DESIGN AND IMPLEMENTATION OF AREA EFFICIENT MULTIPLY-ACCUMULATE UNIT

A Project report submitted in partial fulfilment of the requirements for

the award of the degree of

BACHELOR OF TECHNOLOGY

IN

ELECTRONICS AND COMMUNICATION ENGINEERING

Submitted by

P.SAI RAM(319126512170)

K.MANI KUMAR (320126512L17)

Y.SURESH REDDY (320126512L19)

B.UDAY NARAYANA (319126512136)

Under the guidance of

Dr.K.V.Gowreesrinivas

Assistant Professor

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

ANIL NEERUKONDA INSTITUTE OF TECHNOLOGY AND SCIENCES

(UGC AUTONOMOUS)

(Permanently Affiliated to AU, Approved by AICTE and Accredited by NBA & NAAC with 'A' Grade)

Sangivalasa, Bheemili mandal, Visakhapatnam dist.(A.P)(2022-2023)

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

ANIL NEERUKONDA INSTITUTE OF TECHNOLOGY AND SCIENCES

(Permanently Affiliated to AU, Approved by AICTE and Accredited by NBA & NAAC with 'A' Grade)

Sangivalasa, Bheemili Mandal, Visakhapatnam dist.(A.P)

PRAGNANAM BRAHMA

ANITS

CERTIFICATE

This is to certify that the project report entitled **"DESIGN AND IMPLMENTATION OF AREA EFFICIENT MULTIPLY-ACCUMULATE UNIT"** submitted by **P SAI RAM(319126512170), K MANI KUMAR (320126512L17), Y SURESH REDDY (320126512L19), B UDAY NARAYANA (319126512136)** in partial fulfillment of the requirements for the award of the degree of **Bachelor of Technology in Electronics & Communication Engineering** of Andhra University, Visakhapatnam is a record of Bonafide work carried out under my guidance and supervision.

K.V.G.Srinivas

Project Guide

Dr.K.V.Gowreesrinivas
Assistant Professor
Department of E.C.E
ANITS

Assistant Professor
Department of E.C.E.
Anil Neerukonda
Institute of Technology & Sciences
Sangivalasa, Visakhapatnam-531 162

[illegible]

Head of the Department

Prof. B.Jagadeesh
Professor & HOD
Department of E.C.E
ANITS

Head of the Department
Department of E C E
Anil Neerukonda Institute of Technology & Sciences
Sangivalasa - 531 162

ACKNOWLEDGEMENT

We would like to express our deep gratitude to our project guide **Dr.K.V.Gowreesrinivas** Assistant professor, Department of Electronics and Communication Engineering, ANITS, for his guidance with unsurpassed knowledge and immense encouragement. We are grateful to **Dr. B.Jagadeesh**, Head of the Department, Electronics and Communication Engineering, for providing us with the required facilities for the completion of the project work.

We are very much thankful to the **Principal and Management, ANITS, Sangivalasa ,** for their encouragement and cooperation to carry out this work.

We express our thanks to all **Teaching faculty** of Department of ECE, whose suggestions during reviews helped us in the accomplishment of our project. We would like to thank **all non-teaching staff** of the Department of ECE, ANITS for providing great assistance in accomplishment of our project.

We would like to thank our parents, friends, and classmates for their encouragement throughout our project period. Last but not the least, we thank everyone for supporting us directly or indirectly in completing this project successfully.

PROJECT STUDENTS

P SAI RAM(319126512170)
K MANI KUMAR (320126512L17)
Y SURESH REDDY (320126512L19)
B UDAY NARAYANA (319126512136)

CONTENTS

List of Figures

List of Tables

ABSTRACT

The objective of this paper is to propose an 8-bit MAC using FPGA. In Conventional multiplier, there are many partial steps which reduce the computational speed of a multiplier. Along with accuracy demand for minimizing area, power, and delay of the processor by enhancing speed is the focus point. Vedic mathematics rules and algorithms generate partial products concurrently and save time. The multipliers performance in terms of latency, power, and area is determined by the number of stages used to sum the partial products. In this paper, a new architecture of Vedic multiplier is introduced which reduces the area and increase the speed when compared to the conventional Vedic multiplier. The proposed model is simulated and synthesized using Xilinx Vivado on different FPGA families and delay, area and power are observed. The conclusion drawn from observing the result is that implementation of multiplier using Virtex-6 (lower power) provides optimized outputs in terms of area, delay and power. The model is also implemented on Basys3 Artix7 FPGA with the help of Xilinx Vivado and verified the functionality of the proposed model

CHAPTER-1

INTRODUCTION

One of the most frequently utilized arithmetic knowledge paths in contemporary digital design is the multiplier. An essential and computationally demanding process is multiplication. Several components of a digital system or computing device, most notably signal processing, graphics, and scientific calculation, all benefit from the multiplication operation. The Booth algorithmic programme could be a significant advancement in the signed binary multiplication method. In many digital signal processing (DSP) applications involving multiplications and/or accumulations, MAC units are a necessary component. MAC unit [1] is used for high-performance digital signal processing systems. The execution speed and performance of the complete calculation are determined by the multiplication and addition arithmetic speed since they are essentially performed by repetitive application of multiplication and addition. As a result, the MAC unit's capabilities enable high-speed filtering and other processing that is typical of DSP applications. A multiplier and an accumulator holding the total of the prior sequential products make up a MAC unit. The design of a high-performance 64-bit Multiplier-Accumulator (MAC) is implemented using Verilog HDL . Due of their greater speed performance, compressor adders have been chosen for this method over traditional half-adder and full adder architectures. These compressors behave like adders because they actually serve as counters, counting the number of ones in the input bits. In addition to half-adders and full adders, they use multiplexers, which enable the use of smaller XOR gates and hence higher speed. Some carries are also produced with each resulting bit, which is used to calculate the following final product bits.

1.1 PROJECT OBJECTIVE:

The primary goal of this project is to research, construct, and analyses a higher order compressor adder 16-bit MAC unit using VLSI (Very Large-Scale Integration) design. Xilinx Vivado is used for the Synthesis and Implementation of various compressor adder types. Power, area, and delay comparisons are made for the performance.

1.2 PROJECT OUTLINE:

This project report is presented over the 6 remaining chapters.

Chapter 2 presents Introduction to Vedic Mathematics

Chapter 3 explains Introduction to Multipliers .

Chapter 4 is about Introduction to Compressor Adders and its different types.

Chapter 5 introduction to adder unit

Chapter 6 results

Chapter 7 introduction to Verilog

Chapter 8 xilinx software

Finally, the results of the project work and conclusions are drawn.

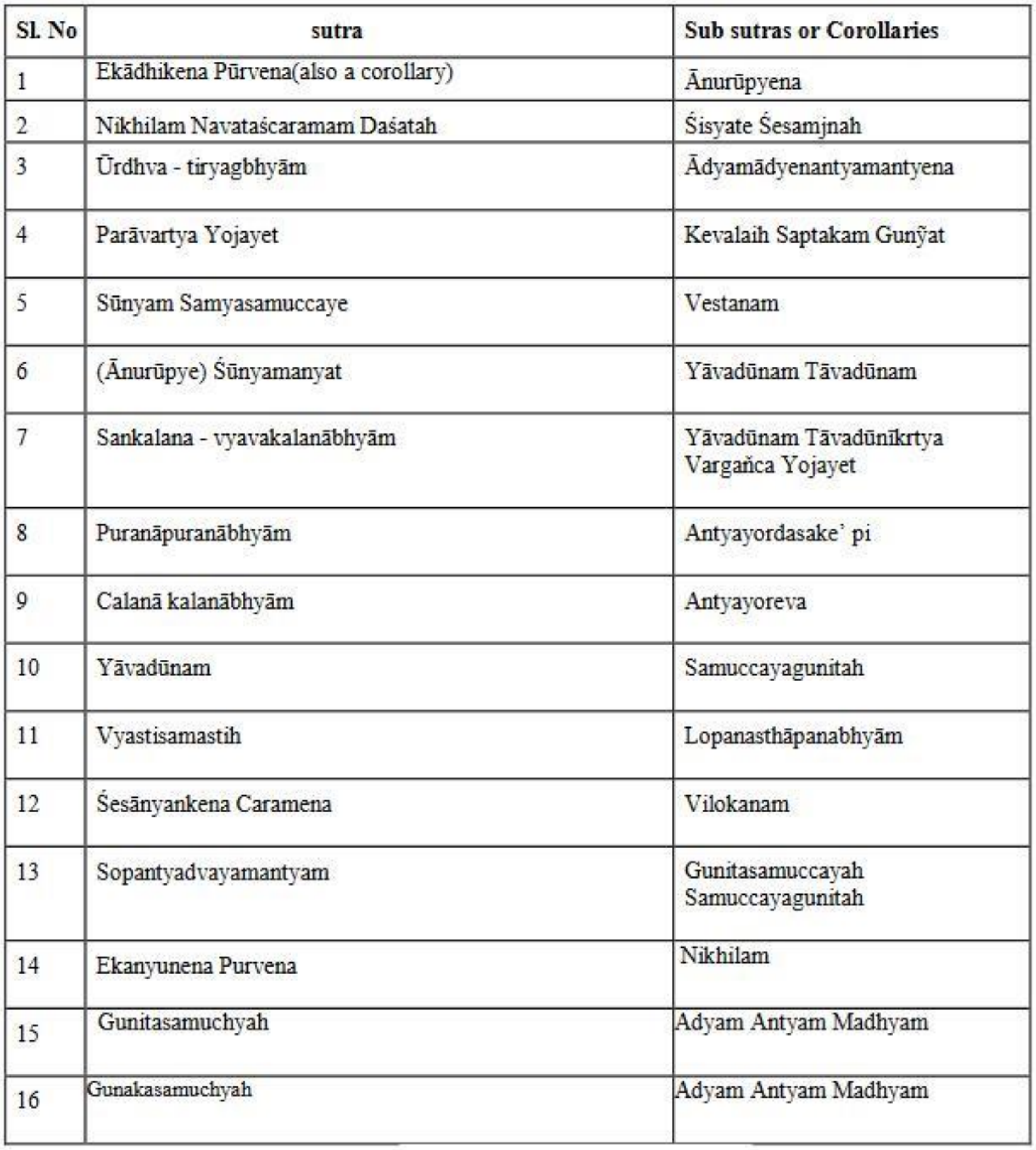
CHAPTER 2

INTRODUCTION TO VEDIC MATHEMATICS

In this chapter, we merely review a few ideas that were previously covered in the Vedic Mathe matics textbook published by Jagadguru Swami Sri Bharati Krsna Tirthaji Maharaja (Sankarach arya of Govardhana Matha, Puri, Orissa, India), General Editor, Dr. V.S. Agrawala.Quick overv iew of his background before.He was born to highly educated and religious parents in March18 84At Tinnevelly (Madras Presidency), his father Sri P Narasimha Shastri served as a Tahsildar before retiring as a Deputy Collector.His greatgrandfather was Justice C. Ranganath Shastri of t he Madras High Court, and his uncle, Sri Chandrasekhar Shastri, served as the principal of the Maharajas College at Vizianagaram.He attended the Hindu College Tinnivelly, Church Mission ary Society College, and National College Tiruchirapalli in Tamil Nadu.He topped the list after he successfully completed the Madras University matriculation exams .In July 1899, the Madra s Sanskrit Association bestowed upon him the title of "Saraswati" in recognition of his exceptio nal command of Sanskrit.After placing first in the B.A. test, Sri Venkataraman applied from th e Bombay Centre for the M.A. examination at the American College of Sciences, Rochester, Ne w York, in 1903.Sanskrit, philosophy, english,mathematics, history, and science were among th e subjects he was tested on.He had an excellent memory.He abruptly left his position as a teach er in 1911 because he was unable to contain his burning desire for spiritual knowledge, practise and attainment at that time. He returned Sacchidananda Shivabhinava Nrisimha Bharati Swam i in Sringeri.The following eight years were devoted to his indepth study of the Brahmasadhana practise and the most cuttingedge Vedanta philosophy. He was appointed to the pontifical throne of Sharada Peetha Sankaracharya in 1921 after serving for a period. Later, in 1925, he was made pontifical head of Sri Govardhan Math Puri, where he spent the remainder of his life.When he eventually made the decision to travel to the United States in 1957, he rewrote the current volume of Vedic Mathematics from memory, providing an overview of the sixteen equa tions he had successfully recreated.The 16 sutras (aphorisms or formulae) and their corollaries a re now provided.According to the editor, the text's introduction includes a list of the primary 16 sutras and any subsutras or corollaries.

The wording also suggests that Sri Swamiji was responsible for their discovery.The editor also believes that since each educated reader should be able to uncover the immense merit of these g uidelines, it is unnecessary to dwell any longer on this point of origin.

Table 1: 16 sutras from the Vedas



Sl. No	sutra	Sub sutras or Corollaries
1	Ekādhikena Pūrvena(also a corollary)	Ānurūpyena
2	Nikhilam Navataścaramam Daśatah	Śisyate Śesamjnah
3	Ūrdhva - tiryagbhyām	Ādyamādyenantyamantyena
4	Parāvartya Yojayet	Kevalaih Saptakam Gunỹat
5	Sūnyam Samyasamuccaye	Vestanam
6	(Ānurūpye) Śūnyamanyat	Yāvadūnam Tāvadūnam
7	Sankalana - vyavakalanābhyām	Yāvadūnam Tāvadūnīkrtya Vargañca Yojayet
8	Puranāpuranābhyām	Antyayordasake' pi
9	Calanā kalanābhyām	Antyayoreva
10	Yāvadūnam	Samuccayagunitah
11	Vyastisamastih	Lopanasthāpanabhyām
12	Śesānyankena Caramena	Vilokanam
13	Sopantyadvayamantyam	Gunitasamuccayah Samuccayagunitah
14	Ekanyunena Purvena	Nikhilam
15	Gunitasamuchyah	Adyam Antyam Madhyam
16	Gunakasamuchyah	Adyam Antyam Madhyam

The fundamental formula is supplied by the phrase "URDHVA TRIYAKBHYAM," which transl ates to "Vertical crosswise Urdhvatriyakbhyam sutra." This formula is applicable to all circumsta nces of multiplication and will subsequently prove to be very helpful in the division of a huge nu mber by another large number.Since there is only one compound word in the entire formula, it is incredibly condensed and simply says, "vertically and crosswise. "This succinct sutra has a wide range of uses. This sutra was selected because it offers a general formula that can be applied to all multiplication scenarios (large bit multiplication, small bit multiplication, and modular multiplication), and it is also very compact when dividing a large number by another large number, such as when dividing a 15-digit number by a 5-digit number.

The underlying mathematic operation is described as follows.

Multiplication Using UrdhvaTriyakbhyamSutra:

Assume we must divide (ax+b) by (cx+d). acx2 + x(ad+bc) + bd is the result. These steps can be taken to get this: The vertical multiplication of a and c yields the coefficient of x2 in step 1.

Step 2: The coefficient of x is derived by adding the two products of the crosswise multiplication of a and d and b and c.

Step 3: The absolute terms b and d are vertically multiplied to obtain the independent term.

The modus operandi thereof will be made clear by a straightforward illustration. Imagine that we need to multiply 12 by 13.

The leftmost part of the answer is calculated by multiplying the leftmost digit 1 of each of the 12 Multiplicands vertically by the leftmost digit 1 of the multiplier, yielding the product 1 1:3 + 2:6 = 156.

The middle and rightmost parts of the answer are calculated as follows:

- We multiply 1 and 3 and 1 and 2 crosswise add the two to get 5 as sum.
- We multiply 2 and 3 vertically to get 6 as their product. Thus 12 ´ 13 = 156.

Example of 8X8 bit multiplication:

Let A be the 8-bit multiplicand and B be the 8-bit multiplier. These can be further divided into 4-bit terms as shown below:

A = A7A6A5A4 A3A2A1A0
X1 X0
B = B7B6B5B4 B3B2B1B0
Y1 Y0

SoA = X1 X0 (8 bit Multiplicand) B = Y1Y0 (8 bit Multiplier where X1, X0, Y1, Y0 are each of 4-bits. Multiplying, we get a 16-bit product, which is further divided into 4 four-bit terms, F, E, D, C.

X1 X0 x Y1Y0 = F E D C

1. CP = X0 x Y0 = C.
2. CP = X1 x Y0 + X0 x Y1 = D
3. CP = X1 x Y1 = F E

where F is the carry of the product of X1 x Y1 and CP is the cross product

Note:

1. Each multiplicand is a parallel 4 x 4 multiply module that is implanted. Each of the multiplication modules generates a carry, which is carried over to the following module.
2. This multiplier architecture benefits from reduced gate delays and more structural regularity.

Examples are used to further clarify the procedure. Decimal numbers are used to illustrate two- and three-digit multiplication examples, and lines are used to represent the multiplication operation. The line's digits are multiplied, the result is added to the carry from the previous carry, and the procedure is repeated.

CHAPTER 3

PROPOSEDMULTIPLIER

3.1 INTRODUCTION

Numerous modern applications, including digital signal processing, rely significantly on multipliers. With the development of technology, many researchers have tried and are still trying to create multipliers that provide one or more of the following design goals: high speed, low power consumption, regularity of layout and thus less area, or even a combination of them in one multiplier, making them suitable for various high speed, low power, and compact VLSI implementation.

