

**DESIGN AND ANALYSIS OF 16-BIT VEDIC MULTIPLICATION USING
COMPRESSOR ADDERS**

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

MADHURI DASARI (317126512034),

B KASI VINAY CHOWDARY (317126512006),

N ADITYA SRI ARSHITH (317126512043).

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)

(2020-2021)

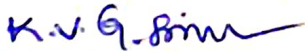
DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING
 ANIL NEERUKONDA INSTITUTE OF TECHNOLOGY AND SCIENCES
 (Permanently Affiliated to AU, Approved by AICTE and Accredited by NBA & NAAC with
 'A' Grade)
 Sangivalasa, Bheemili Mandal, Visakhapatnam dist.(A.P)



CERTIFICATE

This is to certify that the project report entitled "DESIGN AND ANALYSIS OF 16-BIT VEDIC MULTIPLICATION USING COMPRESSOR ADDERS" submitted by MADHURI DASARI (317126512034), B KASI VINAY CHOWDARY (317126512006), N ADITYA SRI ARSHITH (317126512043) in partial fulfillment of the requirements for the award of the degree of Bachelor of Engineering in Electronics & Communication Engineering of Andhra University, Visakhapatnam is a record of Bonafide work carried out under my guidance and supervision.

Project Guide

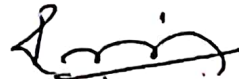

 Dr.K.V.GowreeSrinivas
 Ph.D

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

Head of the Department


 Dr. V.Rajyalakshmi
 M.E, Ph.D, MIEEE, MISTE

Professor
 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 **Mr.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. V. Rajyalakshmi**, 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 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

MADHURI DASARI (317126512034)

B KASI VINAY CHOWDARY (317126512006)

N ADITYA SRI ARSHITH (317126512043)

CONTENTS

LIST OF FIGURES	7
ABSTRACT	8
CHAPTER 1 INTRODUCTION	9
1.1 Project Objective	9
1.2 Project Outline	9
CHAPTER 2 VEDIC MATHEMATICS	11
2.1 Introduction to Vedic Mathematic	11
2.2 Sixteen Sutras	13
2.3 Vedic Mathematics Sutras	14
CHAPTER 3 MULTIPLIERS	16
3.1 Introduction to Multipliers	16
3.2 Types of Multipliers	16
3.2.1 Serial Multipliers	16
3.2.2 Serial/Parallel Multiplier	16
3.2.3 Shift and Add Multiplier	17
3.2.4 Array Multipliers	18
3.2.5 Booth Multipliers	18
3.2.6 Sequential multiplier	19
3.2.7 Wallace tree Multiplier	20
CHAPTER 4 INTRODUCTION TO COMPRESSOR ADDERS	
4.1 What are compressor adders	21
4.2 Types of compressor adders	21
4.2.1 4:2 compressor adder	21
4.2.2 7:2 compressor adder	21
4.2.3 5:3 compressor adder	22
4.2.4 10:4 compressor adder	23
4.2.5 15:4 compressor adder	24
4.2.6 20:5 compressor adder	25
CHAPTER 5 INTRODUCTION TO VERILOG	
5.1 Definition	26
5.2 History of Verilog	26
5.3 Uses of Verilog	26

5.4	Features of Verilog	27
5.5	Data Types	27
5.5.1	Integer and Real data types	27
5.5.2	Non integer data types	27
5.6	Nets	29
5.7	Register	30
5.8	Verilog string	31
5.9	Lexical Tokens	31
5.10	Operators	33
5.11	Operands	35
5.12	Verilog Module	36
CHAPTER 6	XILINX SOFTWARE5	38
6.1	Project Navigator Interface	38
6.2	HDL Based Design	41
6.3	VHDL	41
6.4	Synthesizing the Design	44
CHAPTER 7	SIMULATION RESULTS	48
CONCLUSIONS		57
REFERENCES		58

List of Figures

- Figure 3.1: Serial /parallel Multiplier
- Figure 3.2: Shift and add multiplier
- Figure 3.3: Array multiplier
- Figure 3.4 Booth Multiplier
- Figure 3.5: Sequential multiplier
- Figure 3.6 Wallace tree multiplier
- Fig 4.2.1 circuit diagram of 4-2 compressor adder
- Fig 4.2.2 circuit diagram of 7-2 compressor adder
- Fig 4.2.3(a) circuit diagram of 5-3 compressor adder using gates
- Fig 4.2.3(b) modified circuit diagram of 5-3 compressor adder
- Fig 4.2.4 circuit diagram of 10-4 compressor adder
- Fig 4.2.5 circuit diagram of 15-4 compressor adder
- Fig 4.2.5 circuit diagram of 20-5 compressor adder
- Figure 6.1: Project Navigator
- Figure 6.2: Project navigator Desktop
- Figure 6.3 New project Wizard- Create New Project Page
- Figure 6.4: New Project Wizard- Device Properties Page
- Figure 6.5: Specifying Synthesis Tool
- Figure 6.6: RTL Schematic
- Fig 7.1(a) 5-3 RTL schematic diagram
- Fig 7.1(b) 5-3 synthesis diagram
- Fig 7.1(c) 5-3 power report
- Fig 7.1(d) 5-3 area utilization report
- Fig 7.2(a) 10-4 RTL schematic diagram

- Fig 7.2(b) synthesis diagram
- Fig 7.2(c) power report
- Fig 7.2(d) Area utilization report
- Fig 7.3(a) 15-4 schematic diagram
- Fig 7.3(b) 15-4 synthesis diagram
- Fig 7.3(c) 15-4 power report
- Fig 7.3(d) 15-4 Area utilization report
- Fig 7.4(a) 20-5 RTL schematic diagram
- Fig 7.4(b) 20-5 synthesis diagram
- Fig 7.4(c) 20-5 power report
- Fig 7.4(d) 20-5 Area utilization report
- Fig 7.5 comparison of Area utilization and power of different compressor adders

ABSTRACT

A novel architecture of Vedic multiplier with ‘Urdhava-tiryakbhyam’ methodology for 16-bit multiplier and multiplicand is proposed with the use of compressor adders. Equations for each bit of 32-bit resultant are calculated distinctly and compressor adders are used to implement these equations. They are chosen as they decrease vertical critical delay in comparison to the conventional architectures of compressors implemented using half and full adders only and so make the multiplier fast. The designs are coded in VHDL (Very High-speed Integrated Circuits Hardware Description Language) and synthesized with Xilinx ISE 13.1 using Spartan 3e series of FPGA (Field Programmable Gate Array). The combinational delay calculated for proposed 16×16-bit multiplier is 32 ns. Further speed comparisons of compressor adders with traditional ones and proposed multiplier with popular methods for multiplication are shown. Results clearly indicate the better speed performance of our proposed Vedic multiplier.

CHAPTER-1

INTRODUCTION

Vedic multiplier is built on Vedic mathematics which further is extracted from the ancient Vedas by the Sri Bharati Krishna Tirthaji in between 1911 and 1918. The specialty of Vedic mathematics is that it gives simple way to solve the calculations which can be easily understood by human minds. This Vedic mathematics is divided into 16 sutras which give different rules for the simplification of the problems related to trigonometry, algebra, geometry etc. Designs based on Vedic mathematics have been used in many applications like ALU, MAC etc. and have shown better results in terms of delay, area. Among the 16 sutras, 'Urdhava-tiryakbhyam' is picked as this sutra is a universal method for multiplication and thus always remained favorite method of implementers. Previously this method was used only for multiplication of decimal numbers but from some time it has been used and proved to be better for binary number multiplication. Also, the increase in delay and area with the increase in number of input bits is at a slow rate with respect to other sutras. With the selection of 'Urdhava-tiryakbhyam' sutra, further selection comes with adders to add the partial products generated for the resultant bits (s_0 – s_{31}). These adders decide the speed of the multiplier and thus requirement of high-speed adder becomes the need for concern. In this paper, we have given a novel architecture for the separate calculation of product bits of the multiplication of multiplier and multiplicand. For this method, compressor adders have been used over conventional architectures of half-adders and full adders because of their higher speed performance. These compressors actually act as counters and count the number of 1s in the given bits and thus behave as adders. They make use of multiplexer in addition with half-adders and full adders which allow the use of lesser XOR gates and thus high speed. With each resultant bit some carries are also generated which goes further for the calculation of next final product bits.

1.1 PROJECT OBJECTIVE:

The main objective of this project is to study, design and analysis of 16-bit Vedic multiplier using higher order compressor adders using VLSI (Very Large-Scale Integration) design. The Synthesis and Implementation is done for different types of compressor adders using Xilinx Vivado. The performance is compared in terms of power, area and delay.

1.2 PROJECT OUTLINE:

This project report is presented over the 6 remaining chapters.

Chapter 2 presents Introduction to Vedic Mathematics provides the fundamentals of vedic

multiplication using Urdhava-Tiryakbham sutra.

Chapter 3 explains Introduction to Multipliers and different types of multipliers.

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

Chapter 5 describes Introduction to Verilog.

Chapter 6 mainly gives the description about working with XILINX ISE DESIGN SUITE.

Chapter 7 presents the simulation results which are simulated using Vivado Design Suite simulator.

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

CHAPTER 2

INTRODUCTION TO VEDIC MATHEMATICS

In this chapter we just recall some notions given in the book on Vedic Mathematics written by Jagadguru Swami Sri Bharati Krsna Tirthaji Maharaja (Sankaracharya of Govardhana Matha, Puri, Orissa, India), General Editor, Dr. V.S. Agrawala. Before we proceed to discuss the Vedic Mathematics that he professed we give a brief sketch of his heritage.

He was born in March 1884 to highly learned and pious parents. His father Sri P Narasimha Shastri was in service as a Tahsildar at Tinnevely (Madras Presidency) and later retired as a Deputy Collector. His uncle, Sri Chandrasekhar Shastri was the principal of the Maharajas College, Vizianagaram and his great grandfather was Justice C. Ranganath Shastri of the Madras High Court. Born Venkatraman he grew up to be a brilliant student and invariably won the first place in all the subjects in all classes throughout his educational career. During his school days, he was a student of National College Tiruchirapalli; Church Missionary Society College, Tinivelli and Hindu College Tinnivelly in Tamil Nadu. He passed his matriculation examination from the Madras University in 1899 topping the list as usual. His extraordinary proficiency in Sanskrit earned him the title "Saraswati " from the Madras Sanskrit Association in July 1899. After winning the highest place in the B.A examination Sri Venkataraman appeared for the M.A. examination of the American College of Sciences, Rochester, New York from the Bombay center in 1903. His subject of examination was Sanskrit, Philosophy, English, Mathematics, History and Science. He had a superb retentive memory.

In 1911 he could not anymore resist his burning desire for spiritual knowledge, practice and attainment and therefore, tearing himself off suddenly from the work of teaching, he went back to Sri Sacchidananda Shivabhinava Nrisimha Bharati Swami at Sringeri. He spent the next eight years in the profoundest study of the most advanced Vedanta Philosophy and practice of the Brahmasadhana.

After several years in 1921 he was installed on the pontifical throne of Sharada Peetha Sankaracharya and later in 1925 he became the pontifical head of Sri Govardhan Math Puri

where he served the remainder of his life spreading the holy spiritual teachings of Sanatana Dharma.

In 1957, when he decided finally to undertake a tour of the U.S.A he rewrote from his memory the present volume of Vedic Mathematics giving an introductory account of the sixteen formulae reconstructed by him. This is the only work on mathematics that has been left behind by him.

Now we proceed on to give the 16 sutras (aphorisms or formulae) and their corollaries. As claimed by the editor, the list of these main 16 sutras and of their sub-sutras or corollaries is prefixed in the beginning of the text and the style of language also points to their discovery by Sri Swamiji himself. This is an open acknowledgement that they are not from the Vedas. Further the editor feels that at any rate it is needless to dwell longer on this point of origin since the vast merit of these rules should be a matter of discovery for each intelligent reader.

Sl. No	Sutras	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 Gunṛat
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
15		
16		

The main sutra is given by 'URDHVA TRIYAKBHYAM' which means Vertical cross wise

Urdhva-triyakbhyam sutra which is the General Formula applicable to all cases of multiplication and will also be found very useful later on in the division of a large number by another large number.

The formula itself is very short and terse, consisting of only one compound word and means "vertically and cross-wise." The applications of this brief and terse sutra are manifold.

This sutra has been identified for use in the present work since it gives a general formula that is applicable to all cases of multiplication (large bit multiplication, small bit multiplication and modular multiplication) and is also very compact in the division of a large number by another large number, for example division of a 15-digit number by a 5-digit number. The algebraic principle involved is explained as follows

Multiplication Using Urdhva Triyakbhyam Sutra:

Suppose we have to multiply $(ax+b)$ by $(cx+d)$. The product is $acx^2 + x(ad+bc) + bd$. This can be obtained as follows:

Step 1: The coefficient of x^2 is obtained by the vertical multiplication of a and c

Step 2: The coefficient of x is obtained by the crosswise multiplication of a and d and of b and c and the addition of the two products

Step 3: The independent term is arrived at by vertical multiplication of the absolute terms b and d .

A simple example will suffice to clarify the modus operandi thereof. Suppose we have to multiply 12 by 13.

- (i) We multiply the left hand most digit 1 of the 12 multiplicands vertically by the left hand most digit 1 of the multiplier get their product 1 $1:3 + 2:6 = 156$ and set down as the left hand most part of the answer;
- (ii) We then multiply 1 and 3 and 1 and 2 crosswise add the two get 5 as the sum and set it down as the middle part of the answer; We multiply 2 and 3 vertically get 6 as their product and put it down as the last the right hand most part of the answer. Thus $12 \times 13 = 156$.

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:

$$\begin{array}{l}
 A = A_7A_6A_5A_4 \quad A_3A_2A_1A_0 \\
 \quad \quad X_1 \quad \quad \quad X_0 \\
 B = B_7B_6B_5B_4 \quad B_3B_2B_1B_0 \\
 \quad \quad \quad Y_1 \quad \quad Y_0
 \end{array}$$

$$\begin{array}{l}
 \text{So } A = X_1 X_0 \text{ (8 bit Multiplicand)} \\
 \quad B = Y_1 Y_0 \text{ (8 bit Multiplier)}
 \end{array}$$

where X_1, X_0, Y_1, Y_0 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.

$$X_1 X_0 \times Y_1 Y_0 = F E D C$$

1. $CP = X_0 \times Y_0 = C$.
2. $CP = X_1 \times Y_0 + X_0 \times Y_1 = D$
3. $CP = X_1 \times Y_1 = F E$

where F is the carry of the product of $X_1 \times Y_1$ and CP is the cross product

Note: 1. Each Multiplication operation is an embedded parallel 4 x 4 multiply module.

2. The carry generated in each of the multiplication modules is propagated to the next module. This multiplier architecture has the advantage compared of minimal gate delays and improved regularity of structure.

The process is further explained with the help of examples. Two digit and three digit multiplication examples are explained using decimal numbers and the multiplication process is shown with the help of lines. The digits on either side of the line are multiplied and the result is added to the previous carry and the process is continued.

CHAPTER 3

INTRODUCTION TO MULTIPLIERS

Multipliers play an important role in today's digital signal processing and various other applications. With advances in technology, many researchers have tried and are trying to design multipliers which offer either of the following design targets – high speed, low power consumption, regularity of layout and hence less area or even combination of them in one multiplier thus making them suitable for various high speed, low power and compact VLSI implementation.

The common multiplication method is “add and shift” algorithm. In parallel multipliers number of partial products to be added is the main parameter that determines the performance of the multiplier

To reduce the number of partial products to be added, Modified Booth algorithm is one of the most popular algorithms.

To achieve speed improvements Wallace Tree algorithm can be used to reduce the number of sequential adding stages.

Further by combining both Modified Booth algorithm and Wallace Tree technique we can see advantage of both algorithms in one multiplier.

However, with increasing parallelism, the number of shifts between the partial products and intermediate sums to be added will increase which may result in reduced speed, increase in silicon area due to irregularity of structure and also increased power consumption due to increase in interconnect resulting from complex routing.

On the other hand, “serial-parallel” multipliers compromise speed to achieve better performance for area and power consumption. The selection of a parallel or serial multiplier actually depends on the nature of application. In this lecture we introduce the multiplication algorithms and architecture and compare them in terms of speed, area, power and combination of these metrics.

3.1 TYPES OF MULTIPLIERS

SERIAL MULTIPLIER: Where area and power are of utmost importance and delay can be tolerated the serial multiplier is used. This circuit uses one adder to add the $m * n$ partial products.

