PERFORMANCE ANALYSIS OF MULTIPLIERS BASED ON LOW POWER HYBRID FULL 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

K.SAHITHI (317126512083) **A.DINESH(317126512063)**

G.BENHAR KUMAR(317126512067) **SK.SIRAJUDDIN(318126512L20)**

Under the guidance of

Dr.K.V.G.Srinivas

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)

PRAGNANAM BRAHMA

ANITS

CERTIFICATE

This is to certify that the project report entitled **"PERFORMANCE ANALYSIS OF MULTIPLIERS BASED ON LOW POWER HYBRID FULL ADDERS"** submitted by **K.Sahithi(317126512083), A.Dinesh(317126512063), G.Benhar Kumar (317126512067), SK.Sirajuddin(317126512L20)** 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.

Project Guide

K.V.G.Srinivas

Dr.K.V.G.Srinivas

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

[illegible]

Dr. V.Rajyalakshmi

M.E, Ph.D, MIEEE,MISTE

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.G.Srinivas** 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. At last but not the least, we thank everyone for supporting us directly or indirectly in completing this project successfully.

PROJECT STUDENTS

K. SAHITI(317126512083)

A.DINESH(317126512063)

G.BENHAR KUMAR(317126512067)

SK.SIRAJUDDIN(318126512L20)

CONTENTS

LIST OF FIGURES:

LIST OF TABLES:

ABSTRACT

The growth of the electronics market has driven towards very high integration density. Due to this, critical concerns have been arising to the severe increase in power consumption. In order to overcome these issues, we analysised multipliers based on low power hybrid full adders and compressors. The significance of full hybrid adders is to reduce the power dissipation of the parallel multipliers at the logic level by generating the partial product bits through NAND gates. Baugh-Wooley multipliers are analysised by using this hybrid adders to reduce the number of transistors and array multipliers are designed by using compressors and the power, area, delay of these multipliers are compared. The comparative analysis of multipliers in terms of hybrid adders is included which proves that the analysis is effective in explosive growth of communication.

CHAPTER 1
INTRODUCTION

While the growth of the electronics market has driven the VLSI industry towards very high integration density and system on chip designs and beyond few GHz operating frequencies, critical concerns have been arising to the severe increase in power consumption and the need to further reduce it. Moreover, with the explosive growth in laptops and portable personal communication systems, demanding longer battery operating times and modest weights, the research effort in low power and low area IC designs has been intensified.

Multiplication is one of the most important operations in digital computer systems and digital signal processors. Besides, multipliers are power hungry components, so reducing their power dissipation is a key to satisfying the overall power budget of digital VLSI circuits. Various techniques can be applied externally or internally to reduce power consumption in digital multipliers. External techniques deal with input data characteristics, whereas internal techniques are concerned with the architecture, logic and circuit designs of the multiplier. The basic building block of the multiplier is the full adder cell, thus it has a significant effect on the overall performance and power consumption of the multiplier. Therefore, low power designs of full adders based on pass transistor logic with a low number of transistors were presented. In this work, we introduce five new hybrid full adders, to reduce the power dissipation of the parallel multipliers at the logic/circuit level by allowing the generation of the partial product bits through 4-transistor CMOS NAND gates, rather than the 6-transistor AND gates. This also aims at reducing the number of transistors.

1.1 PROJECT OBJECTIVE:

The main objective of this project is to design and implementation of low power multipliers with hybrid full adders using Xilinx Vivado Software.

1.2 SCOPE OF THE PROJECT:

The scope of this project is to study, design and develop an ALL NAND array multiplier using Verilog HDL The main tasks of this project includes:

1. Background study and literature review on Hybrid Adders.
2. Study on the operation of a Array multiplier.
3. Study and implement the architecture of ALL NAND multiplier using Verilog HDL.
4. Learn and familiarize with the programming language (Verilog HDL).
5. Testing and simulation to verify the results of the new Multiplier
6. Writing Thesis report

1.3 PROJECT OUTLINE:

This Project is presented over the four remaining chapters. Chapter2 provides the overview of Hybrid full Adders. Chapter3 describes about various types of multipliers and their principle of working along with the review of binary and ALL NAND multiplier. Chapter 4 mainly gives the description about working with XILINX VIVADO. Chapter5 describes the simulations and their results.

CHAPTER 2

HYBRID FULL ADDERS

2.1 Regular full adders:

A **full adder** adds binary numbers and accounts for values carried in as well as out. A one-bit full-adder adds three one-bit numbers, often written as A, B, and C_{in}; A and B are the operands, and C_{in} is a bit carried in from the previous less-significant stage. The full adder is usually a component in a cascade of adders, which add 8, 16, 32, etc. bit binary numbers. The circuit produces a two-bit output. Output carry and sum typically represented by the signals C_{out} and S, where the sum equals $2C_{out} + S$.

A full adder can be implemented in many different ways such as with a custom transistor-level circuit or composed of other gates. One example implementation is with $S = A \oplus B \oplus C_{in}$ and $C_{out} = (A \cdot B) + (C_{in} \cdot (A \oplus B))$.

create a byte-wide adder and cascade the carry bit from one adder to the another.

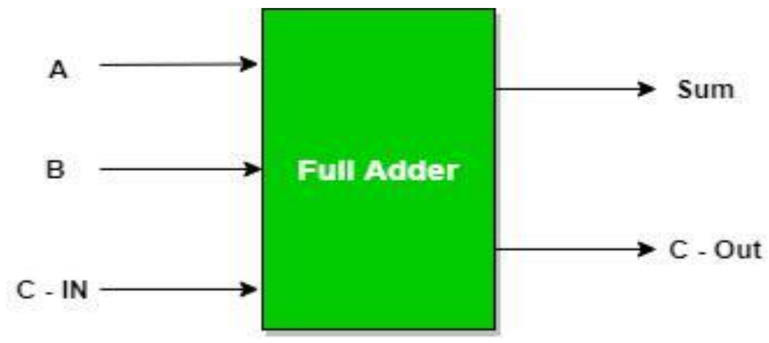


Fig 2.1 Full adder

In this implementation, the final OR gate before the carry-out output may be replaced by an XOR gate without altering the resulting logic. Using only two types of gates is convenient if the circuit is being implemented using simple integrated circuits chips which contain only one gate type per chip.

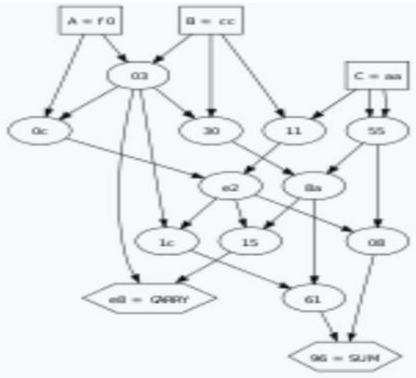


Fig 2.2 Nor full adder

A full adder can also be constructed from two half adders by connecting *A* and *B* to the input of one half adder, then taking its sum-output *S* as one of the inputs to the second half adder and *C* in as its other input, and finally the carry outputs from the two half-adders are connected to an OR gate. The sum-output from the second half adder is the final sum output (*S*) of the full adder and the output from the OR gate is the final carry output (*C*out). The critical path of a full adder runs through both XOR gates and ends at the sum bit *s*. Assumed that an XOR gate takes 1 delays to complete, the delay imposed by the critical path of a full adder is equal to

The critical path of a carry runs through one XOR gate in adder and through 2 gates (AND and OR) in carry-block and therefore, if AND or OR gates take 1 delay to complete, has a delay of

2.1.2 Implementation of Full Adder using NOR gates:

Total 9 NOR gates are required to implement a Full Adder.

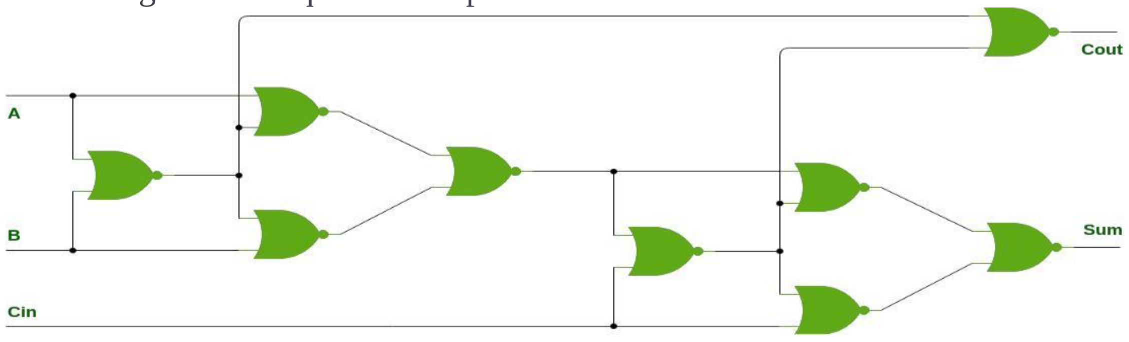


Fig 2.3 Full adder with NOR gate

The truth table for the full adder is:

Table 2.1 Truth table

Inputs			**Outputs**	
A	**B**	**C**in	**C**out	**S**
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

2.1.3 Implementation of Full Adder using Half Adders

2 Half Adders and a OR gate is required to implement a Full Adder.

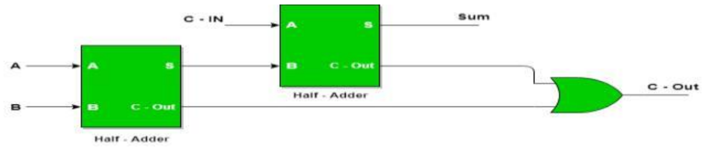


Fig 2.4 Full adder using Half adders

With this logic circuit, two bits can be added together, taking a carry from the next lower order of magnitude, and sending a carry to the next higher order of magnitude

2.1.4 Implementation of Full Adder using NAND gates:

.

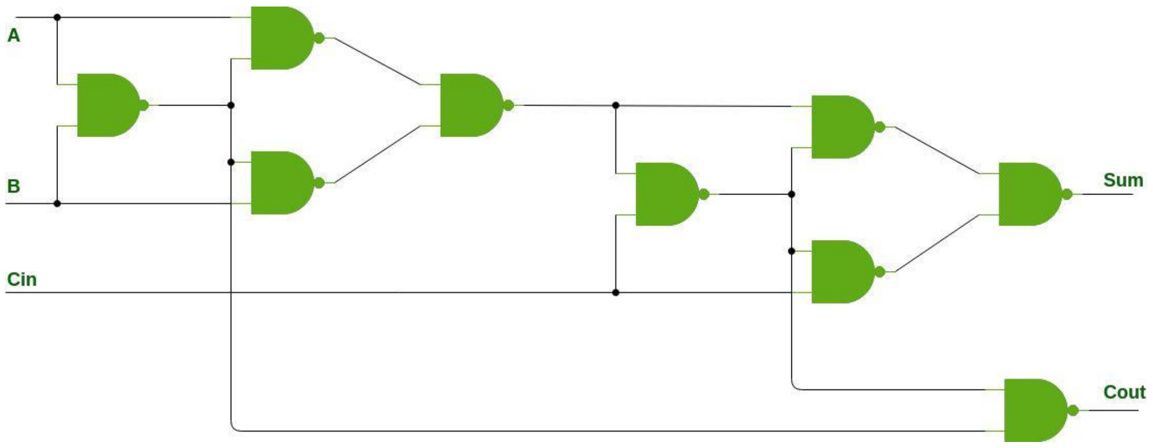


Fig 2.5 Full adders using NAND gates

2.2 Adders supporting multiple bits:

2.2.1 Ripple-carry adder:

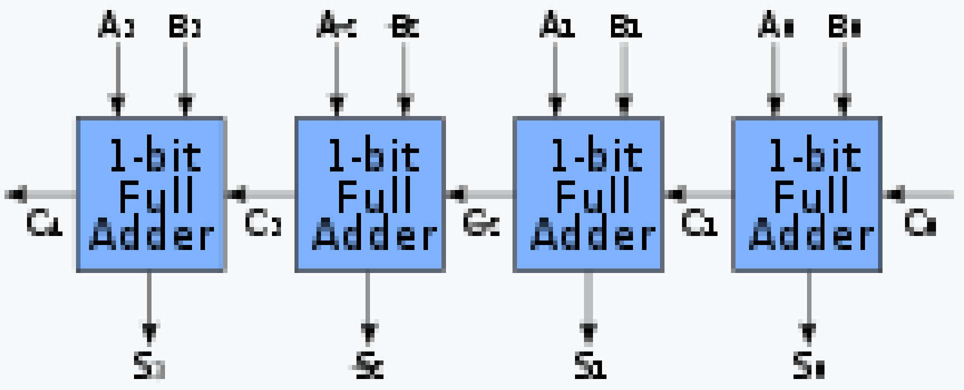


Fig 2.6 4-bit adder with logical block diagram

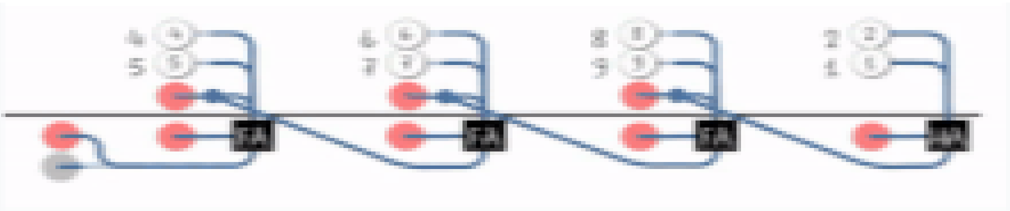


Fig 2.7 Decimal 4-digit ripple carry adder. FA = full adder, HA = half adder.

It is possible to create a logical circuit using multiple full adders to add *N*-bit numbers. Each full adder inputs a C_{in}, which is the C_{out} of the previous adder. This kind of adder is called a **ripple-carry adder** (RCA), since each carry bit "ripples" to the next full adder. Note that the first (and only the first) full adder may be replaced by a half adder (under the assumption that $C_{in} = 0$).

The layout of a ripple-carry adder is simple, which allows fast design time; however, the ripple-carry adder is relatively slow, since each full adder must wait for the carry bit to be calculated from the previous full adder. The gate delay can easily be calculated by inspection of the full adder circuit. Each full adder requires three levels of logic. In a 32-bit ripple-carry adder, there are 32 full adders, so the critical path (worst case) delay is 3 (from input to carry in first adder) + 31 × 2 (for carry propagation in latter adders) = 65 gate delays. The general equation for the worst-case delay for a *n*-bit carry-ripple adder, accounting for both the sum and carry bits, is

A design with alternating carry polarities and optimized AND-OR-Invert gates can be about twice as fast.

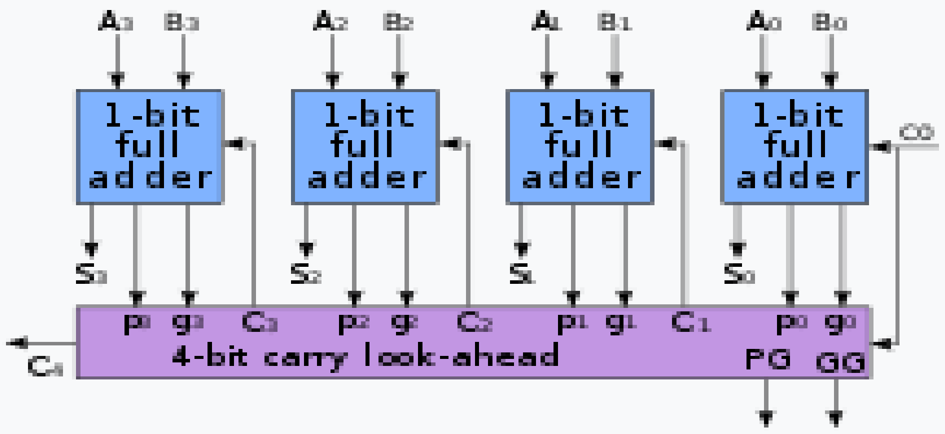


Fig 2.8 4-bit adder with carry look ahead

2.2.2 Carry-look ahead adder:

To reduce the computation time, engineers devised faster ways to add two binary numbers by using carry look ahead adders (CLA). They work by creating two signals (*P* and *G*) for each bit position, based on whether a carry is propagated through from a less significant bit position (at least one input is a 1), generated in that bit position (both inputs are 1), or killed in that bit position (both inputs are 0). In most cases, *P* is simply the sum output of a half adder and *G* is the carry output of the same adder. After *P* and *G* are generated, the carries for every bit position are created. Some advanced carry-look ahead architectures are the Manchester carry chain, Brent-Kung adder (BKA), and the Kogge stone adder (KSA).