The "add and shift" algorithm is a popular technique for multiplying numbers. The required number of partial products has a significant impact on the performance of parallel multipliers. One of the most widely used algorithms is the Modified Booth method, which lowers the amount of incomplete products that must be added.

Improving speed is the goal. One method for reducing the number of sequential addition stages is to use the Wallace Tree algorithm. Additionally, we can observe the benefits of both algorithms in a single multiplier by combining the Wallace Tree methodology and the Modified Booth algorithm.

The processing could get slower as parallelism is raised since there will be more shifts between the partial products and intermediate sums that need to be added. It could also result in an increase in silicon area due to irregular structure, as well as a rise in power consumption due to an increase in interconnect from complex routing.

However, "serial-parallel" multipliers give up speed for improved performance in terms of space and power usage. We describe the architecture and methods for multiplying numbers and contrast them in terms of their speed, area, power, and mashups of these measures.

3.2 PROPOSED MULTIPLIER:

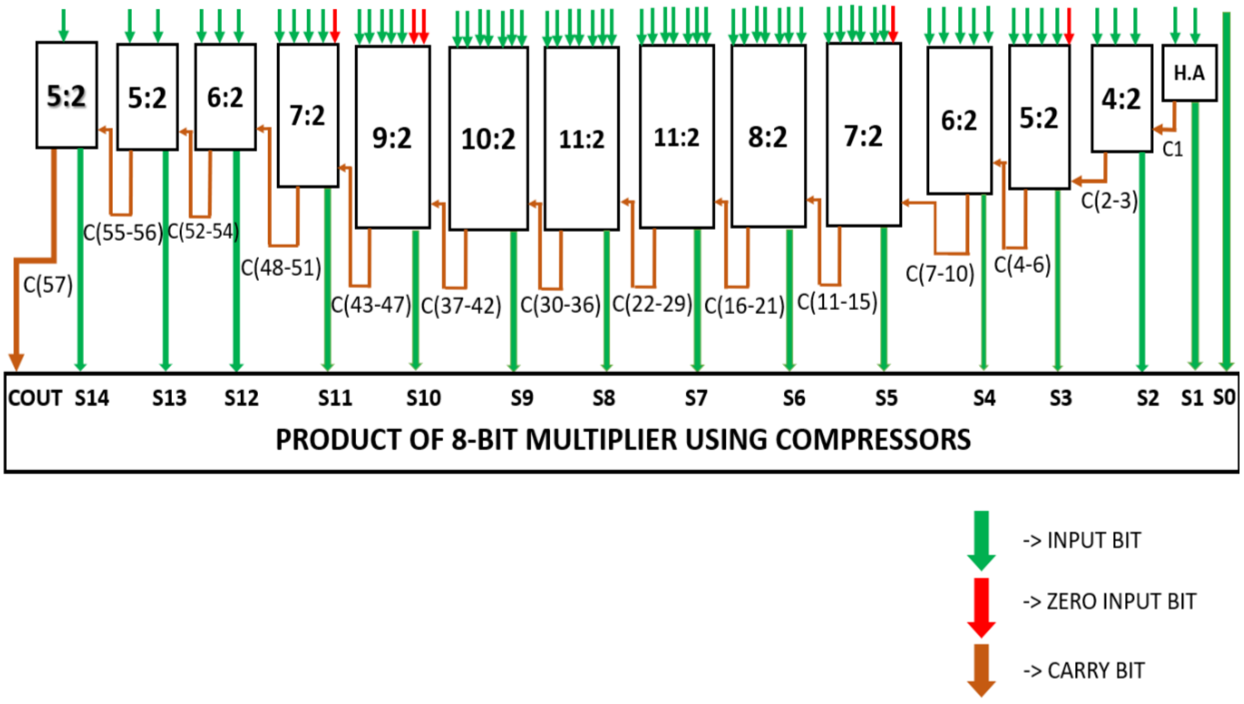


Fig 3.2.1 design of propose multipier

In this proposed multiplier, we have used Vedic Multiplication method to design an area-efficient multiplier,as Vedic multiplication is more efficient when compared toother multipliers. We have designed this multiplier uusing compressorsto reduce the partial products in multiplication.This model consists of various compressors which produce the output in single stages without any partial steps.In normal multiplier methods there are many partial steps which increases the computational speed of a multiplier. So using Vedic Multiplication with compressors reduce the area of the multiplier unit. In this design, the output of the respective bits depend only on the previous carry bits and input bits of that particular stage and do not depend on the carry bits of other stages. Here the carry bits generated in the particular stage will get added up all together to the next respective input bits. So in this way carry propagation takes place in the proposed multiplier which is quite different from the existing models. In the existing models if the carry bits generated at the stage is more than the one then the carry bits are propagated to each and every upcoming stages in a linear fashion which makes the upcoming stages not only depend on the previous carry bits but also on the other carry bits. The size of the compressor depends on the total sum of the input bits and the carry bits transmitted to that specific stage in the compressors employed in this proposed multiplier. The intended multiplier unit's area will be reduced by choosing the compressor in this way.

In the proposed multiplier, various compressors such as 4:2 ,5:2, 7:2 compressors and other higher order compressors are used at certain positions to get the results of 16 bit multiplication.

The block diagram of our proposed multiplier is shown in fig-10. In the fig-10 we can see that there are various compressors used based on input and carry bit size.

The arrows in fig-10 which are in green, brown and red represents input, carry, zero bits.

EQUATIONS:

Let X and Y are two 8 bit binary inputs and S is the output of size 16 bits.

1.S0 = X0Y0

2.S1 = X1Y0 + X0Y1 (C0)

3.S2 = X2Y0 + X1Y1 + X0Y2 + C0 (C1, C2)

4.S3 = X3Y0 + X2Y1 + X1Y2 + X0Y3 + C1 + C2 (C3, C4, C5)

5.S4 = X4Y0 + X3Y1 + X2Y2 + X1Y3 + X0Y4 + C3 + C4 +C5 (C6,C7, C8, C9)

6.S5 = X5Y0 + X4Y1 +X3Y2 +X2Y3 +X1Y4 +X0Y5 + C6 +C7 +C8+ C9 (C10, C11, C12, C13, C14)

7.S6 = X6Y0 + X5Y1 +X4Y2 +X3Y3 + X2Y4 +X1Y5 + X0Y6 +C10+C11 +C12 +C13 +C14 (C15,C16,C17,C18,C19,C20)

8.S7 = X7Y0+X6Y1 + X5Y2 +X4Y3 + X3Y4 + X2Y5 +X1Y6 + X0Y7+C15+C16+C17+C18+C19+C20 (C21,C22,C23,C24,C25,C26,C27,C28)

9.S8 = X7Y1 + X6Y2 + X5Y3 + X4Y4 + X3Y5 + X2Y6 + X4Y7 +C21+C22+C23+C24+C25+C26+C27+C28 (C29,C30,C31,C32,C33,C34,C35,C36)

10.S9 = X7Y2 + X6Y3 + X5Y4 + X4Y5 +X3Y6 + X2Y7 + C29+C30+C31+C32+C33+C34+C35+C36 (C37,C38,C39,C40,C41,C42,C43)

11.S10 = X7Y3 + X6Y4 + X5Y5 + X4Y6 + X3Y7 + C37+C38+C39+C40+C41+C42+C43 (C44,C45,C46,C47,C48,C49)

12.S11 = X7Y4 + X6Y5 + X5Y6 + X4Y7 + C44 +C45+C46+C47+C48+C49 (C50,C51,C52,C53,C54)

13.S12 = X7Y5 + X6Y6 + X5Y7 +C50+C51+C52+C53+C54 (C55,C56,C57,C58)

14.S13 = X7Y6 + X6Y7 +C55+C56+C57+C58(C59,C60,C61)

15.S14 = X7Y7 + C59+C60+C61 (C62,C63)

16.S15 = C62 + C63 (C64).

In this method, various higher order compressors were involved in performing multiplication as per Vedic algorithm. Here, we take two 8-bit binary values, X7 to X0 and Y7 to Y0, where 0 to 7 denote the bits from least to greatest significance and S15 to S0 denote the binary output bits acquired when X and Y are multiplied. The output bits S0 to S15 are computed by using the partial products which are obtained by performing logical "AND" operation between the input bits. While adding the partial products in each stage, the carry bits from C0 to C64 are generated and these carry bits are propagated in such a manner that all the carry bits generated at the stage are propagated to the next upcoming stage. C64 carry bit is neglected as it is superfluous. The individual bits which are obtained from the below mentioned equations are concatenated, which in turn produces the final output of multiplication.

3.3 WORKING OF MULTIPLIER:

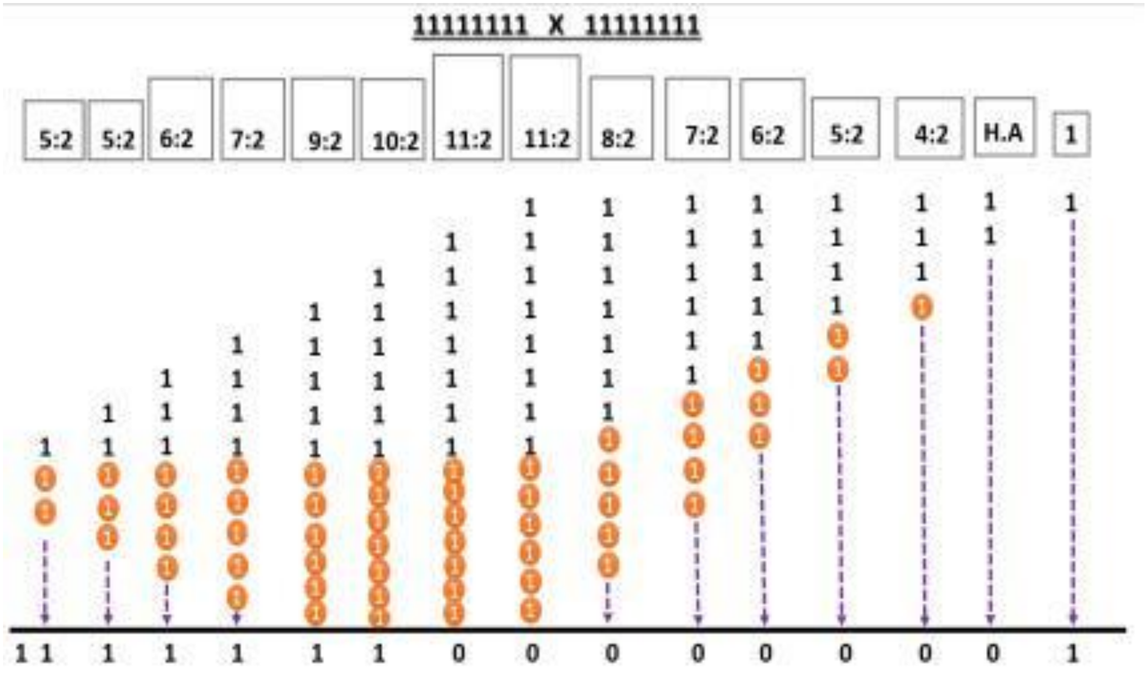


Fig 3.3.1 multiplication of 11111111 x 11111111

EXAMPLE:

From figure 10, it is observed that two 8-bit numbers i.e.(11111111 x 11111111) are multiplied using the proposed model and verified the output using simulation tool. Here in fig 11, the number of carry bits generated at right second most stage is one and that is added to the next upcoming stage and relevant compressor is used for addition at that stage. Similarly the carry bits generated at right third most stage are two and those two carry bits simultaneously added up to the next upcoming stage without any further propagation of the carry and relevant compressor is used at that particular stage. This process will continue till the last stage and output of the two input binary numbers are computed successfully. By propagating all the carry bits at a time without any delay we can reduce the delay time of the multiplier and by using relevant compressors at each and every stage of the process we can reduce the area of the multiplier

CHAPTER-4

INTRODUCTION TO COMPRESSOR ADDERS

4.1 COMPRESSOR ADDERS

In contrast to combinational circuits of half and full adders, compressor adders are fundamental circuits that add bits more than four at a time to produce better delay outcomes. The symbol for compressor design is N r, where 'N' stands for the number of bits fed in and 'r' stands for the overall number of 1s found in N bits. In contrast to adder circuits, it actually reduces gate counts and latency, hence the term compressor. The circuits of lower compressors have been improved in significant part through study. Higher compressors are also used in conjunction with this to add more bits. The most popular compressor architectures are 4-2,7-2, 5-3, 10-4, 15-4, and 20-5.

4.2 VARIEOUS TYPES OF COMPRESSOR ADDERS

4:2 COMPRESSOR ADDER: As the name suggests, a 4-2 compressor reduces four inputs plus one carry bit from the preceding column to two outputs (Sum and Carry) and one intermediate carry bit (Cout), which is sent as Cin to the following column.

The mathematical link between the input and output of the 4-2 compressor is Cin+X4+X3+X2+X1 = Sum+2(Carry+Cout). Implementing a 4-2 compressor with basic cascading of complete adders results in a critical path delay of four XOR gates, as shown in Fig. This 4-2 compressor is regarded as a classic type because, as illustrated in Fig. 2, logical optimisation decreases the critical route time to three XOR gates.

The traditional 4-2 compressor's Boolean equations are as follows: Sum = Cin X4 X3 X2 X1 Carry = (X4 X3 X2 X1) Cin + (X4 X3 X2 X1) X4 Cout = (X2 X1) X3 + (X2 X1) X1

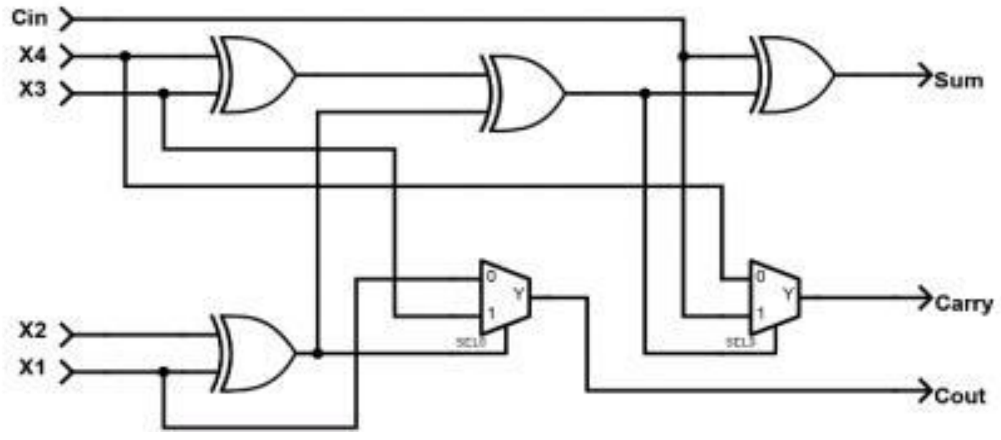


Fig 4.2.1 4:2 compressor adder

5:2 COMPRESSOR ADDER: There are two phases in a 5-2 compressor adder. The groups of two bits are added in the first stage using four half-adders. The carry bits are forwarded to the second stage, and the sum bits are merged to yield a 3-bit result. The carry bits from the first stage and the remaining bits of the input numbers are added together in the second stage using two full-adders. A 6-bit result that represents the sum of the two input numbers is the second stage's output. The 5-2 compressor adder is a straightforward and effective method for adding two 5-bit values. It is frequently utilised in digital circuits where efficiency and speed of operation are crucial factors.

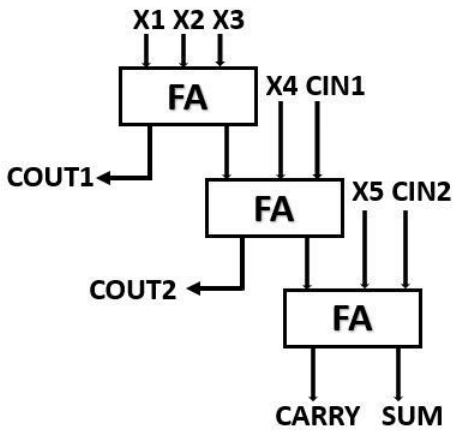
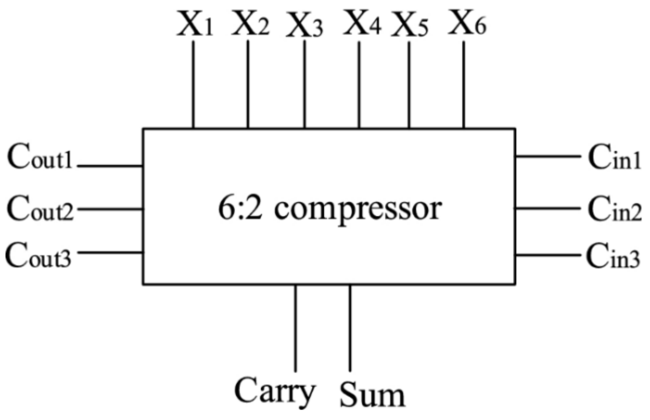


Fig 4.2.2 5:2 compressor

6:2 COMPRESSOR ADDER: The 6-2 compressor adder consists of three stages. In the first stage, six half-adders are used to add together the groups of two bits. Both a carry bit and a sum bit are produced by each half-adder. The carry bits are forwarded to the second stage, and the total bits are merged to yield a 5-bit result. The carry bits from the first stage and the remaining bits of the input numbers are added together in the second stage using three full-adders. A 5-bit result representing the sum of the two input numbers plus the carry bits from the first stage is the result of the second stage. The carry bit from the second stage and the last bit of the input numbers are added together in the third stage using a final full-adder. A 7-bit result that represents the sum of the two input numbers is the third stage's output.



