy Efficient Full Adder for Systolic Array", International Journal of Soft Computing and Engineering, vol.1, no.6, pp.356-360, Jan 2012.

[9] MarojuSaiKumar, P.Samundiswary, "Design and Performance Analysis of Various Multipliers using Verilog HDL", CiiT International Journal of Programmable Device Circuits and Systems, vol.5, no.9, pp.391-398, Sep 2013.

[10] MarojuSaiKumar, P.Samundiswary, “Design and Performance Analysis of Various Multipliers using Verilog HDL”, CiiT International Journal of Programmable Device Circuits and Systems, vol.5, no.9, pp.391-398, Sep 2013.

[11] Nithya J, Sathiyabama G, Revathi K “Comparative Study of Low Power Low Area Bypass Multipliers for Signal Processing Applications” Int. Journal of Engineering Research and Applications Vol. 5, Issue 1(Part 2), pp.95-98, January 2015.

[12] Ron S. Waters and Earl E. Swartzlander, Jr., "A Reduced Complexity Wallace Multiplier Reduction,” IEEE Transactions On Computers, vol. 59, no. 8, pp.11341137, August 2010.

[13] Jasbir Kaur, Kavita, “Structural VHDL Implementation of Wallace Multiplier”, International Journal of Scientific & Engineering Research, vol.4, issue.4, pp.18291833, April 2013.

[14] Anju S, M Saravana, “High Performance Dadda Multiplier Implementation using High Speed Carry Select Adder”, International Journal of Advance Research in Computer and Communication Engineering, vol.2, issue.3, pp.1572-1575, March 2013.

[15] Xilinx13.4, “Synthesis and Simulation Design Guide”, UG626 (v13.4) January 19, 2012.

[16] Xilinx 13.1, “RTL and Technology Schematic Viewers Tutorial”, UG685 (v13.1), March 1, 2011.

[17]

APPENDIX A

Appendix A: Verilog codes

1. Half adder

```
module ha(s,c,a,b);
outputs,c;
inputa,b;
xor x1(s,a,b);
and x2(c,a,b);
endmodule
```

2. Full adder

```
modulefa(s,cout,a,b,cin);
outputs,cout;
inputa,b,cin;
wire [3:1] w;
xor x1(w[1],a,b);
and x2(w[2],a,b);
xor x3(s,w[1],cin);
and x4(w[3],cin,w[1]);
or x5(cout,w[3],w[2]);
endmodule
```