Some other multi-bit adder architectures break the adder into blocks. It is possible to vary the length of these blocks based on the propagation delay of the circuits to optimize computation time. These block based adders include the carry skip (or carry bypass) which will determine *P* and *G* values for each block rather than each bit, and the carry-select adder which pre-generates the sum and carry values for either possible carry input (0 or 1) to the block, using multiplexers to select the appropriate result *when* the carry bit is known.

By combining multiple carry-look ahead adders, even larger adders can be created. This can be used at multiple levels to make even larger adders. For example, the following adder is a 64-bit adder that uses four 16-bit CLAs with two levels of look ahead carry unit

Other adder designs include the carry select adder, conditional sum adder, carry sum adder, and carry-complete adder.

2.2.3 Carry-save adders:

If an adding circuit is to compute the sum of three or more numbers, it can be advantageous to not propagate the carry result. Instead, three-input adders are used, generating two results: a sum and a carry. The sum and the carry may be fed into two inputs of the subsequent 3-number adder without having to wait for propagation of a carry signal. After all stages of addition, however, a conventional adder (such as the ripple-carry or the look ahead) must be used to combine the final sum and carry results.

2.3 Verilog full adders: The full adder is a digital component that performs three numbers an implemented using the logic gates. It is the main component inside an ALU of a processor and is used to increment addresses, table indices, buffer pointers, and other places where addition is required

A one-bit full adder adds three one-bit binary numbers, two input bits, one carry bit, and outputs a sum and a carry bit.

A full adder is formed by using two half adders and ***ORing*** their final outputs. A half adder adds two binary numbers. The full adder is a combinational circuit so that it can be modelled in verilog language.The logical expression for the two outputs ***sum*** and ***carry*** are given below. A, B are the input variables for two-bit binary numbers, Cin is the carry input, and Cout is the output variables for Sum and Carry.

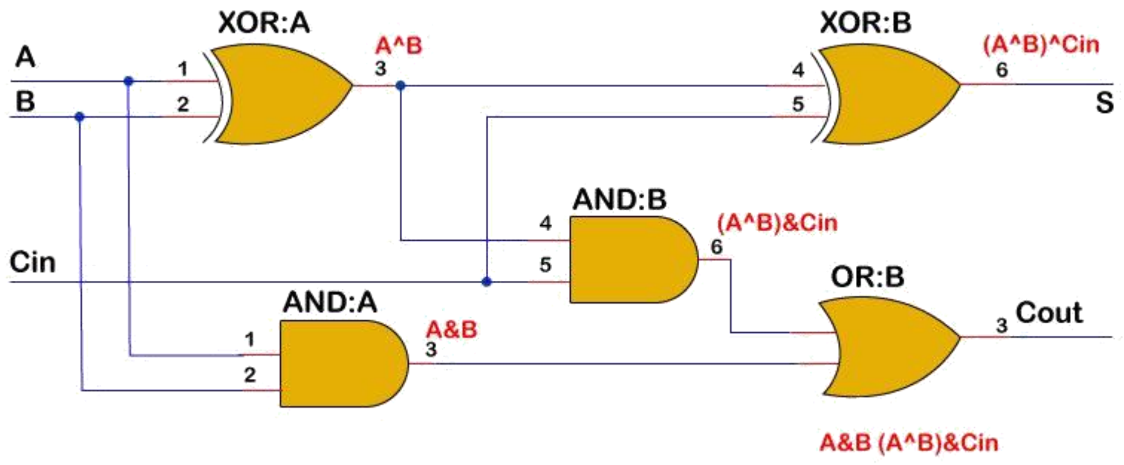


Fig 2.9 Full adders using gates

Truth Table

Table 2.2 Truth table of Verilog full adder

A	B	Cin	Cout	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Example

An example of a 4-bit adder is shown below, which accepts two binary numbers through the signals a and b.

An adder is a combinational circuit. Therefore Verilog can model it using a continuous assignment with ***assign*** or an ***always*** block with a sensitivity list that comprises all inputs

Code:

```
module fulladder (  input [3:0] a,

                    input [3:0] b,
                    input c_in,
                    output c_out,
                    output [3:0] sum);


   assign {c_out, sum} = a + b + c_in;
endmodule
```

Below code shows the uses an ***always*** block which gets executed whenever any of its inputs change value.

Code:

```
module fulladder (  input [3:0] a,
                    input [3:0] b,
                    input c_in,
                    output reg c_out,
                    output reg [3:0] sum);
    always @ (a or b or c_in) begin

    {c_out, sum} = a + b + c_in;
  end
endmodule
```

2.4 Hybrid full adders:

Hybrid technology is the combination of two or more different logic styles. One bit **hybrid full-adder** consists of the CMOS logic design style, transmission gate logic and pass transistor logic.

Firstly five different hybrid adders are introduced to implement multipliers. Each hybrid adder implements a different logic function than that of a regular full adder. Main difference is that some of inputs and outputs are inverted in hybrid adder where as not in regular adder. In a hybrid adder partial product is generated using NAND gate and is connected directly to a bubbled pin without inversion. This is because a bubbled output if connected to a bubbled input, no inverters are needed. Thus hybrid adders allow the use of NAND gates instead instead of AND gates to generate the multipliers partial products. The hybrid adders are named according to the number of bubbles needed at its inputs and outputs .These hybrid adders are optimized ,targeting low power dissipation and full output voltage

2.4.1 Types of hybrid full adders:

1.1-2 hybrid full adder:

1-2 hybrid adder have one bubbled and two bubbled outputs so it is named as 1-2 hybrid full adder

2. 2-1 hybrid full adder:

2-1 hybrid adder have two bubbled inputs and one bubbled output so it is named as 2-1 hybrid full adder

3. 2-2 hybrid full adder:

2-2 hybrid adder have two bubbled input and two bubbled outputs. so it is named as 1-2 hybrid full adder

4. 3-1 hybrid full adder:

3-1 hybrid adder have three bubbled inputs and one bubbled outputs so it is named as 1-2 hybrid full adder.

5. 3-2 hybrid full adder: 3-2 hybrid adder have three bubbled and two bubbled outputs so it is named as 1-2 hybrid full adder

$$S = \overline{X} \oplus Y \oplus \overline{Z} \tag{3}$$

$$\overline{\text{Cout}} = \overline{X} \cdot Y + \overline{Z} \cdot (\overline{X} \oplus Y) \tag{4}$$

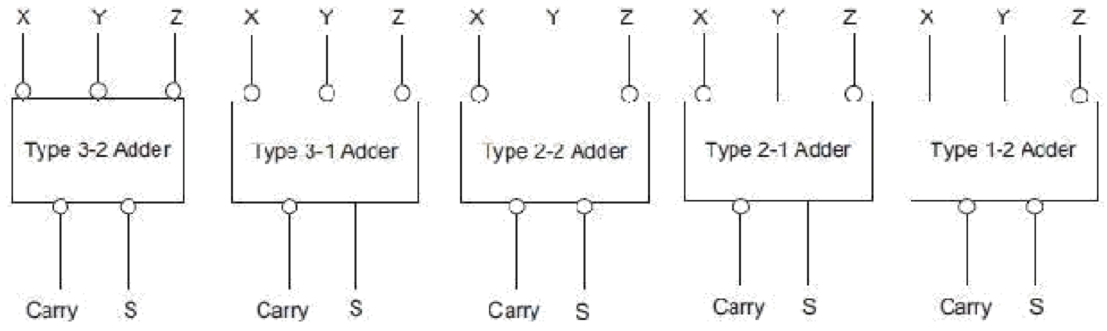


Fig 2.10 Types of hybrid adders

2.5 Compressors:

The 4: 2 compressors were initially designed by an intricate connection of two 3: 2 compressors as shown in Fig. 1a. The structure has a delay of four XORs. The advantage of the structure lies in its carry free nature, whereby the carry from the previous stage is not propagated to the next stage. A novel design of a 4: 2 compressor with XORs and multiplexers (MUX) as building blocks is presented in [5]. This design is based on a modified set of equations for the sum and carries outputs as: Fig2: 4:2 COMPRESSORS

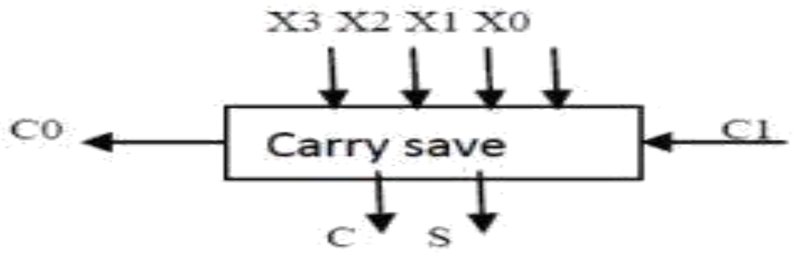


Fig2: 4:2 COMPRESSORS

$$\text{Sum} = x_1 \oplus x_2 \oplus x_3 \oplus x_4 \oplus c_i$$

$$\text{Carry} = (x_1 \oplus x_2 \oplus x_3 \oplus x_4)c_i + \overline{(x_1 \oplus x_2 \oplus x_3 \oplus x_4)}x_4, \text{ and}$$

$$C_o = (x_1 \oplus x_2)x_3 + \overline{(x_1 \oplus x_2)}x_1$$

Fig 2.11 4: 2 compressors

4-2 Compressor The 4-2 Compressor has 5 inputs A, B, C, D and Cin to generate 3 outputs Sum, Carry and Cout as shown in Figure 6(a). The 4 inputs A, B, C and D and the output Sum have the same weight. The input Cin is the output from a previous lower significant compressor and the Cout output is for the compressor in the next significant stage. The conventional approach to implement 4-2 compressors is with 2 full adders connected serially as shown in Figure.

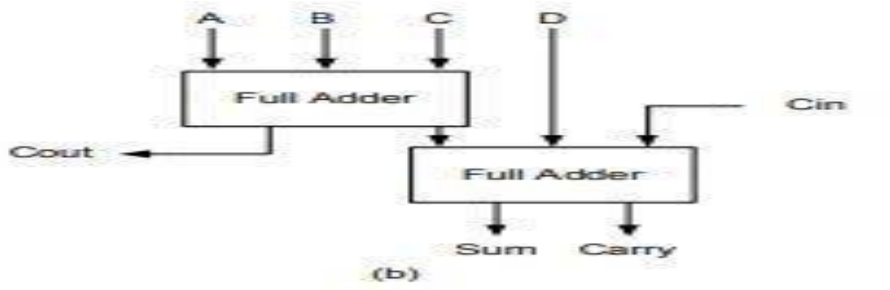


Fig 2.12 4:2 compressors using full adders

In general, compressors reduce N-input bits to a single sum bit of equal weight to that of the inputs but unlike counters, the remaining output bits are all of equal weight: one bit position greater than that of the inputs. Although the 4:2 compressor is not defined as a counter, since it is impossible to use 2 output bits to represent 4 binary input bits, the primitive configuration of 4:2 compressor is based on a 5:3 counter structure, which has 5 inputs and 3 outputs as shown in Figure 25. The four inputs Xo, Xi, X2 and X3, and the output Sum have the same weight. The output Carry is weighted one binary bit order higher.

The 4:2 compressor receives an input CEM from the preceding module of one binary bit order lower in significance, and produce an output Cout to the next compressor module of higher significance. Different structures of 4:2 compressors exist and they all have to abide by the fundamental equation given as follows :

X0 +XX +X2 +X3 +CIN =Sum+ 2- (Carry+Cout) (].

Besides, to accelerate the carry save summation of the partial products, it is imperative that the output Cout be independent of the input Cin.

Compressors are built using the five hybrid adders. They are named base on number of inverted inputs and outputs they have compared to a standard compressor. There are three types of hybrid compressors.

Xi X2 X3 X4 _& ^—ik_ 4:2 Compressor U' 'IN Carry Sum Figure 25 Symbol of 4:2 compressor

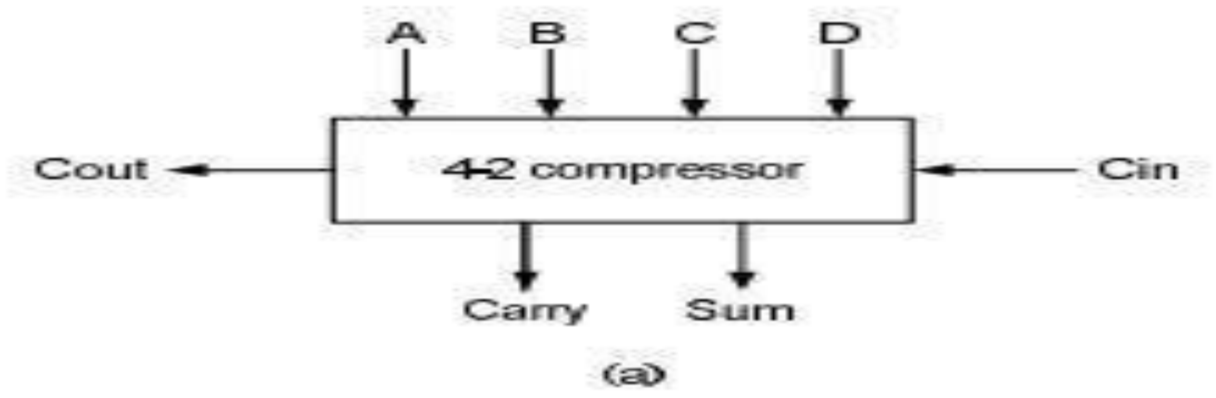


Fig 2.13: 4-2 Compressor

2.5.1 Types of hybrid compressors:

1. v-iii type compressor

This type has 5 inverted inputs and 3 inverted outputs.

2. iv-iii type compressor:

This type has 4 inverted inputs and 3 inverted outputs.

3. iii-iii type compressor

This type has 3 inverted inputs and 3 inverted outputs

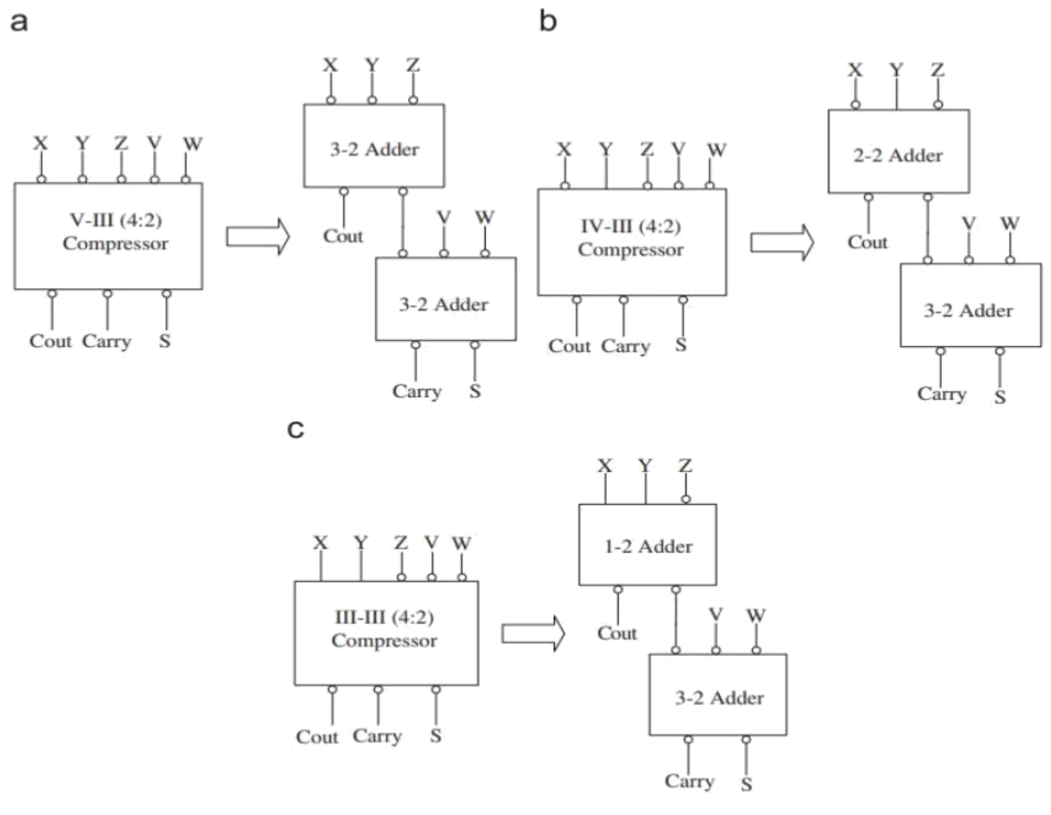


Fig. 4. Hybrid compressors. (a) Type V-III compressor, (b) type IV-III compressor, (c) type III-III compressor.