Serial/Parallel Multiplier: One operand is fed to the circuit in parallel while the other is serial. N partial products are formed each cycle. On successive cycles, each cycle does the addition of one column of the multiplication table of $M*N$ PPs. The final results are stored in the output register after $N+M$ cycles

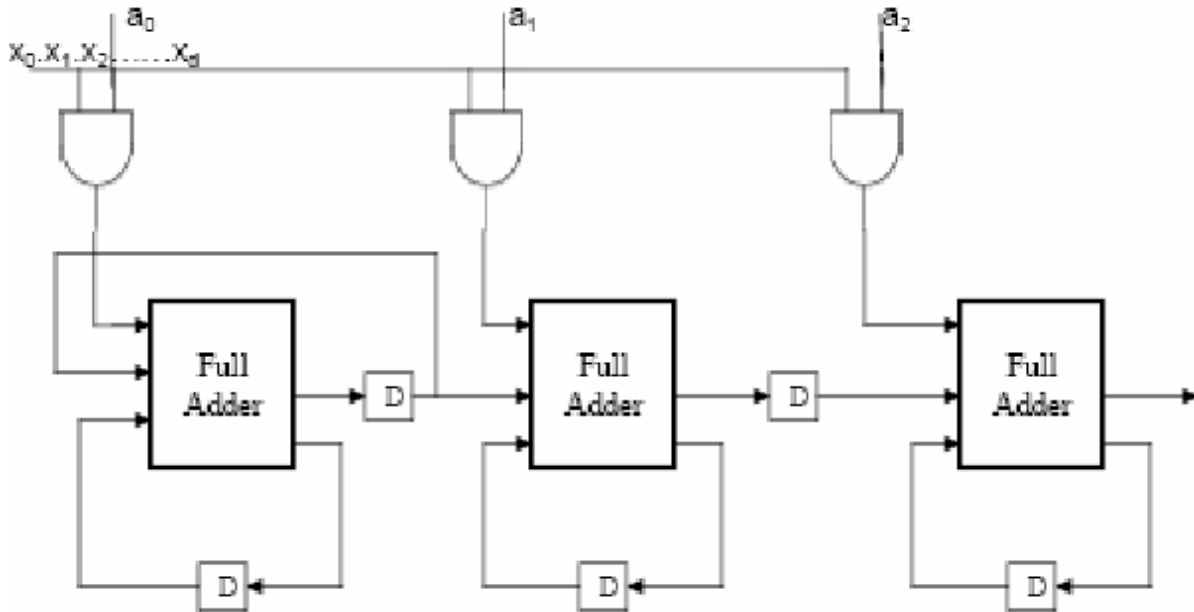


Figure 3.1: Serial /parallel Multiplier

Shift and Add Multiplier: Depending on the value of multiplier LSB bit, a value of the multiplicand is added and accumulated. At each clock cycle the multiplier is shifted one bit to the right and its value is tested. If it is a 0, then only a shift operation is performed. If the value is a 1, then the multiplicand is added to the accumulator and is shifted by one bit to the right. After all the multiplier bits have been tested the product is in the accumulator. The accumulator is $2N (M+N)$ in size and initially the N , LSBs contains the Multiplier. The delay is N cycles maximum. This circuit has several advantages in asynchronous circuits

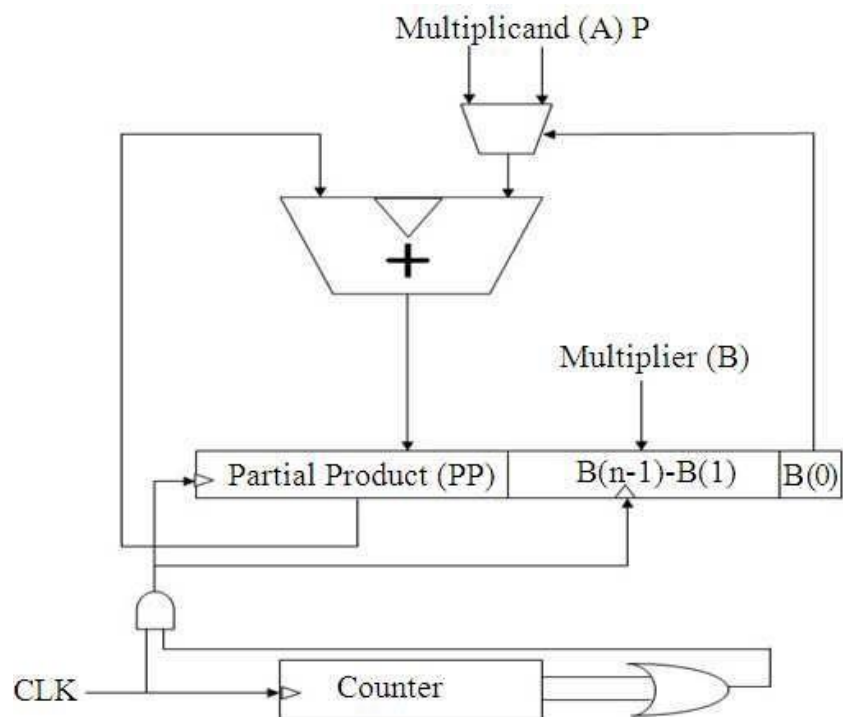


Figure 3.2: Shift and add multiplier

Array Multipliers: Array multiplier is well known due to its regular structure. Multiplier circuit is based on add and shift algorithm. Each partial product is generated by the multiplication of the multiplicand with one multiplier bit. The partial product are shifted according to their bit orders and then added. The addition can be performed with normal carry propagate adder. $N-1$ adders are required where N is the multiplier length.

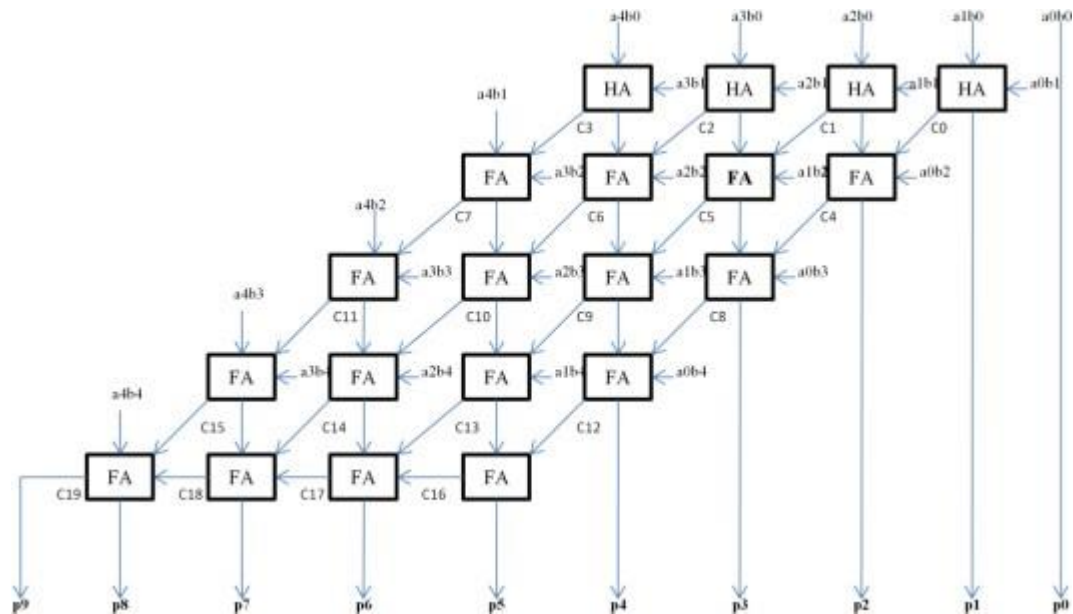


Figure 3.3: Array multiplier

Booth Multipliers: It is a powerful algorithm for signed-number multiplication, which treats both positive and negative numbers uniformly. For the standard add-shift operation, each multiplier bit generates one multiple of the multiplicand to be added to the partial product. If the multiplier is very large, then a large number of multiplicands have to be added. In this case the delay of multiplier is determined mainly by the number of additions to be performed. If there is a way to reduce the number of the additions, the performance will get better. Booth algorithm is a method that will reduce the number of multiplicand multiples. For a given range of numbers to be represented, a higher representation radix leads to fewer digits. Since a k -bit binary number can be interpreted as $K/2$ -digit radix-4 number, a $K/3$ -digit radix-8 number, and so on, it can deal with more than one bit of the multiplier in each cycle by using high radix multiplication.

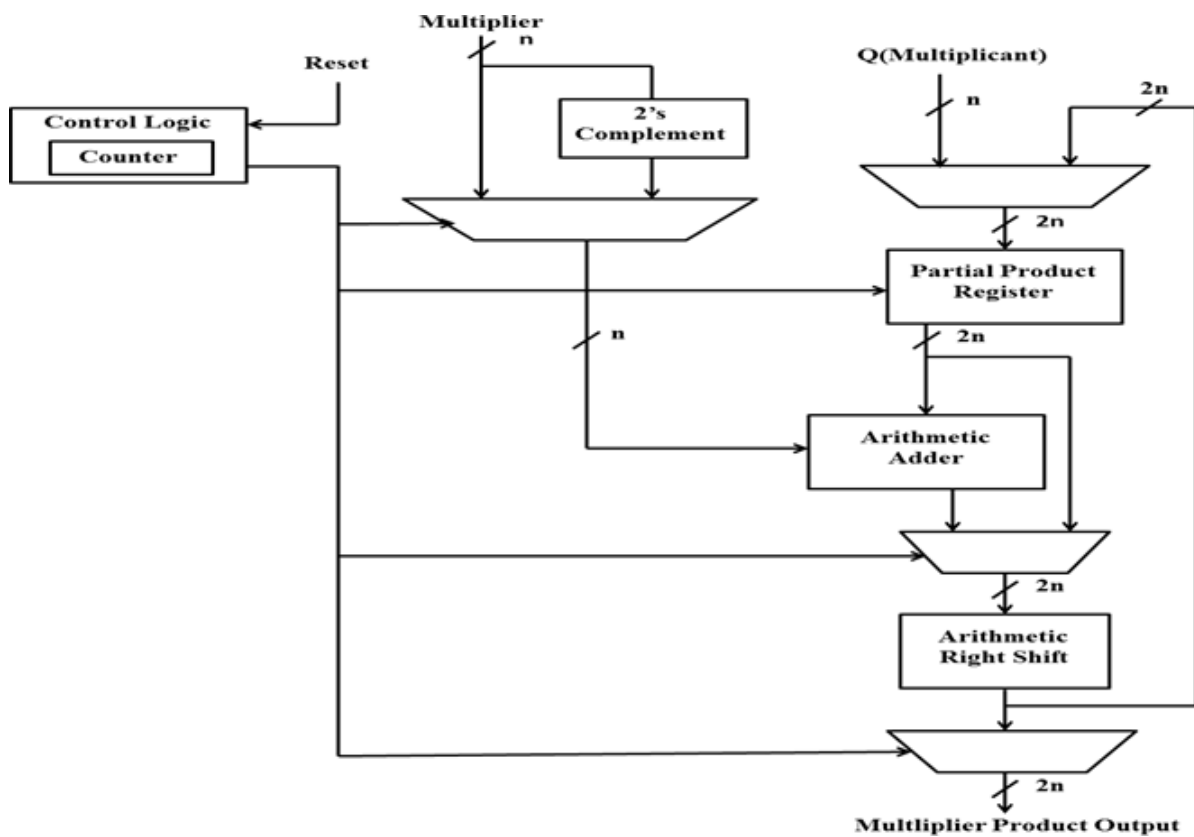


Figure 3.4 Booth Multiplier

Sequential multiplier: If we want to multiply two binary number (multiplicand X has n bits and multiplier Y has m bits) using single n bit adder, we can build a sequential circuit that processes a single partial product at a time and then cycle the circuit m times. This type of circuit is called sequential multiplier. Sequential multipliers are attractive for their low area requirement. In a sequential multiplier, the multiplication process is divided into some sequential steps. In each step some partial products will be generated, added to an accumulated partial sum and partial sum will be shifted to align the accumulated sum with partial product of next steps. Therefore, each step of a sequential multiplication consists of three different operations which are generating partial products, adding the generated partial products to the accumulated partial sum and shifting the partial sum.

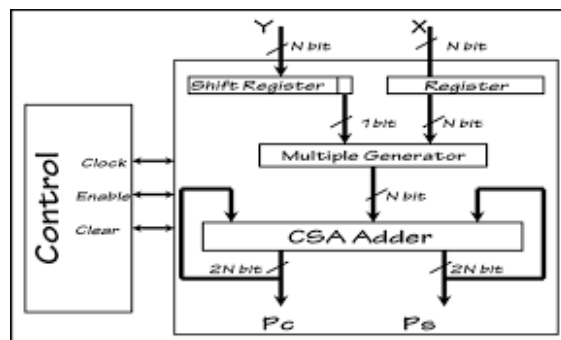


Figure 3.5: Sequential multiplier

Wallace tree Multiplier: A Wallace tree is an efficient hardware implementation of a digital circuit that multiplies two integers. It was devised by the Australian computer scientist Chris Wallace in 1964.

The Wallace tree has three steps:

1. Multiply (that is – AND) each bit of one of the arguments, by each bit of the other, yielding n^2 results. Depending on position of the multiplied bits, the wires carry different weights, for example wire of bit carrying result of is 128 (see explanation of weights below).
2. Reduce the number of partial products to two by layers of full and half adders.
3. Group the wires in two numbers, and add them with a conventional adder

The second step works as follows. As long as there are three or more wires with the same weight add a following layer: -

- Take any three wires with the same weights and input them into a Full adder. The result will be an output wire of the same weight and an output wire with a higher weight for each three input wires.
- If there are two wires of the same weight left, input them into a Half adder
- If there is just one wire left, connect it to the next layer.

The benefit of the Wallace tree is that there are only reduction layers, and each layer has $O(1)$ propagation delay. As making the partial products is $O(1)$ and the final addition is $O(\log n)$, the multiplication is only, not much slower than addition (however, much more expensive in the gate count). Naively adding partial products with regular adders would require $O(\log^2 n)$ time.

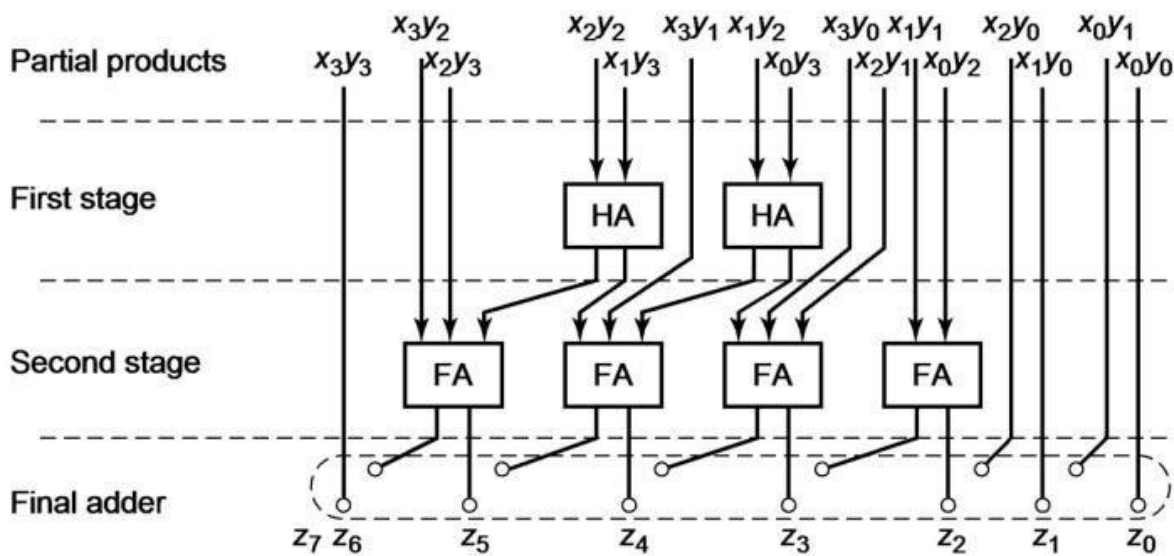


Figure 3.6 Wallace tree multiplier

CHAPTER 4

INTRODUCTION TO COMPRESSOR ADDERS

4.1 WHAT ARE COMPRESSOR ADDERS

Compressor adders are basic circuits which add bits more than four at a time to give better delay results over the combinational circuits of half and full adders. The symbolic representation of compressor architecture is $N - r$ where 'N' represents the number of the bits that are fed and 'r' represents the total count of the 1s present in N bits. It actually reduces the gate counts and delay in comparison to adder circuits and that is why named as compressor. A large part of research has been done in improving the circuits of lower compressors. Along with this, higher compressors are also implemented to add higher number of bits. The main compressor architectures which are used widely are 4-2, 7-2, 5-3, 10-4, 15-4 and 20-5.

4.2 TYPES OF COMPRESSOR ADDERS

4-2 COMPRESSOR ADDER: A 4-2 compressor compresses four inputs plus one carry bit 'Cin' from the previous column into two outputs 'Sum' and 'Carry' plus one intermediate carry bit 'Cout' that is provided as Cin to the next column, as the name implies.

The 4-2 compressor's input and output relationship can be expressed mathematically as $Cin + X4 + X3 + X2 + X1 = Sum + 2(Carry + Cout)$. As shown in Fig., implementing a 4-2 compressor using basic cascading of full adders introduces a critical path delay of four XOR gates. As shown in Fig. 2, logical optimization reduces the critical path delay to three XOR gates, and this 4-2 compressor is considered a traditional model.

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

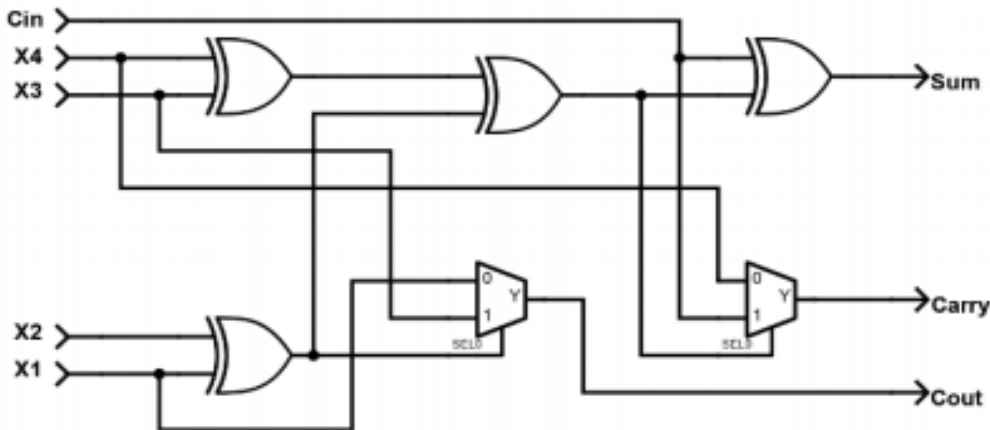


Fig 4.2.1 circuit diagram of 4-2 compressor adder

7-2 COMPRESSOR ADDER: Similar to its 4:2 compressor counterpart, the 7:2 compressor as shown in Fig. 6., is capable of adding 7 bits of input and 2 carries from the previous stages, at a time. In our implementation, we have designed a novel 7:2 compressor utilizing two 4:2 compressors, two full adders and one-half adder. The architecture for the same has been shown in Fig. 7. As mentioned earlier, since the 4:2 compressor shows a significant increase in speed by around 66.6%, utilizing the same in this architecture would improve the efficiency as opposed to a conventional approach of adding nine bits at a time using only full adders and half adders. This leads to a great improvisation in speed of the processor. Through experimentation on a Xilinx Spartan-3e FPGA, it was found that the novel 7:2 compressor adder architecture introduced here is 1.05 times faster than a conventional approach. This result justifies the need of utilizing this compressor in our design.