3. Dadda Multiplier

```
module dadda8(p,a,b);
output [15:0]p;
input [7:0]a,b;
wire [64:0]m;
wire [71:1]s,c;

and k1(m[0],a[0],b[0]);
assign p[0]=m[0];
and k2(m[1],a[0],b[1]);
and k3(m[2],a[1],b[0]);
ha k4(s[1],c[1],m[1],m[2]);
```

```
and k5(m[3],a[0],b[2]);
and k6(m[4],a[1],b[1]);
and k7(m[5],a[2],b[0]);
fa k8(s[2],c[2],m[3],m[4],m[5]);
and k9(m[6],a[0],b[3]);
and k10(m[7],a[1],b[2]);
and k11(m[8],a[2],b[1]);
fa k12(s[3],c[3],m[6],m[7],m[8]);
and k13(m[9],a[0],b[4]);
and k14(m[10],a[1],b[3]);
and k15(m[11],a[2],b[2]);
fa k16(s[4],c[4],m[9],m[10],m[11]);
and k17(m[12],a[0],b[5]);
and k18(m[13],a[1],b[4]);
and k19(m[14],a[2],b[3]);
fa k20(s[5],c[5],m[12],m[13],m[14]);
and k21(m[15],a[0],b[6]);
and k22(m[16],a[1],b[5]);
and k23(m[17],a[2],b[4]);
fa k24(s[6],c[6],m[15],m[16],m[17]);
and k25(m[18],a[0],b[7]);
and k26(m[19],a[1],b[6]);
and k27(m[20],a[2],b[5]);
fa k28(s[7],c[7],m[18],m[19],m[20]);
and k29(m[21],a[1],b[7]);
and k30(m[22],a[2],b[6]);
and k31(m[23],a[3],b[5]);
fa k32(s[8],c[8],m[21],m[22],m[23]);
and k33(m[24],a[2],b[7]);
and k34(m[25],a[3],b[6]);
and k35(m[26],a[4],b[5]);
fa k36(s[9],c[9],m[24],m[25],m[26]);
and k37(m[27],a[3],b[7]);
and k38(m[28],a[4],b[6]);
and k39(m[29],a[5],b[5]);
fa k40(s[10],c[10],m[27],m[28],m[29]);
and k41(m[30],a[4],b[7]);
and k42(m[31],a[5],b[6]);
and k43(m[32],a[6],b[5]);
fa k44(s[11],c[11],m[30],m[31],m[32]);
and k45(m[33],a[5],b[7]);
and k46(m[34],a[6],b[6]);
and k47(m[35],a[7],b[5]);
```

```
fa k48(s[12],c[12],m[33],m[34],m[35]);
and k49(m[36],a[6],b[7]);
and k50(m[37],a[7],b[6]);
ha k51(s[13],c[13],m[36],m[37]);
and k52(m[38],a[7],b[7]);


ha k53(s[14],c[14],s[2],c[1]);
and k54(m[39],a[3],b[0]);
fa k55(s[15],c[15],c[2],s[3],m[39]);
and k56(m[40],a[3],b[1]);
fa k57(s[16],c[16],c[3],s[4],m[40]);
and k58(m[41],a[3],b[2]);
fa k59(s[17],c[17],c[4],s[5],m[41]);
and k60(m[42],a[3],b[3]);
fa k61(s[18],c[18],c[5],s[6],m[42]);
and k62(m[43],a[3],b[4]);
fa k63(s[19],c[19],c[6],s[7],m[43]);
and k64(m[44],a[4],b[4]);
fa k65(s[20],c[20],c[7],s[8],m[44]);
and k66(m[45],a[5],b[4]);
fa k67(s[21],c[21],c[8],s[9],m[45]);
and k68(m[46],a[6],b[4]);
fa k69(s[22],c[22],c[9],s[10],m[46]);
and k70(m[47],a[7],b[4]);
fa k71(s[23],c[23],c[10],s[11],m[47]);
ha k72(s[24],c[24],c[11],s[12]);
ha k73(s[25],c[25],c[12],s[13]);
and k74(m[48],a[7],b[7]);
ha k75(s[26],c[26],c[13],m[48]);

ha k76(s[27],c[27],s[15],c[14]);
and k77(m[49],a[4],b[0]);
and k78(m[50],a[4],b[1]);
and k79(m[51],a[4],b[2]);
and k80(m[52],a[4],b[3]);
and k81(m[53],a[5],b[3]);
and k82(m[54],a[6],b[3]);
and k83(m[55],a[7],b[3]);
fa k84(s[28],c[28],s[16],c[15],m[49]);