Fig 2.13 Types of hybrid compressors

CHAPTER 3

MULTIPLIERS

3.1 INTRODUCTION

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 amount 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.

3.2 TYPES OF MULTIPLIERS

SERIAL MULTIPLIER: Where area and power is 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

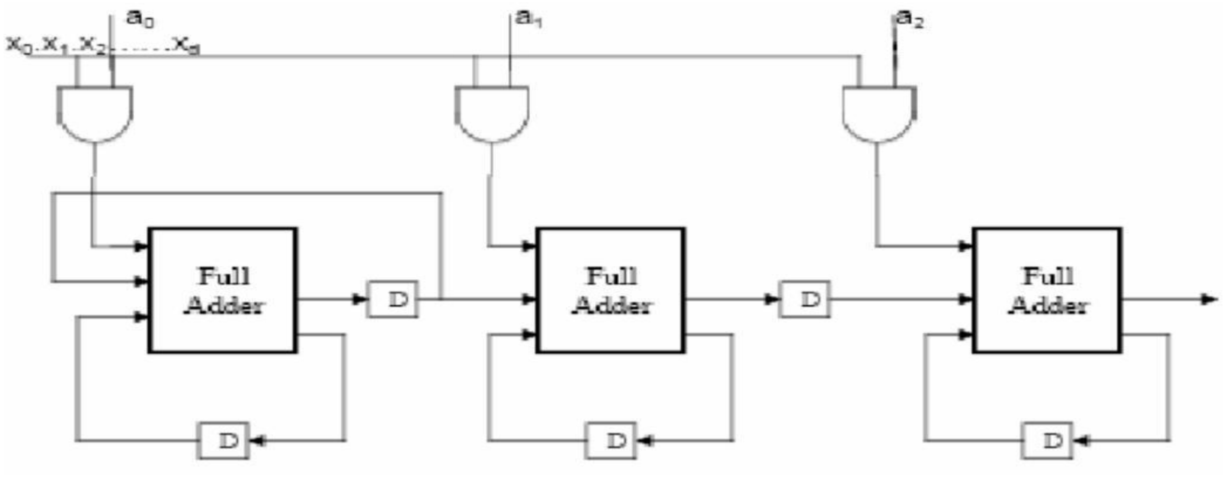


Figure 3.1: Serial /parallel Multiplier

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

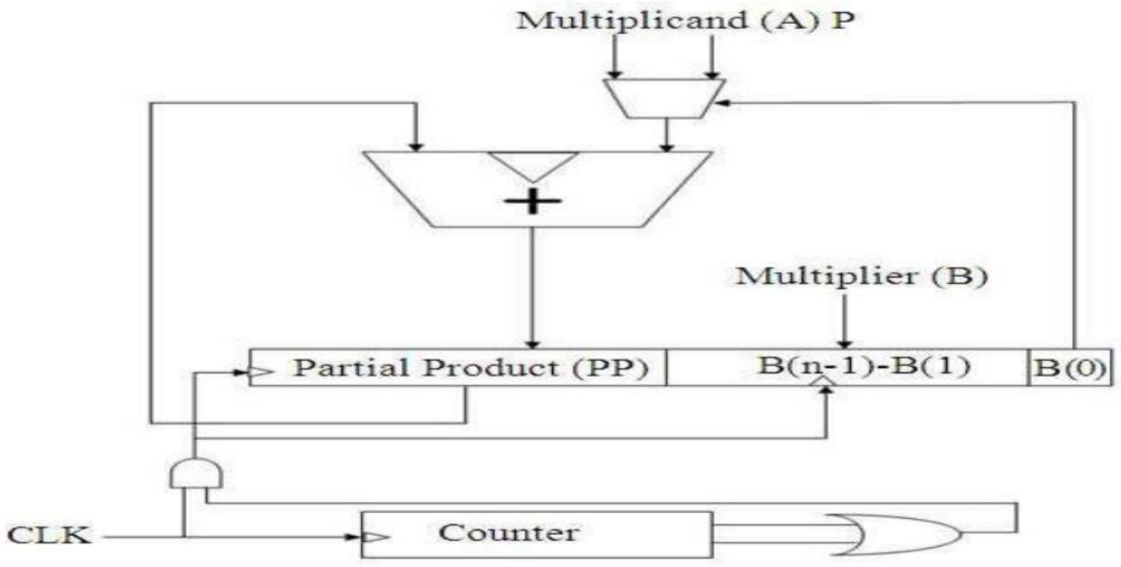


Figure 3.2 : Shift and add multiplier

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.

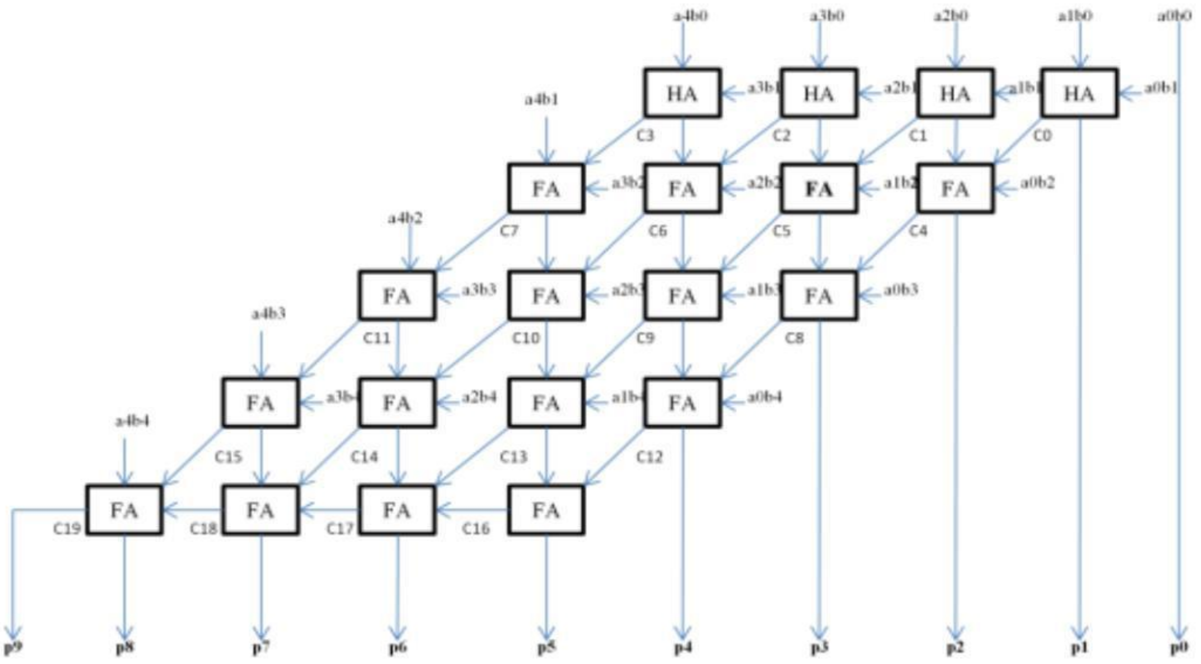


Figure 3.3: Array multiplier

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.

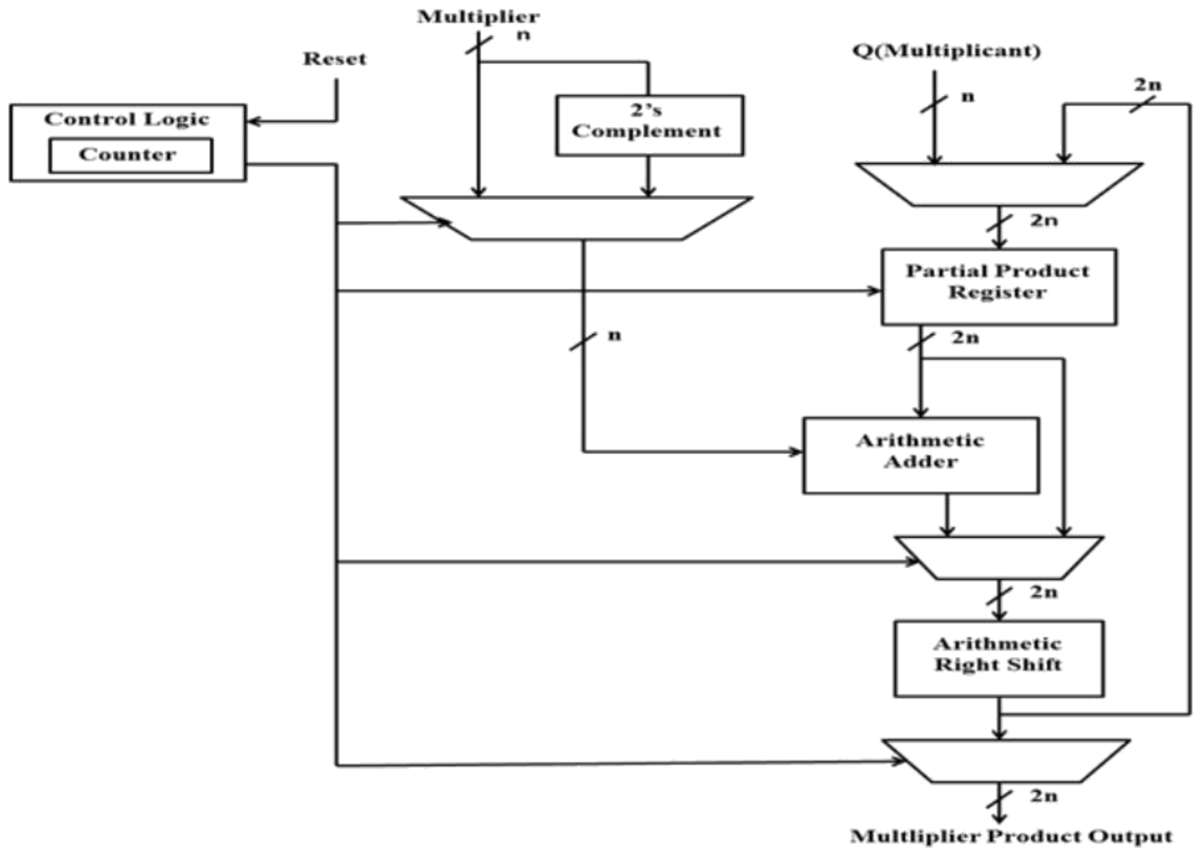


Figure 3.4 Booth Multiplier

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 built 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.

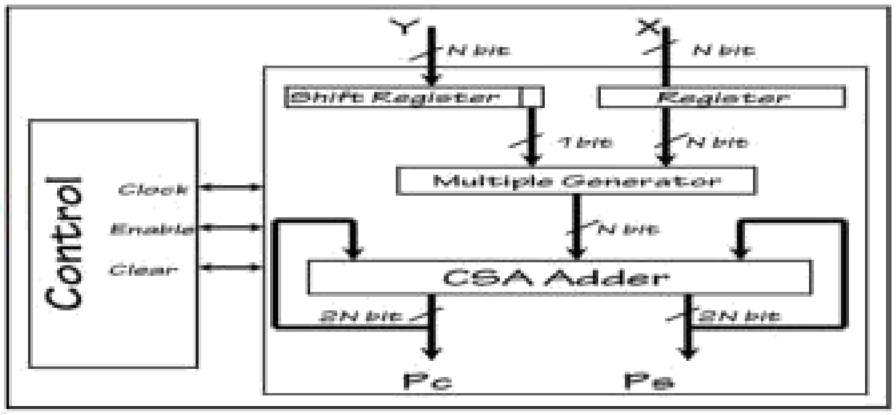


Figure 3.5: Sequential multiplier

<u>Wallace tree Multiplier</u>: A Wallace tree is a 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(log2 n) time.

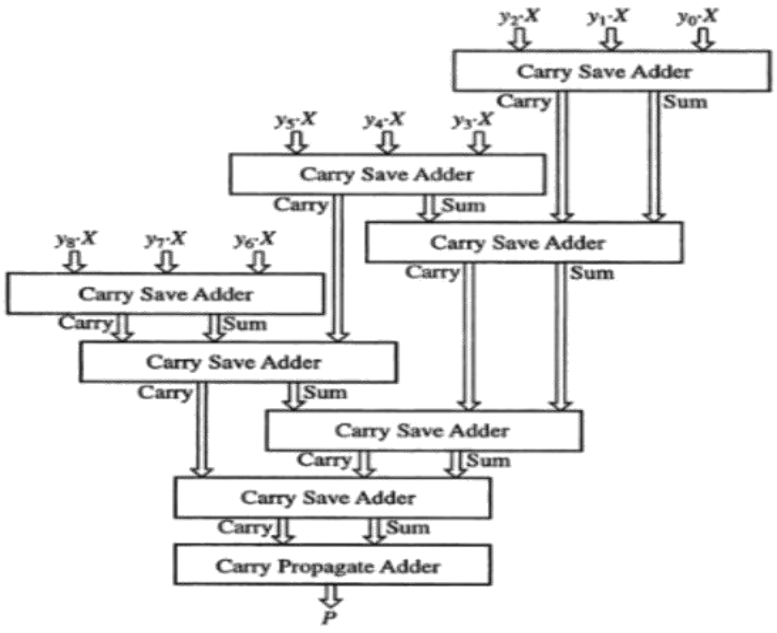


Figure 3.6 : Wallace Tree Multiplier

3.3 The ALL-NAND array multipliers:

The proposed ALL-NAND multipliers use the five hybrid full bit adders, and the three new (4:2) compressors, to enable the employment of the 4-transistor NAND gates in generating the partial product bits, instead of the 6-transistor AND gates. The goal is the reduction of both the number of transistors and the power dissipation of the parallel multipliers.

3.3.1 Review of array multipliers: Parallel multiplication, in binary number system, involves three tasks: the generation of PP bits, the accumulation of PP bits into two rows, and the computing of the final product commonly via a carry propagate adder (CPA) [8]. In signed array and tree multipliers, a regular PP bit is generated by a logical AND operation while an inverted PP is generated by a NAND gate.

The simplification of the multiplication of two operands A and B suggests the use of n^2 - 2(n-1) AND gates to generate the regular PPs and 2(n-1) NAND gates to generate the inverted PPs in an array multiplier.

3.3.2 ALL NAND signed array multiplier: A new array multiplier is proposed where NAND gates are used to generate the PP bits. The dashed arrows indicate that their associated signals are inverted, relative to those of a standard array multiplier, allowing the use of NAND gates. The bits ‘1’ in the first row are the inverted values of the original bits ‘0’. This allows the use of two low power type 3-2 adders and one type 2-2 adder instead of two type 2-2 adders and one type 1-2 adder to lower the power consumption. The CPA uses two type 2-1 hybrid adders and two type 3-1 hybrid adders to compute the final result of the multiplier.