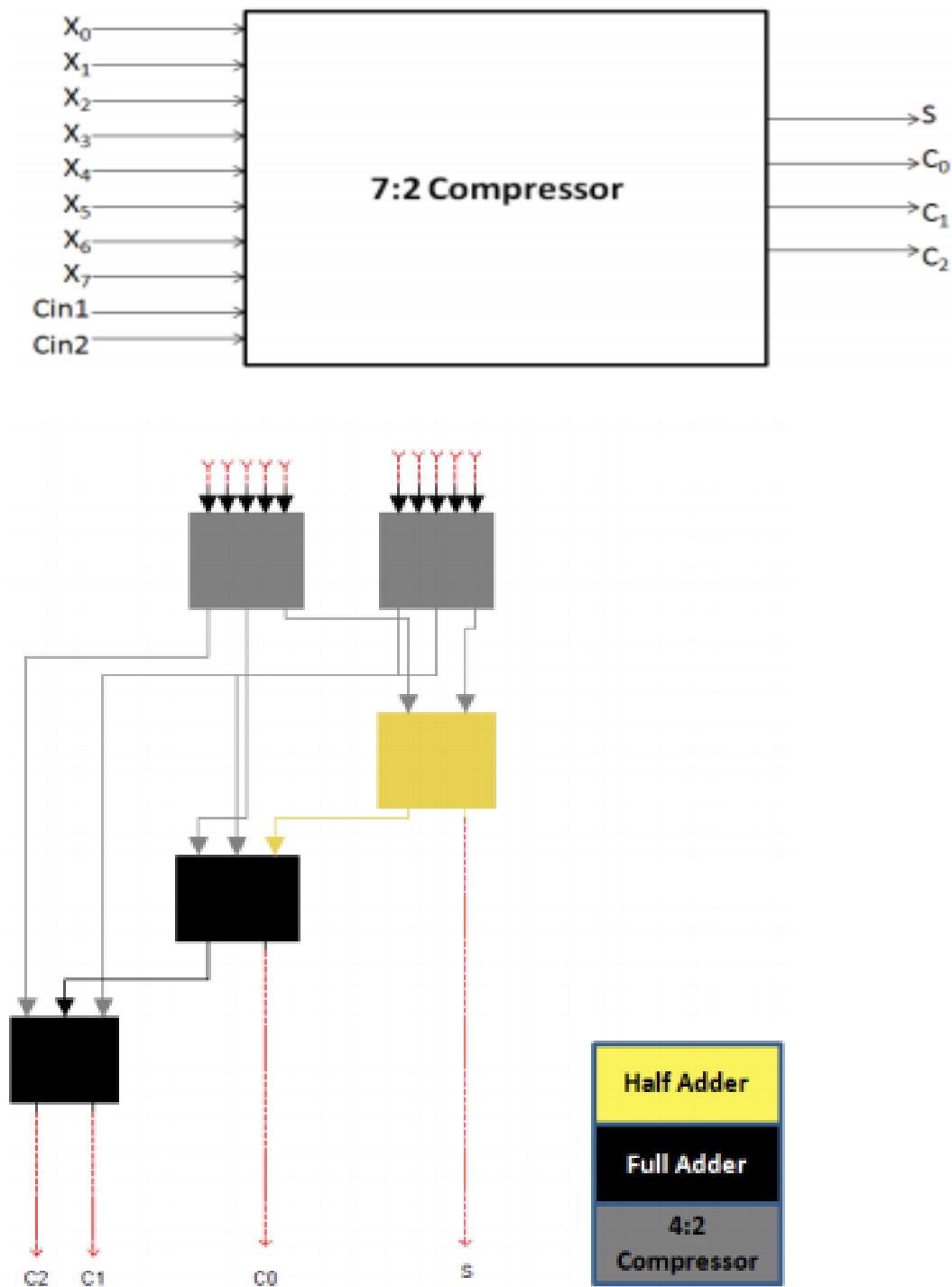


Fig 4.2.2 circuit diagram of 7-2 compressor adder

5-3 COMPRESSOR ADDER: A 5-3 compressor adder is a logical circuit in which maximum five bits can be added at the same time and three bits resultant of maximum value 101 is obtained. The circuit uses three 4:1 multiplexer. This multiplexer allows only one output to be high at a time and this property makes the multiplier fast and low power consuming circuit [13–15]. The circuit is reorganized in such a way that only 3 XOR operations are used instead of 5 XOR operations (in case of conventional 5-3 compressor) and other two inputs (X_3 and X_4) acts as a control signal. The conventional and the modified 5-3 compressor adder circuit are shown in

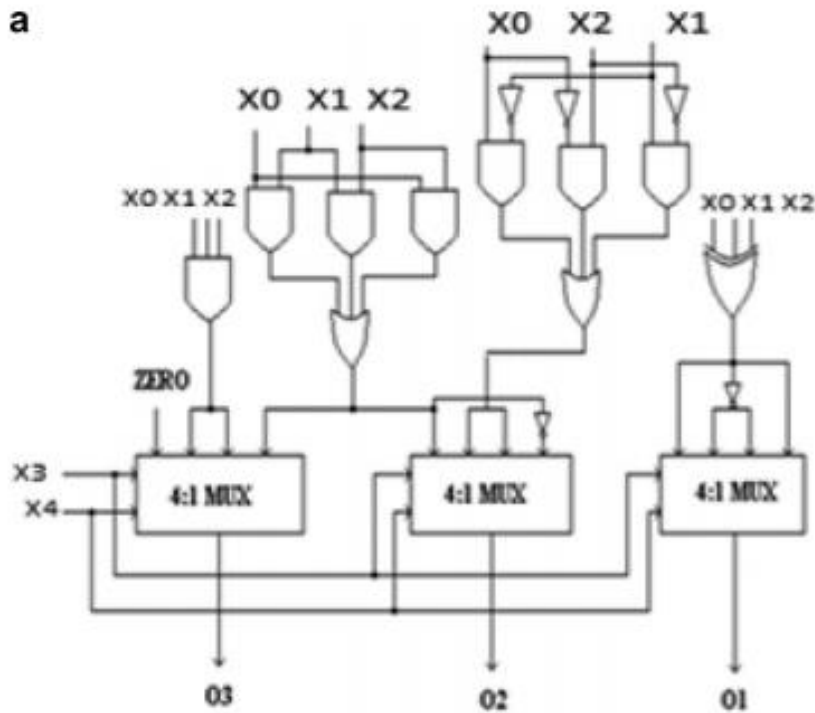


Fig 4.2.3(a) circuit diagram of 5-3 compressor adder using gates

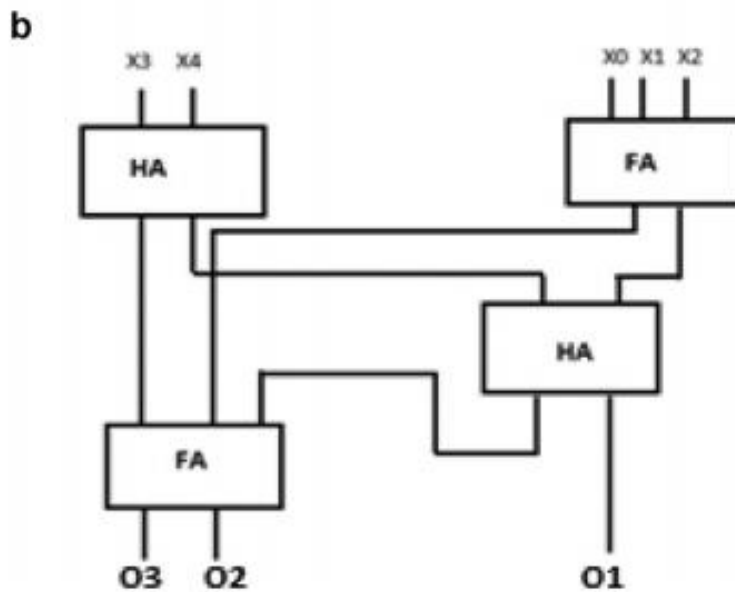


Fig 4.2.3(b) modified circuit diagram of 5-3 compressor adder

10-4 COMPRESSOR ADDER: Its circuitry takes ten inputs, adds them and gives four-bit output. The maximum resultant can be 1010. It makes use of two 5-3 compressors, two full adders and a half adder at the required position. Because of the use of modified 5-3 compressor circuitry, this compressor shows lesser delay and gate counts making the multiplier fast and ultimately the processor. The given below represents the modified circuitry of the 10-4 compressor.

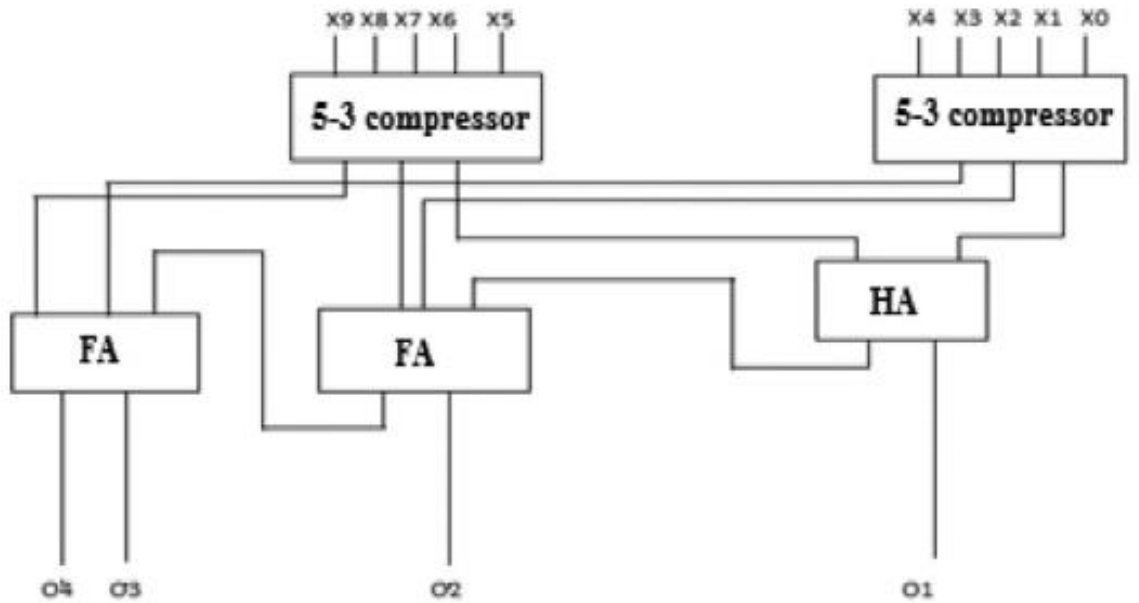


Fig 4.2.4 circuit diagram of 10-4 compressor adder

15-4 COMPRESSOR ADDER: Similar to the 5-3, 10-4 compressor adders, 15-4 compressors feed 15 bits at a time and give four-bit resultant which can go to extreme value of 1111. Its circuitry contains two 5-3 compressors, five full adders and one 4-bit parallel adder. The inputs are given in a group of three to the five full adders. Then the sum bits of all five full adder are added using one 5-3 compressor and the carry bits of the full adders are fed to another 5-3 compressor. Further a 4-bit parallel adder is used to add the outputs of these two 5-3 compressor and gives the final result. The given below represents the circuit of 15-4 compressor adder.

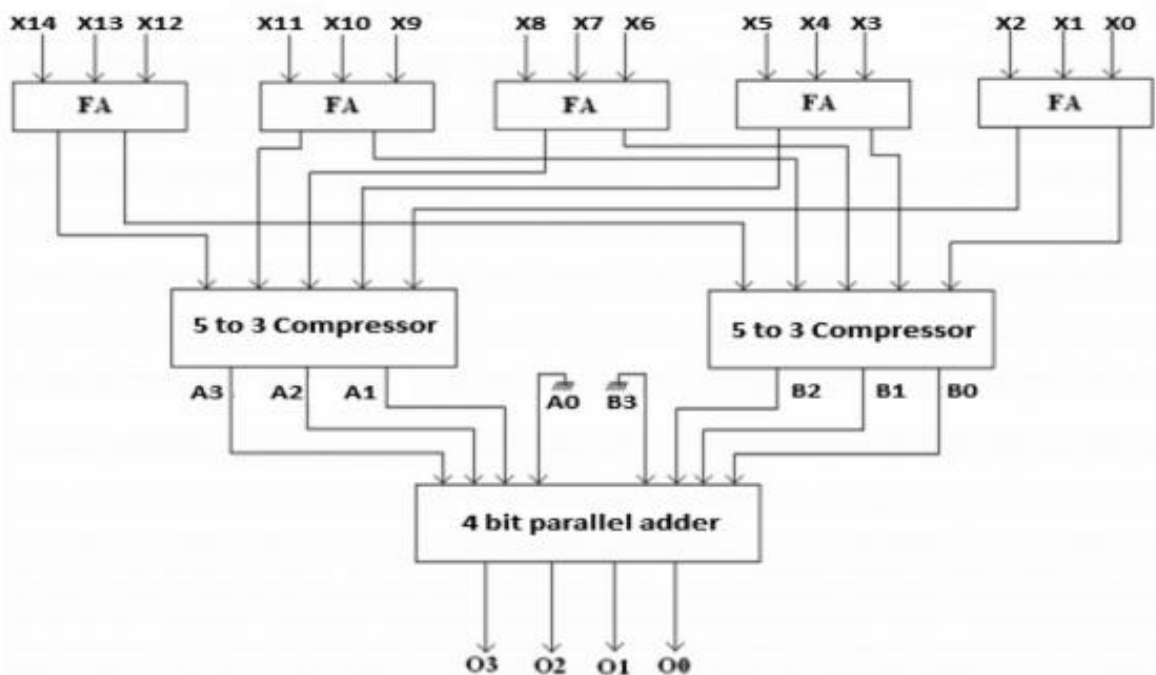


Fig 4.2.5 circuit diagram of 15-4 compressor adder

20-5 COMPRESSOR ADDER: In the proposed design we need to add 19 bits at the same time and so need to use a higher compressor adder circuit. In 20-5 compressor circuit, it converts 20 partial products into five output bits having maximum value of 10010. This makes use of one 15-4 compressor, one 5-3 compressor, two half-adders and two full adders. The improved architectures of lower compressors 15-4 and 5-3 bring speed improvement in its circuit in comparison with conventional architectures containing only full and half-adders. The 20-5 compressor adder circuit is shown in the figure

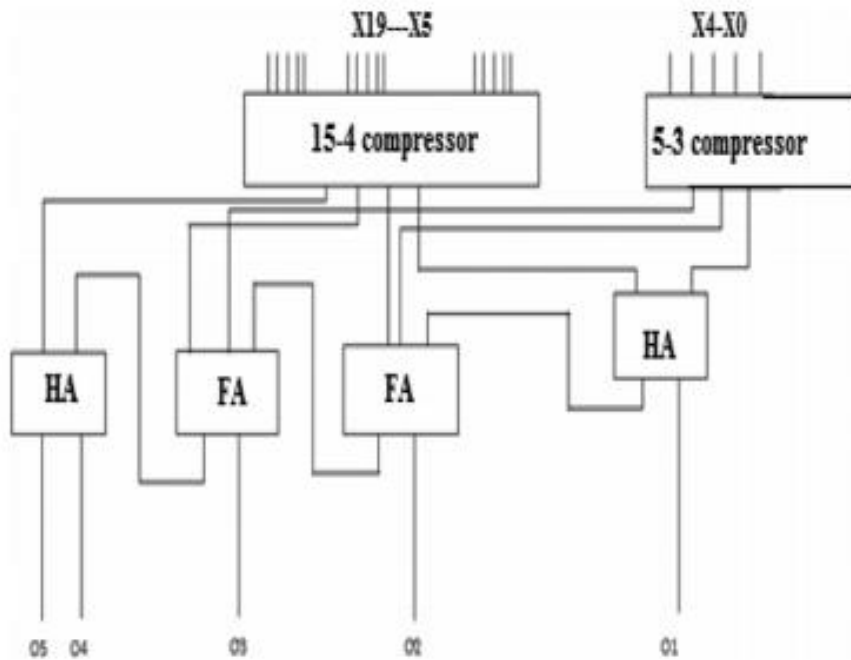


Fig 4.2.5 circuit diagram of 20-5 compressor adder

CHAPTER 5

INTRODUCTION TO VERILOG

5.1 DEFINITION:

Verilog is a **HARDWARE DESCRIPTION LANGUAGE (HDL)**, which is used to describe a digital system such as a network switch or a microprocessor or a memory a flip-flop. Verilog was developed to simplify the process and make the HDL more robust and flexible. Today, Verilog is the most popular HDL used and practiced throughout the semiconductor industry was developed to enhance the design process by allowing engineers to describe the desired hardware's functionality and let automation tools convert that behavior into actual hardware elements like combinational gates and sequential logic. Verilog is like any other hardware description language. It permits the designers to design the designs in either Bottom-up or Top-down methodology.