fa k85(s[29],c[29],s[17],c[16],m[50]);
fa k86(s[30],c[30],s[18],c[17],m[51]);
fa k87(s[31],c[31],s[19],c[18],m[52]);
```

```
fa k88(s[32],c[32],s[20],c[19],m[53]);
fa k89(s[33],c[33],s[21],c[20],m[54]);
fa k90(s[34],c[34],s[22],c[21],m[55]);
ha k91(s[35],c[35],s[23],c[22]);
ha k92(s[36],c[36],s[24],c[23]);
ha k93(s[37],c[37],s[25],c[24]);
ha k94(s[38],c[38],s[26],c[25]);

ha k95(s[39],c[39],c[27],s[28]);
and k96(m[56],a[5],b[0]);
and k97(m[57],a[5],b[1]);
and k98(m[58],a[5],b[2]);
and k99(m[59],a[6],b[2]);
and k100(m[60],a[7],b[2]);
fa k101(s[40],c[40],c[28],s[29],m[56]);
fa k102(s[41],c[41],c[29],s[30],m[57]);
fa k103(s[42],c[42],c[30],s[31],m[58]);
fa k104(s[43],c[43],c[31],s[32],m[59]);
fa k105(s[44],c[44],c[32],s[33],m[60]);
ha k106(s[45],c[45],s[34],c[33]);
ha k107(s[46],c[46],s[35],c[34]);
ha k108(s[47],c[47],s[36],c[35]);
ha k109(s[48],c[48],s[37],c[36]);
ha k110(s[49],c[49],s[38],c[37]);
ha k111(s[50],c[50],c[26],c[38]);

ha k112(s[51],c[51],c[39],s[40]);
and k113(m[61],a[6],b[0]);
and k114(m[62],a[6],b[1]);
and k115(m[63],a[7],b[1]);
fa k116(s[52],c[52],c[40],s[41],m[61]);
fa k117(s[53],c[53],c[41],s[42],m[62]);
fa k118(s[54],c[54],c[42],s[43],m[63]);
ha k119(s[55],c[55],c[43],s[44]);
ha k120(s[56],c[56],c[44],s[45]);
ha k121(s[57],c[57],c[45],s[46]);
ha k122(s[58],c[58],c[46],s[47]);
ha k123(s[59],c[59],c[47],s[48]);
ha k124(s[60],c[60],c[48],s[49]);
ha k125(s[61],c[61],c[49],s[50]);

ha k126(s[62],c[62],c[51],s[52]);
and k127(m[64],a[7],b[0]);
```

```
fa k128(s[63],c[63],c[52],s[53],m[64]);
ha k129(s[64],c[64],c[53],s[54]);
ha k130(s[65],c[65],c[54],s[55]);
ha k131(s[66],c[66],c[55],s[56]);
ha k132(s[67],c[67],c[56],s[57]);
ha k133(s[68],c[68],c[57],s[58]);
ha k134(s[69],c[69],c[58],s[59]);
ha k135(s[70],c[70],c[59],s[60]);
ha k136(s[71],c[71],c[60],s[61]);
//ha k137(s[72],c[72],c[50],c[61]);

//ha k138(s[73],c[73],c[62],s[63]);
//ha k139(s[74],c[74],c[63],s[64]);
//ha k140(s[75],c[75],c[64],s[65]);
//ha k141(s[76],c[76],c[65],s[66]);
//ha k142(s[77],c[77],c[66],s[67]);
//ha k143(s[78],c[78],c[67],s[68]);
//ha k144(s[79],c[79],c[68],s[69]);
//ha k145(s[80],c[80],c[69],s[70]);
//ha k146(s[81],c[81],c[70],s[71]);
//ha k147(s[82],c[82],c[71],s[72]);
assign
p[15:1]={s[71],s[70],s[69],s[68],s[67],s[66],s[65],s[64],s[63],s[62],s[51],s[39],s[27],s[14],s[1]};

endmodule


modulefa(s,cout,a,b,cin);
outputs,cout;
inputa,b,cin;
wire [3:1] w;
xor x1(w[1],a,b);
and x2(w[2],a,b);
xor x3(s,w[1],cin);
and x4(w[3],cin,w[1]);
or x5(cout,w[3],w[2]);
endmodule


module ha(s,c,a,b);
outputs,c;
inputa,b;
```

```
xor x1(s,a,b);
and x2(c,a,b);
endmodule
```