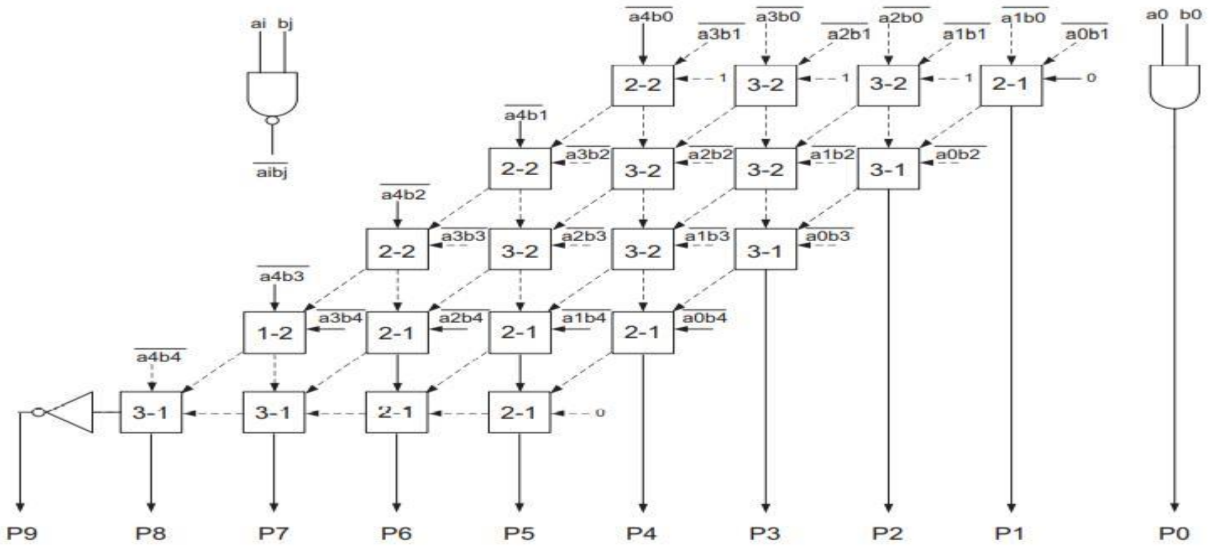


Figure 3.7 : New Architecture of Array Multiplier

As compared to signed array's conventional PP generation scheme, where $[(n-1)^2 +1]$AND gates and 2(n-1) NAND gates are needed, our new realization of the multiplier requires [n(n-1)] NAND gates and only (one) AND gate to generate the PP bits. This results in reducing the number of transistors by $(n-1)^2$ in addition to eliminating an inverter from the critical path by using an NAND gate instead of an AND gate.

When designing a multiplier, a uniform layout is of great importance. The layout uniformity will not be greatly affected by using five different FAs because a multiplier with a large n consists mostly of 3-2 adders and 2-2 adders. Furthermore, the hybrid FA cells use the same number of transistors and have a uniform number of inputs and outputs. Consequently, they all have layouts of approximately the same size and stackable shapes. One decisive factor regarding layout is the organization of input and outputs of each cell to make the routing among adjacent cells as simple as possible.

CHAPTER 4

VERILOG HDL

4.1 INTRODUCTION

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.

HDL 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.

4.2 Verilog Abstraction Levels

Verilog supports a design at many levels of abstraction, such as:

- Behavioral level
- Register-transfer level
- Gate level

4.2.1 Behavioral Level: The behavioral level describes a system by concurrent algorithms behavioral. Every algorithm is sequential, which means it consists of a set of executed instructions one by one. Functions, tasks, and blocks are the main elements. There is no regard for the structural realization of the design.

4.2.2 Register -Transfer Level: Designs using the Register-Transfer Level specify a circuit's characteristics using operations and the transfer of data between the registers. The modern definition of an RTL code is "Any code that is synthesizable is called RTL code".

4.2.3 Gate Level: The characteristics of a system are described by logical links and their timing properties within the logical level. All signals are discrete signals. They can only have definite logical values (`0', `1', `X', `Z`).

The usable operations are predefined logic primitives (basic gates). Gate level modeling may not be the right idea for logic design. Gate level code is generated using tools such as synthesis tools, and his netlist is used for gate-level simulation and backend.

4.3 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.

4.3.1 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.

4.3.2 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)

4.3.3 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)

4.3.4 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)

4.3.5 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)

4.4 Operands

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

4.4.1 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.

4.4.2 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 "[ ]".

4.4.3 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.

4.5 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.

4.5.1 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.Supply0andSupply1
Supply0 and supply1 define wires tied to logic 0 (ground) and logic 1 (power).

4.5.2 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.

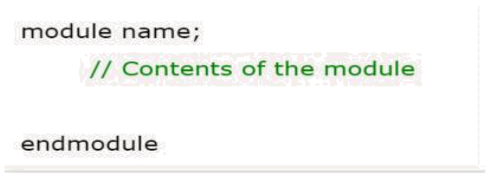
- **Reg** is used for describing logic.
- **An integer** is general-purpose variables. They are used mainly loops-indices, parameters, and constants. They store data as signed numbers, whereas explicitly declared reg types store them as unsigned. If they hold numbers that are not defined at compile-time, 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.
- The reg variables are initialized to x at the start of the simulation. Any wire variable not connected to anything has the x value.
- When the reg or wire size is more than one bit, then register and wire are declared vectors.

4.6 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 ***endmodule*** 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.



```
module name;
      // Contents of the module

endmodule
```

Fig 4.1 Syntax of module

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

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.

4.7 RTL Verilog

In the digital circuit design, **register-transfer level (RTL)** is a design abstraction which models a synchronous digital circuit in terms of the data flow between hardware register, and the logical operations performed on those signals.

Register-transfer-level abstraction is used in HDL to create high-level representations of a circuit, from which lower-level representations and ultimately actual wiring can be derived. Design at the RTL level is a typical practice in modern digital design.

While designing digital integrated circuits with a hardware description language, the designs are usually arranged at a higher level of abstraction than the transistor level or logic gate level.

In HDLs, the designer declares the registers, which roughly correspond to variables in the programming languages and describes the combinational logic by using constructs such as if-then-else and arithmetic operations.

This level is called the ***register-transfer level*** or ***RTL***. The term RTL focuses on describing the flow of signals between registers.

This description can usually be directly translated into an equivalent hardware implementation file using an EDA tool for synthesis. The synthesis tool also performs logic optimization.

At the register-transfer level, some types of circuits can be recognized. If there is a cyclic path of logic from a register's output to its input, then the circuit is called a ***state machine*** or sequential logic.

If there are logic paths from a register to another without a cycle, then it is called a ***pipeline***.

4.8 Verilog Ports

Port is an essential component of the Verilog module. Ports are used to communicate for a module with the external world through input and output.

It communicates with the chip through its pins because of a module as a fabricated chip placed on a PCB.

Every port in the port list must be declared as ***input, output*** or ***inout***. All ports declared as one of them is assumed to be wire by default to declare it, or else it is necessary to declare it again.

Ports, also referred to as pins or terminals, are used when wiring the module to other modules.

- If the module does not exchange any signals with the environment, there are no ports in the list.
- Consider a 4-bit full adder that is instantiated inside a top-level module.
- The module fulladd4 takes input on ports a, b, and c_in and produces an output on ports sum and c_out.

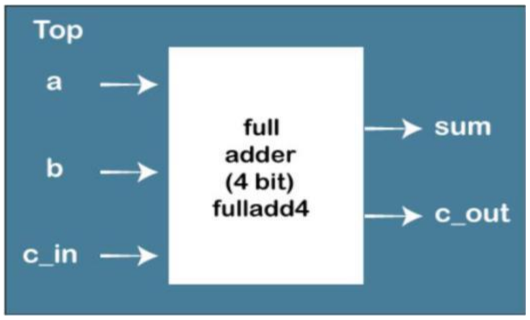


Fig 4.2: I/O ports for full adder

Each port in the port list is defined as input, output, or inout based on the port signal's direction.

If a port declaration includes the net or variable types, then that port is considered completely declared. It is illegal to declare the same port in a net or variable type declaration.

And if the port declaration does not include a net or variable type, then the port can be declared again in a net or variable type declaration.

For example:

```
module fulladd4(sum, c_out, a, b, c_in);

output [3:0] sum;

output c_out;

input [3:0] a, b;

input c_in;

<module internals>

endmodule
```

4.9 Verilog Assign Statement

Assign statements are used to drive values on the net. And it is also used in ***Data Flow Modeling***.

Signals of type wire or a data type require the continuous assignment of a value. As long as the +5V battery is applied to one end of the wire, the component connected to the other end of the wire will get the required voltage.

This concept is realized by the assign statement where any wire or other similar wire (data-types) can be driven continuously with a value. The value can either be a constant or an expression comprising of a group of signals.

Syntax

The assignment syntax starts with the keyword assign, followed by the signal name, which can be either a signal or a combination of different signal nets.

```
module combo (input a, b, c, d, output  o);
assign o = ~((a & b) | c ^ d);
endmodule
```

The combinational digital circuit for the above code is as follows:

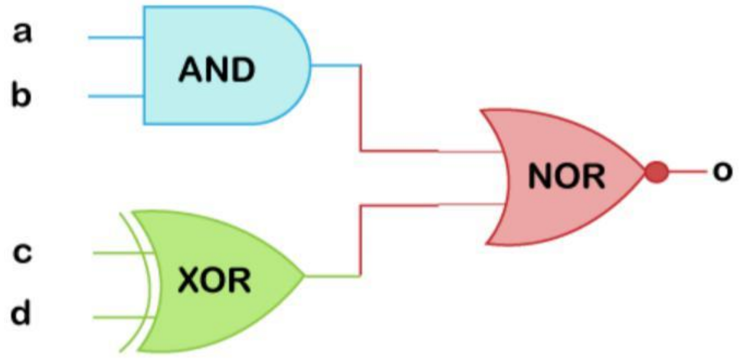


Fig 4.3: Combinational digital circuit

An assigned statement satisfies the purpose because the output o is updated whenever any of the inputs on the right-hand side change.

4.9.1 Rules

Some rules need to be followed during the use of an assign statement

- LHS should always be a scalar, vector, or a combination of scalar and vector nets but never a scalar or vector register.
- RHS can contain scalar or vector registers and function calls.
- Whenever any operand on the RHS changes in value, LHS will be updated with the new value.
- Assign statements are also called continuous assignments.

4.10 Advantages

The major benefit of the language is fast design and better verification. The Top-down design and hierarchical design method allows the design time; design cost and design errors to be reduced. Another major advantage is related to complex designs, which can be managed and verified easily. HDL provides the timing information and allows the design to be described in gate level and register transfer level. Reusability of resources is one of the other advantage.

CHAPTER 5

XILINX VIVADO SOFTWARE

5.1 INTRODUCTION

Xilinx Vivado software is to create a simple digital circuit using Verilog HDL. A typical design flow consists of creating model(s), creating user constraint file(s), creating a Vivado project, importing the created models, assigning created constraint file(s), optionally running behavioral simulation, synthesizing the design, implementing the design, generating the bitstream, and finally verifying the functionality in the hardware by downloading the generated bitstream file. You will go through the typical design flow targeting the Artix-100 based Nexys4 board. The typical design flow is shown below.

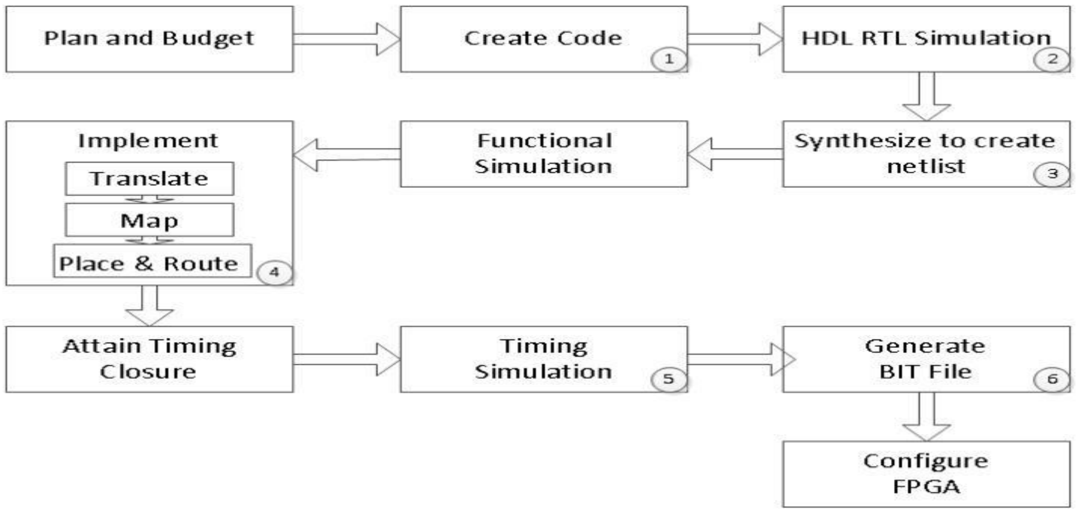
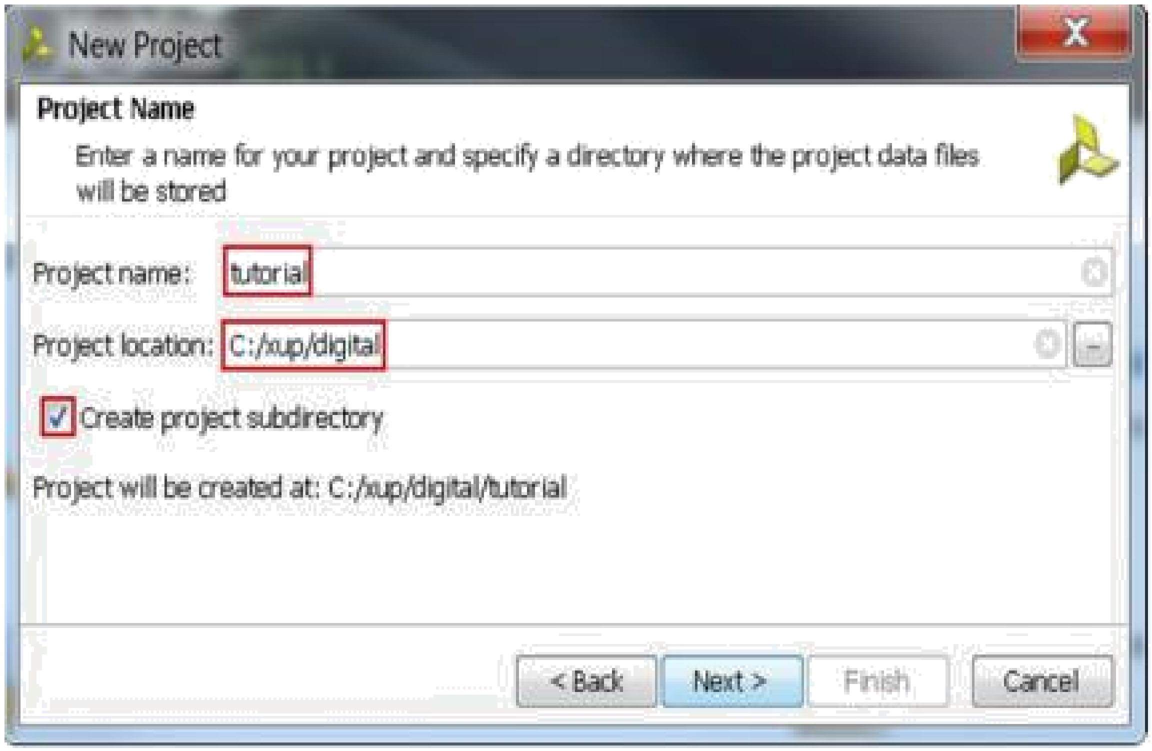


Fig 5.1: Typical Design flow

5.2 Creating a Vivado project using IDE

i. Open Vivado by selecting **Start** > **All Programs** > **Xilinx Design Tools** > **Vivado 2013.3** > **Vivado 2013.3.**
ii. Click **Create New Project** to start the wizard. You will see Create A New Vivado Project dialog box. Click **Next.**
iii. Click the Browse button of the Project location field of the **New Project** form, browse to **c:\xup\digital**, and click **Select.**
iv. Enter **tutorial** in the Project name field. Make sure that the Create Project Subdirectory box is checked. Click **Next.**


New Project

Project Name

Enter a name for your project and specify a directory where the project data files will be stored