Bottom-Up Design:

The traditional method of electronic design is bottom-up. Each design is performed at the gate-level using the standards gates. This design gives a way to design new structural, hierarchical design methods.

Top-Down Design:

It allows early testing, easy change of different technologies, and structured system design and offers many other benefits.

5.2 HISTORY OF VERILOG:

Verilog HDL's history goes back to the 1980s when a company called Gateway Design Automation developed a logic simulator, Verilog-XL, and a hardware description language. Cadence Design Systems acquired Gateway in 1989 and with it the rights to the language and the simulator. In 1990, Cadence put the language into the public domain, with the intention that it should become a standard, non-proprietary language. The Verilog HDL is now maintained by a nonprofit making organization, Accellera, formed from the merger of Open Verilog International (OVI) and VHDL International. OVI had the task of taking the language through the IEEE standardization procedure. In December 1995, Verilog HDL became IEEE Std. 1364-1995. A significantly revised version was published in 2001: IEEE Std. 1364-2001.

There was a further revision in 2005, but this only added a few minor changes.

Accellera has also developed a new standard, System Verilog, which extends Verilog.

System Verilog became an IEEE standard (1800-2005) in 2005.

5.3 USES OF VERILOG:

Verilog creates a level of abstraction that helps hide away the details of its implementation and technology. For example, a D flip-flop design would require the knowledge of how the transistors need to be arranged to achieve a positive-edge triggered FF and what the rise, fall, and CLK-Q times required to latch the value onto a flop among much other technology-oriented details. Power dissipation, timing, and the ability to drive nets and other flocs would also require a more thorough understanding of a transistor's physical characteristics. Verilog helps us to focus on the behavior and leave the rest to be sorted out later.

5.4 FEATURES OF VERILOG:

- Verilog is case sensitive.
- In Verilog, Keywords are defined in lower case.
- In Verilog, most of the syntax is adopted from "C" language.
- Verilog can be used to model a digital circuit at Algorithm, RTL, Gate and Switch level.
- There is no concept of package in Verilog.
- It also supports advanced simulation features like TEXTIO, PLI, and UDPs.

5.5 DATA TYPES:

Verilog introduces several new data types. These data types make RTL descriptions easier to write and understand. The data storage and transmission elements found in digital hardware are represented using a set of Verilog Hardware Description Language (HDL) data types. In Verilog, data types are divided into NETS and Registers. These data types differ in the way that they are assigned and hold values, and also, they represent different hardware structures.

The Verilog HDL value set consists of four basic 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

**5.5.1
INTE
GER
AND
REAL**

DATA TYPES:

Many data types will be familiar to C programmers. The idea is that algorithms modeled in C can be converted to Verilog if the two languages have the same data types. Verilog introduces new two-state

data types, where each bit is 0 or 1 only. Using two-state variables in RTL models may enable simulators to be more efficient. And they are not affecting the synthesis results.

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

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

❖ F
our-
state
integer

types:

We preferred logic because it is better than reg. We can use logic where we have used reg or wire.

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

5.5.2 NON-INTEGER DATA TYPES:

❖ Arrays:

In Verilog, we can define scalar and vector nets and variables. We can also define memory arrays, which are one-dimensional arrays of a variable type. Verilog allowed multi-dimensioned arrays of both nets and variables and removed some of the restrictions on memory array usage. Verilog takes this a stage further and refines the concept of arrays and permits more operations on arrays. In Verilog, arrays may have either packed or unpacked dimensions, or both.

❖ **Packed dimensions:**

- Are guaranteed to be laid out contiguously in memory.
- It can be copied on to any other packed object.
- Can be sliced ("part-selects").
- Are restricted to the "bit" types (bit, logic, int, etc.), some of which (e.g., int) have a fixed size.

❖ **Unpacked dimensions:**

- It can be arranged in memory in any way that the simulator chooses. We can reliably copy an array on to another array of the same type.
- For arrays with different types, we have to use a cast, and there are rules for how an unpacked type is cast to a packed type.
- Verilog permits several operations on complete unpacked arrays and slices of unpacked arrays.
- For these, the arrays or slices involved must have the same type and shape, i.e., the same number and lengths of unpacked dimensions.
- The packed dimensions may differ, as long as the array or slice elements have the same number of bits.

The permitted operations are:

- Reading and writing the whole array.
- Reading and writing array slices.
- Reading and writing array elements.
- Equality relations on arrays, slices, and elements

Verilog also includes dynamic arrays (the number of elements may change during simulation) and associative arrays (which have a non-contiguous range). Verilog includes several arrays of querying functions and methods to support all these array types.

5.6 NETS:

Nets are used to connect between hardware entities like logic gates and hence do not store any value.

The net variables represent the physical connection between structural entities such as logic gates.

These variables do not store values except trireg. These variables have the value of their drivers, which changes continuously by the driving circuit. Some net data types are wire, tri, wor, trior, wand, triand, tri0, tri1, supply0, supply1, and trireg.

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

The output of some devices drives it.

It is declared as an input or in-out port.

On the left-hand side of a continuous assignment.

1. Wire:

A wire represents a physical wire in a circuit and is used to connect gates or modules. The value of a wire can be read, but not assigned to, in a function or block. A wire does not store its value but must be driven by a continuous assignment statement or by connecting it to the output of a gate or module.

2. Wand (wired-AND)

The value of a wand depends on logical AND of all the drivers connected to it.

3. Wor (wired-OR)

The value of wor depends on the logical OR of all the drivers connected to it.

4. Tri (three-state)

All drivers connected to a tri must be z, except one that determines the tri's value.

5. Supply0 and Supply1

Supply0 and supply1 define wires tied to logic 0 (ground) and logic 1 (power).

5.7 REGISTERS:

A register is a data object that stores its value from one procedural assignment to the next. They are used only in functions and procedural blocks. An assignment statement in a procedure acts as a trigger that changes the value of the data storage element.

Reg is a Verilog variable type and does not necessarily imply a physical register. In multi-bit registers, data is stored as unsigned numbers, and no sign extension is done for what the user might have thought were two's complement numbers. Some register data types are **reg**, **integer**, **time**, and **real**. **reg** is the most frequently used type. **Reg** is used for describing logic. An integer is general-purpose variables. They are used mainly loops-indices, parameters, and constants. They store data as signed numbers, whereas explicitly declared **reg** types store them as unsigned. If they hold numbers that are not defined at compile-time, their size will default to 32-bits. If they hold constants, the synthesizer adjusts them to the minimum width needed at compilation. **Real** in system modules. **Time** and **Realtime** for storing simulation times in test benches. **Time** is a 64-bit quantity that can be used in conjunction with the **\$time** system task to hold simulation time.

Note: A **reg** need not always represent a flip-flop because it can also represent combinational logic. The **reg** variables are initialized to x at the start of the simulation. Any wire variable not connected to anything has the x value. The size of a register or wire may be specified during the declaration. When the **reg** or wire size is more than one bit, then register and wire are declared vectors.

5.8 VERILOG STRING:

Strings are stored in reg, and the width of the reg variable has to be large enough to hold the string. Each character in a string represents an ASCII value and requires 1 byte. If the variable's size is smaller than the string, then Verilog truncates the leftmost bits of the string. If the variable's size is larger than the string, then Verilog adds zeros to the left of the string.

5.9 LEXICAL TOKENS:

Lexical conventions in Verilog are similar to the C programming language. Verilog language source text files are a stream of lexical tokens. A lexical token may consist of one or more characters, and every single character is in exactly one token.

The tokens can be **keywords, comments, numbers, white space, or strings**. All lines should be terminated by a semi-colon (;). Verilog HDL is a case-sensitive language. And all keywords are in lowercase.

White Space

White space can contain the characters for tabs, blanks, newlines, and form feeds. These characters are ignored except when they serve to separate other tokens. However, blanks and tabs are significant in strings.

Comments

There are two types to represent the comments, such as:

Single line comments begin with the token // and end with a carriage return.

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

Multi-Line comments begin with the token /* and end with the token */

For example, /* this is multiline syntax*/

Numbers

We can specify constant numbers in binary, decimal, hexadecimal, or octal format. Negative numbers are represented in 2's complement form. The question mark (?) character is the Verilog alternative for the z character when used in a number. The underscore character (_) is legal anywhere in a number, but it is ignored as the first character.

1. Integer Number

Verilog HDL allows integer numbers to be specified as:

- **Sized or unsized numbers** (Unsize size is 32 bits).
- In a radix of **decimal, hexadecimal, binary or octal**.
- Radix and hex digits (a,b,c,d) are case insensitive.

- Spaces are allowed between the radix, size, and value.

Syntax

The syntax is given as:

<size>'<radix><value>

2. Real Numbers

- Verilog supports real constants and variables.
- Verilog converts real numbers to integers by rounding.
- Real Numbers can not contain 'A' and 'X'.
- Real numbers may be specified in either decimal or scientific notation.

< value >.< value >

< mantissa >E< exponent >

- Real numbers are rounded off to the nearest integer when assigning to an integer.

3. Signed and Unsigned Numbers

Verilog supports both the type of numbers, but with certain restrictions. In C language, we don't have int and uint types to say if a number is signed integer or unsigned integer. Any number that does not have a negative sign prefix is positive. Or indirect way would be "Unsigned".

Negative numbers can be specified by putting a minus sign before the size for a constant number, thus become signed numbers. Verilog internally represents negative numbers in 2's complement format. An optional signed specifier can be added for signed arithmetic.

4. Negative Numbers

Negative numbers are specified by placing a minus (-) sign before the size of a number. It is illegal to have a minus sign between base format and number.

Identifiers

- The identifier is the name used to define the object, such as a function, module, or register. Identifiers should begin with alphabetical characters or underscore characters.
For example, A_Z and a_z.
- Identifiers are a combination of alphabetic, numeric, underscore, and \$ characters. They can be up to 1024 characters long.
- Identifiers must begin with an alphabetic character or the underscore character (a-z A-Z_).
- Identifiers may contain alphabetic characters, numeric characters, the underscore, and the dollar sign (a-z A-Z 0-9 _ \$).
- Identifiers can be up to 1024 characters long.

Escaped Identifiers

Verilog HDL allows any character to be used in an identifier by escaping the identifier. Escaped identifiers are including any of the printable ASCII characters in an identifier. The decimal values 33 through 126, or 21 through 7E in hexadecimal.

Escaped identifiers begin with the backslash (\). The backslash escapes the entire identifier. The escaped identifier is terminated by white space characters such as commas, parentheses, and semicolons become part of the escaped identifier unless preceded by white space. Terminate escaped identifiers with white space. Otherwise, characters that should follow the identifier are considered part of it.

5.10 OPERATORS

Operators are special characters used to put conditions or to operate the variables. There are one, two, and sometimes three characters used to perform operations on variables.

1. Arithmetic Operators

These operators perform arithmetic operations. The + and - are used as either unary (x) or binary (z-y) operators. The operators included in arithmetic operation are addition, subtraction, multiplication, division, and modulus.

2. Relational Operators

These operators compare two operands and return the result in a single bit, 1 or 0. The Operators included in relational operation are:

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

3. Bit-wise Operators

Bit-wise operators do a bit-by-bit comparison between two operands. The Operators included in Bit-wise operation are:

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

4. Logical Operators

Logical operators are bit-wise operators and are used only for single-bit operands. They return a single bit value, 0 or 1. They can work on integers or groups of bits, expressions and treat all non-zero values as 1. Logical operators are generally used in conditional statements since they work with expressions.

The operators included in Logical operation are:

!	logical NOT
&&	logical AND
	logical OR

5.Reduction Operators

Reduction operators are the unary form of the bitwise operators and operate on all the bits of an operand vector. These also return a single-bit value. The operators included in Reduction operation are:

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

6. Shift Operators

Shift operators are shifting the first operand by the number of bits specified by the second operand in the syntax. Vacant positions are filled with zeros for both directions, left and right shifts (There is no use sign extension). The Operators included in Shift operation are:

<<	shift left
>>	shift right

7. Concatenation Operator

The concatenation operator combines two or more operands to form a larger vector. The operator included in Concatenation operation is: { }

8. Replication Operator

The replication operator is making multiple copies of an item. The operator used in Replication operation is:

{n{item}} (n fold replication of an item)

9. Conditional Operator

Conditional operator synthesizes to a multiplexer. It is the same kind as is used in C/C++ and evaluates one of the two expressions based on the condition. The operator used in Conditional operation is:

(Condition)?:

5.11 OPERANDS

Operands are expressions or values on which an operator operates or works. All expressions have at least one operand.

1. Literals

Literals are constant-valued operands that are used in Verilog expressions. The two commonly used Verilog literals are:

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

Numeric: A constant number of the operand is specified in binary, octal, decimal, or hexadecimal number.

2. Wires, Regs, and Parameters

Wires, regs, and parameters are the data types used as operands in Verilog expressions. Bit-Selection "x [2]" and Part-Selection "x [4:2]"

Bit-selects and part-selects are used to select one bit and multiple bits, respectively, from a wire, regs or parameter vector using square brackets "["].

3. Function Calls

In the Function calls, the return value of a function is used directly in an expression without first

assigning it to a register or wire. It just places the function call as one of the types of operands. It is useful to know the bit width of the return value of the function call.

5.12 VERILOG MODULE:

A module is a block of Verilog code that implements certain functionality. Modules can be embedded within other modules, and a higher-level module can communicate with its lower-level modules using their input and output ports.

Syntax

A module should be enclosed within a module and end module keywords. The name of the module should be given right after the module keyword, and an optional list of ports may be declared as well.

Note: The ports declared in the list of port declarations cannot be re-declared within the module's body.

```
module <name> ([port_list]);
```

```
    // Contents of the module
```

```
end module
```

```
    // A module can have an empty port list
```

```
module name;
```

```
    // Contents of the module
```

```
end module
```

All variable declarations, functions, tasks, dataflow statements, and lower module instances must be defined within the module and end module keywords.

Purpose of a Module

A module represents a design unit that implements specific behavioral characteristics and will get converted into a digital circuit during synthesis. Any combination of inputs can be given to the module, and it will provide a corresponding output.

It allows the same module to be reused to form more significant modules that implement more complex hardware.

Hardware Schematic

Instead of building up smaller blocks to form bigger design blocks, the reverse process can also be done.

Consider the breakdown of a simple GPU engine into smaller components such that each can be

represented as a module that implements a specific feature

CHAPTER 6

XILINX SOFTWARE

The ISE® Design Suite controls all aspects of the design flow. Through the Project Navigator interface, you can access all of the design entry and design implementation tools. You can also access the files and documents associated with your project.

6.1 Project Navigator Interface

By default, the Project Navigator interface is divided into four panel sub-windows, as seen in Figure 4.1. On the top left are the Start, Design, Files, and Libraries panels, which include display and access to the source files in the project as well as access to running processes for the currently selected source. The Start panel provides quick access to opening projects as well as frequently access reference material, documentation and tutorials. At the bottom of the Project Navigator are the Console, Errors, and Warnings panels, which display status messages, errors, and warnings. To the right is a multi-document interface (MDI) window referred to as the Workspace. The Workspace enables you to view design reports, text files, schematics, and simulation waveforms. Each window can be resized, undocked from Project Navigator, moved to a new location within the main Project Navigator window, tiled, layered, or closed. You can use the **View > Panels** menu commands to open or close panels. You can use the **Layout > Load Default Layout** to restore the default window layout. These windows are discussed in more detail in the following sections.

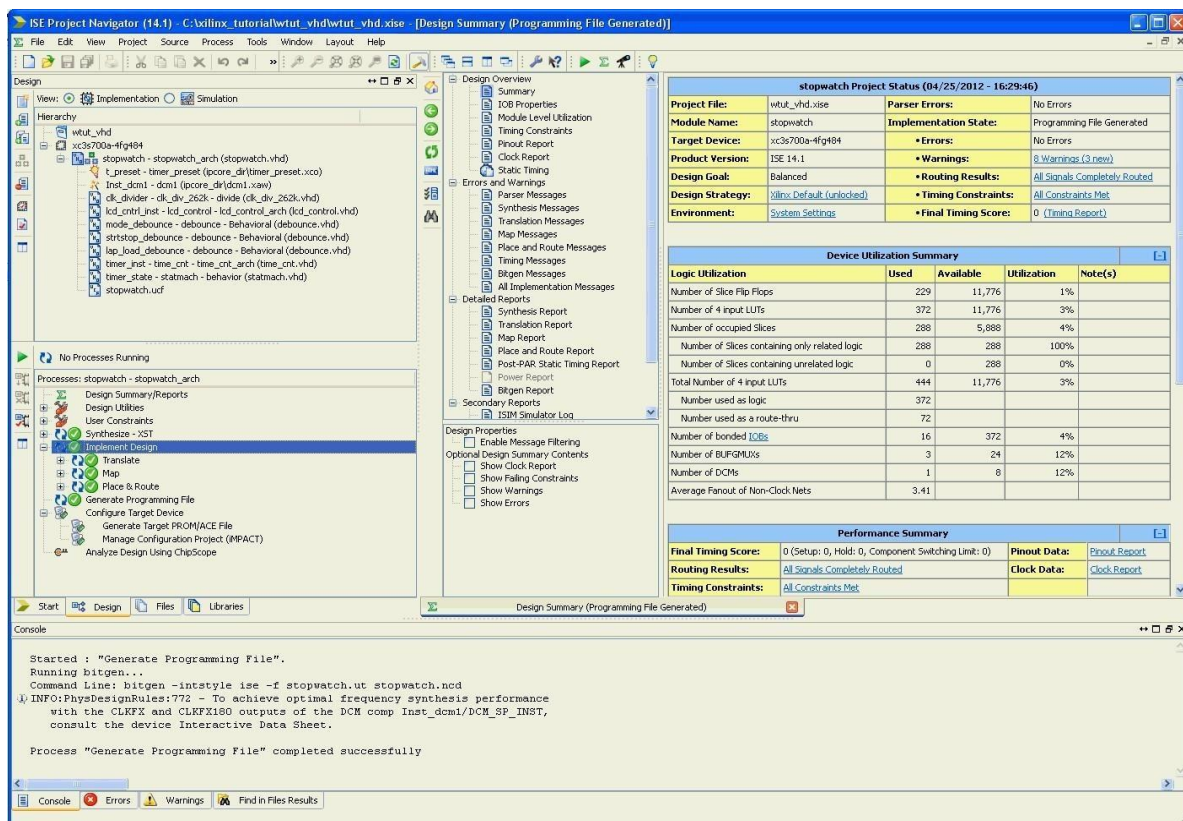


Figure 6.1: Project Navigator

Design Panel

The design panel provides access to the view, Hierarchy and process panes.