4. LOFA

```
modulelofa(sum,carry,a,b,c);
outputsum,carry;
inputa,b,c;
wire x;

xor x1(x,a,b);
xor x2(sum,x,c);
mux1 x3(carry,b,c,x);
endmodule

module mux1(out,in1,in2,sel);
output out;
input in1,in2;
inputsel;
assign out=(sel==0)?in1:in2;
endmodule
```

5. Convolution

```
`timescale 1ns / 1ps

module conv(I1,I2,I3,I4,clk,sf_e,e,rs,rw,d,c,b,a);
//module conv(x1,x2,h1,h2,y1,y2,y3);
//output [15:0]y1,y2,y3;
(*LOC = "N17"*)      input   I1;

(*LOC = "H18"*)      input    I2;

(*LOC = "L14"*)      input   I3;

(*LOC = "L13"*)      input    I4;
```

```
////TestLCD.v    test LCD of sparten 3E board , XCS500E    model ,            320-pin            pack
(* LOC="C9"*)                    input clk;        // pin C9 is the 50-Hz on-board clock

(* LOC="D16"*)       output regsf_e; //1 LCD access (0 StartaFlashaccess )

(* LOC="M18"*)       output reg e;  //           enable(1)

(* LOC="L18"*)       output regrs; //   Register Select (1 data bit for R/W)

(* LOC="L17"*)       output regrw;//            read/Write , 1/0

(* LOC="M15"*)       output reg d; // 4th data bits    (to form a nibble)

(* LOC="P17"*)       output reg c;  // 3rd data bits   (to form a nibble)

(* LOC="R16"*)       output reg b;  // 2nd data bits  (to form a nibble)

(* LOC="R15"*)       output reg a; // 1st data bits    (to form a nibble)


reg[26:0] count=0;                          // 27- bit count ,0-(128M-1) over 2 secs
reg [5:0] code;                             // 6-bit  different signals  to give output
reg refresh;                                 //  refresh LCD rate @about 25Hz
//

//reg [15:0]add;
//reg [15:0]mul;
wire [7:0]x1, x2,h1,h2;
//input [7:0]x1, x2,h1,h2;
wire [15:0]y1,y2,y3,y4,y5,y6,y7;
wire[15:0]temp1,temp2,temp3,temp4,temp5,temp6,temp7,temp8,tenp9,temp10,temp11,temp12;
wire c1,c2,c3,c4,c5,c6,c7,c8,c9;
wire [15:0]s1;
//regcout;
//regcin = 0;
wirecin = 0;

//        if (I1 == 0)
//                x1 = 0;
//        else
//                x1 = 1;
//        if (I2 == 0)
//                x2 = 0;
```

```
//	else
//		x1 = 1;
//	if (I3 == 0)
//		h1 = 0;
//	else
//		h1 = 1;
//	if (I4 == 0)
//		h2 = 0;
//	else
//		h2 = 1;
//
//for first output of convolution = x[0]h[0]
//dadda8 P1(y1,x1,h1);
dadda8 P1(y1,I1,I1);
//dadda8 P1(y1,x1,h1);

//for second output  = x[1]h[0] + x[0]h[1]
//dadda8	p2(temp1, x2, h1);
//dadda8	p3(temp2, x1, h2);
//adder16	a1(y2, c1, temp1, temp2, cin);
dadda8		p2(temp1, I1, I2);
//dadda8	p3(temp2, I2, I1);
adder16		a1(y2, c1, temp1, temp1, cin);

//for the third output =
dadda8		p3(temp2, I1, I3);
dadda8		p4(temp3, I2, I2);
adder16		a2(temp4, c2, temp2, temp2, cin);
adder16		a3(y3, c3, temp4, temp3, c2);

//for the fourth output
dadda8		p5(temp5, I1, I4);
dadda8		p6(temp6, I2, I3);
adder16		a4(temp7, c4, temp5, temp5, cin);
adder16		a5(temp8, c5, temp6, temp6, c4);
adder16		a6(y4, c6, temp7, temp8, c5);

//for the fifth output
dadda8		p7(temp9, I2, I4);
dadda8		p8(temp10, I3, I3);
adder16		a7(temp11, c7, temp9, temp9, cin);
adder16		a8(y5, c8, temp11, temp10, c7);
```

```
//for the sixth output
dadda8          p9(temp12, I3, I4);
adder16         a9(y6, c9, temp12, temp12, cin);