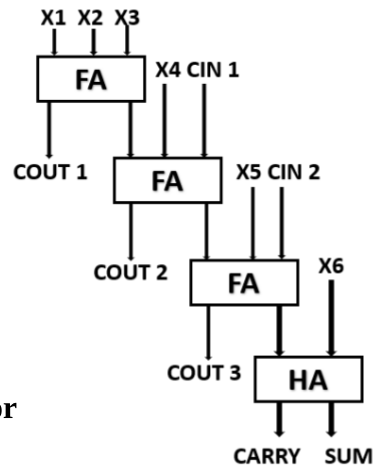
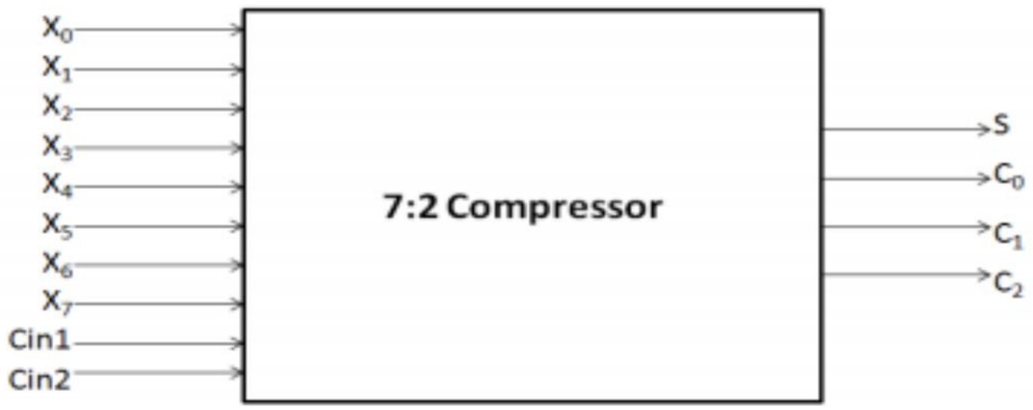


Fig 4.2.3 6:2 compressor

7:2 COMPRESSOR ADDER: The 7:2 compressor, as depicted in Figure 6, works similarly to its 4:2 counterpart in that it may add 7 bits of input data and 2 carries from earlier stages at once. In our approach, two 4:2 compressors, two full adders, and one half adder are combined to create a novel 7:2 compressor. Fig. 7 depicts the architecture for the same. As mentioned earlier, the adoption of the 4:2 compressor in this architecture would boost efficiency as contrasted to the conventional way of adding nine bits at a time using only full adders and half adders because the 4:2 compressor demonstrates a sizable speed gain of approximately 66.6%. This causes the processor's speed to improve greatly.



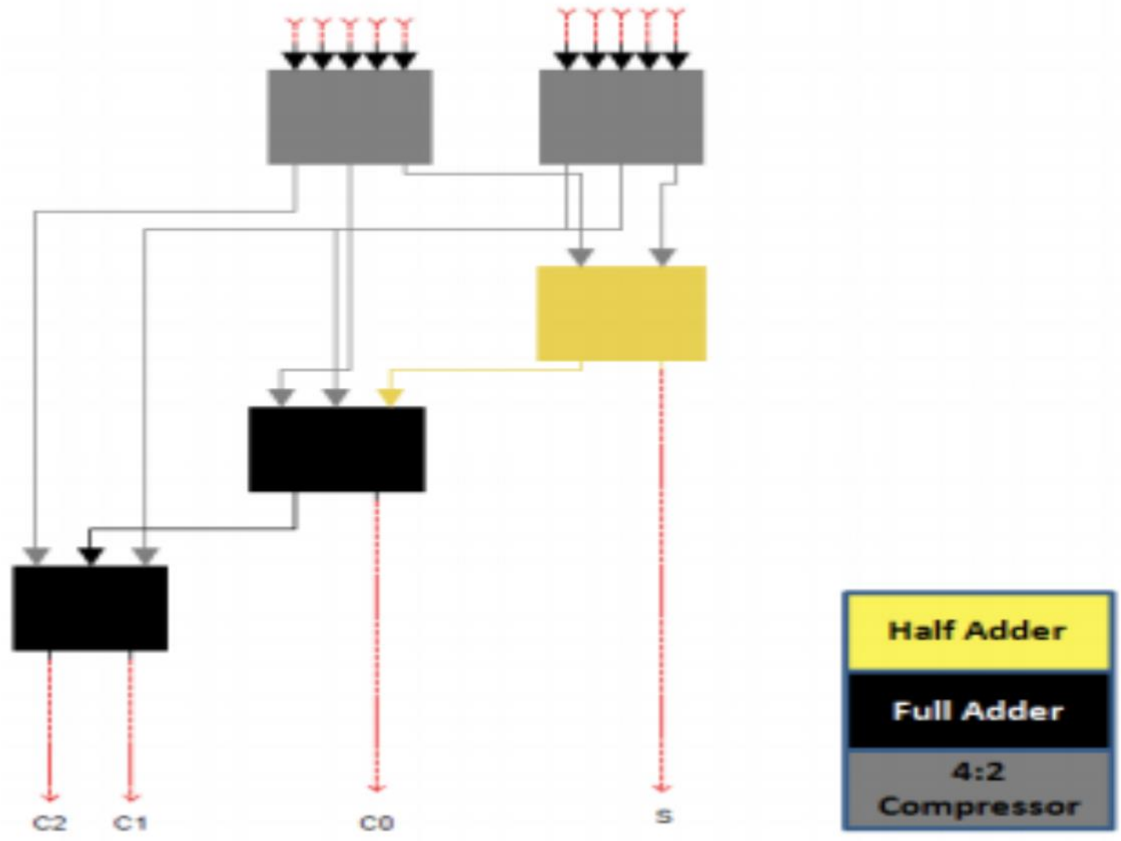


Fig 4.2.4 circuit diagram of 7:2 compressor adder

8:2 COMPRESSOR ADDER: A The 8-2 compressor adder consists of four stages. In the first stage, eight half-adders are used to add together the groups of two bits. Both a carry bit and a sum bit are produced by each half-adder. The carry bits are forwarded to the second stage, and the total bits are merged to yield a 7-bit result.The carry bits from the first stage and the remaining bits of the input numbers are added together in the second stage using four full-adders. The output of the second stage is a 6-bit result that represents the sum of the two input numbers plus the carry bits from the first stage.In the third stage, two full-adders are used to add together the carry bits from the second stage along with the remaining bits of the input numbers. The output of the third stage is a 7-bit result that represents the sum of the two input numbers plus the carry bits from the first and second stages.Finally, in the fourth stage, a final full-adder is used to add together the carry bit from the third stage along with the remaining bit of the input numbers. The output of the fourth stage is a 9-bit result that represents the sum of the two input numbers.The 8-2 compressor adder is a more complex circuit than the 5-2 and 6-2 compressor adders, but it is still an efficient way to perform addition of two 8-bit numbers. It is commonly used in digital circuits where low power consumption and fast operation are important considerations.

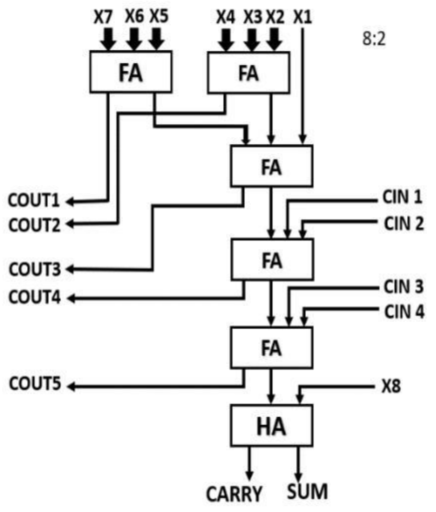


Fig 4.2.5 8:2 compressor adder

9:2 COMPRESSOR ADDER: The 9-2 compressor adder consists of five stages. In the first stage, 10 half-adders are used to add together the groups of two bits. Each half-adder produces a sum bit and a carry bit. The sum bits are combined to produce an 8-bit result, while the carry bits are passed on to the second stage.

In the second stage, five full adders are used to add the carry bits from the first stage and the remaining bits of the input numbers. The output of the second stage is a 7-bit result that represents the sum of the two input numbers plus the carry bits from the first stage. In the third stage, threefull adderss are used to add the carry bits from the second stage and the remaining bits of the input numbers.

The output of the third stage is an 8-bit result that represents the sum of the two input numbers plus the carry bits from the first and second stages. In the fourth stage, twofull adderss are used to add the carry bits from the third stage and the remaining bits of the input numbers.

The output of the fourth stage is a 9-bit result that represents the sum of the two input numbers plus the carry bits from the first, second, and third stages. Finally, in the fifth stage, a final full-adder is used to add together the carry bit from the fourth stage along with the remaining bit of the input numbers. The output of the fifth stage is a 10-bit result that represents the sum of the two input number.

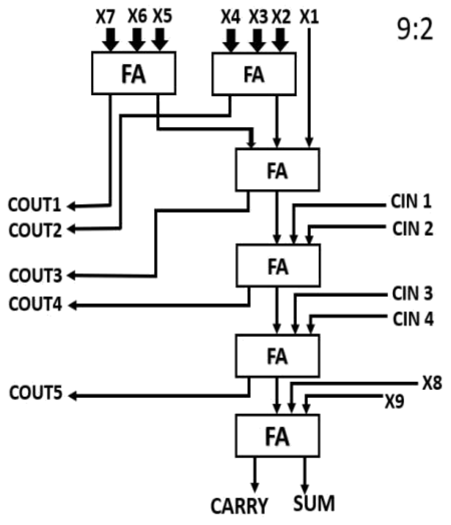


Fig 4.2.6 9:2 compressor adder

10:2 COMPRESSOR ADDER: The 10-2 compressor adder consists of six stages. In the first stage, 11 half-adders are used to add together the groups of two bits. Both a carry bit and a sum bit are produced by each half-adder. The carry bits are forwarded to the second stage, and the sum bits are merged to yield a 9-bit result. The carry bits from the first stage and the remaining bits of the input numbers are added in the second stage using five complete adders. An 8-bit result representing the sum of the two input integers plus the carry bits from the first stage is the outcome of the second stage. The carry bits from the second standits of the input numbers are added together in the third stage using three full-adders. A 9-bit result representing the sum of the two input numbers plus the carry bits from the first and second stages is the third stage's output. The carry bits from the third stand it sits of the input numbers are added together in the fourth stage using two full-adders. The output of the fourth stage is a 10-bit result that represents the sum of the two input numbers plus the carry bits from the first, second, and third stages. In the fifth stage, a full-adder is used to add together the carry bit from the fourth stage along with the remaining bit of the input numbers. The output of the fifth stage is an 11-bit result that represents the sum of the two input numbers plus the carry bits from the first, second, third, and fourth stages. Finally, in the sixth stage, another full-adder is used to add together any remaining carry bits from the previous stages. The output of the sixth stage is the final 11-bit result.

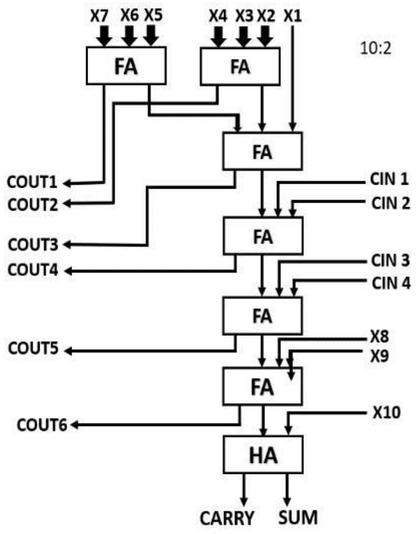


Fig 4.2.7 10:2 compressor adder

11:2 COMPRESSOR ADDER: The 11-2 compressor adder consists of seven stages. In the first stage, 12 half-adders are used to add together the groups of two bits. Both a carry bit and a sum bit are produced by each half-adder. The carry bits are sent to the second stage, and the total bits are merged to yield a 10-bit output.The carry bits from the first stage and the remaining bits of the input numbers are added in the second stage using five full-adders. A 9-bit result representing the sum of the two input numbers plus the carry bits from the first stage is the result of the second stage.The carry bits from the second stage are combined with the remaining bits of the input numbers in the third stage using three full-adders. The output of the third stage is a 10-bit result that represents the sum of the two input numbers plus the carry bits from the first and second stages. In the fourth stage, two full-adders are used to add together the carry bits from the third stage along with the remaining bits of the input numbers. The output of the fourth stage is an 11-bit result that represents the sum of the two input numbers plus the carry bits from the first, second, and third stages.In the fifth stage, another full-adder is used to add together the carry bit from the fourth stage along with the remaining bit of the input numbers. The output of the fifth stage is a 12-bit result that represents the sum of the two input numbers plus the carry bits from the first, second, third, and fourth stages.The sixth and seventh stages are used to propagate the carry bits and ensure that the final result is correctly calculated. In the sixth stage, a carry-lookahead adder is used to propagate the carry bits from the previous stages. In the seventh stage, an additional full-adder is used to combine the carry bits with the final result and produce the correct 12-bit output.

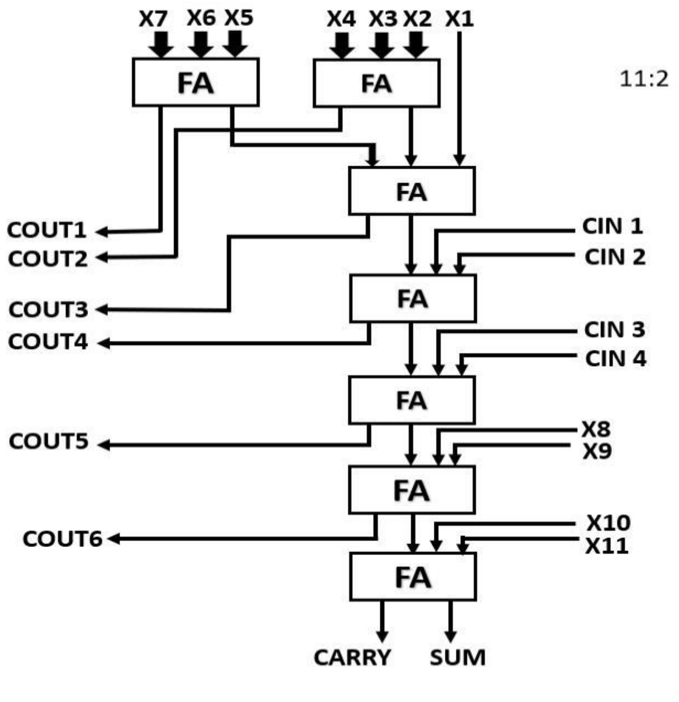


Fig 4.2.8 circuit diagram of 11:2 compressor adder

CHAPTER 5

ADDER UNIT

5.1 INTRODUCTION:

A digital circuit known as an adder is used to add two binary digits. One of the most basic arithmetic circuits used in digital systems is this one. Most digital systems, such as microprocessors, digital signal processors, and other digital circuits, use adders.

An adder generates a sum output from two binary inputs of 1s and 0s each. The fundamental working of an adder is that it adds the two inputs bit by bit, beginning with the least significant bit (LSB), propagating the carry to the next significant bit (NSB), and so forth, until it reaches the most significant bit (MSB).

There are various varieties of adders, including full, ripple, carry-lookahead, carry-select, and half-adders. Depending on the needs for speed, area, power, and complexity, several types of adders are employed in different applications. Each form of adder has advantages and disadvantages of its own.

In digital systems, adders are a fundamental building component for arithmetic operations and are important for carrying out numerous arithmetic and logical processes. They are also utilised in numerous other fields of digital electronics, including digital signal processing, computer graphics, and cryptography.

5.2 CARRY SELECT ADDER :

To add two binary numbers, a carry-select adder (CSLA), a form of digital adder, is employed. Because it can add several bits simultaneously, decreasing the delay time and accelerating adding speed, it is a faster and more effective adder than a ripple-carry adder (RCA).

A two-stage procedure is used by the carry-select adder to perform addition. Two distinct carry propagation chains—one for the high-order bits and one for the low-order bits—are established in the first stage. Every chain is made to determine the carry-out at every bit point under the assumption that the carry-in is 0.

Based on the value of the carry-in, a multiplexer is utilised in the second stage to choose the carry-out from the appropriate chain. The carry-out is chosen from the low-order chain if the carry-in is 0. The carry-out from the high-order chain is chosen if the carry-in is 1.

The two binary numbers' sum and a carry-out are the output. Any two equal-length binary numbers can be added together using the carry-select adder.