View Pane

The View pane radio buttons enable you to view the source modules associated with the Implementation or Simulation Design View in the Hierarchy pane. If you select Simulation, you must select a simulation phase from the drop-down

Hierarchy Pane

The Hierarchy pane displays the project name, the target device, user documents, and design source files associated with the selected Design View. The View pane at the top of the Design panel allows you to view only those source files associated with the selected Design View, such as Implementation or Simulation.

Each file in the Hierarchy pane has an associated icon. The icon indicates the file type (HDL file, schematic, core, or text file, for example). For a complete list of possible sources types and their associated icons, see the “Source File Types” topic in the ISE Help. From Project Navigator, select **Help > Help Topics** to view the ISE Help.

If a file contains lower levels of hierarchy, the icon has a plus symbol (+) to the left of the name. You can expand the hierarchy by clicking the plus symbol (+). You can open a file for editing by double-clicking on the filename.

Processes Pane

The Processes pane is context sensitive, and it changes based upon the source type selected in the Sources pane and the top-level source in your project. From the Processes pane, you can run the functions necessary to define, run, and analyze your design. The Processes pane provides access to the following functions:

- Design Summary/Reports

Provides access to design reports, messages, and summary of results data. Message filtering can also be performed.

- Design Utilities

Provides access to symbol generation, instantiation templates, viewing command line history, and simulation library compilation.

- User Constraints

Provides access to editing location and timing constraints.

- Synthesis

Provides access to Check Syntax, Synthesis, View RTL or Technology Schematic, and synthesis reports. Available processes vary depending on the synthesis tools you use.

- Implement Design

Provides access to implementation tools and post-implementation analysis tools.

- Generate Programming File

Provides access to bitstream generation.

Configure Target Device

Provides access to configuration tools for creating programming files and programming the device.

The Processes pane incorporates dependency management technology. The tools keep track of which processes have been run and which processes need to be run. Graphical status indicators display the state of the flow at any given time. When you select a process in the flow, the software automatically runs the processes necessary to get to the desired step. For example, when you run the Implement Design process, Project Navigator also runs the Synthesis process because implementation is dependent on up-to-date synthesis results.

To view a running log of command line arguments used on the current project, expand Design Utilities and select **View Command Line Log File**

Files Panel

The Files panel provides a flat, sortable list of all the source files in the project. Files can be sorted by any of the columns in the view. Properties for each file can be viewed and modified by right-clicking on the file and selecting **Source Properties**.

Libraries Panel

The libraries panel enables you to manage HDL libraries and their associated HDL source files. You can create, view, and edit libraries and their associated sources.

Console Panel

The Console provides all standard output from processes run from Project Navigator. It displays errors, warnings, and information messages. Errors are signified by a red X next to the message; while warnings have a yellow exclamation mark (!).

Errors Panel

The Errors panel displays only error messages. Other console messages are filtered out.

Warnings Panel

The Warnings panel displays only warning messages. Other console messages are filtered out.

Error Navigation to Source

You can navigate from a synthesis error or warning message in the Console, Errors, or Warnings panel to the location of the error in a source HDL file. To do so, select the error or warning message, right-click the mouse, and select **Go to Source** from the right-click menu. The HDL source file opens, and the cursor moves to the line with the error.

Error Navigation to Answer Record

You can navigate from an error or warning message in the Console, Errors, or Warnings panel to relevant Answer Records on the [Product Support and Documentation](#) page of the Xilinx® website. To navigate to the Answer Record, select the error or warning message, right-click the mouse, and select **Search for Answer Record** from the right-click menu. The default Web browser opens and displays all Answer Records applicable to this message.

Workspace

The Workspace is where design editors, viewers, and analysis tools open. These include ISE Text Editor, Schematic Editor, Constraint Editor, Design Summary/Report Viewer, RTL and Technology Viewers, and Timing Analyzer.

Other tools such as the PlanAhead™ tool for I/O planning and floorplanning, ISim, third-party text editors, XPower Analyzer, and iMPACT open in separate windows outside the main Project Navigator environment when invoked.

Design Summary/Report Viewer

The Design Summary provides a summary of key design data as well as access to all of the messages and detailed reports from the synthesis and implementation tools. The summary lists high-level information about your project, including overview information, a device utilization summary, performance data gathered from the Place and Route (PAR) report, constraints information, and summary information from all reports with links to the individual reports. A link to the System Settings report provides information on environment variables and tool settings used during the design implementation. Messaging features such as message filtering, tagging, and incremental messaging are also available from this view.

6.2 HDL Based Design:

Overview of HDL-Based Design

This chapter guides you through a typical HDL-based design procedure using a design of a runner's stopwatch. The design example used in this tutorial demonstrates many device features, software features, and design flow practices you can apply to your own design. This design targets a Spartan®-3A device; however, all of the principles and flows taught are applicable to any Xilinx® device family, unless otherwise noted.

The design is composed of HDL elements and two cores. You can synthesize the design using Xilinx Synthesis Technology (XST), Synplify/Synplify Pro, or Precision software

Required Software

To perform this tutorial, you must have Xilinx ISE® Design Suite installed.

This tutorial assumes that the software is installed in the default location `c:\xilinx\release_number\ISE_DS\ISE`. If you installed the software in a different location, substitute your installation path in the procedures that follow.

Note: For detailed software installation instructions, refer to the *Xilinx Design Tools: Installation and Licensing Guide (UG798)* available from the Xilinx website

VHDL:

This software supports both VHDL and Verilog designs and applies to both designs simultaneously, noting differences where applicable. You will need to decide which HDL language you would like to work through for the tutorial and download the appropriate files for that language. XST can synthesize a mixed-language design. However, this tutorial does not cover the mixed language feature.

Starting the ISE Design Suite

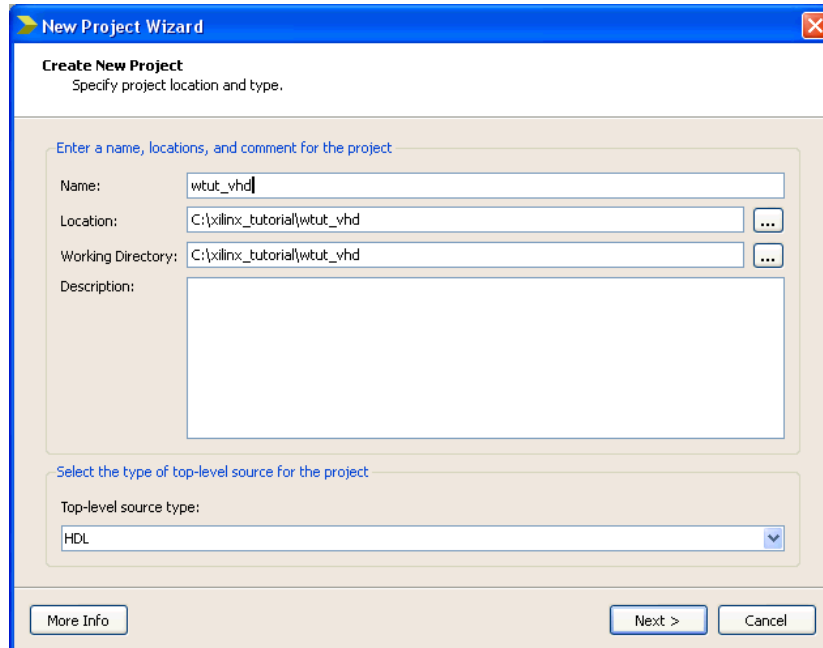
To start the ISE Design Suite, double-click the Project Navigator icon on your desktop, or select **Start > All Programs > Xilinx ISE Design Suite > Xilinx Design Suite 14 > ISE Design Tools > Project Navigator**.



Figure 6.2: Project navigator Desktop

Creating a New Project

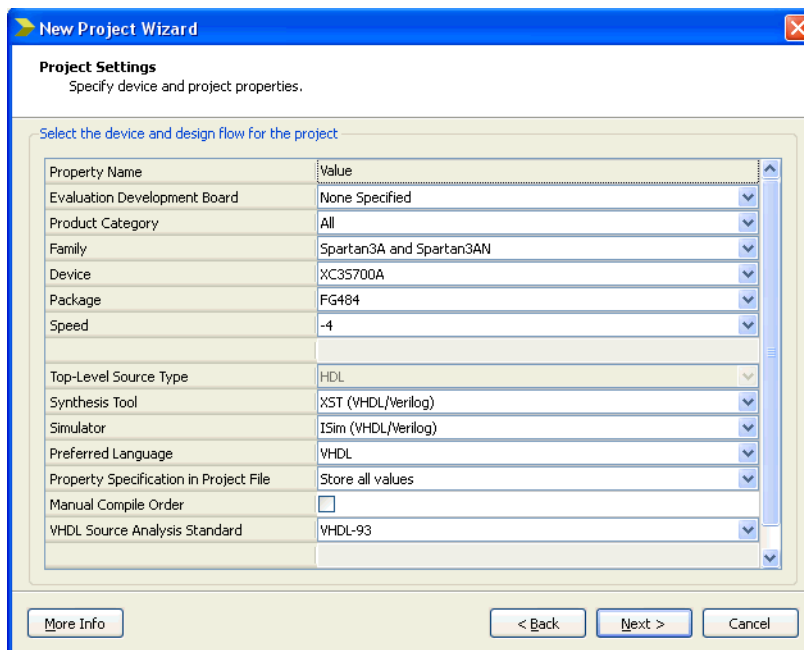
To create a new project using the New Project Wizard, do the following:



From Project Navigator, select **File > New Project**. The New Project Wizard appears.

Figure 6.3 New project Wizard- Create New Project Page

1. In the Location field, browse to c:\xilinx_tutorial or to the directory in which you installed the project.
2. In the Name field, enter **wtut_vhd** or **wtut_ver**.
3. Verify that **HDL** is selected as the Top- Level Source Type, and click **Next**
4. The New Project Wizard—Device Properties page appears.



- 1.
5. Figure 6.4: New Project Wizard- Device Properties Page
 1. Select the following values in the New Project Wizard—Device Properties page:
 2. Product Category: **All**
 3. Family: **Spartan3A and Spartan3AN**
 4. Device: **XC3S700A**

5. Package: **FG484**
6. Speed: **-4**
7. Synthesis Tool: **XST (VHDL/Verilog)**
8. Simulator: **ISim (VHDL/Verilog)**
9. Preferred Language: **VHDL** or **Verilog** depending on preference. This will determine the default language for all processes that generate HDLfiles.
10. Other properties can be left at their default values.
11. Click **Next**, then **Finish** to complete the project creation.
- 12.

13. Stopping the Tutorial

14. You may stop the tutorial at any time and save your work by selecting **File > Save All**.

15. Design Description

16. The design used in this tutorial is a hierarchical, HDL-based design, which means that the top-level design file is an HDL file that references several other lower-level macros. The lower-level macros are either HDL modules or IP modules.

17. The design begins as an unfinished design. Throughout the tutorial, you will complete the design by generating some of the modules from scratch and by completing others from existing files. When the design is complete, you will simulate it to verify the design functionality.

In the runner's stopwatch design, there are five external inputs and four external output buses. The system clock is an externally generated signal. The following list summarizes the input and output signals of the design.

INPUT

The following are input signals for the tutorial stopwatch design:

1. strtstop

Starts and stops the stopwatch. This is an active low signal which acts like the start/ stop button on a runner's stopwatch.

1. reset

Puts the stopwatch in clocking mode and resets the time to 0:00:00.

2. clk

Externally generated system clock.

3. mode

Toggles between clocking and timer modes. This input is only functional while the clock or timer is not counting.

4. lap_load

This is a dual function signal. In clocking mode, it displays the current clock value in the 'Lap' display area. In timer mode, it loads the pre-assigned values from the ROM to the timer display when the timer is not counting.

OUTPUT

The following are outputs signals for the design:

1. lcd_e, lcd_rs, lcd_rw

These outputs are the control signals for the LCD display of the Spartan-3A demo board used to display the stopwatch times.

1. sf_d[7:0]

Provides the data values for the LCD display.

Functional Blocks

The completed design consists of the following functional blocks:

1. clk_div_262k

Macro that divides a clock frequency by 262,144. Converts 26.2144 MHz clock into 100 Hz 50% duty cycle clock.

2. dcm1

Clocking Wizard macro with internal feedback, frequency-controlled output, and duty-cycle correction. The CLKFX_OUT output converts the 50 MHz clock of the Spartan-3A demo board to 26.2144 MHz.

3. debounce

Schematic module implementing a simplistic debounce circuit for the strtstop, mode, and lap_load input signals

1. lcd_control

Module controlling the initialization of and output to the LCD display.

2. statmach

State machine HDL module that controls the state of the stopwatch.

3. timer_preset

CORE Generator™ tool 64x20 ROM. This macro contains 64 preset times from 0:00:00 to 9:59:99 that can be loaded into the timer.

4. time_cnt

Up/down counter module that counts between 0:00:00 to 9:59:99 decimal. This macro has five 4-bit outputs, which represent the digits of the stopwatch time.

Synthesizing the Design

So far you have been using Xilinx Synthesis Technology (XST) for syntax checking. Next, you will synthesize the design using either XST, Synplify/Synplify Pro, or Precision software. The synthesis tool uses the design's HDL code and generates a supported netlist type (EDIF or NGC) for the Xilinx implementation tools. The synthesis tool performs the following general steps (although all synthesis tools further break down these general steps) to create the netlist:

Analyze/Check Syntax

Checks the syntax of the source code.

2. Compile

Translates and optimizes the HDL code into a set of components that the synthesis tool can recognize.

3. Map

Translates the components from the compile stage into the target technology's primitive components.

The synthesis tool can be changed at any time during the design flow. To change the synthesis tool, do the following:

1. In the Hierarchy pane of the Project Navigator Design panel, select the targeted part.
2. Right-click and select **Design Properties**.
3. In the Design Properties dialog box, click the Synthesis Tool value and use the pull-down arrow to select the desired synthesis tool from the list

Note: If you do not see your synthesis tool among the options in the list, you may not have the software installed or may not have it configured in the ISE Design Suite. The synthesis tools are configured in the Preferences dialog box. Select **Edit > Preferences**, expand **ISE General**, and click **Integrated Tools**.

Changing the design flow results in the deletion of implementation data. You have not yet created any implementation data in this tutorial. For projects that contain implementation data, Xilinx recommends that you make a copy of the project using **File > Copy Project** if you would like to make a backup of the project before continuing.

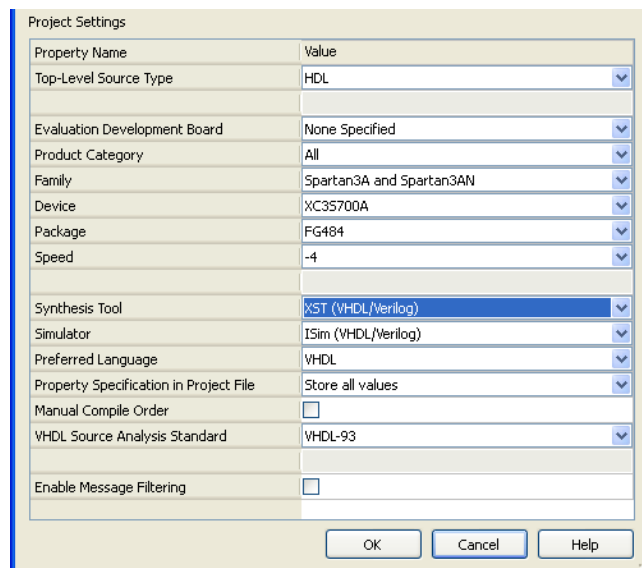


Figure 6.5: Specifying Synthesis Tool Synthesizing the Design Using XST

Now that you have created and analyzed the design, the next step is to synthesize the design. During synthesis, the HDL files are translated into gates and optimized for the target architecture.

Processes available for synthesis using XST are as follows:

- View RTL Schematic

Generates a schematic view of your RTL netlist.

- View Technology Schematic

Generates a schematic view of your technology netlist.

- Check Syntax

Verifies that the HDL code is entered properly.

- Generate Post-Synthesis Simulation Model

Creates HDL simulation models based on the synthesis netlist. Entering

Synthesis Options

Synthesis options enable you to modify the behavior of the synthesis tool to make optimizations according to the needs of the design. One commonly used option is to control synthesis to make optimizations based on area or speed. Other options include controlling the maximum fanout of a flip-flop output or setting the desired frequency of the design.

To enter synthesis options, do the following:

1. In the Hierarchy pane of the Project Navigator Design panel, select stopwatch.vhd (Or stopwatch.v).
2. In the Processes pane, right-click the **Synthesize** process, and select **Process Properties**.
3. Under the Synthesis Options tab, set the Netlist Hierarchy property to a value of Rebuilt.