//for the seventh output
dadda8          p10(y7, I4, I4);

//for third output = x[1]h[1]
//dadda8        p4(y3, x2, h2);
//dadda8        p4(y3, I2, I4);
//      dadda8          p4(y3, x2, h2);
always@(posedgeclk)
begin
                count<= count+1;

                case (count[26:21]) // as top 6 bits change
                                //power-on init can be carried out before this loop to         avoid
the flicker
                                        0: code<= 6'h03; // power-on init sequence
                                        1: code<= 6'h03;        // this is needed atleast once
                                        2: code<= 6'h03;        // when LCD's power on
                                        3: code<= 6'h02;   // it flickers existing char display

                                // Table 5-3, Function set
                                // send 00 and upper nibble 0000, then 00 and lower nibble 10xx

                                        4: code<= 6'h02;        // Function set
                                        5: code<= 6'h08;        // lower nibble 1000(10xx)

                                // Table 5-3,  Entry mode
                                // send 00 and upper nible 0000, then 00 and lower nibble  0 1 I/D
S
                                //      last 2 bits of lower nibble: I/D bit(Incr 1, Decr 0),S bit (shit
1,0 no)
                                        6: code<= 6'h00;        // see table upper nible 0000, then
lower nibble:
                                        7: code<= 6'h06; // 0110, Incr shift disabled

                                //Table 5-3, Display on/off
                                //send 00 and upper nible 0000, then 00 and lower niblle 1DCB:
                                // D:1 show char represnted by code in DDR, 0 dont but code
reain:
                                //C:1 show cursor, 0 dont
```

```
//B:1 show cursor blinks(if shown), 0 dont(if shown)
     8: code<= 6'h00;        //Display on/off, upper nibble 0000
     9: code<= 6'h0C;        //lower nibble 1100 (1 D C B)

//Table 5-3, clear dsiplay, 00 and upper nibble 0000, 00 and lower
nibble 0001
     10: code<= 6'h00; // clear display , 00 and upper nible 0000
     11: code<= 6'h01; // then 00 and lower nibble 0001

// characters are given out, the cursor will advance to right
//Table 5-3, write data to DD RAM (or CG RAM)
// Fig.5-4, H send 10 and upper nible 0100, then 10 and lower
nible 1000
     //12: code<= 6'h24;     //H: high nibble
     //13: code<= 6'h28;     //H: lower nibble


     12: code<= 6'h24;       //O
     13: code<= 6'h2F;

     14: code<= 6'h25;       //U
     15: code<= 6'h25;

     16: code<= 6'h25;       //T
     17: code<= 6'h24;

     18: code<= 6'h25;       //P
     19: code<= 6'h20;

     20: code<= 6'h25;       //U
     21: code<= 6'h25;

     22: code<= 6'h25;       //T
     23: code<= 6'h24;


//Table 5-3 set DD RAM(DDR)Address
//position the cursor onto the start of the second line
// send 00, and upper nible 1???, ????is the highest of the 3 bits of
DDR
//adderess to move the cursor to, then 00 and lower 4bits   of the