The carry-select adder has the advantage over the ripple-carry adder in that it allows several bits to be added in parallel, which decreases the delay time and speeds up addition. In contrast to the ripple-carry adder, it is more difficult and expensive to implement.

A N-bit carry choose adder is designed to prevent the carry from spreading from bit to bit sequentially. We might choose between the outputs of the two parallel adders using the real carry input created if we had two adders operating in parallel, one with a carry input of 0 and the other with a carry input of 1. This implies that all adders may be running calculations concurrently. Since it is rather inefficient to employ two adders for each result bit, we may configure the N-bit adder to use two concurrent 2*N/M-1 M-bit ripple carry adders. There is no need for parallel addition in this scenario because the adder for the least significant bits will always have a carry input of 0.

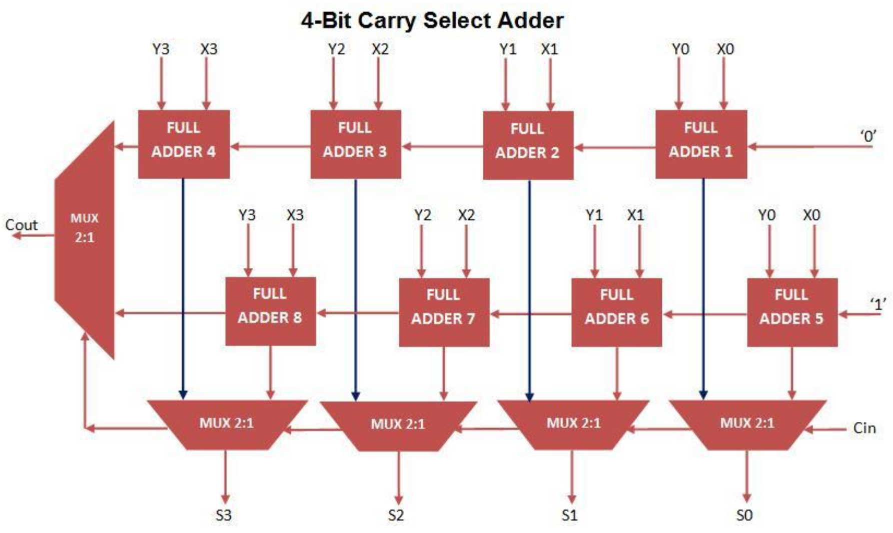


Fig 5.2.1 internal diagram of carry select adder

CHAPTER 6

RESULTS

6.1 Simulation Results of 8 bit multiplier:

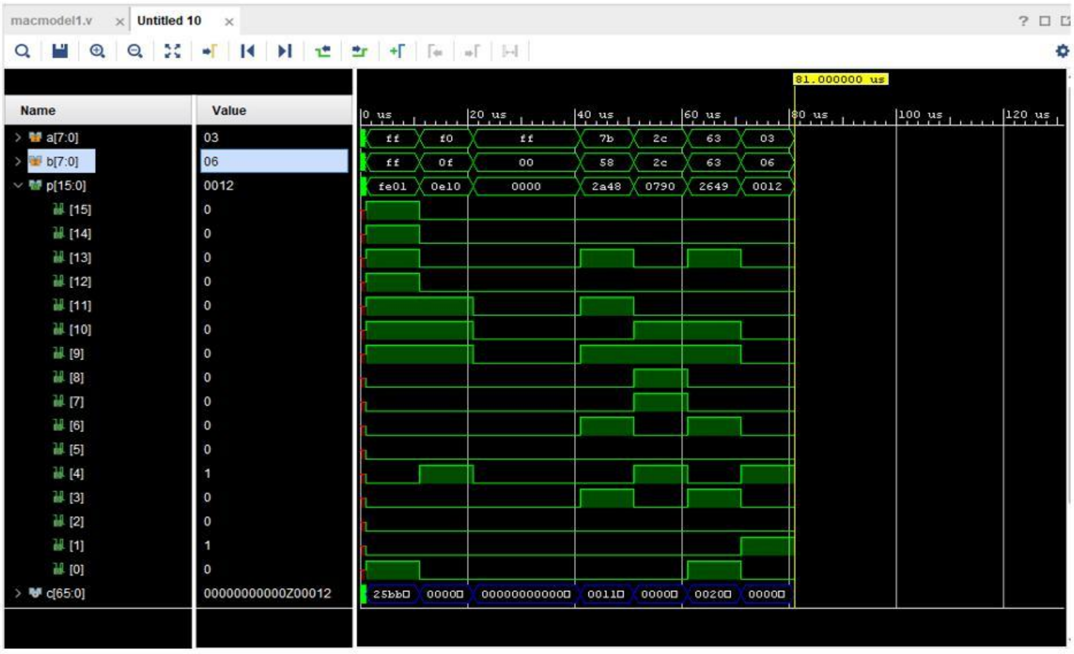


Fig 6.1.1 : Simulation waveform of 8-bit multiplication

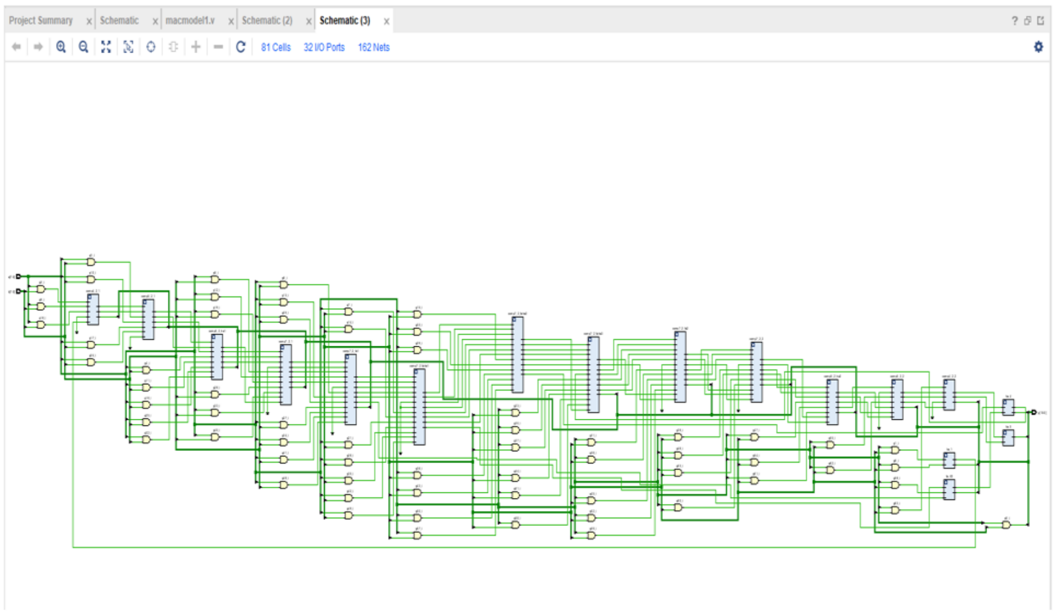


Fig 6.1.2: RTL Diagram of 8-bit multiplication

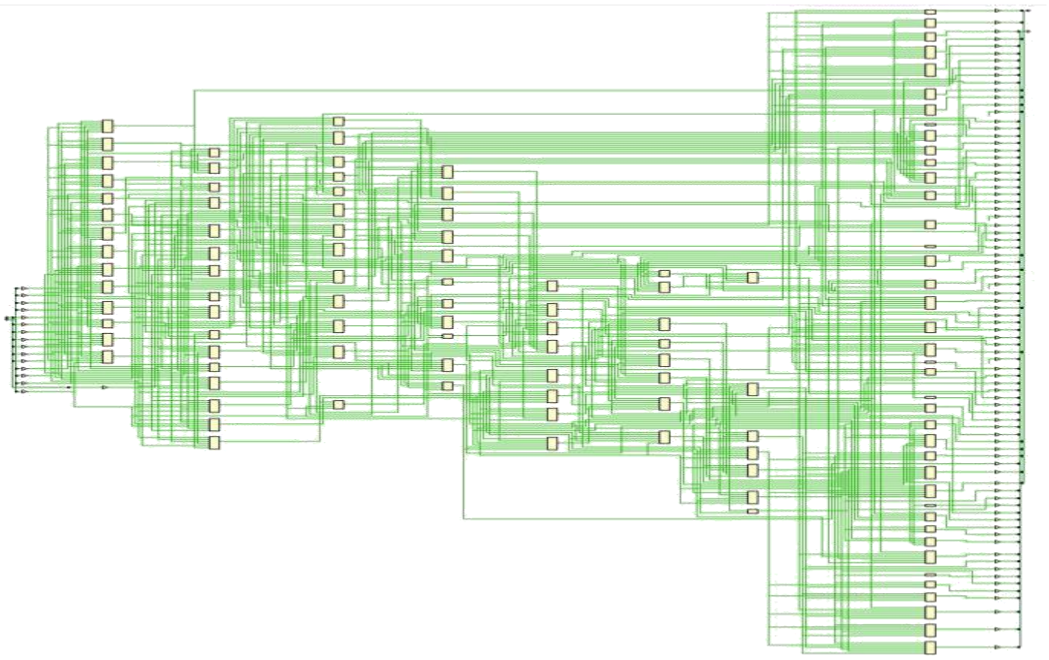
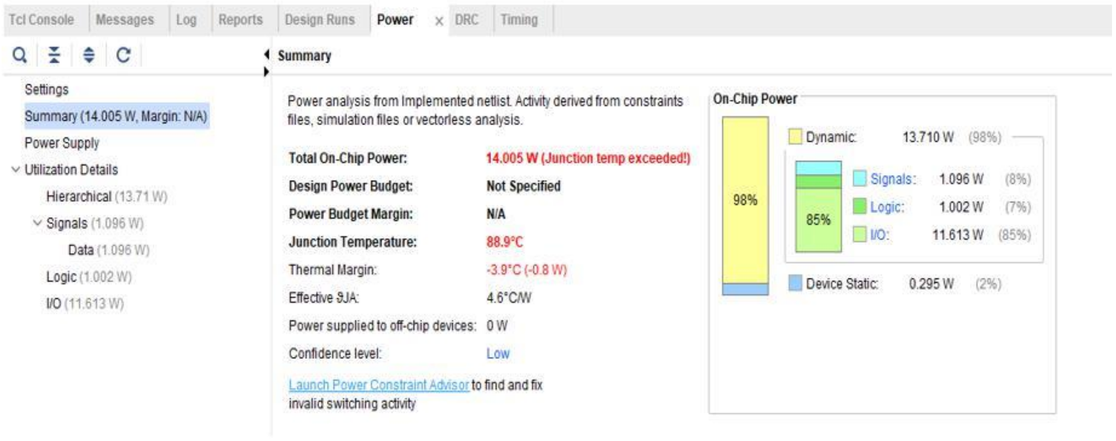


Fig.6.1.3: Synthesized schematic of 8-bit multiplication



Tcl Console | Messages | Log | Reports | Design Runs | Power | DRC | Timing

Settings
Summary (14.005 W, Margin: N/A)
Power Supply
Utilization Details
Hierarchical (13.71 W)
Signals (1.096 W)
Data (1.096 W)
Logic (1.002 W)
I/O (11.613 W)

Summary

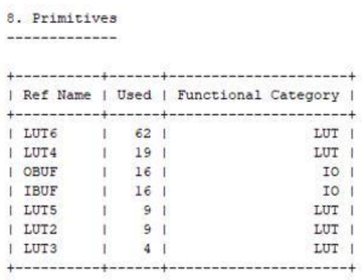
Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	14.005 W (Junction temp exceeded!)
Design Power Budget:	Not Specified
Power Budget Margin:	N/A
Junction Temperature:	88.9°C
Thermal Margin:	-3.9°C (-0.8 W)
Effective ϑJA:	4.6°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power

🟨	Dynamic:	13.710 W	(98%)
🟦	Signals:	1.096 W	(8%)
🟩	Logic:	1.002 W	(7%)
🟩	I/O:	11.613 W	(85%)
🟦	Device Static:	0.295 W	(2%)



```
8. Primitives
-------------

+----------+------+--------------------+
| Ref Name | Used | Functional Category |
+----------+------+--------------------+
| LUT6     |   62 |                 LUT |
| LUT4     |   19 |                 LUT |
| OBUF     |   16 |                  IO |
| IBUF     |   16 |                  IO |
| LUT5     |    9 |                 LUT |
| LUT2     |    9 |                 LUT |
| LUT3     |    4 |                 LUT |
+----------+------+--------------------+
```

Fig.6.1.4: Power report and LUTs utilized for 8-bit multiplication

The proposed model is simulated and synthesized using Xilinx Vivado and implemented on Artix-7 FPGA using The simulation results and RTL schematic of the proposed model is shown in figure 11 and figure 12 respectively . The designs are simulated and verified its functionality by providing different set of inputs and the corresponding output is verified. For example the first inputs are ff x ff which are in hexadecimal produced output as fe01 in hexadecimal which implies 255 x 255 = 65025.Register Transfer

Level (RTL) is an abstraction for defining the digital portions of a design.The figure 12represents the RTL diagram of 8-bit multiplier designed using proposed method. From the figure 13 and figure 14 it is portrayed that synthesized schematic and power report of the proposed model.

6.1.1 Comparison Environment:

In this section, various comparisons have included in Table1 and Table2. Table1 projects the blocks required by the proposed 8-bit vedic multiplier

Table2: Manual Resource utilization of 8-bit proposed Multiplier

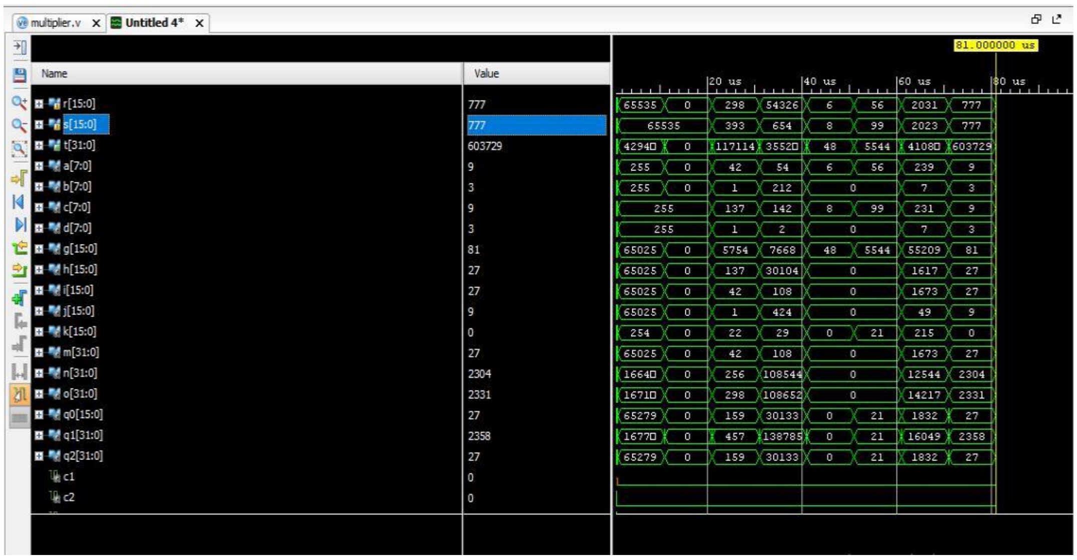
Multiplier	**Area (LUTs)**	**Delay (ns)**
Proposed Multiplier	95	12.735
Conventional 8-Bit Vedic Multiplier [23]	167	27.650
8-Bit Vedic Multiplier Using Compressors [24]	153	13.480

Table3: Comparison among various FPGA families

FPGA Family	**No. of Slices**	**Delay(ns)**	**Power(W)**
Artix-7	50	12.735	0.082
Automotive Artix-7	50	10.216	0.082
Spartan-6	40	18.639	**0.081**
Vertex-6	35	9.141	4.447
Vertex-6 lower power	**33**	**9.404**	1.412

Percentage improvement: From the results it is portrayed that implementation of multiplier using Virtex-6 (lower power) provides optimized results in terms of area, delay and power compared to other FPGA families. By using Vertex-6 (lower power) FPGA we can observe that there is 26.15% improvement in delay and 34% reduction in area compared to Artix 7, 17.5% reduction in area and 49.54% improvement in delay compared to Spartan 6. As far as power concern, Spartan 6 and Artix 7 provide better performance compared to Virtex 6 family.