Note: To use this property, you must set the Property display level to **Advanced**.

4. Click **OK**.

6.4 Synthesizing the Design

Now you are ready to synthesize your design. To take the HDL code and generate a compatible netlist, do the following:

1. **In the Hierarchy pane, select stopwatch.vhd (or stopwatch.v). In the Processes pane, double-click the Synthesize process *Using the RTL/Technology***

Viewer

XST can generate a schematic representation of the HDL code that you have entered. A schematic view of the code helps you analyze your design by displaying a graphical connection between the various components that XST has inferred. Following are the two forms of schematic representation:

- RTL View

Pre-optimization of the HDL code.

- Technology View

Post-synthesis view of the HDL design mapped to the target technology. To view a schematic representation of your HDL code, do the following:

1. In the Processes pane, expand **Synthesize**, and double-click **View RTL Schematic** or **View Technology Schematic**.
2. If the Set RTL/Tech Viewer Startup Mode dialog appears, select **Start with the Explorer Wizard**.
3. In the Create Schematic start page, select the **clk_divider** and **lap_load_debounce** components from the Available Elements list, and then click the **Add** button to move the selected items to the Selected Elements list.
4. Click **Create Schematic**.

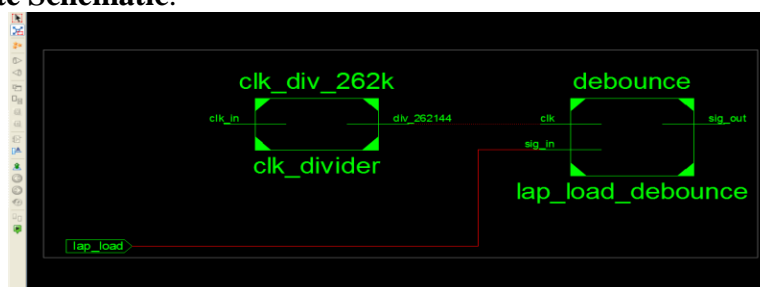


Fig 6.6: RTL Schematic

The schematic viewer allows you to select the portions of the design to display as schematics. When the schematic is displayed, double-click on the symbol to push into the schematic and

view the various design elements and connectivity. Right-click the schematic to view the various operations that can be performed in the schematic viewer.

CHAPTER 7

SIMULATIONS AND RESULTS

In this chapter various compressor adders like 5-3,10-4,15-4,20-5 compressor adders and 16-bit Vedic Multiplier using compressor adders are simulated using Xilinx Vivado 2020.2 design suite and results are presented and also performance comparison of the compressor adders in terms of area and power utilization are presented.

7.1 SYNTHESIS RESULTS OF 5: 3 COMPRESSOR ADDER:

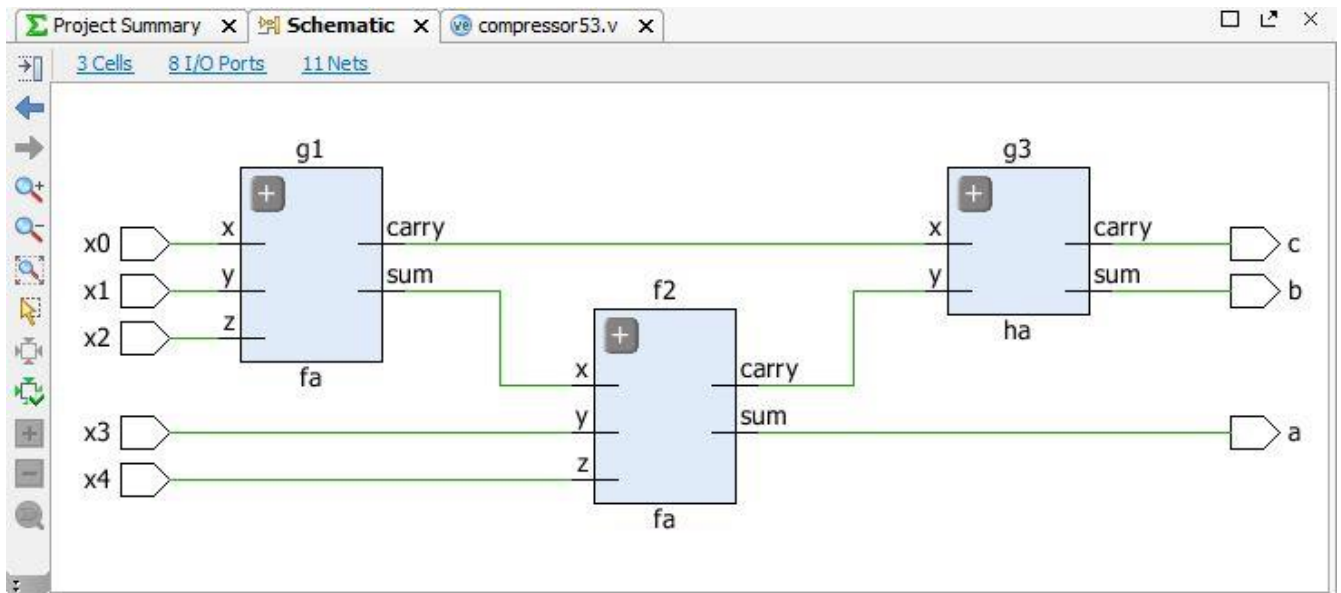


Fig 7.1(a) 5-3 Compressor RTL schematic diagram.

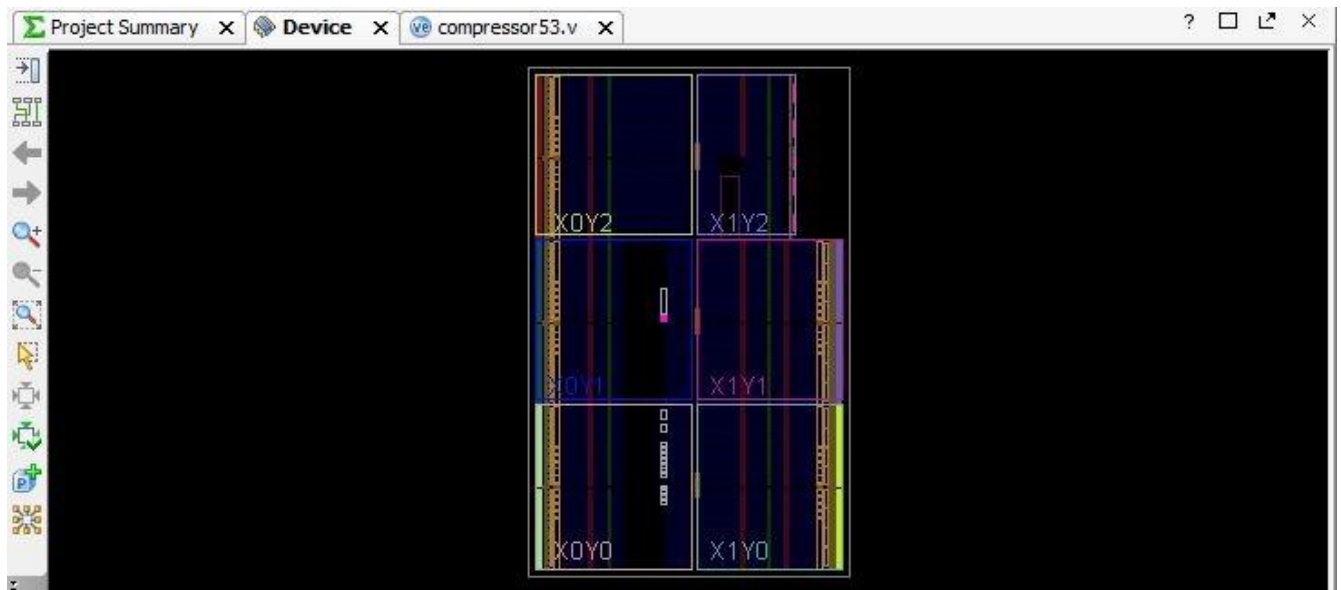


Fig 7.1(b) 5-3 Compressor Synthesis Diagram

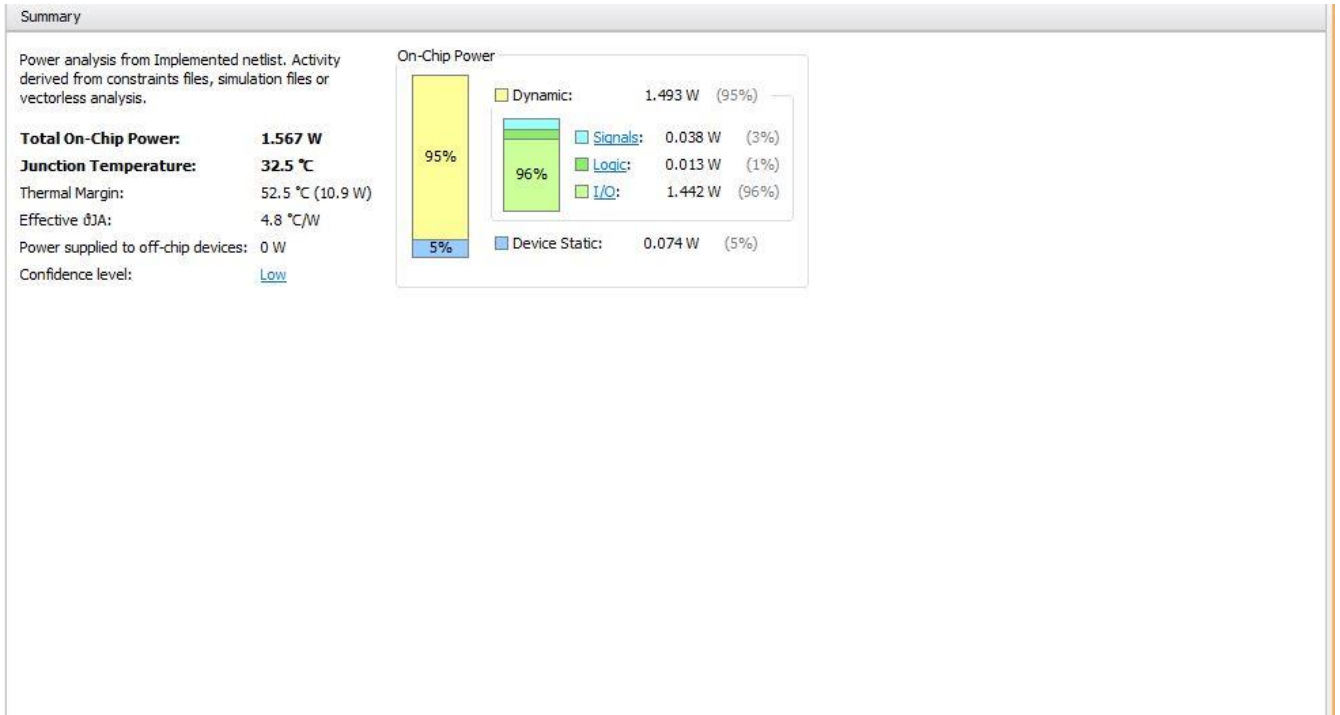


Fig 7.1(c) 5-3 Compressor Power Report

Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Failed Routes	LUT	FF	BRAM	URAM	DSP	Start	Elapsed
synth_1	constrs_1	synth_design Complete!							2	0	0	0	0	1/1/05 12:34 AM	00:00:3
impl_1	constrs_1	route_design Complete!	NA	NA	NA	NA	NA		0	2	0	0	0	1/1/05 12:36 AM	00:02:1