addr
// so??? is 100 and then 0000          for h40
```

```
        24: code<= 6'b001100; //pos cursor to second line upper
        25: code<= 6'b000000; // lower nibble: h0

// characters are the given out, the cursor will advance to the

        26: code<= 6'h23;
        27:
                if(y1 == 1)
                        code<= 6'h21;
                else
                        code<= 6'h20;

        28: code<= 6'h22;
        29: code<= 6'h20;

        30: code<= 6'h23;
        31:
                //if(y2 == 3)
                        //code<= 6'h23;
                if(y2 == 2)
                        code<= 6'h22;
                else if (y2 == 1)
                        code<= 6'h21;
                else
                        code<= 6'h20;


        32: code<= 6'h22;        //
        33: code<= 6'h20;

        34: code<= 6'h23;
        35:
                if(y3 == 3)
                        code<= 6'h23;
                else if(y3 == 2)
                        code<= 6'h22;
                else if (y3 == 1)
                        code<= 6'h21;
                else
                        code<= 6'h20;

        36: code<= 6'h22;        //
        37: code<= 6'h20;
```

```
38: code<= 6'h23;
39:
        if(y4 == 4)
                code<= 6'h24;
        else if(y4 == 3)
                code<= 6'h23;
        else if(y4 == 2)
                code<= 6'h22;
        else if (y3 == 1)
                code<= 6'h21;
        else
                code<= 6'h20;

40: code<= 6'h22;       //
41: code<= 6'h20;

42: code<= 6'h23;
43:
        if(y5 == 3)
                code<= 6'h23;
        else if(y5 == 2)
                code<= 6'h22;
        else if (y5 == 1)
                code<= 6'h21;
        else
                code<= 6'h20;

44: code<= 6'h22;       //
45: code<= 6'h20;

46: code<= 6'h23;
47:
        if(y6 == 2)
                code<= 6'h22;
        else if (y6 == 1)
                code<= 6'h21;
        else
                code<= 6'h20;

48: code<= 6'h22;       //
49: code<= 6'h20;
```

```
                                    50: code<= 6'h23;
                                    51:
                                         if (y7 == 1)
                                              code<= 6'h21;
                                         else
                                              code<= 6'h20;

                                    52: code<= 6'h22;       //
                                    53: code<= 6'h20;




                                    // Table 5-3 , read busy flag and address
                                    // send 01 BF (busy flag) x xx, the 01xxxX
                                    //idling
                               default: code<= 6'h10;        // the rest un-used time
                    endcase
          // refresh (enable) te LCD when bit 20 of the count is 1
          // (it flips when counted upto 2M, and flips again after another 2M)
         refresh<=count[20]; // flip rate about 25 (50MHz/2^21=2M)
         sf_e<=1;
          {e,rs,rw,d,c,b,a}<={refresh,code};
end       // always

endmodule


//16 bit adder using lofa