6.2 Simulation Results of 16-bit multiplier:



Name	Value
r[15:0]	777
s[15:0]	777
t[31:0]	603729
a[7:0]	9
b[7:0]	3
c[7:0]	9
d[7:0]	3
g[15:0]	81
h[15:0]	27
i[15:0]	27
j[15:0]	9
k[15:0]	0
m[31:0]	27
n[31:0]	2304
o[31:0]	2331
q0[15:0]	27
q1[31:0]	2358
q2[31:0]	27
c1	0
c2	0

Fig.6.2.1: Simulation waveform of 16-bit multiplication

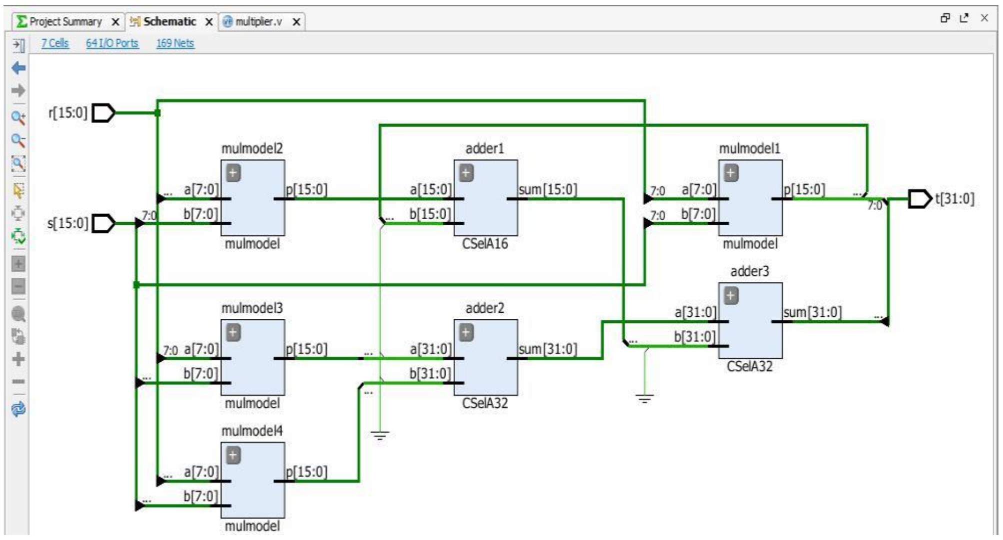


Fig.6.2.2: RTL Diagram of 16-bit multiplication

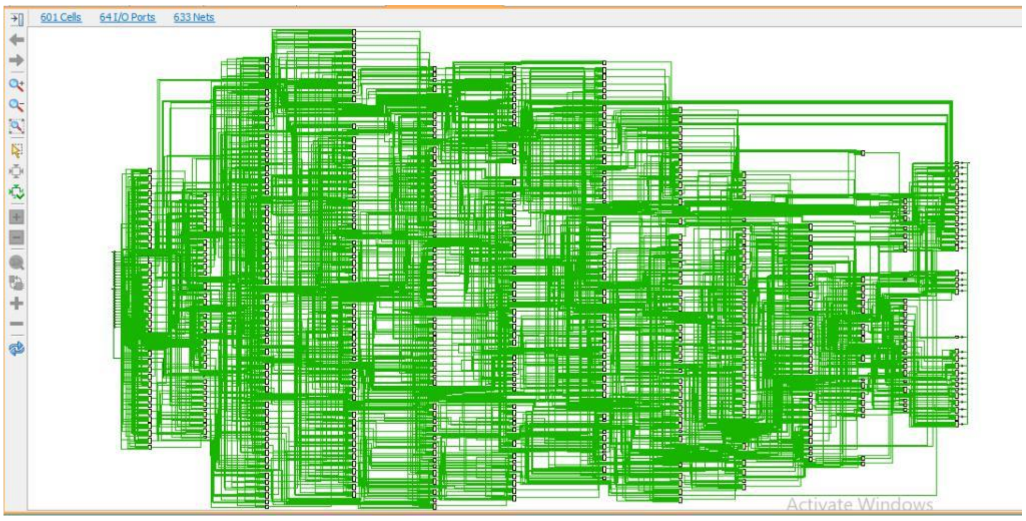
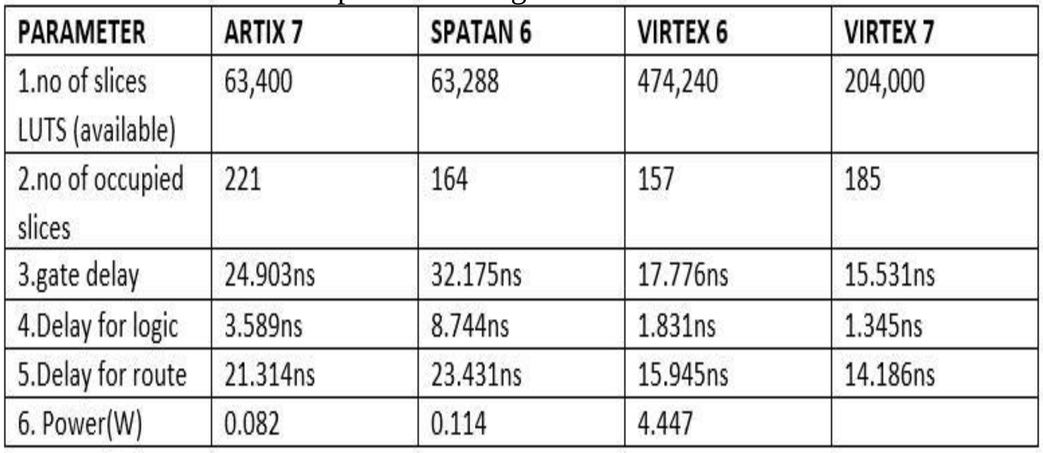


Fig6.2.3: Synthesized schematic of 8-bit multiplication

The designs are simulated and verified its functionality by providing different set of inputs and the corresponding output is verified. For example the first inputs are ffff x ffff which are in hexadecimal produced output as fe01 in hexadecimal which implies 65535 x 65535 = 4294836225. Register Transfer Level (RTL) is an abstraction for defining the digital portions of a design. The figure 13 represents the RTL diagram of 16-bit multiplier designed using proposed method.

6.2.1 Comparison observation:

Table4 : Comparison among various FPGA families



PARAMETER	ARTIX 7	SPATAN 6	VIRTEX 6	VIRTEX 7
1.no of slices LUTS (available)	63,400	63,288	474,240	204,000
2.no of occupied slices	221	164	157	185
3.gate delay	24.903ns	32.175ns	17.776ns	15.531ns
4.Delay for logic	3.589ns	8.744ns	1.831ns	1.345ns
5.Delay for route	21.314ns	23.431ns	15.945ns	14.186ns
6. Power(W)	0.082	0.114	4.447	

From the results it is portrayed that implementation of multiplier using Artix 7 provides optimized results in terms of power compared to other FPGA families. By using Vertex-6 FPGA we can observe that the number of occupied slices are less when compared to the other FPGA families.Virtex 7 provides optimized results in terms of gate delay,delay for logic and delay for route.

Table5 : Manual Resource utilization of 16-bit

Multiplier	**Area (no of occupied slices)**	**Delay (ns)**
Reference Paper (25)	415	25.044
Reference Paper (26)	-	36.71
Reference Paper (27)	-	32
Reference Paper (28)	436	-
Proposed Design	221	24.90

From the above table, we observed that the area of the multiplier is reduced nearly 49 % when compared to the reference paper (25 and 26).The delay of the proposed multiplier is reduced nearly 32% when compared to the reference paper (26). So, it is observed that the proposed multiplier is efficient in terms of delay and area when compared to other existing multipliers.

6.3 Simulation Results of 8 bit MAC:

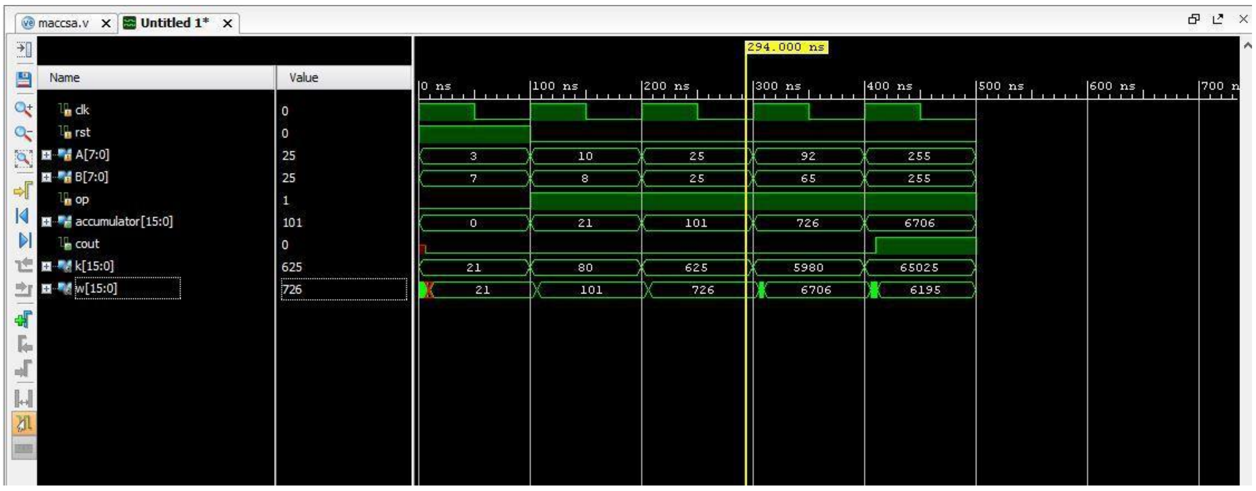


Fig 6.3.1: Simulation waveform of 8-bit MAC

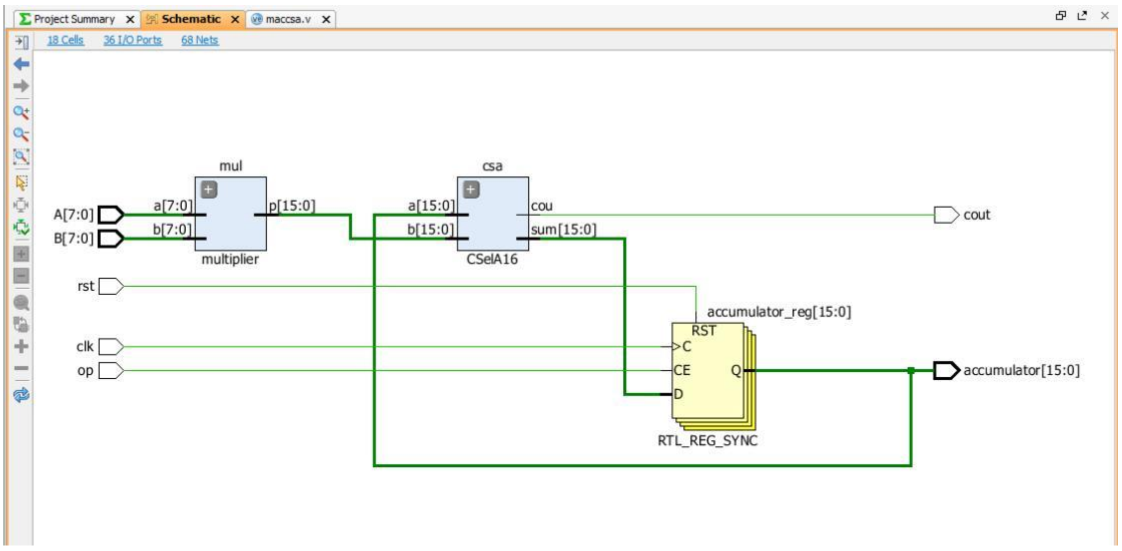


Fig 6.3.2: RTL diagram of 8-bit MAC

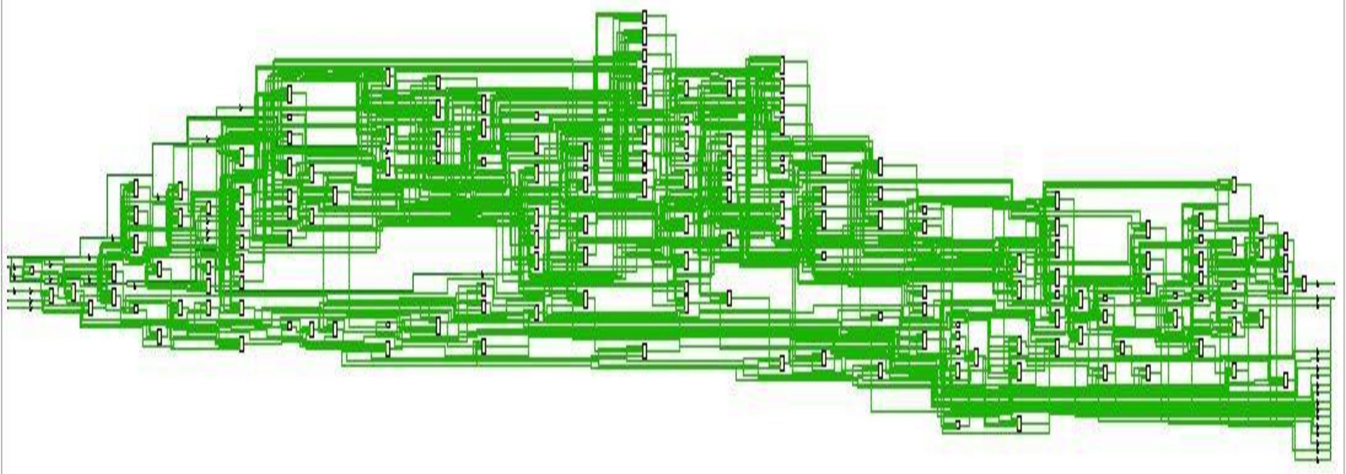
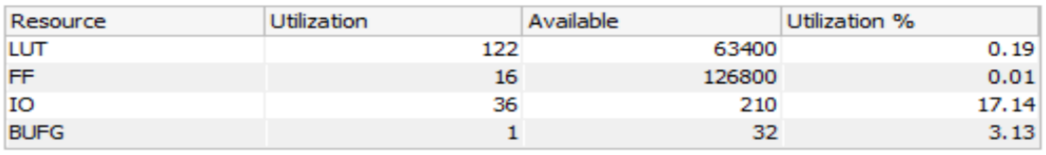
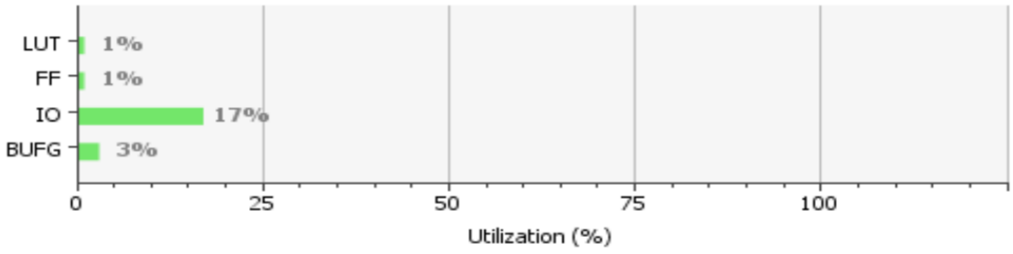
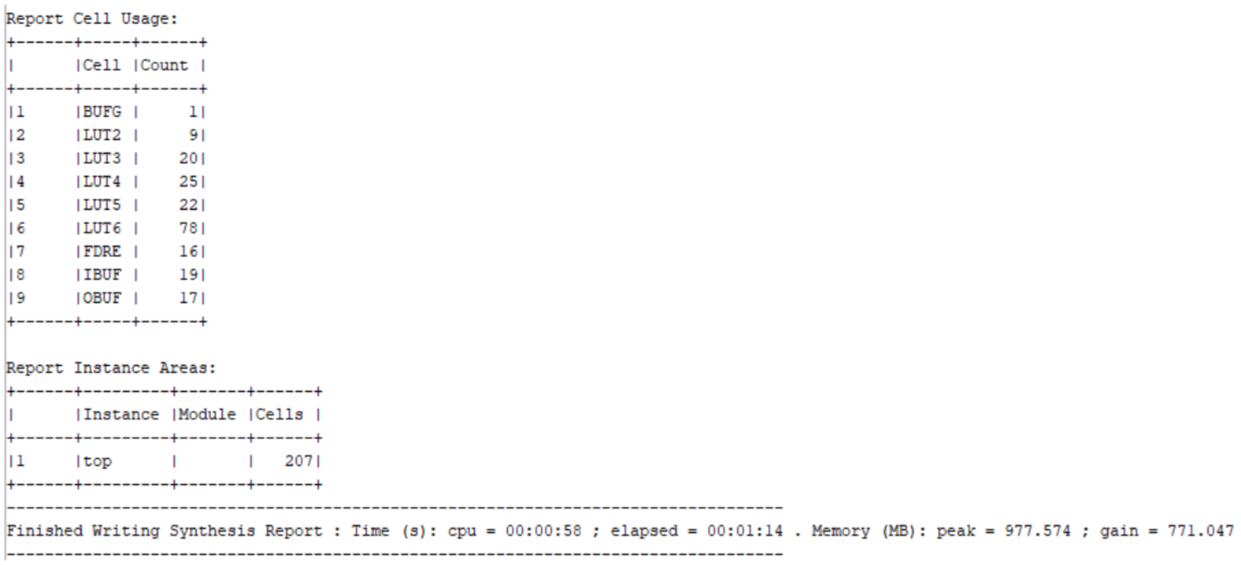


Fig 6.3.3: Synthesized schematic of 8-bit MAC



Resource	Utilization	Available	Utilization %
LUT	122	63400	0.19
FF	16	126800	0.01
IO	36	210	17.14
BUFG	1	32	3.13