Fig 7.1(d) 5-3 Compressor Area Utilization Report

7.2 SYNTHESIS RESULTS OF 10: 4 COMPRESSOR ADDER:

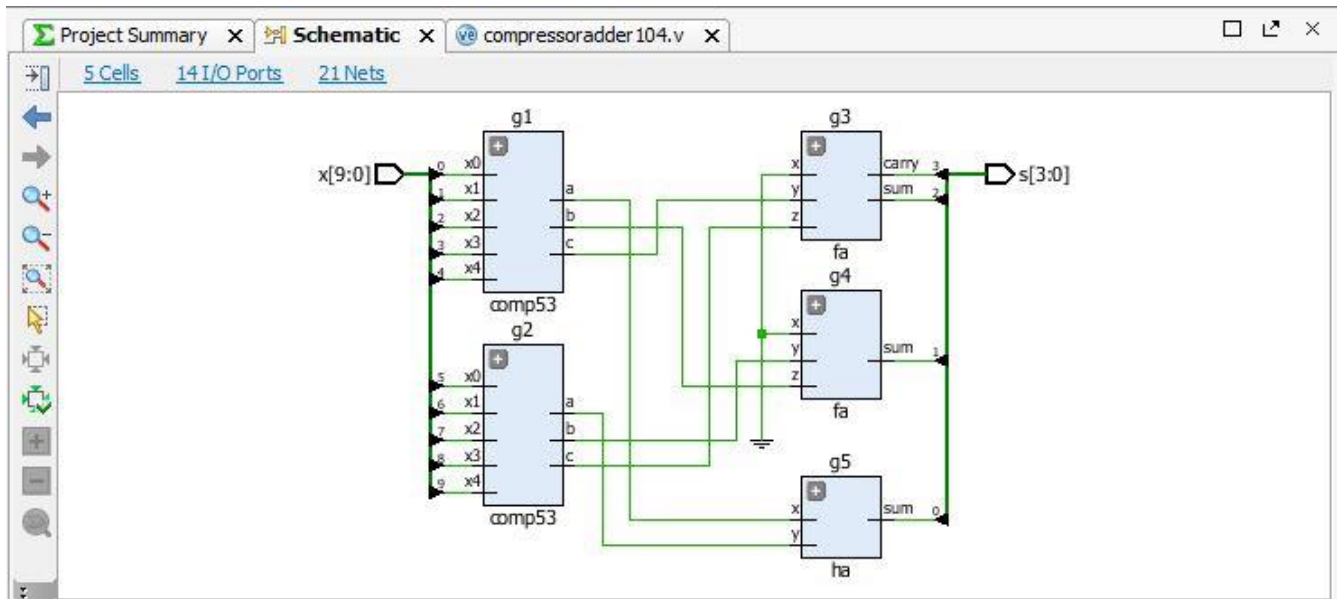


Fig 7.2(a) 10-4 Compressor RTL Schematic Diagram

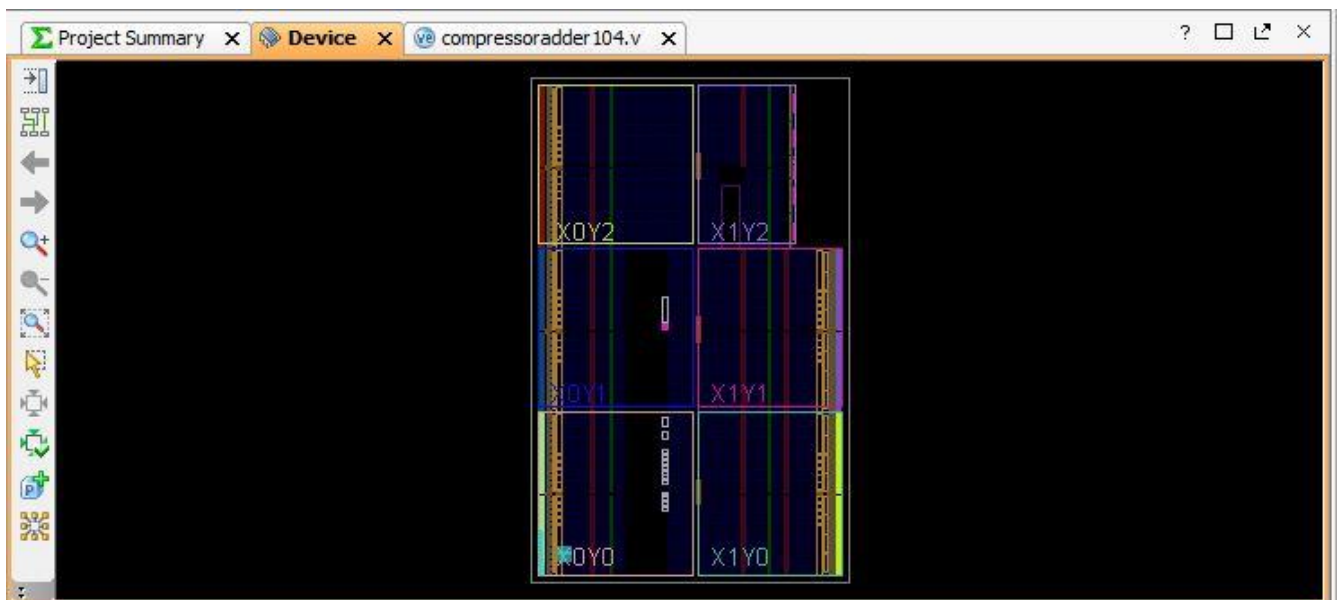


Fig 7.2(b) 10-4 Compressor Synthesis Diagram

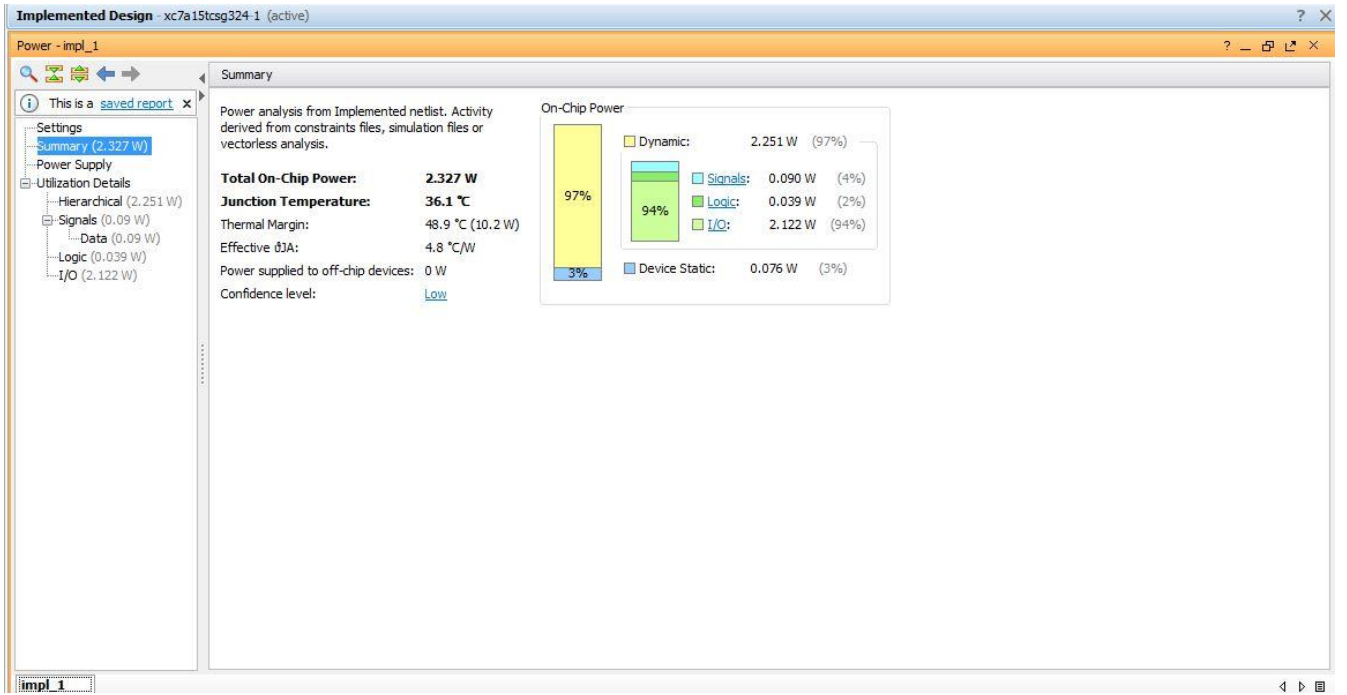


Fig 7.2(c) 10-4 Compressor Power Report

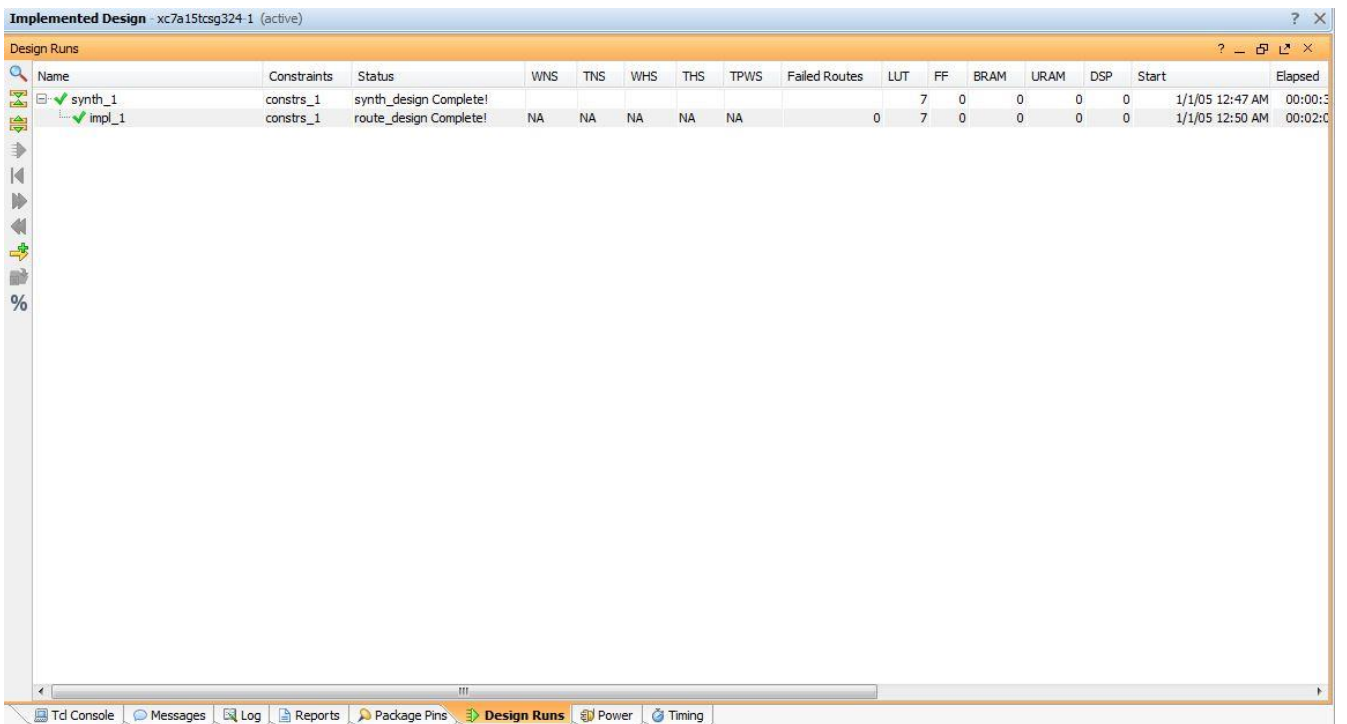


Fig 7.2(d) Compressor Area Utilization Report

7.3 SYNTHESIS RESULTS OF 15: 4 COMPRESSOR ADDER:

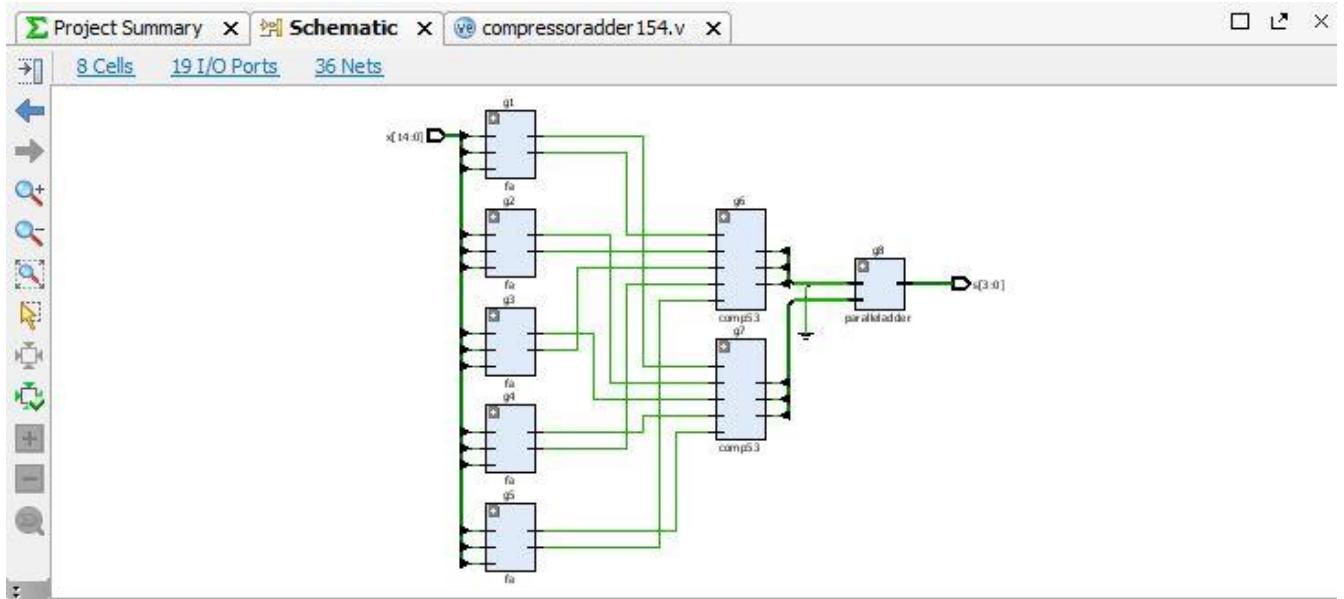


Fig 7.3(a) 15-4 Compressor Schematic Diagram



Fig 7.3(b) 15-4 Compressor Synthesis Diagram

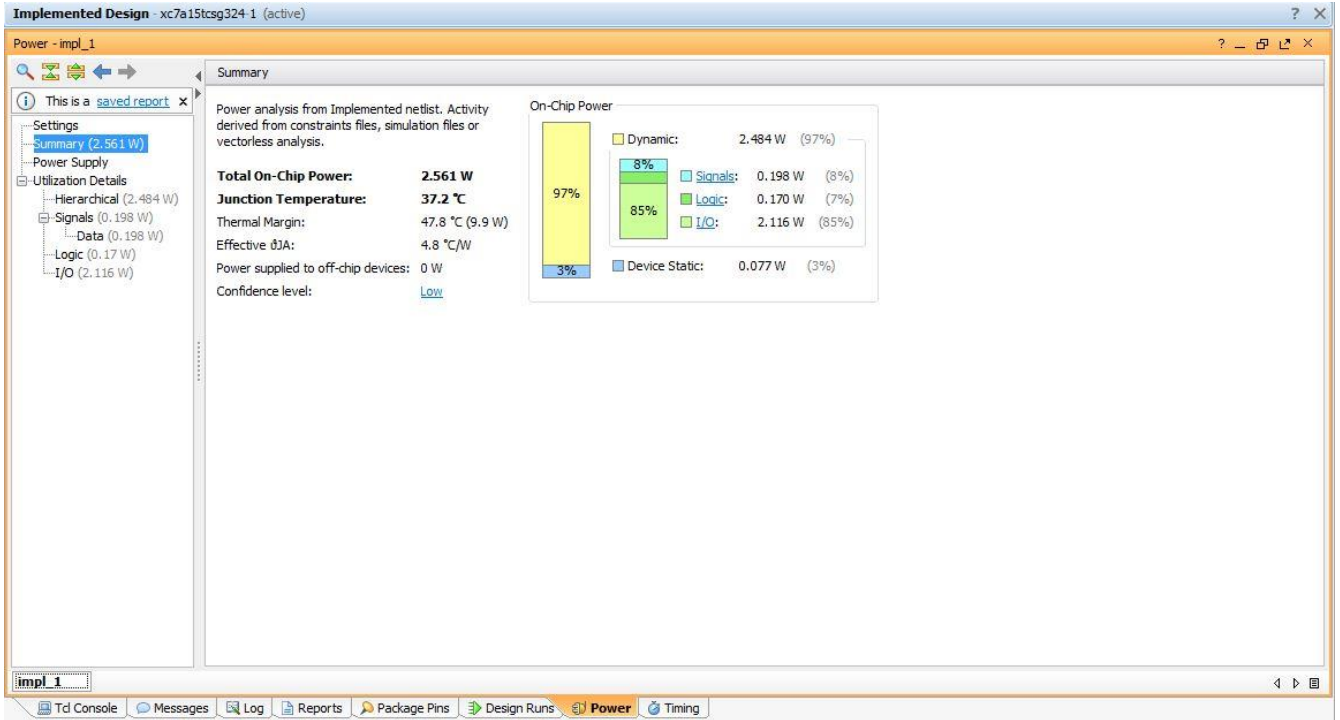


Fig 7.3(c) 15-4 Compressor Power Report

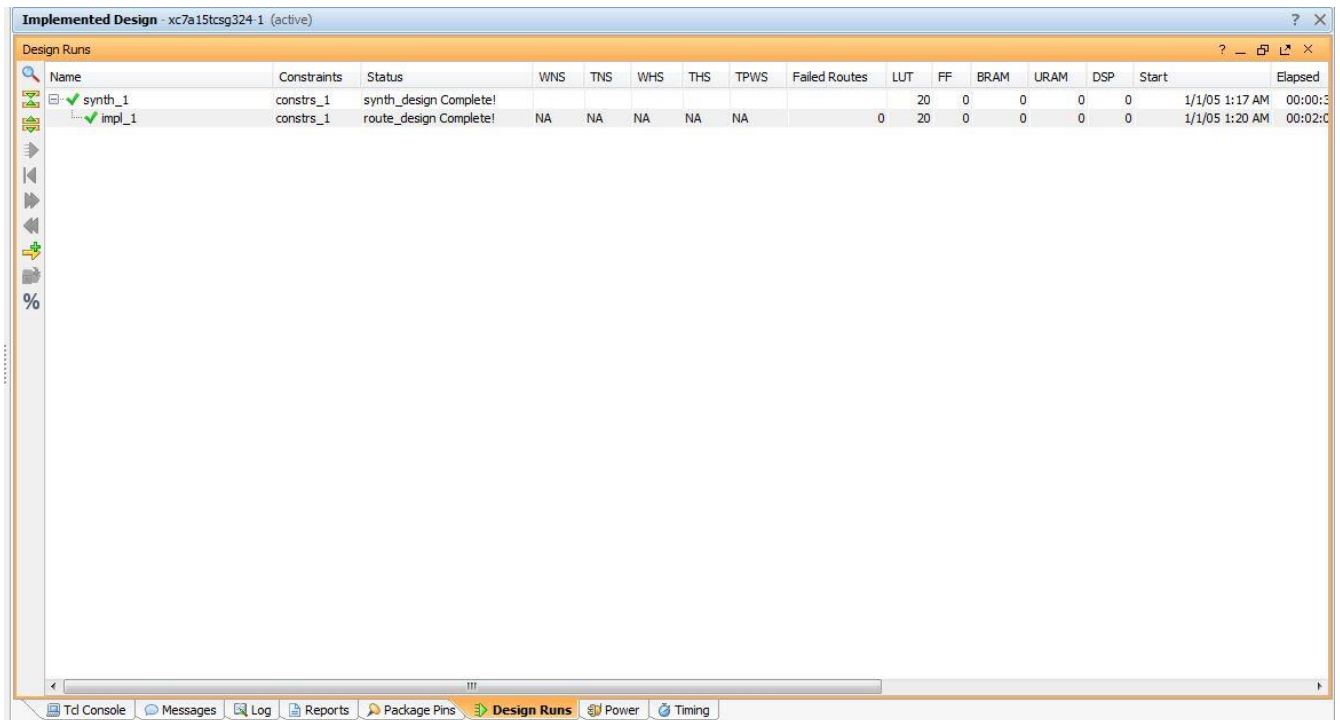


Fig 7.3(d) 15-4 Compressor Area Utilization Report

7.4 SYNTHESIS RESULTS OF 20: 5 COMPRESSOR ADDER:

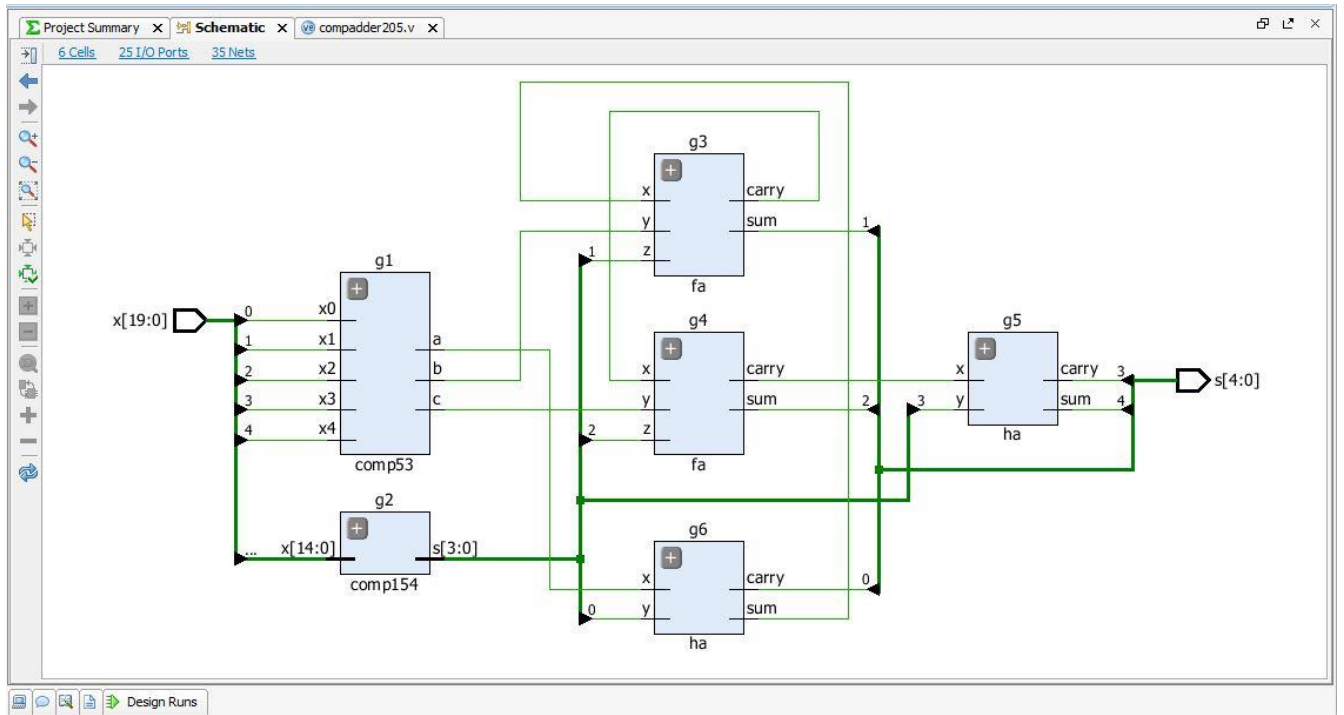


Fig 7.4(a) 20-5 Compressor RTL Schematic Diagram

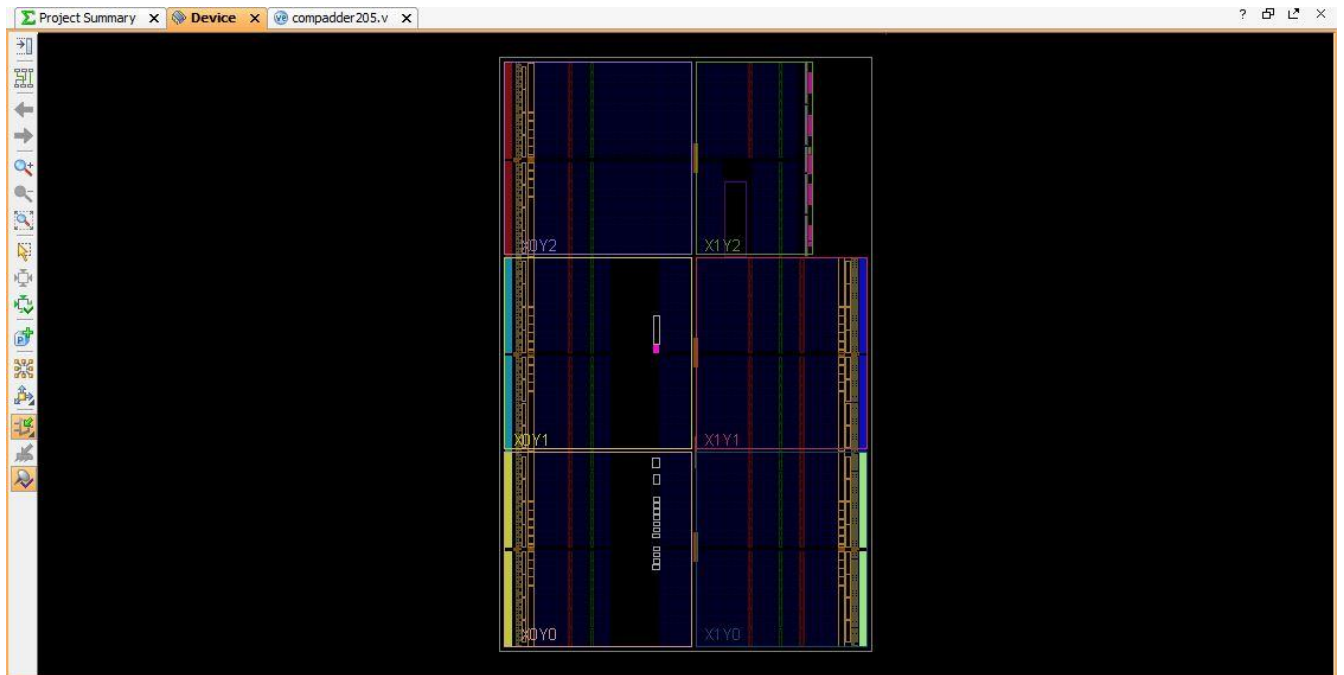


Fig 7.4(b) 20-5 Compressor Synthesis Diagram

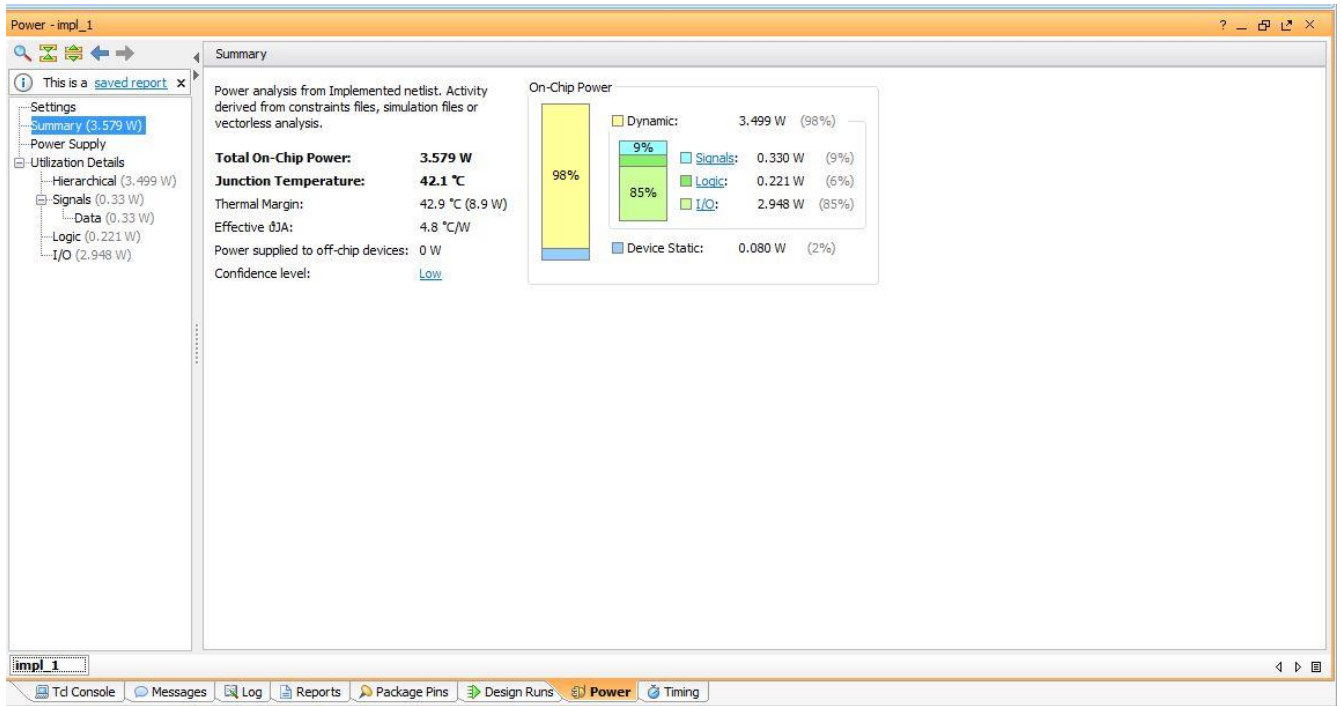


Fig 7.4(c) 20-5 Compressor Power Report

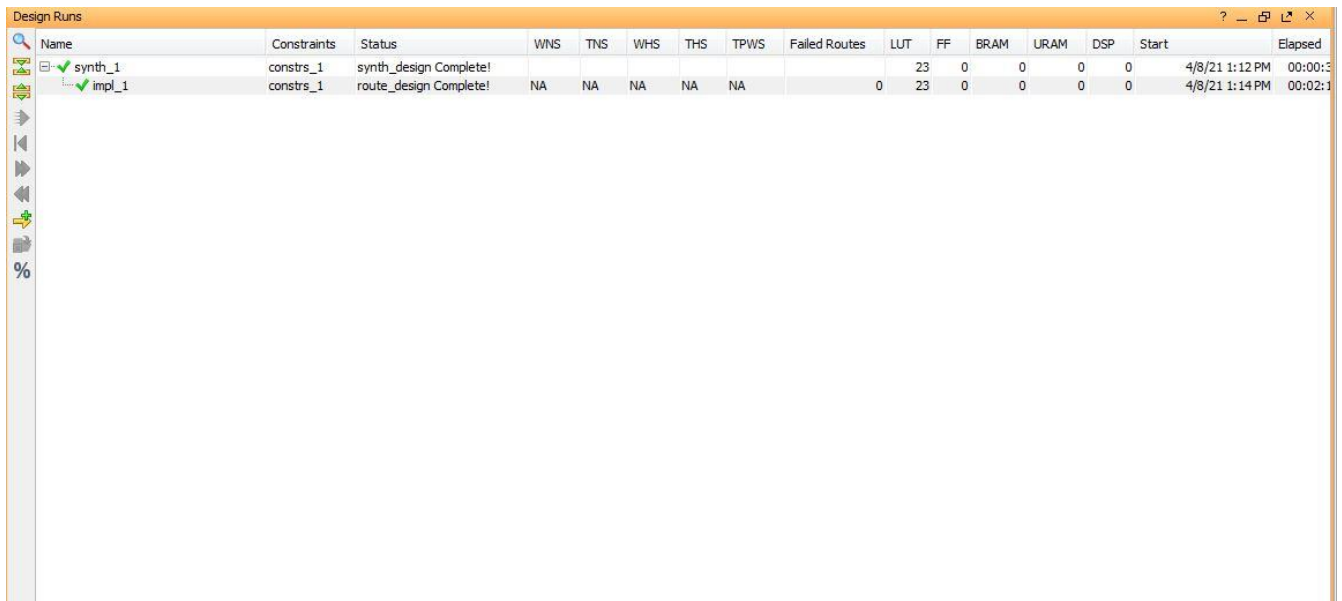


Fig 7.4(d) 20-5 Compressor Area Utilization Report

Table 1: Comparison of Area Utilization and Power of different Compressor Adders

Compressor Adders	Area Utilization	Power (W)
5-3	2	1.567
10-4	7	2.327
15-4	20	2.561
20-5	23	3.579

Proposed Work

In this project, we have analyzed 16-bit Vedic multiplier using higher order compressors. Figure 7.5 depicts that how partial products are grouped to compressor adders for addition.

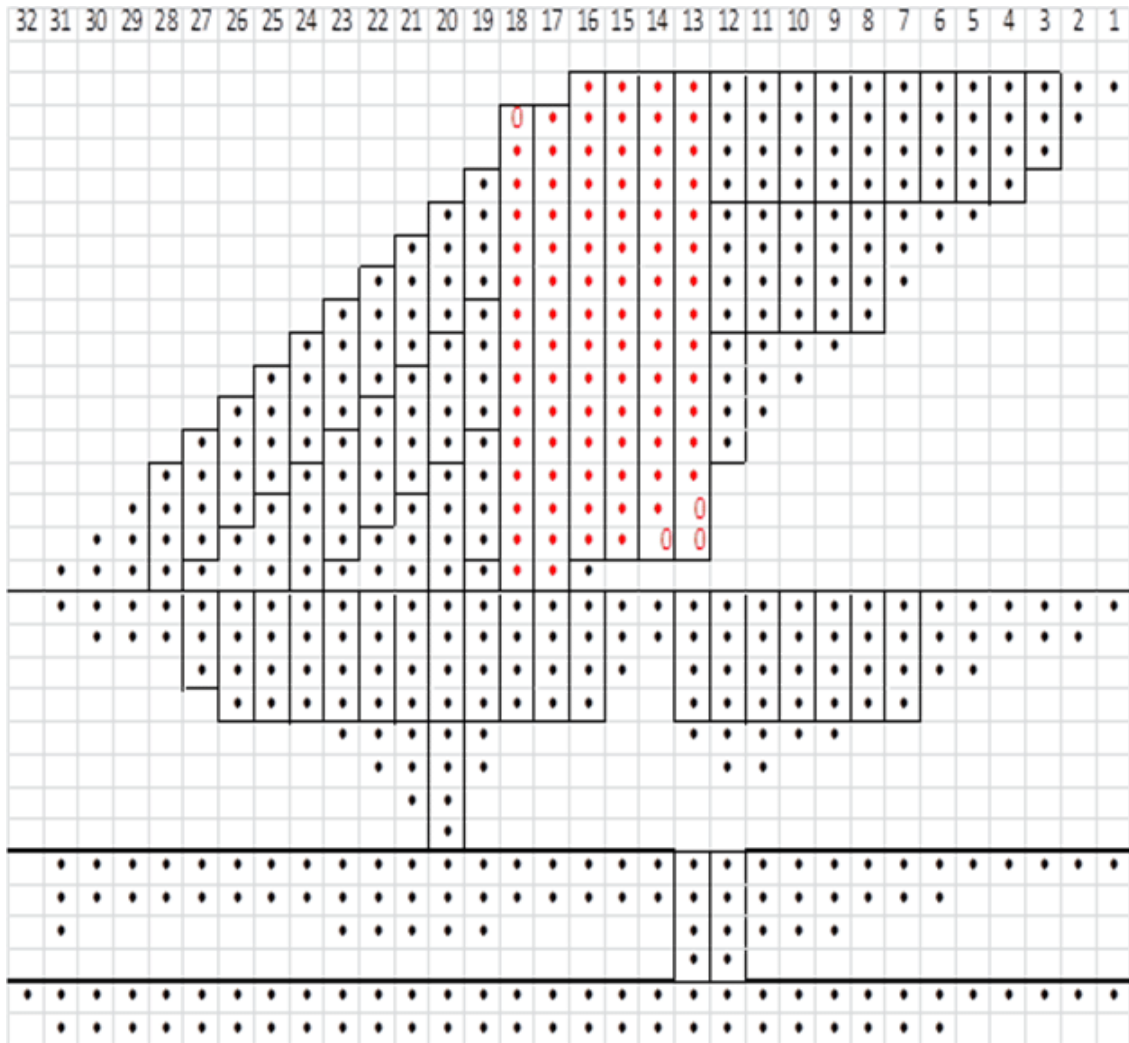


Fig 7.5: Compressor based 16-bit multiplier

Simulation results of 16-bit Vedic Multiplier:

Device Utilization Summary				
Logic Utilization	Used	Available	Utilization	Note(s)
Number of 4 input LUTs	562	1,920	29%	
Number of occupied Slices	472	960	49%	
Number of Slices containing only related logic	472	472	100%	
Number of Slices containing unrelated logic	0	472	0%	
Total Number of 4 input LUTs	562	1,920	29%	
Number of bonded IOBs	48	66	72%	
Average Fanout of Non-Clock Nets	4.09			

Fig 7.6: Area analysis of 16-bit VM using regular adders

Device Utilization Summary				
Logic Utilization	Used	Available	Utilization	Note(s)
Number of 4 input LUTs	398	1,920	35%	
Number of occupied Slices	327	960	41%	
Number of Slices containing only related logic	327	392	100%	
Number of Slices containing unrelated logic	0	392	0%	
Total Number of 4 input LUTs	398	1,920	35%	
Number of bonded IOBs	48	66	72%	
Average Fanout of Non-Clock Nets	3.12			

Fig 7.7: Area analysis of 16-bit VM using compressor adders

CONCLUSION

In digital signal processing multiplication is a key operation which determines the overall performance of the multiplier. Vedic multiplier is used as the multiplication for 16-bit operands, is explained. Architecture is proposed with the sutra, 'Urdhava-tiryakbhyam', from Vedic Mathematics which is a basic multiplication method for multiplication. The unwanted multiplication steps are removed and makes the parallel generation of partial products with the help of this sutra. Increase in area and delay is less with increase