module adder16(sum,cout,a,b,c);
output [15:0]sum;
outputcout;
input [15:0]a,b;
input c;

wirecout;
wire [15:0]sum;

wire c1,c2,c3,c4,c5,c6,c7,c8,c9,c10,c11,c12,c13,c14,c15;

lofa l0(sum[0], c1, a[0], b[0], c);
lofa l1(sum[1], c2, a[1], b[1], c1);
lofa l2(sum[2], c3, a[2], b[2], c2);
```

```
lofa l3(sum[3], c4, a[3], b[3], c3);
lofa l4(sum[4], c5, a[4], b[4], c4);
lofa l5(sum[5], c6, a[5], b[5], c5);
lofa l6(sum[6], c7, a[6], b[6], c6);
lofa l7(sum[7], c8, a[7], b[7], c7);
lofa l8(sum[8], c9, a[8], b[8], c8);
lofa l9(sum[9], c10, a[9], b[9], c9);
lofa l10(sum[10], c11, a[10], b[10], c10);
lofa l11(sum[11], c12, a[11], b[11], c11);
lofa l12(sum[12], c13, a[12], b[12], c12);
lofa l13(sum[13], c14, a[13], b[13], c13);
lofa l14(sum[14], c15, a[14], b[14], c14);
lofa l15(sum[15], cout, a[15], b[15], c15);

endmodule

//Dadda Multiplier
module dadda8(p,a,b);
output [15:0]p;
input [7:0]a,b;
wire [64:0]m;
wire [71:1]s,c;

and k1(m[0],a[0],b[0]);
assign p[0]=m[0];
and k2(m[1],a[0],b[1]);
and k3(m[2],a[1],b[0]);
ha k4(s[1],c[1],m[1],m[2]);
and k5(m[3],a[0],b[2]);
and k6(m[4],a[1],b[1]);
and k7(m[5],a[2],b[0]);
fa k8(s[2],c[2],m[3],m[4],m[5]);
and k9(m[6],a[0],b[3]);
and k10(m[7],a[1],b[2]);
and k11(m[8],a[2],b[1]);
fa k12(s[3],c[3],m[6],m[7],m[8]);
and k13(m[9],a[0],b[4]);
and k14(m[10],a[1],b[3]);
and k15(m[11],a[2],b[2]);
fa k16(s[4],c[4],m[9],m[10],m[11]);
and k17(m[12],a[0],b[5]);
and k18(m[13],a[1],b[4]);
and k19(m[14],a[2],b[3]);
```

```
fa k20(s[5],c[5],m[12],m[13],m[14]);
and k21(m[15],a[0],b[6]);
and k22(m[16],a[1],b[5]);
and k23(m[17],a[2],b[4]);
fa k24(s[6],c[6],m[15],m[16],m[17]);
and k25(m[18],a[0],b[7]);
and k26(m[19],a[1],b[6]);
and k27(m[20],a[2],b[5]);
fa k28(s[7],c[7],m[18],m[19],m[20]);
and k29(m[21],a[1],b[7]);
and k30(m[22],a[2],b[6]);
and k31(m[23],a[3],b[5]);
fa k32(s[8],c[8],m[21],m[22],m[23]);
and k33(m[24],a[2],b[7]);
and k34(m[25],a[3],b[6]);
and k35(m[26],a[4],b[5]);
fa k36(s[9],c[9],m[24],m[25],m[26]);
and k37(m[27],a[3],b[7]);
and k38(m[28],a[4],b[6]);
and k39(m[29],a[5],b[5]);
fa k40(s[10],c[10],m[27],m[28],m[29]);
and k41(m[30],a[4],b[7]);
and k42(m[31],a[5],b[6]);
and k43(m[32],a[6],b[5]);
fa k44(s[11],c[11],m[30],m[31],m[32]);
and k45(m[33],a[5],b[7]);
and k46(m[34],a[6],b[6]);
and k47(m[35],a[7],b[5]);
fa k48(s[12],c[12],m[33],m[34],m[35]);
and k49(m[36],a[6],b[7]);
and k50(m[37],a[7],b[6]);
ha k51(s[13],c[13],m[36],m[37]);
and k52(m[38],a[7],b[7]);


ha k53(s[14],c[14],s[2],c[1]);
and k54(m[39],a[3],b[0]);
fa k55(s[15],c[15],c[2],s[3],m[39]);
and k56(m[40],a[3],b[1]);
fa k57(s[16],c[16],c[3],s[4],m[40]);
and k58(m[41],a[3],b[2]);
fa k59(s[17],c[17],c[4],s[5],m[41]);
and k60(m[42],a[3],b[3]);
```

```
fa k61(s[18],c[18],c[5],s[6],m[42]);
and k62(m[43],a[3],b[4]);
fa k63(s[19],c[19],c[6],s[7],m[43]);
and k64(m[44],a[4],b[4]);
fa k65(s[20],c[20],c[7],s[8],m[44]);
and k66(m[45],a[5],b[4]);
fa k67(s[21],c[21],c[8],s[9],m[45]);
and k68(m[46],a[6],b[4]);
fa k69(s[22],c[22],c[9],s[10],m[46]);
and k70(m[47],a[7],b[4]);
fa k71(s[23],c[23],c[10],s[11],m[47]);
ha k72(s[24],c[24],c[11],s[12]);
ha k73(s[25],c[25],c[12],s[13]);
and k74(m[48],a[7],b[7]);
ha k75(s[26],c[26],c[13],m[48]);