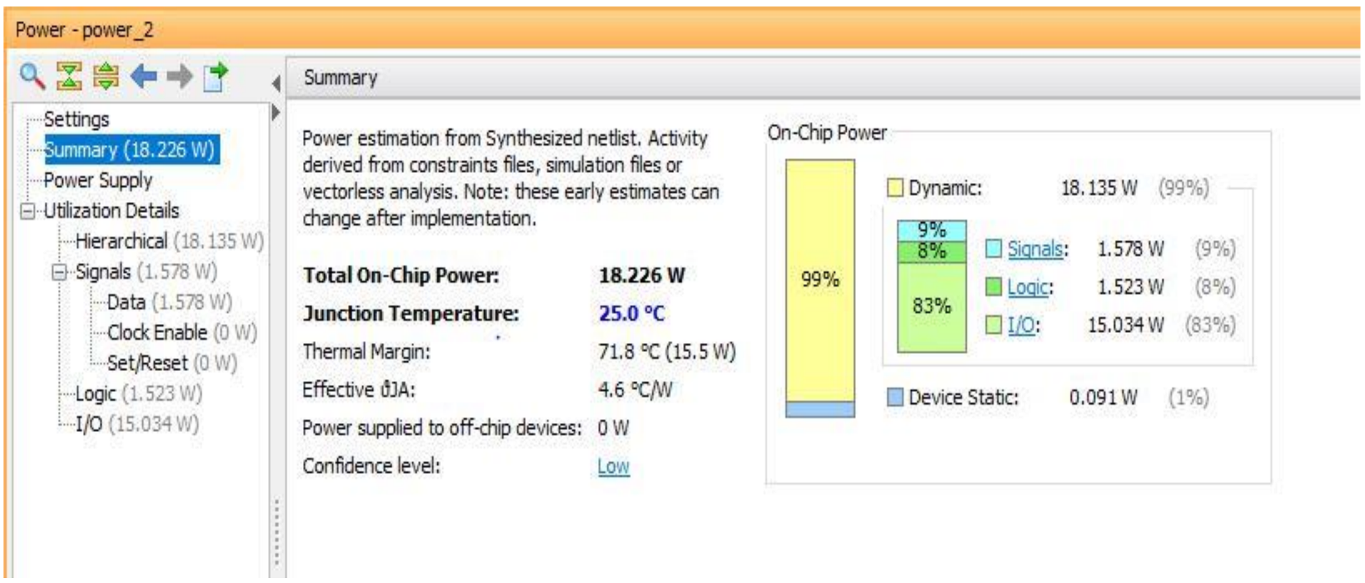




```
Report Cell Usage:
+------+-----+------+
|      |Cell |Count |
+------+-----+------+
|1     |BUFG |     1|
|2     |LUT2 |     9|
|3     |LUT3 |    20|
|4     |LUT4 |    25|
|5     |LUT5 |    22|
|6     |LUT6 |    78|
|7     |FDRE |    16|
|8     |IBUF |    19|
|9     |OBUF |    17|
+------+-----+------+

Report Instance Areas:
+------+---------+-------+------+
|      |Instance |Module |Cells |
+------+---------+-------+------+
|1     |top      |       |   207|
+------+---------+-------+------+
---------------------------------------------------------------------------------
Finished Writing Synthesis Report : Time (s): cpu = 00:00:58 ; elapsed = 00:01:14 . Memory (MB): peak = 977.574 ; gain = 771.047
---------------------------------------------------------------------------------
```

Fig 6.3.4: LUTs utilized for 8-bit MAC



Power - power_2

Settings
Summary (18.226 W)
Power Supply
Utilization Details
Hierarchical (18.135 W)
Signals (1.578 W)
Data (1.578 W)
Clock Enable (0 W)
Set/Reset (0 W)
Logic (1.523 W)
I/O (15.034 W)

Summary

Power estimation from Synthesized netlist. Activity derived from constraints files, simulation files or vectorless analysis. Note: these early estimates can change after implementation.

Total On-Chip Power:	**18.226 W**
Junction Temperature:	**25.0 °C**
Thermal Margin:	71.8 °C (15.5 W)
Effective ϑJA:	4.6 °C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

On-Chip Power

Dynamic:	18.135 W	(99%)
Signals:	1.578 W	(9%)
Logic:	1.523 W	(8%)
I/O:	15.034 W	(83%)
Device Static:	0.091 W	(1%)

Fig 6.3.5 : Power report for 8-bit MAC

The performance of an 8-bit MAC operation depends on the speed of the multiplier and adder circuits, as well as the accuracy of the result. Different techniques can be used to optimize the performance and accuracy of the MAC operation, such as using parallel multipliers and adders, implementing high-speed algorithms for multiplication and accumulation, and using precision arithmetic techniques to reduce rounding errors.

The result of an 8-bit MAC operation is a single 8-bit number that represents the accumulated product of the two input numbers. The result can be represented in binary, decimal, or hexadecimal format, depending on the requirements of the application.

In conclusion, an 8-bit MAC operation involves multiplying two 8-bit numbers and accumulating the result with a previously accumulated value. The result is an 8-bit number that represents the accumulated product of the two input numbers. The performance and accuracy of the operation depend on the speed of the multiplier and adder circuits, as well as the techniques used to optimize the calculation. The result can be represented in binary, decimal, or hexadecimal format, and may require saturation and/or rounding-off depending on the application requirements. Overall, the 8-bit MAC operation is a fundamental building block of digital signal processing and other numerical applications, and is widely used in various embedded systems and microcontroller-based applications.

6.4 Simulation Results of 16-bit MAC:

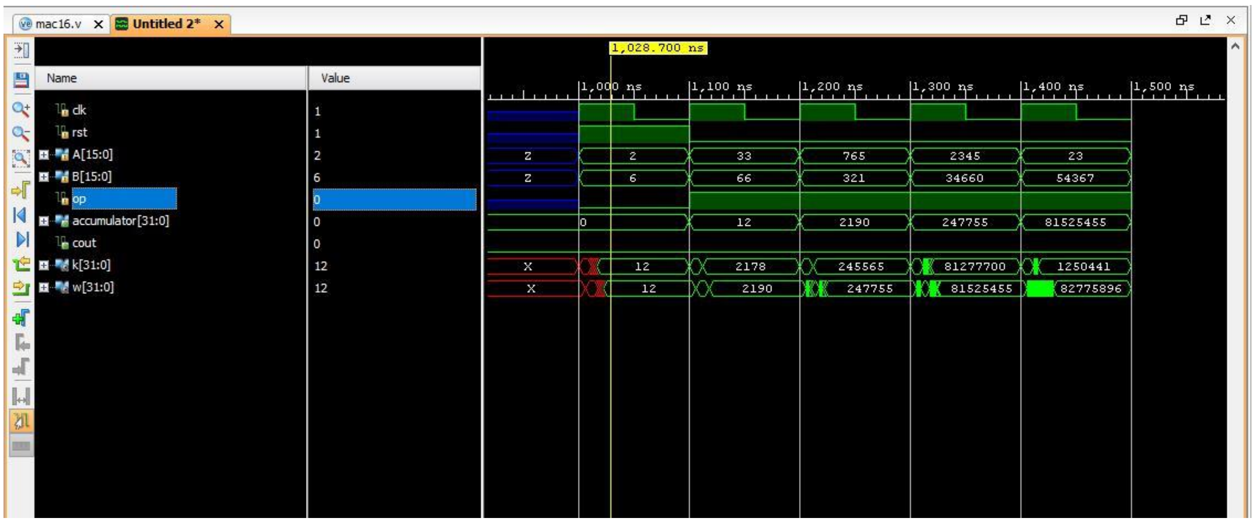


Fig 6.4.1 : Simulation waveform of 16-bit MAC

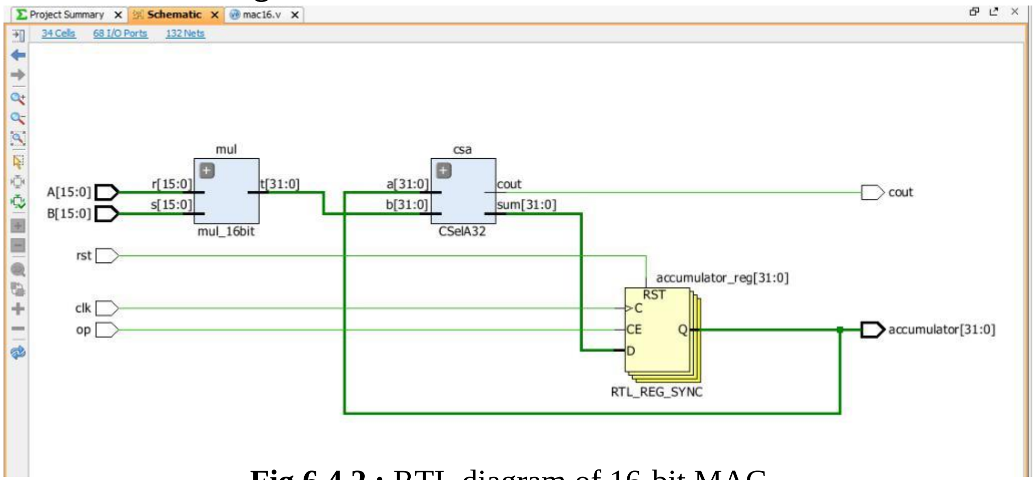


Fig 6.4.2 : RTL diagram of 16-bit MAC

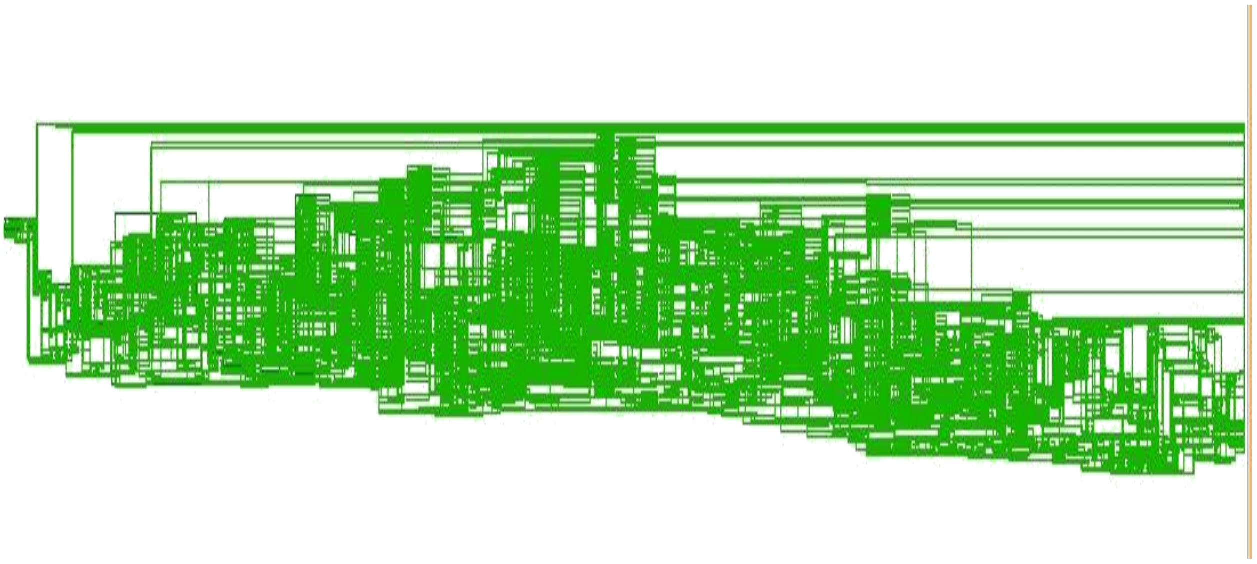
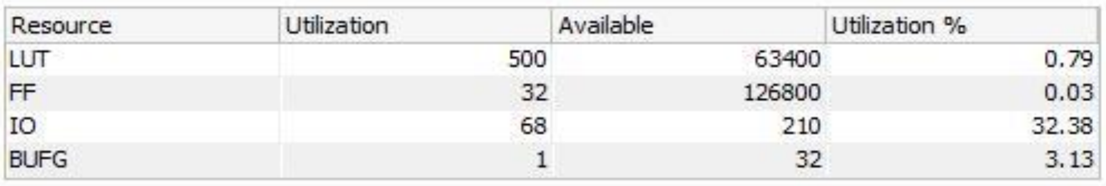


Fig 6.4.3: Synthesized schematic of 16-bit MAC



Resource	Utilization	Available	Utilization %
LUT	500	63400	0.79
FF	32	126800	0.03
IO	68	210	32.38
BUFG	1	32	3.13

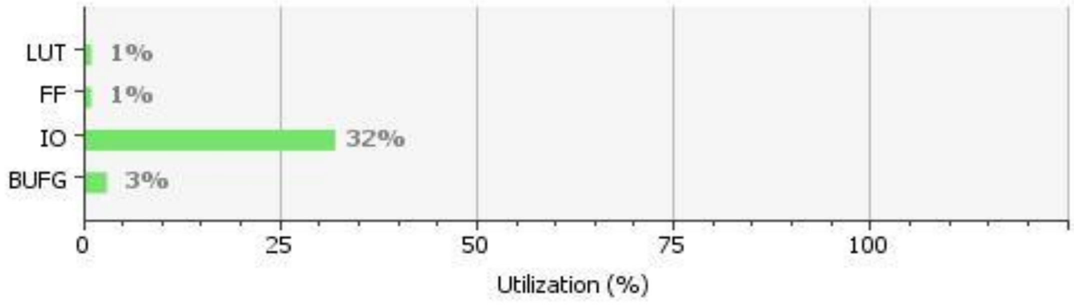
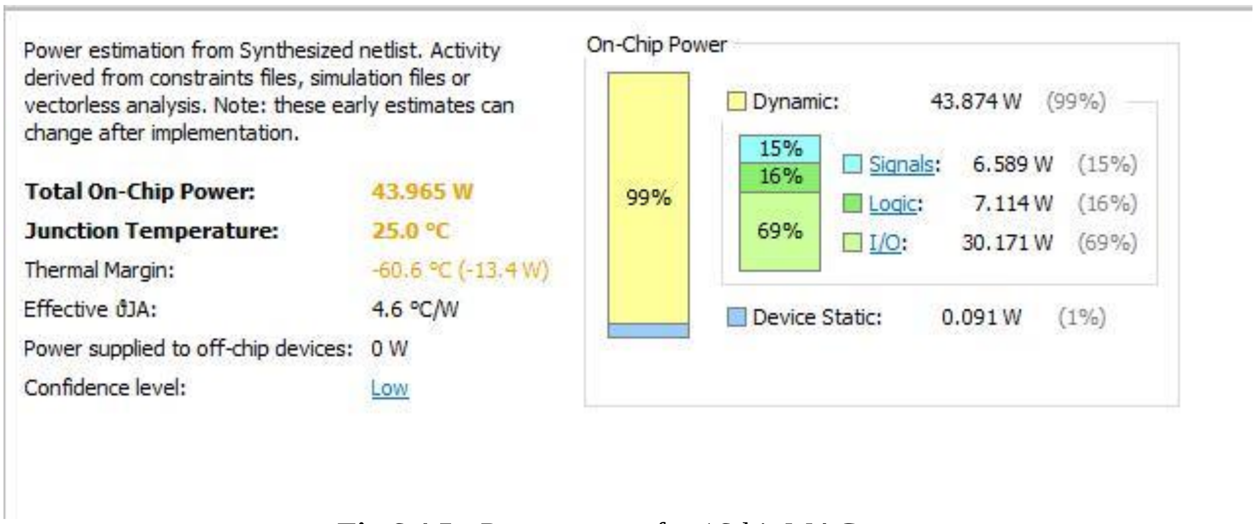


Fig 6.4.4 : LUTs utilized for 16-bit MAC



Power estimation from Synthesized netlist. Activity derived from constraints files, simulation files or vectorless analysis. Note: these early estimates can change after implementation.

Total On-Chip Power:	43.965 W
Junction Temperature:	25.0 °C
Thermal Margin:	-60.6 °C (-13.4 W)
Effective ϑJA:	4.6 °C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

On-Chip Power

Dynamic:	43.874 W	(99%)
Signals:	6.589 W	(15%)
Logic:	7.114 W	(16%)
I/O:	30.171 W	(69%)
Device Static:	0.091 W	(1%)

Fig 6.4.5 : Power report for 16-bit MAC

A 16-bit MAC (Multiply-Accumulate) operation involves multiplying two 16-bit numbers and accumulating the result with a previously accumulated value. This operation is commonly used in digital signal processing and other numerical applications that require higher precision than an 8-bit MAC operation.

The performance of a 16-bit MAC operation depends on the speed of the multiplier and adder circuits, as well as the accuracy of the result. Different techniques can be used to optimize the performance and accuracy of the MAC operation, such as using parallel multipliers and adders, implementing high-speed algorithms for multiplication and accumulation, and using precision arithmetic techniques to reduce rounding errors.

The result of a 16-bit MAC operation is a single 16-bit number that represents the accumulated product of the two input numbers. The result can be represented in binary, decimal, or hexadecimal format, depending on the requirements of the application.