Project name: tutorial

Project location: C:/xup/digital

[✓] Create project subdirectory

Project will be created at: C:/xup/digital/tutorial

Fig 5.2: Project Name and Location Entry

v. Select **RTL Project** option in the Project Type form, and click **Next.**

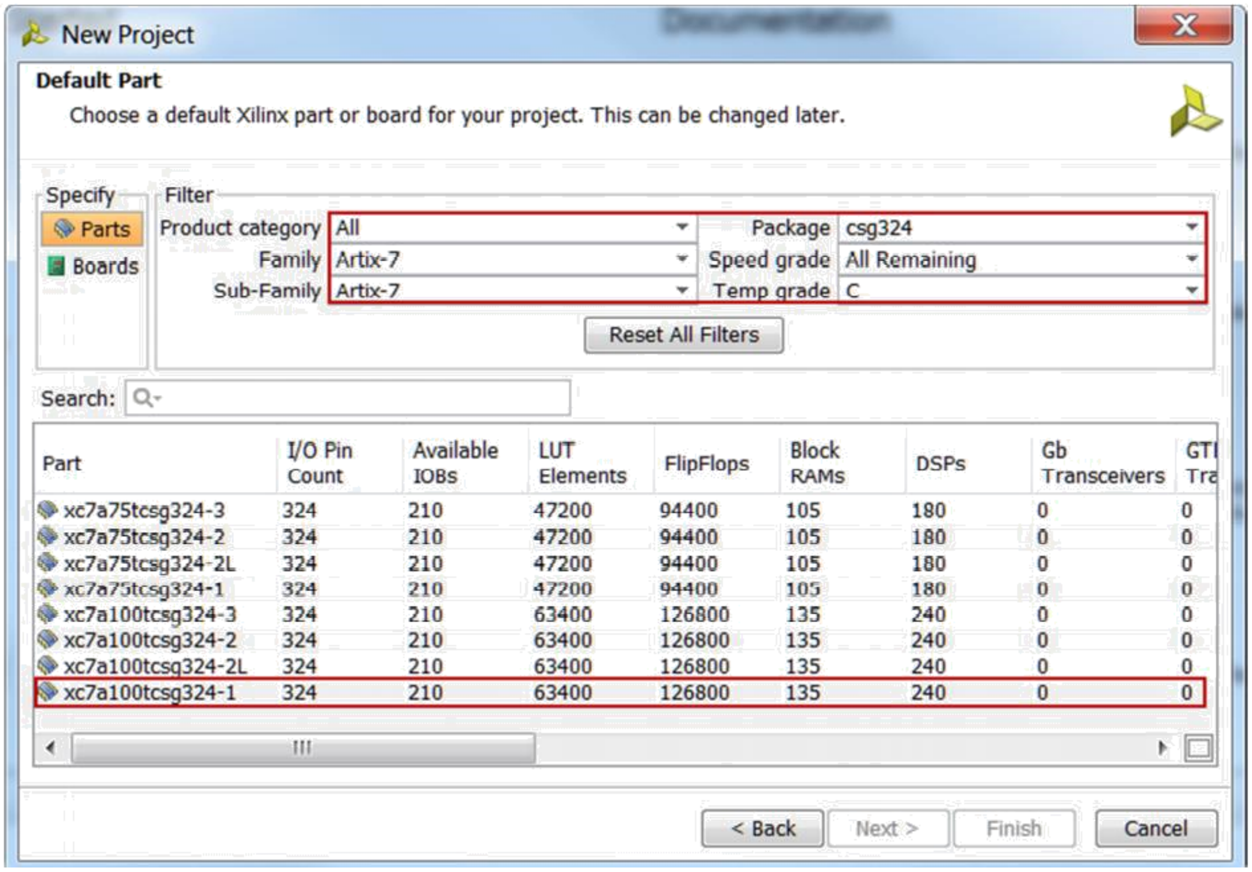
vi. Select **Verilog** as the Target language and Simulator language in the **Add Sources** form.

vii. Click on the **Add Files…** button, browse to the **c:\xup\digital\sources\tutorial directory,** select **tutorial.v**, click **Open**, and then click **Next.**

viii. Click **Next** to get to the **Add Constraints** form.

ix. Click **Next** if the entry is already auto-populated, otherwise click on the **Add Files…** button, browse to the **c:\xup\digital\sources\turorial directory** and select **tutorial.xdc,** and click **Open.**

x. In the Default Part form, using the **Parts** option and various drop-down fields of the **Filter** section, select the **XC7A100TCSG324-1** part. Click **Next**.


New Project

Default Part

Choose a default Xilinx part or board for your project. This can be changed later.

Specify: Parts, Boards

Filter

Product category: All

Family: Artix-7

Sub-Family: Artix-7

Package: csg324

Speed grade: All Remaining

Temp grade: C

Reset All Filters

Search:

Part	I/O Pin Count	Available IOBs	LUT Elements	FlipFlops	Block RAMs	DSPs	Gb Transceivers	GTI Tra
xc7a75tcsg324-3	324	210	47200	94400	105	180	0	0
xc7a75tcsg324-2	324	210	47200	94400	105	180	0	0
xc7a75tcsg324-2L	324	210	47200	94400	105	180	0	0
xc7a75tcsg324-1	324	210	47200	94400	105	180	0	0
xc7a100tcsg324-3	324	210	63400	126800	135	240	0	0
xc7a100tcsg324-2	324	210	63400	126800	135	240	0	0
xc7a100tcsg324-2L	324	210	63400	126800	135	240	0	0
xc7a100tcsg324-1	324	210	63400	126800	135	240	0	0

Fig 5.3: Part Selection

xi. Click **Finish** to create the Vivado project.
xii. Open the **tutorial.v** source and analyze the content.
xiii. In the **Sources pane**, double-click the tutorial.v entry to open the file in text mode.

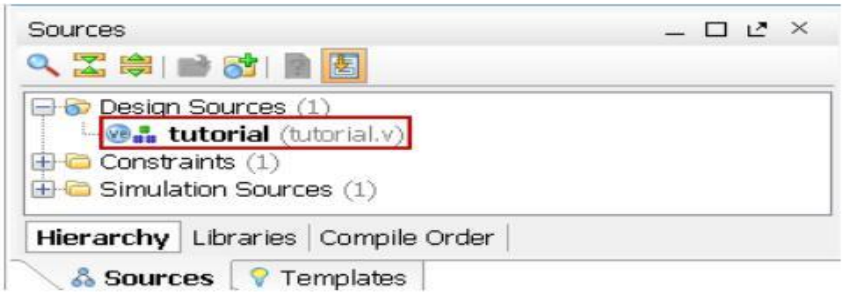


Fig 5.4: Opening the Source file

xiv. Write the required Verilog Code in the Dialog box and check for the errors.
xv. Save the code after rectifying the errors.

5.3 SIMULATING THE DESIGN

i. Add the **tutorial_tb.v** testbench file.
ii. Click **Add Sources** under the Project Manager tasks of the Flow Navigator pane.

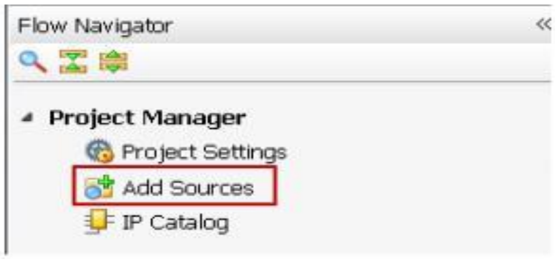
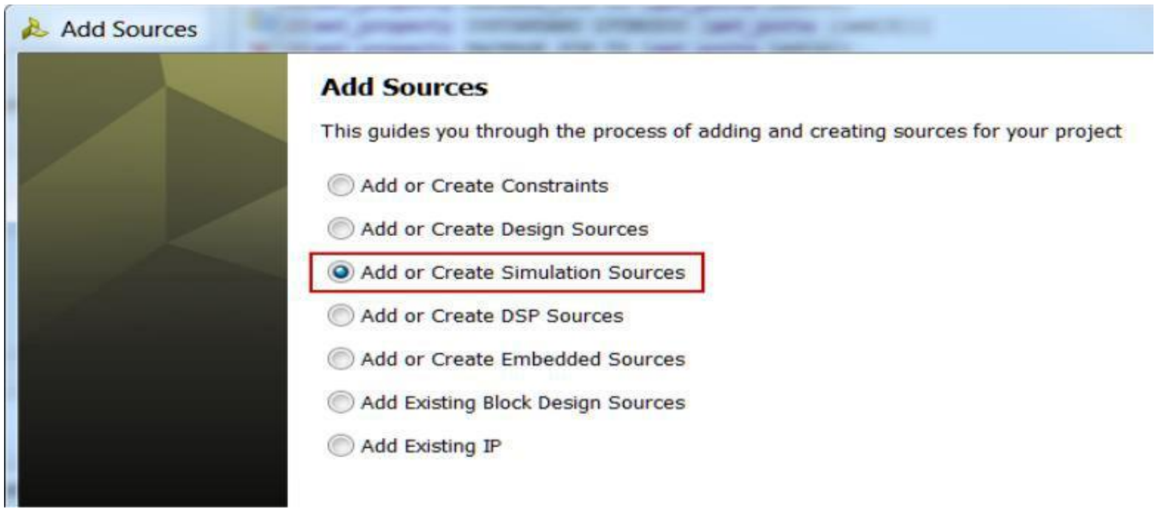


Fig 5.5 Adding Sources

iii. Select the **Add or Create** Simulation Sources option and click **Next.**



Add Sources

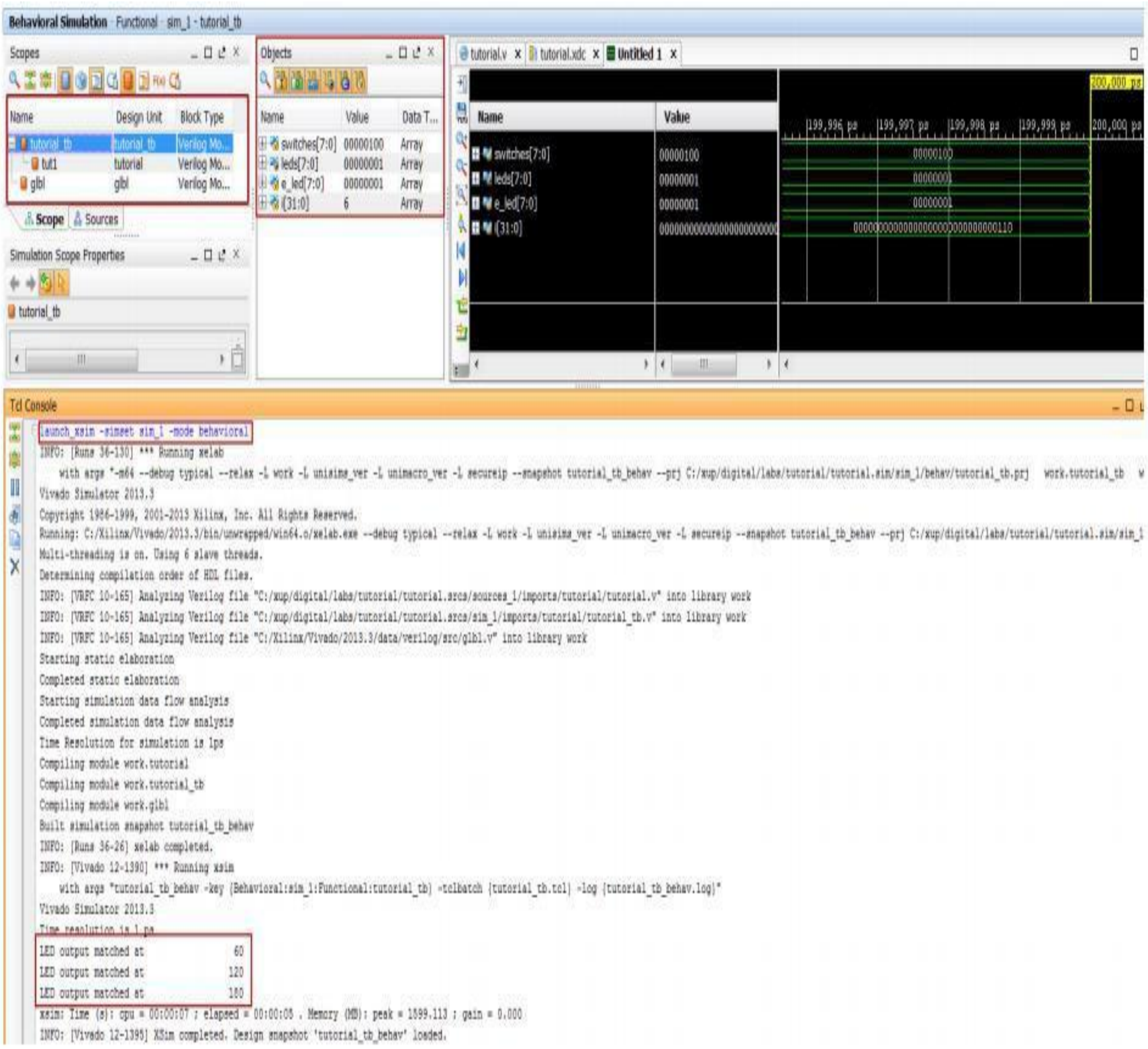
This guides you through the process of adding and creating sources for your project

Add or Create Constraints
Add or Create Design Sources
[✓ Add or Create Simulation Sources]
Add or Create DSP Sources
Add or Create Embedded Sources
Add Existing Block Design Sources
Add Existing IP

Fig 5.6 Selecting Simulating Sources

iv. In the Add Sources Files form, click the **Add Files**… button.
v. Browse to the **c:\xup\digital\sources** folder and select **tutorial_tb.v** and click **OK.**
vi. Click **Finish.**
vii. Select **Simulation Settings** under the Project Manager tasks of the Flow Navigator pane. A Project Settings form will appear showing the Simulation properties form.
viii. Select the **Simulation tab**, and set the Simulation Run Time value to 200 ns and click **OK.**
ix. Click on **Run Simulation** > **Run Behavioral Simulation** under the Project Manager tasks of the Flow Navigator pane.

The testbench and source files will be compiled and the XSim simulator will be run (assuming no errors). You will see a simulator output similar to the one shown below.



Name	Design Unit	Block Type
tutorial_tb	tutorial_tb	Verilog Mo...
tut1	tutorial	Verilog Mo...
glbl	glbl	Verilog Mo...

Name	Value	Data T...
switches[7:0]	00000100	Array
leds[7:0]	00000001	Array
e_led[7:0]	00000001	Array
[31:0]	6	Array

Name	Value
switches[7:0]	00000100
leds[7:0]	00000001
e_led[7:0]	00000001
[31:0]	000000000000000000000000

```
launch_xsim -simset sim_1 -mode behavioral
INFO: [Runs 36-130] *** Running xelab
    with args "-m64 --debug typical --relax -L work -L unisims_ver -L unimacro_ver -L secureip --snapshot tutorial_tb_behav --prj C:/xup/digital/labs/tutorial/tutorial.sim/sim_1/behav/tutorial_tb.prj   work.tutorial_tb   w
Vivado Simulator 2013.3
Copyright 1986-1999, 2001-2013 Xilinx, Inc. All Rights Reserved.
Running: C:/Xilinx/Vivado/2013.3/bin/unwrapped/win64.o/xelab.exe --debug typical --relax -L work -L unisims_ver -L unimacro_ver -L secureip --snapshot tutorial_tb_behav --prj C:/xup/digital/labs/tutorial/tutorial.sim/sim_1
Multi-threading is on. Using 6 slave threads.
Determining compilation order of HDL files.
INFO: [VRFC 10-165] Analyzing Verilog file "C:/xup/digital/labs/tutorial/tutorial.srcs/sources_1/imports/tutorial/tutorial.v" into library work
INFO: [VRFC 10-165] Analyzing Verilog file "C:/xup/digital/labs/tutorial/tutorial.srcs/sim_1/imports/tutorial/tutorial_tb.v" into library work
INFO: [VRFC 10-165] Analyzing Verilog file "C:/Xilinx/Vivado/2013.3/data/verilog/src/glbl.v" into library work
Starting static elaboration
Completed static elaboration
Starting simulation data flow analysis
Completed simulation data flow analysis
Time Resolution for simulation is 1ps
Compiling module work.tutorial
Compiling module work.tutorial_tb
Compiling module work.glbl
Built simulation snapshot tutorial_tb_behav
INFO: [Runs 36-26] xelab completed.
INFO: [Vivado 12-1390] *** Running xsim
    with args "tutorial_tb_behav -key {Behavioral:sim_1:Functional:tutorial_tb} -tclbatch {tutorial_tb.tcl} -log {tutorial_tb_behav.log}"
Vivado Simulator 2013.3
Time resolution is 1 ps
LED output matched at                60
LED output matched at               120
LED output matched at               180
xsim: Time (s): cpu = 00:00:07 ; elapsed = 00:00:05 . Memory (MB): peak = 1599.113 ; gain = 0.000
INFO: [Vivado 12-1395] XSim completed. Design snapshot 'tutorial_tb_behav' loaded.
```

Fig 5.7 Simulation Results

x. Click on the **Zoom** Fit button and observe the output.
xi. Close the simulator by selecting **File** > **Close Simulation.**
xii. Click **OK** and then click **No** to close it without saving the waveform.