ha k76(s[27],c[27],s[15],c[14]);
and k77(m[49],a[4],b[0]);
and k78(m[50],a[4],b[1]);
and k79(m[51],a[4],b[2]);
and k80(m[52],a[4],b[3]);
and k81(m[53],a[5],b[3]);
and k82(m[54],a[6],b[3]);
and k83(m[55],a[7],b[3]);
fa k84(s[28],c[28],s[16],c[15],m[49]);
fa k85(s[29],c[29],s[17],c[16],m[50]);
fa k86(s[30],c[30],s[18],c[17],m[51]);
fa k87(s[31],c[31],s[19],c[18],m[52]);
fa k88(s[32],c[32],s[20],c[19],m[53]);
fa k89(s[33],c[33],s[21],c[20],m[54]);
fa k90(s[34],c[34],s[22],c[21],m[55]);
ha k91(s[35],c[35],s[23],c[22]);
ha k92(s[36],c[36],s[24],c[23]);
ha k93(s[37],c[37],s[25],c[24]);
ha k94(s[38],c[38],s[26],c[25]);

ha k95(s[39],c[39],c[27],s[28]);
and k96(m[56],a[5],b[0]);
and k97(m[57],a[5],b[1]);
and k98(m[58],a[5],b[2]);
and k99(m[59],a[6],b[2]);
and k100(m[60],a[7],b[2]);
fa k101(s[40],c[40],c[28],s[29],m[56]);
```

```
fa k102(s[41],c[41],c[29],s[30],m[57]);
fa k103(s[42],c[42],c[30],s[31],m[58]);
fa k104(s[43],c[43],c[31],s[32],m[59]);
fa k105(s[44],c[44],c[32],s[33],m[60]);
ha k106(s[45],c[45],s[34],c[33]);
ha k107(s[46],c[46],s[35],c[34]);
ha k108(s[47],c[47],s[36],c[35]);
ha k109(s[48],c[48],s[37],c[36]);
ha k110(s[49],c[49],s[38],c[37]);
ha k111(s[50],c[50],c[26],c[38]);

ha k112(s[51],c[51],c[39],s[40]);
and k113(m[61],a[6],b[0]);
and k114(m[62],a[6],b[1]);
and k115(m[63],a[7],b[1]);
fa k116(s[52],c[52],c[40],s[41],m[61]);
fa k117(s[53],c[53],c[41],s[42],m[62]);
fa k118(s[54],c[54],c[42],s[43],m[63]);
ha k119(s[55],c[55],c[43],s[44]);
ha k120(s[56],c[56],c[44],s[45]);
ha k121(s[57],c[57],c[45],s[46]);
ha k122(s[58],c[58],c[46],s[47]);
ha k123(s[59],c[59],c[47],s[48]);
ha k124(s[60],c[60],c[48],s[49]);
ha k125(s[61],c[61],c[49],s[50]);

ha k126(s[62],c[62],c[51],s[52]);
and k127(m[64],a[7],b[0]);
fa k128(s[63],c[63],c[52],s[53],m[64]);
ha k129(s[64],c[64],c[53],s[54]);
ha k130(s[65],c[65],c[54],s[55]);
ha k131(s[66],c[66],c[55],s[56]);
ha k132(s[67],c[67],c[56],s[57]);
ha k133(s[68],c[68],c[57],s[58]);
ha k134(s[69],c[69],c[58],s[59]);
ha k135(s[70],c[70],c[59],s[60]);
ha k136(s[71],c[71],c[60],s[61]);
//ha k137(s[72],c[72],c[50],c[61]);

//ha k138(s[73],c[73],c[62],s[63]);
//ha k139(s[74],c[74],c[63],s[64]);
//ha k140(s[75],c[75],c[64],s[65]);
//ha k141(s[76],c[76],c[65],s[66]);
```

```
//ha k142(s[77],c[77],c[66],s[67]);
//ha k143(s[78],c[78],c[67],s[68]);
//ha k144(s[79],c[79],c[68],s[69]);
//ha k145(s[80],c[80],c[69],s[70]);
//ha k146(s[81],c[81],c[70],s[71]);
//ha k147(s[82],c[82],c[71],s[72]);
assign
p[15:1]={s[71],s[70],s[69],s[68],s[67],s[66],s[65],s[64],s[63],s[62],s[51],s[39],s[27],s[14],s[1]};

endmodule


modulefa(s,cout,a,b,cin);
outputs,cout;
inputa,b,cin;
wire [3:1] w;
xor x1(w[1],a,b);
and x2(w[2],a,b);
xor x3(s,w[1],cin);
and x4(w[3],cin,w[1]);
or x5(cout,w[3],w[2]);
endmodule



module ha(s,c,a,b);
outputs,c;
inputa,b;
xor x1(s,a,b);
and x2(c,a,b);
endmodule


//LOFA ADDER

modulelofa(sum,carry,a,b,c);
outputsum,carry;
inputa,b,c;
wire x;

xor x1(x,a,b);
xor x2(sum,x,c);
mux1 x3(carry,b,c,x);
```

```
endmodule

//multiplexer module
module mux1(out,in1,in2,sel);
 output out;
 input in1,in2;
 inputsel;
 assign out=(sel==0)?in1:in2;
endmodule
```