In summary, a 16-bit MAC operation is a more precise version of the 8-bit MAC operation, involving the multiplication of two 16-bit numbers and accumulation of the result with a previously accumulated value. The result is a 16-bit number that represents the accumulated product of the two input numbers. The performance and accuracy of the operation depend on the speed of the multiplier and adder circuits, as well as the techniques used to optimize the calculation. The result can be represented in binary, decimal, or hexadecimal format and may require saturation and/or rounding-off depending on the application requirements.

In conclusion, a 16-bit MAC (Multiply-Accumulate) operation is a fundamental building block of digital signal processing and other numerical applications that require higher precision than an 8-bit MAC operation. It involves multiplying two 16-bit numbers and accumulating the result with a previously accumulated value. The operation can be performed using a binary multiplier circuit and an adder circuit, and may require saturation and/or rounding-off to ensure the accuracy of the result. The performance of the operation depends on the speed of the circuits and the techniques used to optimize the calculation. The result can be represented in binary, decimal, or hexadecimal format, and is commonly used in various embedded systems and microcontroller-based applications.

CHAPTER 7
INTRODUCTION TO VERILOG

7.1 DEFINITION:

A flip-flop, microprocessor, network switch, or other digital system can be described using Verilog, a HARDWARE DESCRIPTION LANGUAGE (HDL). Verilog was developed to simplify the process and increase the HDL's durability and adaptability. By enabling engineers to specify the functionality of desired hardware and letting automation tools translate that behaviour into actual hardware pieces like combinational gates and sequential logic, Verilog was developed to speed up the design process. The semiconductor industry today uses and practises Verilog more than any other HDL. Verilog has applications, much as every other hardware description language. It gives designers the option of creating designs top-down or bottom-up.

Bottom-Up Design:

Electronic design is traditionally done from the bottom up. Each design is carried out utilising the conventional gates at the gate level. With the help of this design, new structural, hierarchical design approaches can be created.

Top-Down Design:

Numerous other advantages include early testing, simple switching between different technologies, and structured system architecture.

7.2 HISTORY OF VERILOG:

The creation of a hardware description language and a logic simulator called Verilog-XL by a company by the name of Gateway Design Automation in the 1980s might be credited as the beginning of Verilog HDL. Cadence Design Systems bought Gateway in 1989, along with the licence for the language and simulator. The language was placed in the public domain by Cadence in 1990 with the intention of it becoming a widely used, non-proprietary language. Since Open Verilog International (OVI) and VHDL International merged, Accellera, a nonprofit company, has taken over maintenance of the Verilog HDL. The language had to go through the IEEE standardisation process, which was the responsibility of OVI. Verilog HDL became IEEE Std. 1364-1995 in December 1995. IEEE Std. 1364-2001 is a significantly improved version that was released in 2001.

A subsequent revision was made in 2005, however it only made a few small adjustments.

Additionally, Accellera has created a brand-new Verilog extension standard called System Verilog.

In 2005, System Verilog was added to the IEEE standard (1800-2005).

7.3 USES OF VERILOG:

Verilog's level of abstraction makes it difficult to understand the nuances of how it uses technology. To produce a positive-edge triggered FF, for example, a D flip-flop design would need to understand how the transistors should be arranged as well as the rise, fall, and CLK-Q times necessary to latch the value onto the flop. Controlling power consumption, timing, and the ability to drive nets and other flops would also require a deeper understanding of the physical characteristics of transistors. We can focus on the behaviours with Verilog and take care of the rest later.

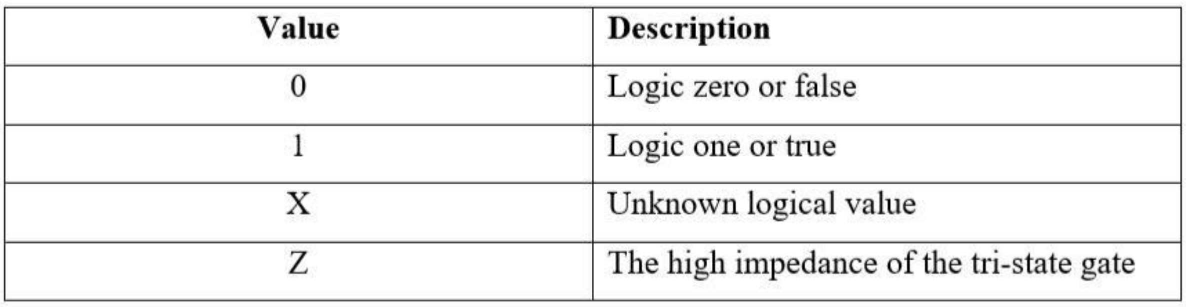
7.4 FEATURES OF VERILOG:

- Verilog takes case into account.
- Lowercase letters are used to specify keywords in Verilog.
- Verilog borrows most of its grammar from the "C" language.
- Verilog can be used to simulate a digital circuit at the algorithm, RTL, gate, and switch levels.
- Verilog lacks the concept of a package but nevertheless supports modern simulation tools like TEXTIO, PLI, and UDPs.

7.5 DATA TYPES:

Several new data types are introduced by Verilog. RTL descriptions are simpler to write and comprehend because to these data types. Verilog Hardware Description Language (HDL) data types are used to represent the data storage and transport components found in digital hardware.Data types in Verilog are separated into NETS and Registers. These data types represent various hardware architectures and differ in how values are assigned to and stored in them.

There are four fundamental values in the Verilog HDL value set: **Table 6**: the description of values



Value	Description
0	Logic zero or false
1	Logic one or true
X	Unknown logical value
Z	The high impedance of the tri-state gate

7.5.1 INTEGER AND REAL DATA TYPES:

For the most part, C programmers are familiar with data types. The idea is that algorithms developed in C can be converted into Verilog if the two languages' data types are the same. Each bit in the brand-new two-state data types that Verilog offers can only be either 0 or 1. RTL models may perform better in simulators when two-state variables are used. Additionally, they have no influence on the synthesis's results.

Table 7 : the description of types

Types	Description
bit	user-defined size
byte	8 bits, signed
shortint	16 bits, signed
int	32 bits, signed
longint	64 bits, signed

✓ **Two-state integer types:**

Unlike in C, Verilog specifies the number of bits for the fixed-width types **Table 8** : the description of Two-state integer types

Types	Description
reg	user-defined size
logic	identical to reg in every way
integer	32 bits, signed

❖ **Four-state integer types:**

We preferred logic because it is better than reg. We can use logic where we have used reg or wire. **Table 9**: the description of four-state integer types

Type	Description
time	64-bit unsigned
shortreal	like a float in C
shortreal	like double in C
Realtime	identical to real

7.5.2 NON-INTEGER DATA TYPES:

✓ **Arrays:**

In Verilog, variables, scalar and vector nets can all be defined. You can also define memory arrays, which are one-dimensional arrays of a variable type.. Verilog lifted some of the limitations on memory array utilization and permitted multi-dimensional arrays of both nets and variables. This is improved in Verilog, which also allows for more array operations and refines the idea of arrays. Arrays in Verilog can have both packed and unpacked dimensions.

✓ **Packed dimensions:**

- Are certain to be organised sequentially in memory.
- Any other packed object can receive a copy of it.
- It is slicable ("part-selects").
- Are limited to "bit" kinds such as bit, logic, int, etc.; some of these types, like int, have fixed sizes.

✓ **Unpacked dimensions:**

- It can be set up in memory however the simulator sees fit. An array can be dependably copied onto another array of the same kind.
- There are restrictions for how an unpacked type is cast to a packed type when dealing with arrays of various types.
- On entire unpacked arrays and slices of unpacked arrays, Verilog supports a number of operations.
- The arrays or slices in question for these must be of the same type and shape, i.e., have the same quantity and length of unpacked dimensions.
- The packed dimensions can differ as long as the array or slice elements have the same number of bits.

The permitted operations are:

Reading and writing array elements, slices, and the entire array, as well as equality relations on the array's elements, slices, and elements.

Associative arrays and dynamic arrays, both of which have variable element counts during simulation, are also supported in Verilog. To accommodate all of these array kinds, Verilog provides a wide variety of arrays of querying procedures and methods.

7.6 NETS:

Nets don't hold any data because they are used to connect hardware components like logic gates. The physical link between structural elements, such as logic gates, is represented by the net variables. Except for trireg, these variables don't save any values. The value of these variables' drivers is continuously altered by the driving circuit. Wire, tri, wor, trior, wand, triand, tri0, tri1, supply0, supply1, and trireg are a few examples of net data types.

A net data type must be used when a signal is:

It is driven by some gadgets' output.

It is designated as either an input or an out-of port.

within a continuous assignment, on the left.

1. Wire:

A wire in a circuit acts as a fake wire when linking gates or modules. The value of a wire can be read but not assigned within a function or block. Because a wire cannot keep its value, it can only be driven by a continuous assignment statement or by connecting it to a gate or module's output.

2. Wand (wired-AND)

A wand's value is determined by the logical AND of all the drivers attached to it.

3. Wor (wired-OR)

The logical OR of all the drivers connected to wor determines its value.

4. Tri (three-state)

Except for the driver that decides the value of the tri, all drivers attached to it must be z.

5. Supply0 and Supply1

Supplies 0 and 1 specify the wires connected to logics 0 (ground) and 1 (power).

7.7 REGISTERS:

An instance of a data object is a register, which preserves a value for upcoming procedural assignments. They are only used by functions and procedural blocks. An assignment statement in a procedure acts as a trigger to change the value of the data storage element.

Reg is not by definition a physical register in Verilog; instead, it is a variable type. Data is stored in multi-bit registers as unsigned numbers, and no sign extension is applied to numbers that the user could have assumed to be two's complement. Register data types include reg, integer, time, and real. The type that is most frequently used is reg. Reg is the term used to describe logic. A general-purpose variable is an integer. In particular, loops--including indices, arguments, and constants--use them. Unlike officially specified reg types, which store data as unsigned numbers, they store data as signed numbers. Their size will automatically default to 32 bits if they store numbers that are not declared at compile time. The synthesiser adjusts them to the minimum width required at compilation if they hold constants in system modules real. Use Time and Realtime to store simulation times in test benches. The $time system task and time, a 64-bit quantity, can be used to store simulation time.A reg can also express combinational logic, therefore it need not always represent a flip-flop. At the beginning of the simulation, the reg variables are set to x. The value x is present in any wire variable that isn't attached to anything. During the declaration, the size of a register or wire may be defined. Registers and wires are designated as vectors when their size is greater than one bit.

7.8 VERILOG STRING:

Reg is used to store strings, and the reg variable's width needs to be sufficient to hold the string. A string has one byte and one ASCII value for each character. Verilog truncates the string's leftmost bits if the variable's size is less than the string's. Verilog inserts zeros to the left of the string if the variable's size is greater than the string's size.

7.9 LEXICAL TOKENS:

Verilog's lexical rules are comparable to those of the C programming language. The source text files for the Verilog language are a stream of lexical tokens. One or more characters may make up a lexical token, and each character appears in a single token.

The tokens may be strings, keywords, comments, numbers, or white space. A semicolon (;) should be used to end each line. A case-sensitive language is Verilog HDL. Furthermore, all terms are lowercase.

White Space

Tab, blank, newline, and form feed characters can all be found in white space. Except when they are used to divide other tokens, these characters are ignored. Tabs and blank spaces, however, matter in strings.

Comments

The remarks can be divided into two categories, including:

Comments on a single line start with the symbol // and end with a carriage return.

For instance, the single-line syntax is //this.

The tokens /* and */ mark the beginning and end of multi-line comments, respectively.

/* this is multiline syntax/, for instance.

Identifiers

The name given to an object, such as a module, register, or function, is its identifier. Identifiers must start with an alphabetical character or an underscore.A_Z and a_z, for instance.Identifiers are made up of an alphabetical combination, a number value, an underscore, and the symbol $. They have a maximum character count of 1024.Identifiers must start with an alphabetic character (a-z A-Z_) or an underscore. Identifiers may also include underscores, dollar signs, and alphabetic and numeric characters (a-z A-Z 0-9 _ $).Identifiers can have a maximum of 1024 characters.

Escaped Identifiers

By escaping the identification, Verilog HDL enables the use of any character in an identifier. Any readable ASCII character can be used in an identifier, which is what is meant by "escaped identifiers." the hexadecimal digits 21 through 7E, or the decimal values 33 through 126. Identifiers that have been escape start with a backslash (/). The backslash causes the whole identifier to escape. Commas, brackets and semicolons constitute a part of the escaped identifier unless they are preceded by white space, which ends the escaped identifier. Put a space after any escaped identifiers. If not, characters that ought to come after the identifier are seen as being a part of it.

7.10 OPERATORS

Unique characters called operators are used to specify conditions or control variables. One, two, and occasionally three characters are used to control variables.

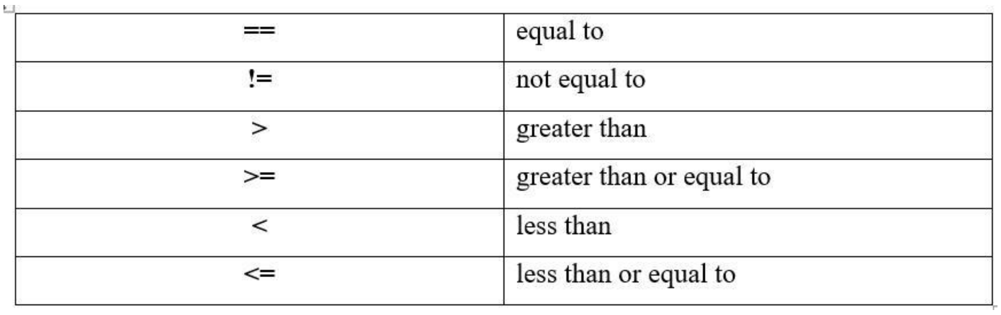
1. Arithmetic Operators

These operators carry out calculations. Both the + and - can be employed as binary (z-y) or unary (x) operators. Addition, subtraction, multiplication, division, and modulus are all arithmetic operators.

2. Relational Operators

It returns the result in single bit either 1 or 0. These are the operations :

Table 10 : Relational Operators

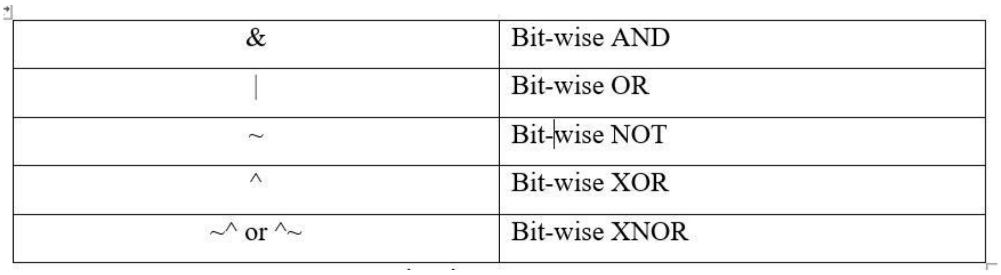


==	equal to
!=	not equal to
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to

3. Bit-wise Operators

It compares both operands bit by bit. These are the operations:

Table 11 : Bit-wise Operators

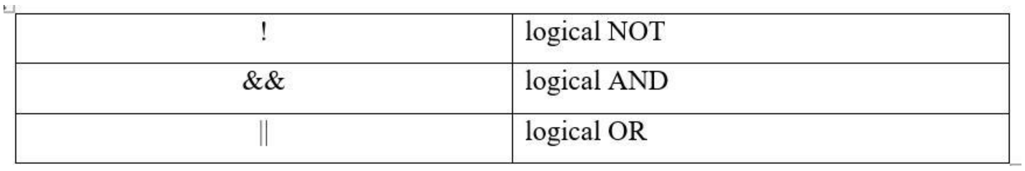


&	Bit-wise AND
|	Bit-wise OR
~	Bit-wise NOT
^	Bit-wise XOR
~^ or ^~	Bit-wise XNOR

4. Logical Operators

Logical operators, which are bit-wise operators, only use single-bit operands. One of the two-bit values, 0, or 1, is returned. All non-zero values can be treated as 1, and they can work with integers, bits, or expressions. Logical operators are usually used in conditional statements since they work with expressions. The following operators are a part of logical operation:

Table 12 : logical Operators

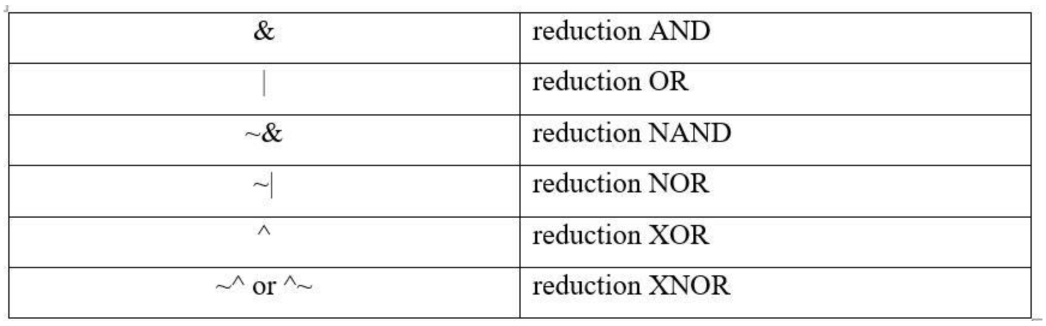


!	logical NOT
&&	logical AND
||	logical OR

5.Reduction Operators

Reduction operators work on all the bits in an operand vector and are the unary version of bitwise operators. These also give off a value of one bit. The following operators are a part of the reduction operation:

Table 13 : Reduction Operators

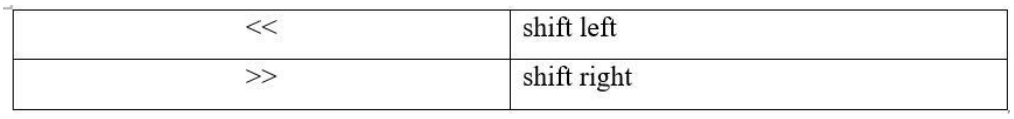


&	reduction AND
|	reduction OR
~&	reduction NAND
~|	reduction NOR
^	reduction XOR
~^ or ^~	reduction XNOR

6. Shift Operators

Shift operators shift the first operand by the number of bits indicated by the second argument in the syntax. Zeros are substituted for empty positions in both left and right shifts (sign extension is not used). The following operators work during shifts:

Table 14 : Shift Operators



<<	shift left
>>	shift right

7. Concatenation Operator

In order to create a larger vector, the concatenation operator joins two or more operands together. The following operator is a part of the concatenation operation: {}

8. Replication Operator

The replication operator duplicates an item in many ways. The Replication operation's operator is: **{n{item}}** (n fold replication of an item)

9. Conditional Operator

A multiplexer is created using the conditional operator. It is of the same type as that used in C/C++ and evaluates one of the two expressions based on the condition. Conditional operations employ the following operator: **(Condition)?:**

7.11 OPERANDS

Expressions or values that an operator manipulates or performs operations on are referred to as operands. Every expression requires at least one operand.