5.4 RTL ANALYSIS

i. Open the **tutorial.xdc** source and analyze the content.
ii. In the Sources pane, expand the **Constraints folder** and double-click the **tutorial.xdc** entry to open the file in text mode.

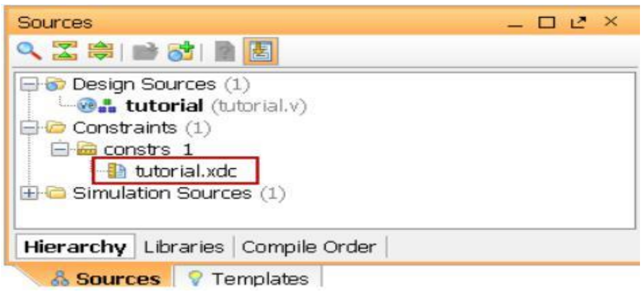


Fig 5.8 Opening the Constraint File

iii. Perform **RTL analysis** on the source file.
iv. Expand the Open **Elaborated Design** entry under the RTL Analysis tasks of the Flow Navigator pane and click on **Schematic.** The model (design) will be elaborated and a logic view of the design is displayed.

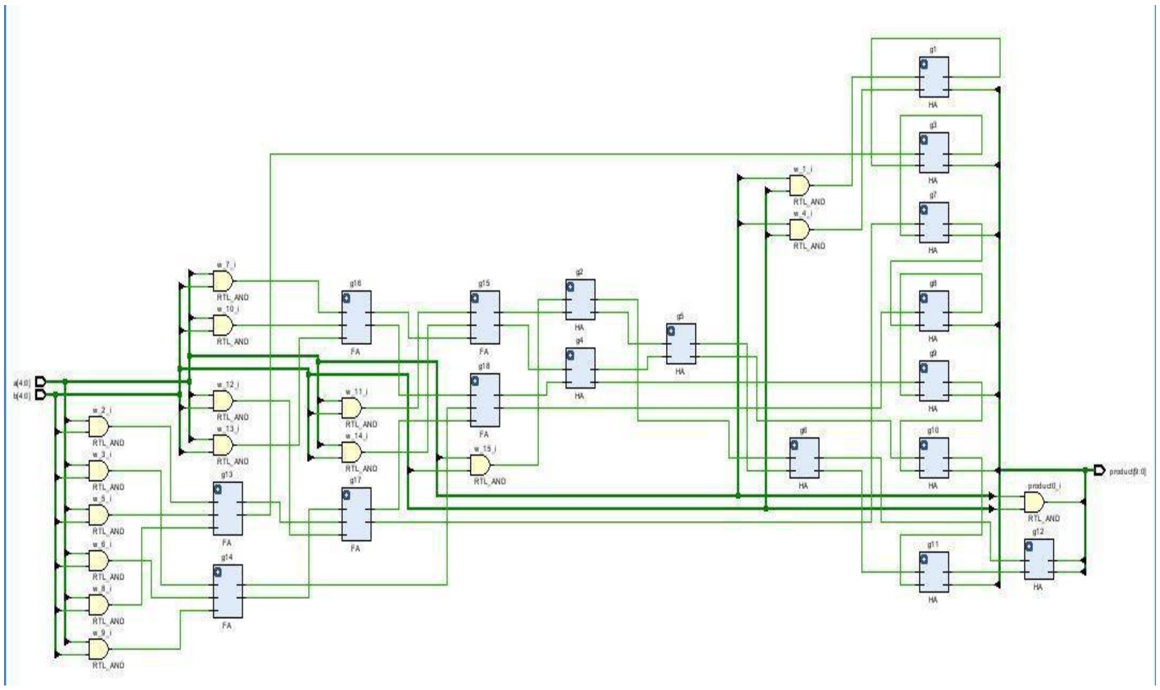


Fig 5.9 A Logic view of the Full Adder

5.5 SYNTHESIS OF DESIGN

i. Synthesize the design with the Vivado synthesis tool and analyze the Project Summary output.
ii. Click on **Run Synthesis** under the Synthesis tasks of the Flow Navigator pane. The synthesis process will be run on the **tutorial.v file** (and all its hierarchical files if they exist). When the process is completed a Synthesis Completed dialog box with three options will be displayed.
iii. Select the **Open Synthesized Design** option and click **OK** as we want to look at the synthesis output before progressing to the implementation stage. Click **Yes** to close the elaborated design if the dialog box is displayed.
iv. Click on **Schematic** under the **Open Synthesized Design** tasks of Synthesis tasks of the Flow Navigator pane to view the synthesized design in a schematic view.

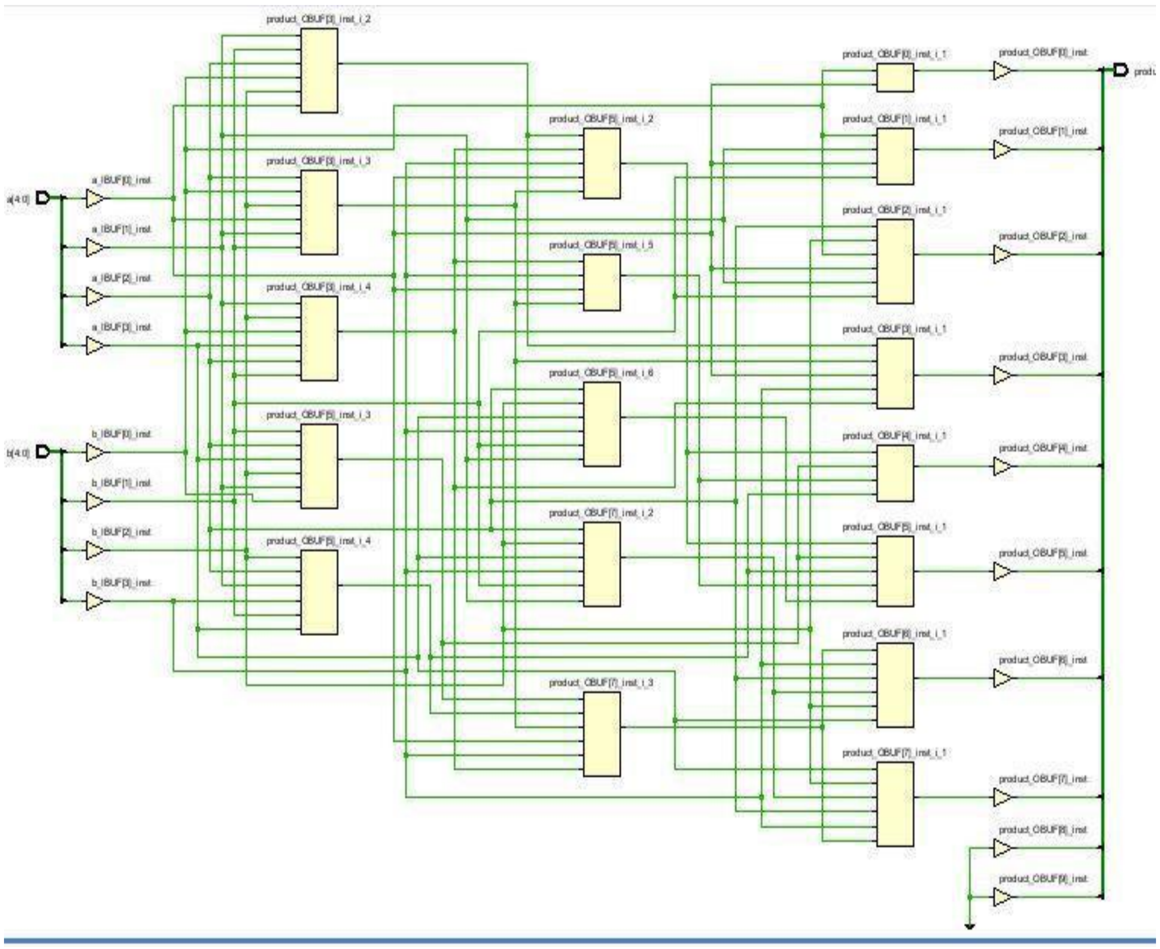
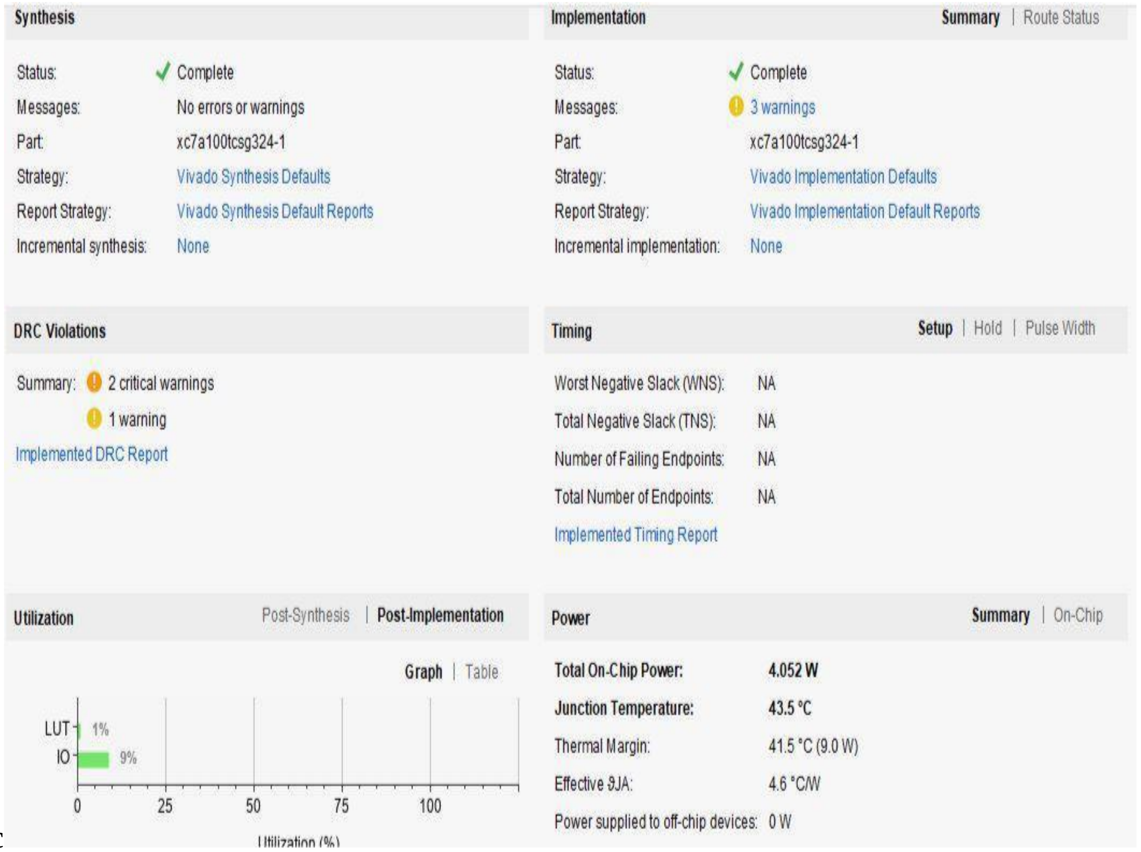


Fig 5.10 Synthesis Schematic of a Full Adder

5.6 IMPLEMENTATION OF DESIGN

i. Implement the design with the Vivado Implementation Defaults (Vivado Implementation 2013) settings and analyze the Project Summary output.
ii. Click on **Run Implementation** under the Implementation tasks of the Flow Navigator pane. The implementation process will be run on the synthesis output files. When the process is completed an Implementation Completed dialog box with three options will be displayed.
iii. Select **Open implemented design** and click **OK** as we want to look at the implemented design in a Device view tab.
iv. Click **Yes** to **close the synthesized design.** The implemented design will be opened.
v. Close the implemented design view and select the **Project Summary tab** (you may have to change to the Default Layout view) and observe the results.



Synthesis

Status:	✓ Complete
Messages:	No errors or warnings
Part:	xc7a100tcsg324-1
Strategy:	Vivado Synthesis Defaults
Report Strategy:	Vivado Synthesis Default Reports
Incremental synthesis:	None

Implementation — Summary | Route Status

Status:	✓ Complete
Messages:	3 warnings
Part:	xc7a100tcsg324-1
Strategy:	Vivado Implementation Defaults
Report Strategy:	Vivado Implementation Default Reports
Incremental implementation:	None

DRC Violations

Summary: 2 critical warnings
1 warning
Implemented DRC Report

Timing — Setup | Hold | Pulse Width

Worst Negative Slack (WNS):	NA
Total Negative Slack (TNS):	NA
Number of Failing Endpoints:	NA
Total Number of Endpoints:	NA

Implemented Timing Report

Utilization — Post-Synthesis | Post-Implementation

Power — Summary | On-Chip

Total On-Chip Power:	4.052 W
Junction Temperature:	43.5 °C
Thermal Margin:	41.5 °C (9.0 W)
Effective ϑJA:	4.6 °C/W
Power supplied to off-chip devices:	0 W

c

Fig 5.11 Implementation Results

vi. Select **the Reports tab**, and double-click on the **Utilization Report** entry under the Place Design section. The report will be displayed in the **auxiliary view pane** showing resources utilization.

5.7 SIMULATION PLATFORM

The Vivado software tool can be used to perform a complete design flow. The project was created using the supplied source files (HDL model and user constraint file). A behavioral simulation was done to verify the model functionality. The model was then synthesized, implemented.

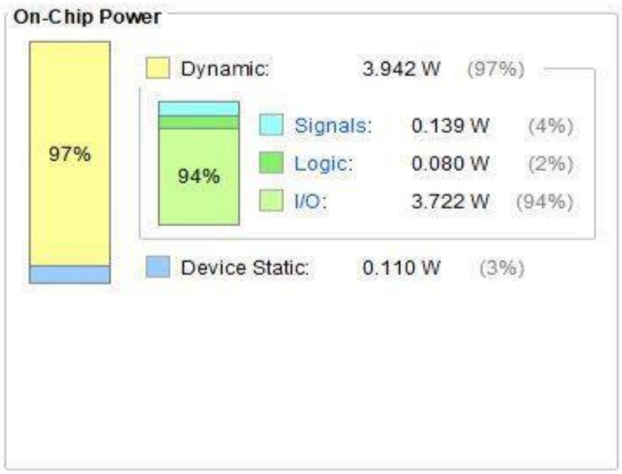
CHAPTER 6

SIMULATION RESULTS

In order to implement Low power multipliers, Hybrid adders are to be simulated in XILINX VIVADO software. Below are the simulation results for the Hybrid Adders. After simulation of all hybrid adders and full adders simulated values for power dissipation, area, look up table values and number of cells are obtained.