in number of bits. The designed architecture involves two steps. First is the computing each bit's resultant equation. Second, with the help of compressor adders executing of equation takes place. The compressor adders result in reduction of delay by adding 4 bits at a time which uses multiplexers in their circuit in reducing the operations of XOR Gate. Speed improvement is seen in compressor adders 5-3, 10-4, 15-4, and 20-5. Finally, we can justify that Vedic multiplier that are compressor dependent proves to have a convenient and useful way over conventional multipliers in the VLSI circuits.

References

1. Saokar SS, Banakar RM, Siddamal S. High-speed signed multiplier for digital signal processing applications. In: Proceedings of signal processing, computing and control (ISPCC); 2012. p. 1–6. doi:10.1109/ISPCC.2012.6224373.
2. Kumar A, Raman A. Low power ALU design by ancient mathematics. In: Proceedings of IEEE international conference on aerospace and aviation engineering (ICAAE); 2010. p. 862–5.
3. Hanumantharaju MC, Jayalaxmi H, Renuka RK, Ravishankar M. A high-speed block convolution using ancient indian Vedic mathematics. In: Proceedings of IEEE conference on computational intelligence and multimedia applications (ICCIMA); 2007. p. 169–73. doi:10.1109/ICCIMA.2007.332.
4. Prakash AR, Kirubaveni S. Performance evaluation of FFT processor using conventional and Vedic algorithm. In: Proceedings of IEEE conference on emerging trends in computing, communication and nanotechnology (ICE-CCN); 2013. p. 89–94. doi:10.1109/ICE-CCN.2013.6528470.
5. Saha P, Banerjee A, Dandapat A, Bhattacharyya P. ASIC design of a high-speed low power circuit for factorial calculation using ancient Vedic mathematics. *Microelectron J* 2011;42:1343–52.
6. Ramalatha M, Thanushkodi K, Deena Dayalan K, Dharani P. A novel time and energy efficient cubing circuit using Vedic mathematics for finite field arithmetic. In: Proceedings of advances in recent technologies in communication and computing; 2009. p. 873–5. doi:10.1109/ARTCom.2009.227.
7. Aliparast P, Koozehkanani ZD, Khianvi AM, Karimian G, Bahar HB. A new very high-speed MOS 4-2 compressor for fast digital arithmetic circuits. In: Proceedings of mixed design of integrated circuits and systems (MIXDES); 2010. p. 191–4.
8. Jaina D, Sethi K, Panda R. Vedic mathematics Based Multiply Accumulate Unit. In: Proceedings of computational intelligence and communication systems (CICN); 2011. p. 754–7. doi:10.1109/CICN.2011.167.
9. Kunchigi V, Kulkarni L, Kulkarni S. High-speed and area efficient Vedic multiplier. In: Proceedings of international conference on devices, circuits and systems (ICDCS); 2012. p. 360–4. doi:10.1109/ICDCSyst.2012.6188747.
10. Gu J, Chang CH. Low voltage, low power (5-2) compressor cell for fast arithmetic circuits. In: Proceedings of international conference on acoustics, speech, and signal processing (ICASSP); 2003. p. II-661-664. doi:10.1109/ICASSP.2003.1202453.
11. Aliparast P. and Koozehkanani Z.D., Khiavi A.M., Karimian G., Bahar H.B.. A very high-speed CMOS 4-2 compressor using fully differential current-mode circuit technique. *Analog Integr Circ Sig Process* (2011); 66: pp. 235-243.