1. Literals

In Verilog expressions, literals are operands with constant values. The following are the top two Verilog literals:

String: A literal string operand is a one-dimensional array of letters enclosed in double quotation marks ("").

Numeric: The operand's constant number is given as a binary, octal, decimal, or hexadecimal number.

2. Wires, Regs, and Parameters

Data types such as wires, regs, and parameters are utilised as operands in Verilog expressions.. "x [2]" for both the bit- and part-selection and "x [4:2]"

Square brackets ("[]") are used to choose one bit or several bits from a wire, a set of rules, or a parameter vector, respectively.

3. Function Calls

Function calls employ the return value of a function directly, as opposed to assigning it to a register or wire beforehand. Simply listed as one of the types of operands is the function call. It is useful to know the bit width of the return value from the function call.

7.12 VERILOG MODULE:

A piece of code that implements a certain feature is known as a Verilog module. Modules can be embedded within other modules, and a higher-level module's input and output ports can be used to communicate with its lower-level modules.

Syntax

A piece of code that implements a certain feature is known as a Verilog module. Using its input and output ports, a higher-level module can communicate with its lower-level modules, and modules can be nested inside of other modules.

Purpose of a Module

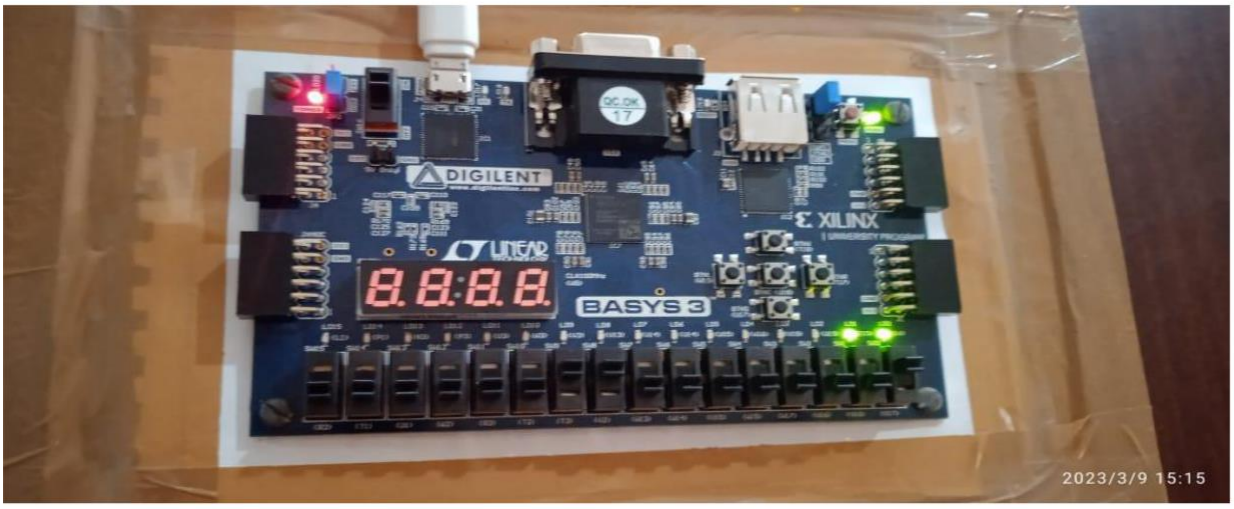
During synthesis, a module is a design element that carries out a specific set of behavioral characteristics before being converted into a digital circuit. Any number of inputs may be provided to the module, and it will output data in response.It makes it possible to reuse a module to build bigger, more significant modules that utilise more advanced hardware.

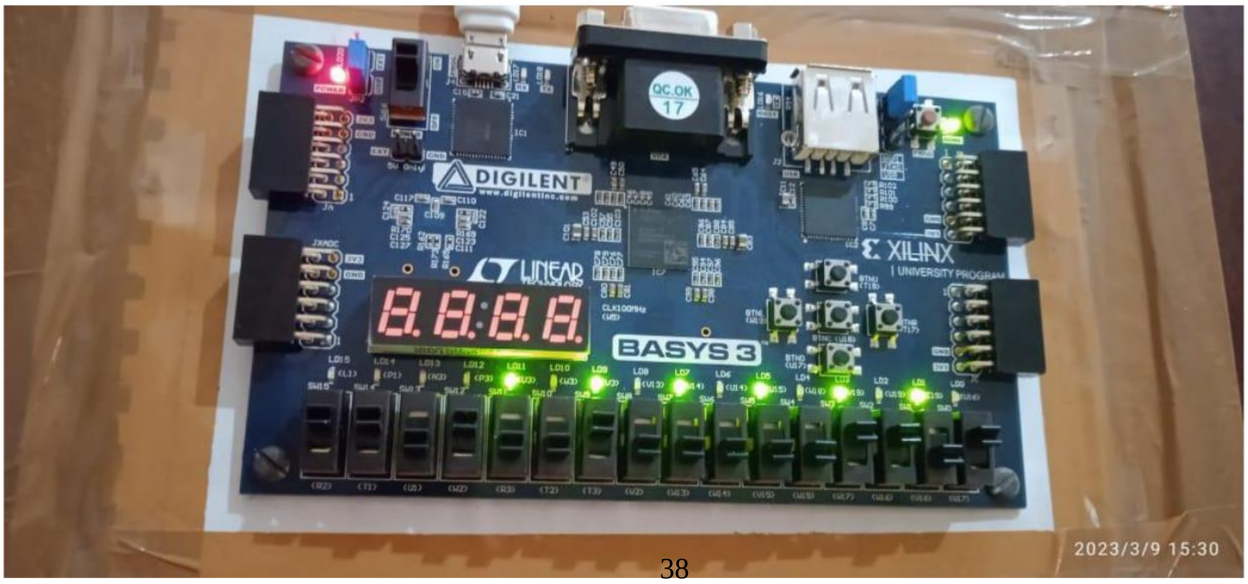
Hardware Schematic

The technique can be reversed as opposed to using smaller design blocks as a foundation for larger ones. Consider dividing a simple GPU engine into smaller components.

CHAPTER 8

HARDWARE IMPLEMENTATION





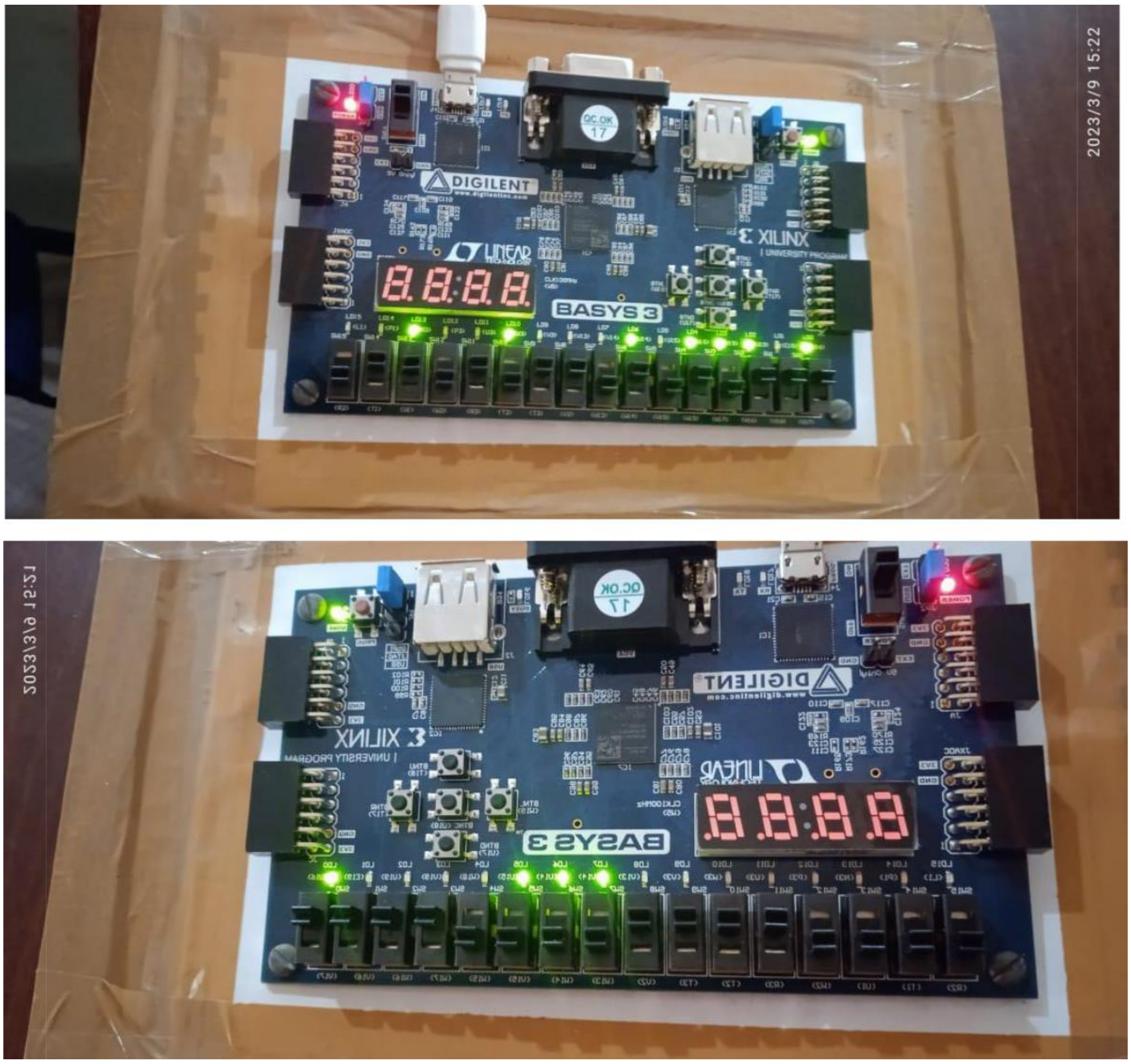


Fig.8 Hardware implementation of the proposed multiplier

BASYS 3 ARTIX-7 FPGA BOARD

An entry-level FPGA development board with the Xillinx Artix 7 FPGA architecture, the Basys 3 was created specifically for the vivado design suite.

The Basys 3 has the same functionalities that are present on all Basys boards. All necessary FPGA support circuits, fully functional hardware that is ready to use, a sizable collection of hardware, a sizable collection of on-board input-output devices, and a free version of development tools. The use of Basys 3 board in this project and the salient features are

- Inputs are given to the switches and each binary digit used as a one switch.
- Total 8x8 switches are used. Multiplier is used for remaining 8 switches.
- Output are represented by the LEDs.
- In this project USB connection is used.

- Switch acts as Input and Output indicated by LEDs.
- For the proper use of vivado, follow the
  steps: (i)Achieve the bitstream successfully.
  (ii)Connecting the hardware manager.

1) 00001101 x 11010010 => 0000101010101010 In Binary Form
   13 x 20 => 2730 In decimal Form
2) 01010111 x 01101011=> 0010010001011101 In Binary Form
   87 x 107 => 9309 In Decimal Form
3) 00001111 x 00001111 => 0000000011100001 In Binary Form
   15 x 15 => 225 In decimal Form
4) 00000001 x 00000011 => 0000000000000011 In Binary Form
   1 x 3 => 3 In Decimal Form
5) 11111111 x 11111111 => 1111111000000001 In Binary Form
   255 x 255 =>65025 In Decimal Form

CONCLUSION

In this project , a 8-bit MAC and 16-bit MAC are designed with successful implementation 8-bit and 16-bit proposed multiplier. The proposed multiplier was synthesized in different FPGA families using the Xilinx tool and the response of each FPGA platform was observed. In the proposed 16-bit multiplier, it is concluded that multiplier implementation using Artix 7 offers optimized results in terms of power when compared to other FPGA families. When compared to other FPGA families, the number of occupied segments is lower when using Vertex-6 FPGA. Virtex 7 produces optimized outcomes in terms of gate delay, logic delay, and route delay. According to the findings, implementing a 8-bit multiplier using Virtex-6 (lower power) offers optimised results in terms of area, delay, and power when compared to other FPGA families. We can see that using Vertex-6 (lower power) FPGA results in increase in delay and a reduction in area when compared to Artix 7, and Spartan 6. Spartan 6 and Artix 7 outperform the Virtex 6 family in terms of battery consumption.The compressors in this proposed multiplier are used in such a way that the size of the compressor is determined by the entire sum of both input bits and carry bits propagated to that level. By deciding on the compressor in this manner, the area of the suggested multiplier unit is reduced.To obtain the results of 16 bit multiplication, different compressors such as 4:2, 5:2, 7:2, and other higher order compressors are used at various positions in the proposed multiplier.

References

[1] Y. Kim, Y. Zhang, and P. Li, "An energy efficient approximate adder with carry skip for error resilient neuromorphic VLSI systems," in proc. of International Conference on Computer-Aided Design (ICCAD), USA, 2013, pp. 130-137.

[2] D. Baran, M. Aktan, and V.G. Oklobdzija, "Energy Efficient Implementation of Parallel CMOS Multipliers with Improved Compressors,", in proc. ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED), USA, 2010, pp. 147-152.

[3] S. Veeramachaneni, K. Krishna M, L. Avinash, S. R. Puppala, and M.B. Srinivas, "Novel Architectures for High-Speed and Low-Power 3-2, 4-2, and 5-2 Compressors", in proc. of International Conference on VLSI Design (VLSID), Bangalore, 2007, pp.324-329.

[4] R. Menon, and D. Radhakrishnan, "High performance 5: 2 compressor architectures," in proc. of IEE - Circuits, Devices and Systems, vol. 153,no. 5, pp. 447-452, Nov 2006.

[5] A. Pishvaie, G. Jaberipur, and A. Jahanian,"High Performance CMOS (4:2) compressors," Int. journal of electronics, vol. 101, no. 11,1511-1525, Jan. 2014.

[6] O. Kwan, K. Nawka, and E. SwartzlanderJr," A 16 bit by 16 bit MAC Design Using Fast 5:3 Compressor Cells", J. of VLSI Signal Processing, vol. 31, no. 2, 77-89,July 2002.

[7] S. Mehrabi, R.F Mirzaee, S. Zamanzadeh, K. Navi, and O. Hashemipour,"Design, analysis, and implementation of partial product reduction phase by using wide m:3 (4 m 10) compressors", Int. Journal of High Performance System Arch, vol. 4, no. 4,231-241,Jan. 2013.

[8] A. Dandapat, P.Bose, S. Ghosh, P Sarkar, and D. Mukhopadhyay,(March 2009), "A 1.2-ns 16 x 16 bit binary multiplier using high speed compressors," World Academy of Science, Engineering and Technology, vol. 39, 627-632,March 2009.

[9] R. Marimuthu, M. Pradeep kumar, D. Bansal, S. Balamurugan, and P.S Mallick,"Design of high speed and low power 15-4 compressor," in proc. International Conference on Communication and Signal Processing (ICCSP),Melmaruvathur, 2013, pp. 533-536.

[10] J. Liang, J. Han,and F. Lombardi, New metrics for the reliability of approximate and probabilistic adders," IEEE Trans. on Computers, vol. 63, no. 9,1760 – 1771,Sep.2013.

[11] K. Bhardwaj, P.S. Mane, and J. Henkel,"Power- and area-efficient Approximate Wallace Tree Multiplier for error resilient systems," in proc. of 15th International Symposium on Quality Electronic Design (ISQED),USA, 2014, pp. 263 – 269.