6.1 Power Analysis of Hybrid Adders:

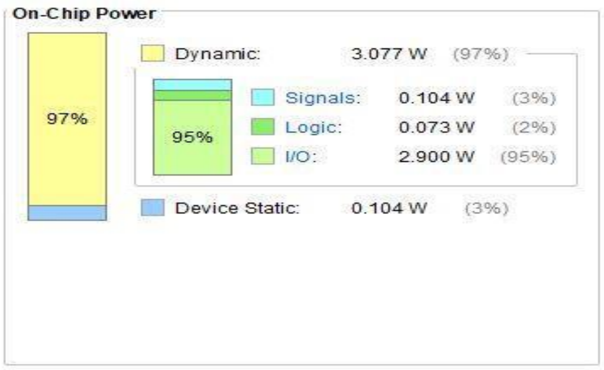
After Simulating the hybrid adders in the XILINX software the power analysis of the adders are as follows



On-Chip Power

Dynamic:	3.942 W	(97%)	97%
Signals:	0.139 W	(4%)	94%
Logic:	0.080 W	(2%)	
I/O:	3.722 W	(94%)	
Device Static:	0.110 W	(3%)	

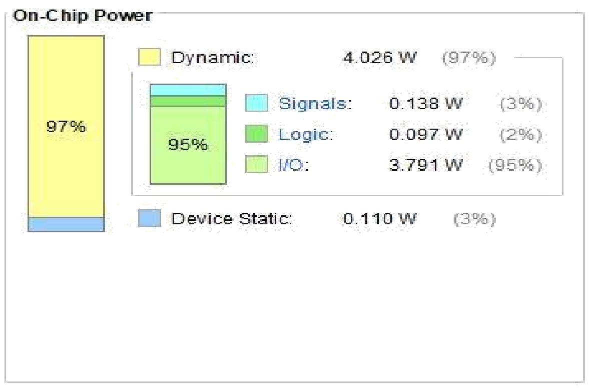
Fig 6.1 Power analysis of Full Adder



On-Chip Power

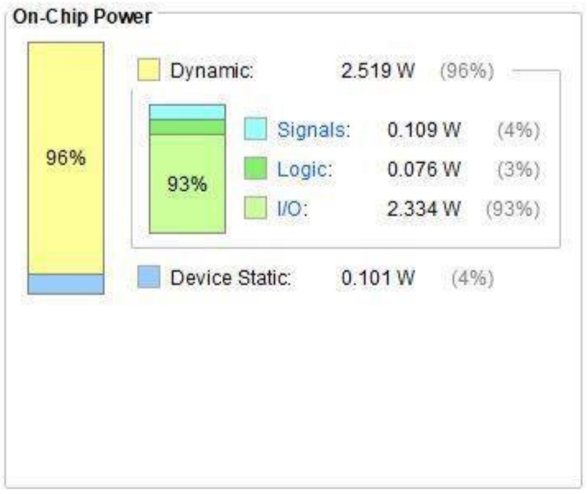
Dynamic:	3.077 W	(97%)	97%
Signals:	0.104 W	(3%)	95%
Logic:	0.073 W	(2%)	
I/O:	2.900 W	(95%)	
Device Static:	0.104 W	(3%)	

Fig 6.2 Power Analysis of 1-2 Adder



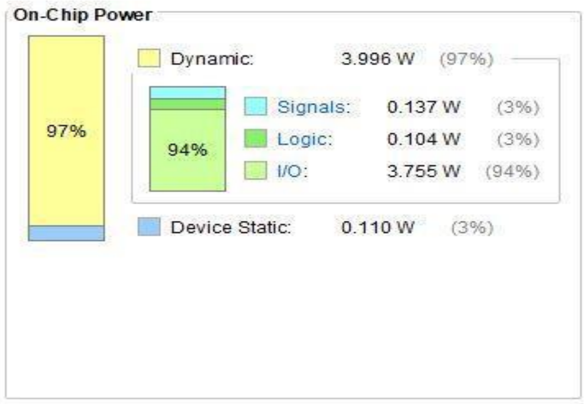
Dynamic:	4.026 W	(97%)
Signals:	0.138 W	(3%)
Logic:	0.097 W	(2%)
I/O:	3.791 W	(95%)
Device Static:	0.110 W	(3%)

Fig 6.3 Power Analysis of 2-1 adder



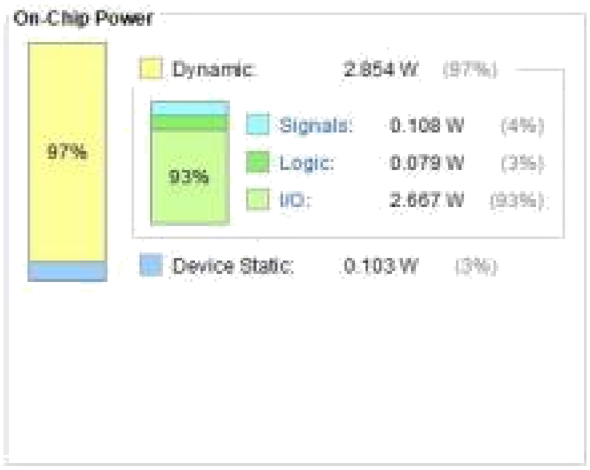
Dynamic:	2.519 W	(96%)
Signals:	0.109 W	(4%)
Logic:	0.076 W	(3%)
I/O:	2.334 W	(93%)
Device Static:	0.101 W	(4%)

Fig 6.4 Power Analysis of 2-2 Adder



Dynamic:	3.996 W	(97%)
Signals:	0.137 W	(3%)
Logic:	0.104 W	(3%)
I/O:	3.755 W	(94%)
Device Static:	0.110 W	(3%)

Fig 6.5 Power Analysis of 3-1 Adder



On-Chip Power

🟨	Dynamic:	2.854 W	(97%)
🟦	Signals:	0.108 W	(4%)
🟩	Logic:	0.079 W	(3%)
🟩	I/O:	2.667 W	(93%)
🟦	Device Static:	0.103 W	(3%)

Fig 6.6 Power Analysis of 3-2 Adder

From the above power analysis we can conclude that using of 3-2 adders and 2-2 adders results in low power consumption at 43.5^0 C Junction temperature.

6.2 Schematic Representation of Hybrid Adders

The Schematic Representation of the Hybrid Adders gives the information about the Nets, Cells and I/O ports used and the LUT'S required.

1. Full Adder:

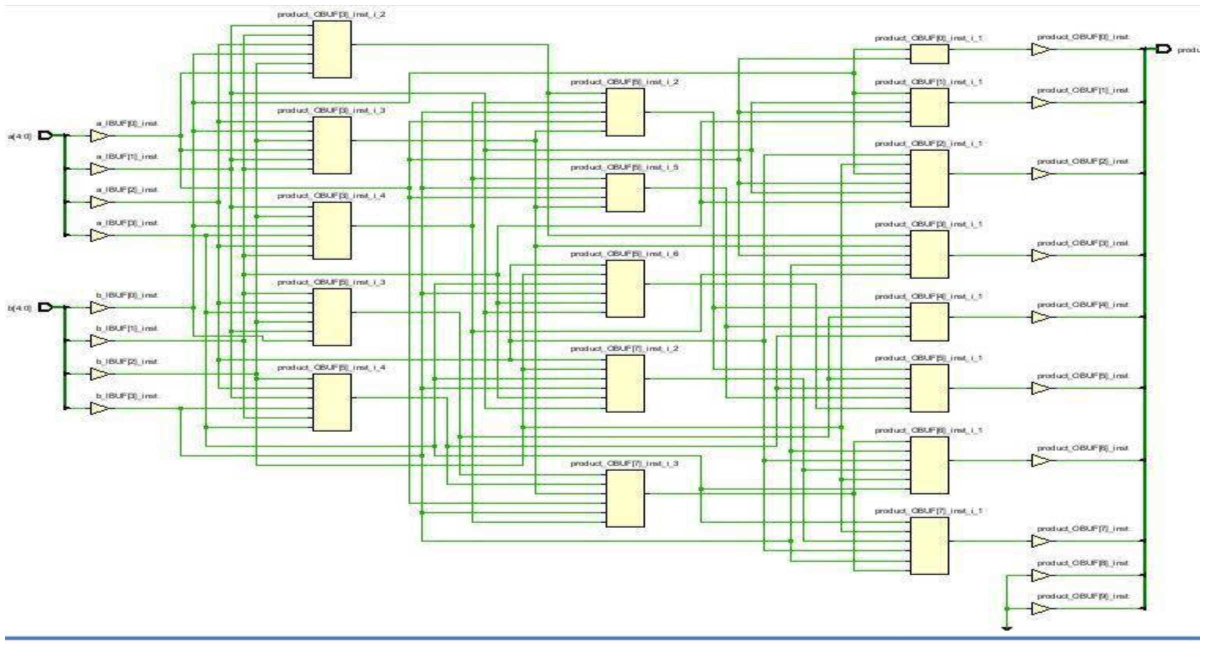


Fig 6.7 Schematic Representation of 4 bit Full Adder

Nets Required: 45

Cells Required:36

I/O ports :20

2. 1-2 Hybrid Adder:

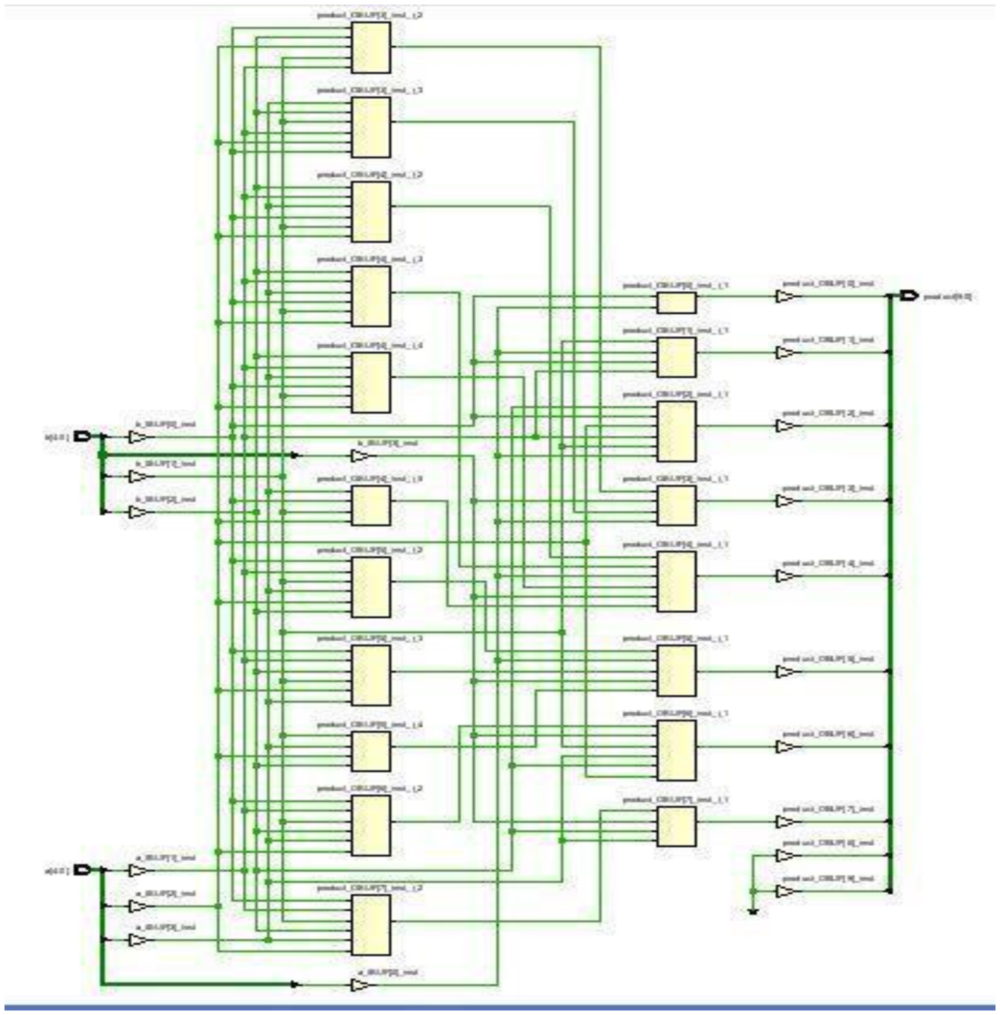


Fig 6.8 Schematic Representation of 4 bit 1-2 Hybrid Adder

Nets Required :46

Cells Required:37

I/O ports :20

3. 2-1 Hybrid Adder:

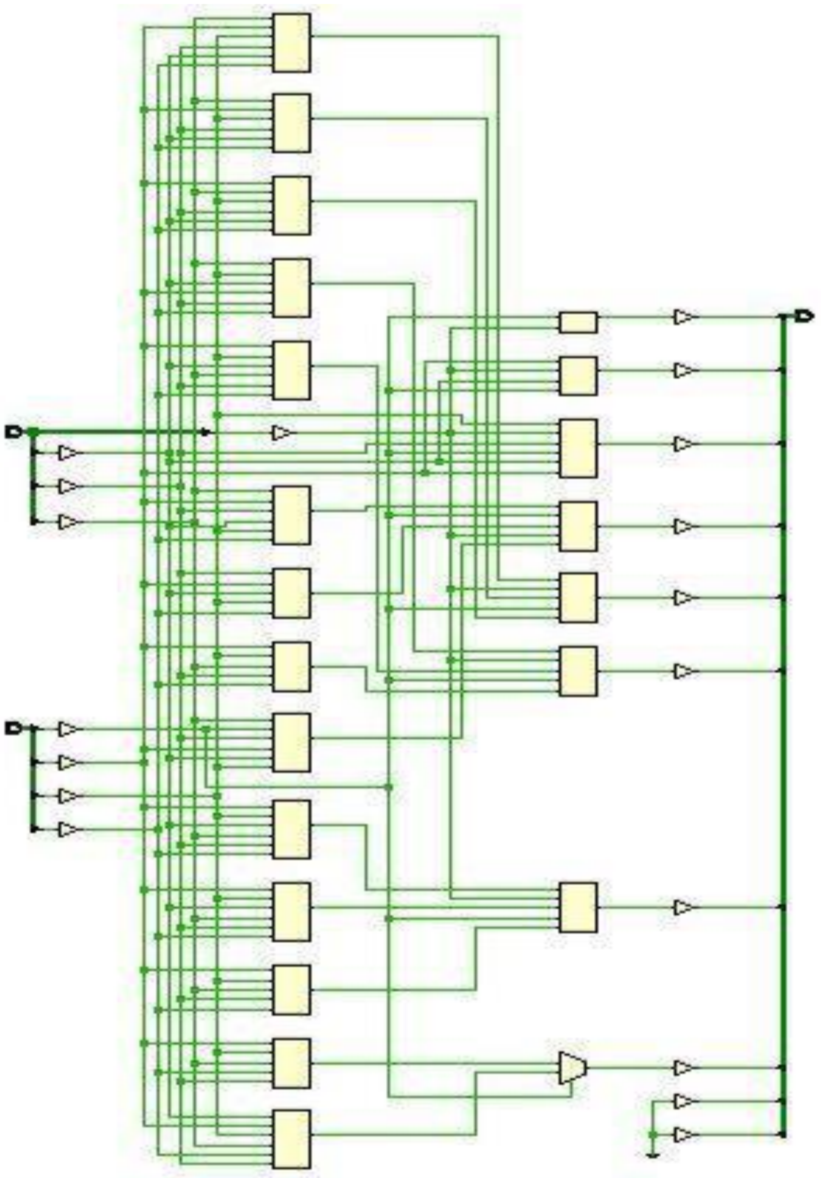


Fig 6.9: Schematic Representation of 4 bit 2-1 Hybrid Adder

Nets Required: 49

Cells Required:40

I/O ports :20

4. 2-2 Hybrid Adder

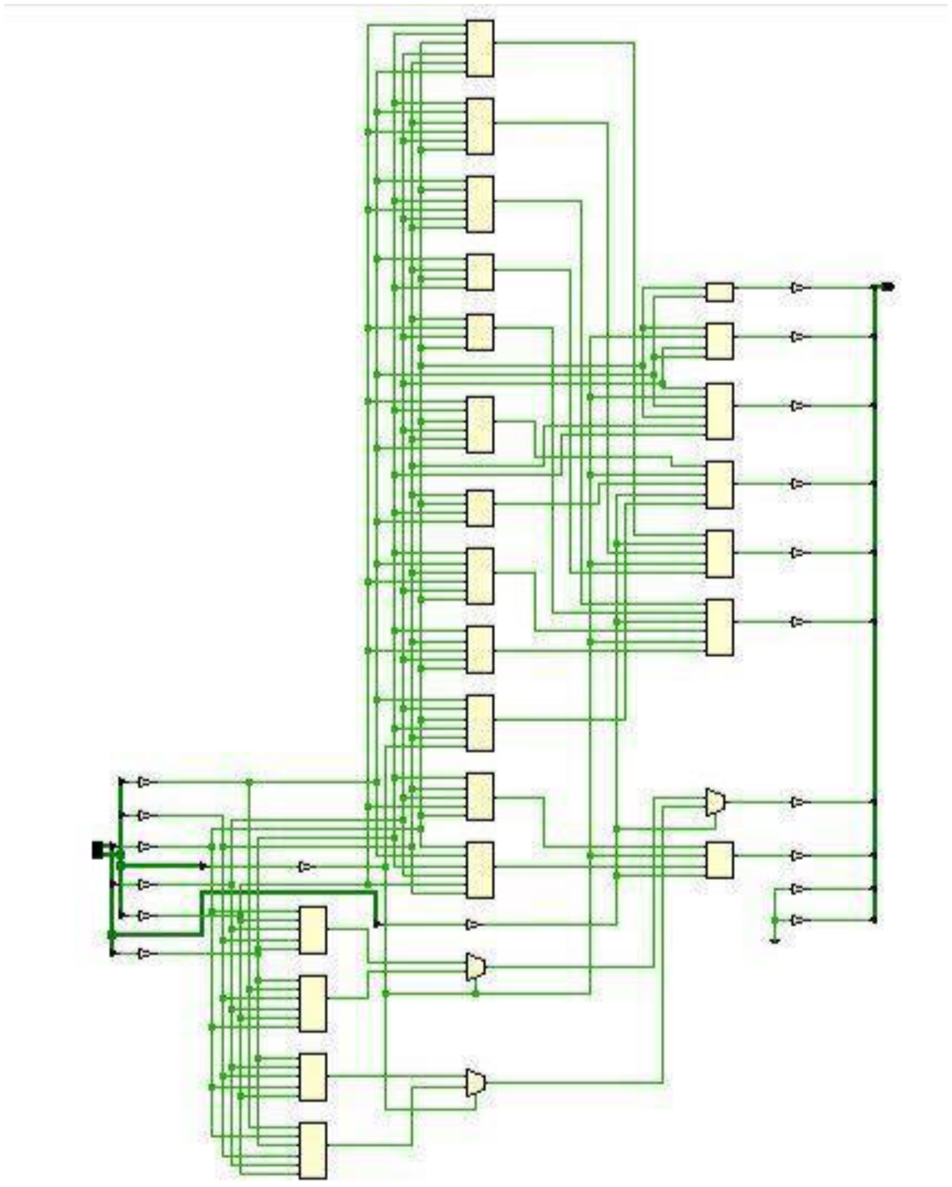


Fig 6.10 Schematic representation of 4 bit 2-2 Hybrid Adder

Nets Required: 53

Cells Required: 44

I/O ports : 20

5. 3-1 Hybrid Adder:

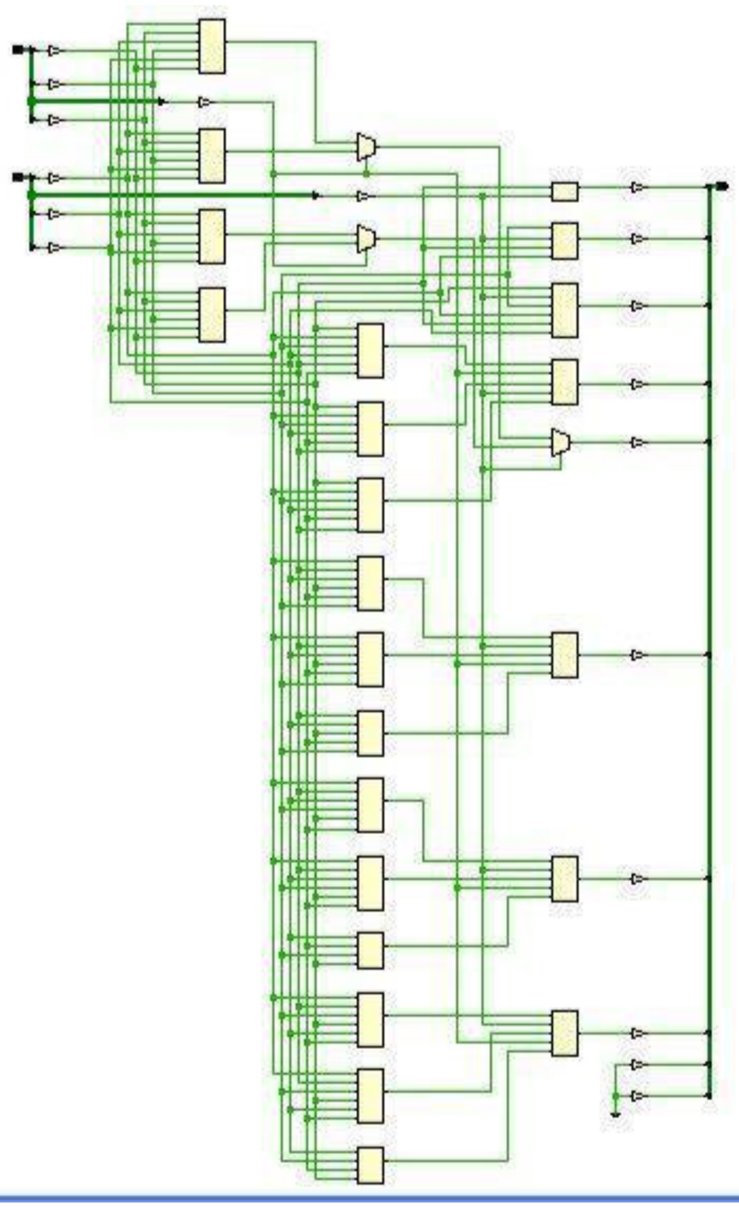


Fig 6.11 Schematic Representation of 4 bit 3-1 Hybrid Adder

Nets Required : 53

Cells required: 44

I/O ports : 20

6. 3-2 Hybrid Adder

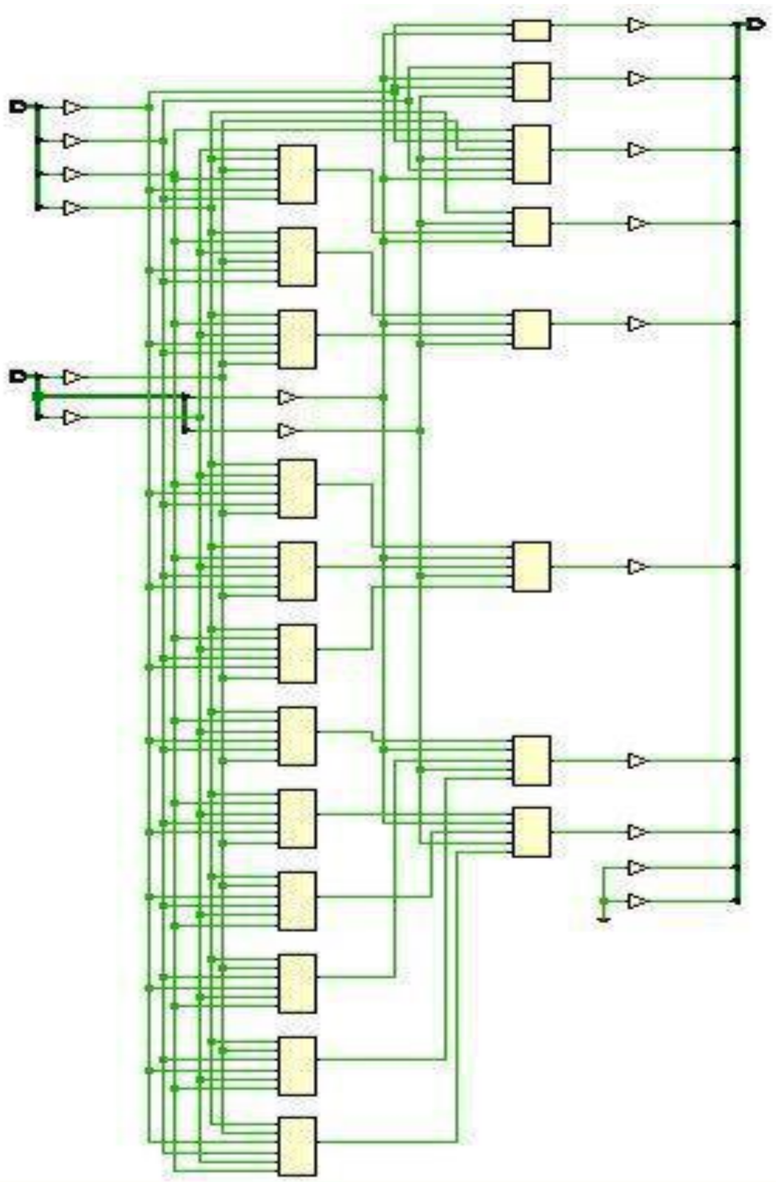


Fig 6.12 Schematic Representation of 4 bit 3-2 Hybrid Adder

Nets Required:49

Cells required:40

I/O ports :20

The Overall Performance Analysis of the 4 bit Hybrid adders are as follows:

Table 6.1: Performance Analysis of 4 bit Hybrid Adders

Type of Hybrid Adder	Area	Power	LUT
Full Adder	36 cells 20 I/O ports 45 Nets	4.052W	15
1-2 Hybrid Adder	37 Cells 20 I/O ports 46 Nets	3.182W	17
2-1 Hybrid Adder	40 cells 20 I/O ports 49 Nets	4.136W	19
2-2 Hybrid Adder	44 Cells 20 I/O ports 53 Nets	2.621W	20
3-1 Hybrid Adder	44 cells 20 I/O ports 53 Nets	4.106W	21
3-2 Hybrid Adder	40 Cells 20 I/O ports 49 Nets	2.957W	19

Based on the above analysis results show that 2-2 Hybrid Adder and 3-2 Hybrid Adder has low power consumption. 3-1,2-1,1-2 Hybrid Adders and full Adders show degraded performance compared to the 3-2 and 2-2 Hybrid Adder. Nevertheless since type 3-2 and

2-2 will be the most used in our multiplier and since NAND gates will replace AND gates to generate partial product bits, we expect the multiplier to offer low power consumption and lower area. The new architecture of the multiplier using Hybrid Adders is as follows

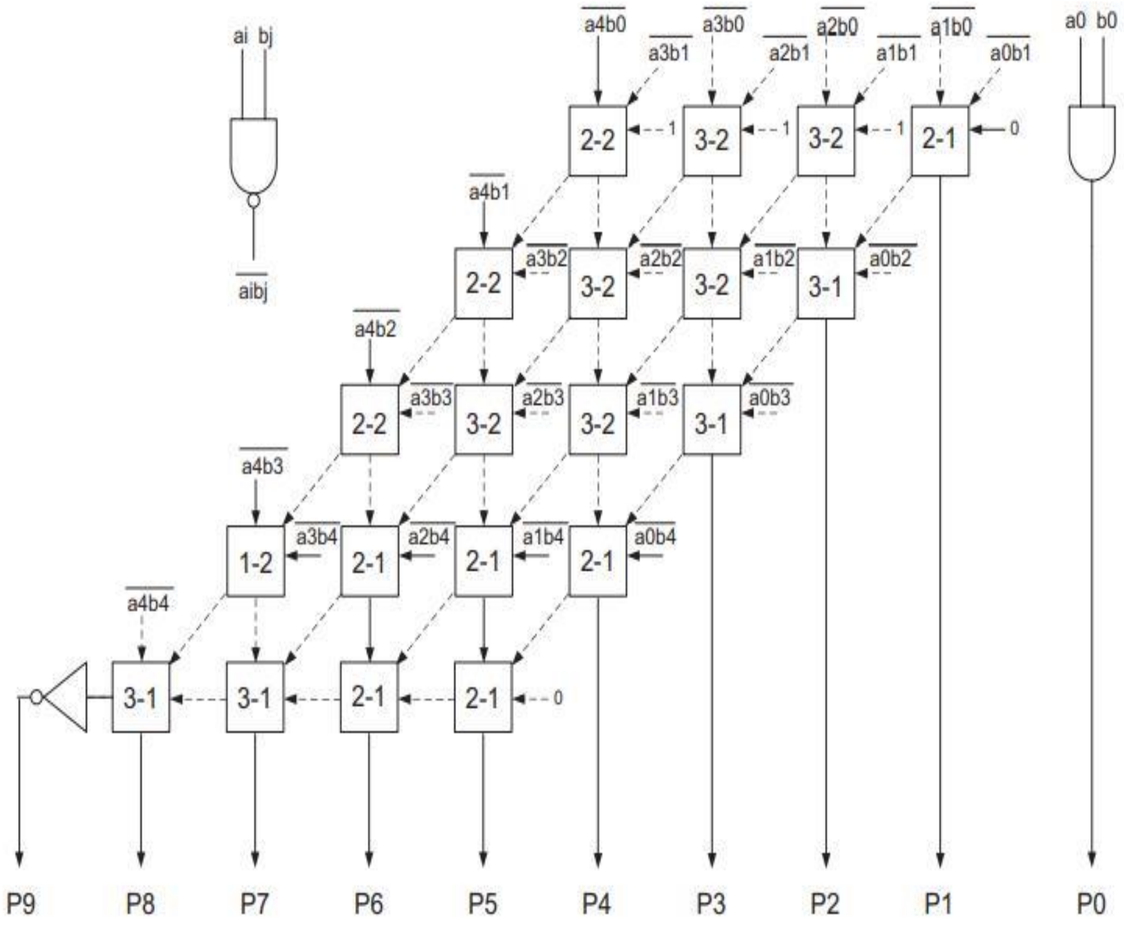


Fig 6.13 New Architecture of ALL NAND Multiplier

The ALL NAND array multiplier achieves low reduction in terms of power consumption and reduction in transistor count compared to Baugh Wooley's. This changes are mainly due to the hybrid adders since they have low power consumption and low transistor count.

CHAPTER 7

CONCLUSIONS

Five Hybrid Adders using the 4:2 Compressors were implemented. These are used to implement 4*4 bits array multipliers in which partial product bits are generated by means of NAND instead of AND gates. The proposed ALL-NAND array multiplier shows a decrease in power dissipation and in transistor count compared to Baugh Wooley. Therefore, our All-NAND signed array multipliers are promising for low area and low power applications.

REFERENCES

1. Z. Huang, High-level optimization techniques for low-power multiplier design, Ph.D. Thesis, University of California, 2003.
2. J.-M.W.S.-C. Fang, W.-S. Feng, New efficient designs for XOR and XNOR functions on the transistor level, IEEE J. Solid State Circuits 29 (1994) 780–786.
3. H. Lee, G.E. Sobelman, A new low-voltage full adder circuit, in: 7th Great Lakes Symposium on VLSI, 1997, pp. 88–92.
4. A.M. Shams, M.A. Bayoumi, A novel high-performance CMOS 1-bit full-adder cell, IEEE Trans. Circuits Syst. II: Analog Digital Signal Process. 47 (2000) 478–481.
5. L. Junming, S. Yan, L. Zhenghui, W. Ling, A novel 10-transistor low-power highspeed full adder cell, in: 6th International Conference on Solid-State and Integrated-Circuit Technology, 2001, pp. 1155–1158.
6. Y. Jiang, A. Al-Sheraidah, Y. Wang, E. Sha, J.-G. Chung, A novel multiplexer based low-power full adder, IEEE Trans. Circuits Syst. II: Express Briefs 51 (2004) 345–348.
7. C.-H. Chang, M. Zhang, J. Gu, A novel low power low voltage full adder cell, in: 3rd International Symposium on Image and Signal Processing and Analysis, IEEE, New York, 2003, pp. 454–458.
8. N.H.E. Weste, D. Harris, CMOS VLSI Design: A Circuits and Systems Perspective, Addison-Wesley, Reading, MA, 2005.
9. Z. Abid, H. El-Razouk, D.A. El-Dib, “Low power multipliers based on new hybrid full adders”, Microelectrosnics Journal, 39 (2008) 1509– 1515