FPGA Implementation of Canonical Signed Digit Algorithm Based Floating Point Multiplication

A Project report submitted in partial fulfillment of the requirements for the award of the degree of

BACHELOR OF TECHNOLOGY

IN

ELECTRONICS AND COMMUNICATION ENGINEERING

Submitted by

G.Sai Bhavani (316126512184) P.Vamsi Krishna(316126512162)

S.Kamala Kumar (31612651294) P.Pavani (316126512210)

Under the guidance of

Dr. K.V.Gowreesrinivas

Assistant Professor

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)

2019-2020

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 **"FPGA Implementation of Canonical Signed Digit Algorithm Based Floating Point Multiplication"** submitted by **G.Sai Bhavani(316126512184), P.Vamsi Krishna (316126512162), S.Kamala Kumar(316126512194), P.Pavani (316126512210)** in partial fulfillment of the requirements for the award of the degree of **Bachelor of Technology** in **Electronics & Communication Engineering** of Andhra University, Visakhapatnam is a record of bonafide work carried out under my guidance and supervision.

K.V.G. Srinivas	[illegible]
Project Guide	**Head of the Department**
Dr. K.V.Gowreesrinivas	**Dr. V.Rajyalakshmi**
Assistant Professor	**Professor**
Department of E.C.E	Department of E.C.E
ANITS	ANITS

ACKNOWLEDGEMENT

We would like to express our deep gratitude to our project guide **Dr. K.V.Gowreesrinivas** Assistant Professor, Department of Electronics and Communication Engineering, ANITS, for his/her 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
G.Sai Bhavani(316126512184),
P.Vamsi Krishna(316126512162),
S.KamalaKumar(31612651194),
P.Pavani (316126512210).

CONTENTS

ABSTRACT

In today’s generation the requirement of very high-speed operations in processors is increased. To speed up the process of computation, arithmetic operations such as addition ,subtraction and multiplication are used in various digital circuits. Floating point multiplication is a critical operation in high power computing applications such as image processing, digital signal processing.

The main multiplier characteristics are good accuracy, increase in speed, reduction in area and less power consumption. As speed is always a constraint in the multiplication operation, increase in speed can be achieved by reducing the number of stages in the calculation process. Since the multiplier requires the longest delay among the basic operational blocks in digital system, the critical path is determined more by the multiplier. Furthermore, multiplier consumes much area and dissipates more power. Hence designing multipliers which offer either of the following design targets – high speed, lower power consumption, less area or even a combination of them is our prime concern.

The main aim of the project is to achieve the above design targets of a floating point multiplier using Canonical Signed Digit(CSD) algorithm.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVATIONS

DSP	Digital Signal Processing
FPM	Floating Point Multiplication
CSD	Canonical Signed Digit
FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
LSB	Least Significant Bit
IEEE	Institute of Electrical and Electronics Engineers
RCA	Ripple Carry Adder
CLA	Carry Look -Ahead Adder
CSA	Carry Save Adder
CA	Carry-Select Adder
SPST	Spurious Power Suppression Technique
UT	Urdhva Triyakbhyam
DDR	Double Data Rate
LUT	Look Up Tables
PLL	Phase-Locked Loop
HR	High Range
ODT	On-Die Terminations
PHY	Physical Layer
MiB	mebibyte
RTL	Register Transfer Level
EDA	Electronic Design Automation

CHAPTER1
INTRODUCTION

The multiplication operation is one of the important operations in digital signal processing (DSP) used in applications such as the Discrete Fourier Transform, Fast Fourier Transform, convolution or digital filters. Thus, there has been a continuous research for refining predefined multipliers and developing new approaches for efficient multiplier architecture. There are many existing multiplier architectures such as Shift and Add multiplier, Booth multiplier, Wallace tree multiplier,Array multiplier etc. Shift and Add multiplier is the simplest among all however, large number of gates are required making time inefficient architecture. High performance multipliers are preferred with regular structures or reduced number of partial products. The array multiplier and Wallace tree multiplier are based on the arrangement of adders aiming reduction in overall critical path of the multiplication operation.

Many significant amounts of work have been published using the concept of Vedic mathematics, obtaining efficient multiplier, divider, squaring and cube architectures. The efficiency of multiplier is based on factors: (i) larger operation is reduced to the smaller operation. (ii) the speed of addition operation in accumulation of partial products is increased using carry select adders. The multiplier architecture is found to be efficient as compared to the state-of-art Vedic multipliers .The reduction in number of stages in a Vedic multiplication process leads to significant reduction in the propagation delay along with switching power consumption. The multiplication operation is based on Urdhva Tiryagbhyam algorithm of Vedic mathematics.

Generally, most of the multipliers consumes much area,consumes more power and work with less accuracy.hence designing multipliers which offer either of the following design targets-high speed,low power consumption,less area or even a combination of all these is of substantial research interest.

PROJECT OBJECTIVE

The main aim of the project is to simulate and implement FPM(Floating Point Multiplication) multiplication using CSD(Canonical Signed Digit)algorithm and compare it with existing multipliers.Vivado design suite is used for simulation. Synthesis and implementation is done using Nexys 4 DDR board,which is ready to use digital cir cuit development platform based on the latest Artix -7 Field Programmable Gate Array (FPGA) (XC7A100T-1CSG324C) from Xilinx. The performance is compared in terms of power,area and delay.

PROJECT OUTLINE

This project report is presented over the 4 remaining remaining chapters.Chapter 2 presents Floating Point Multiplication and provides the fundamentals of various adders and multipliers. Chapter 3 explains about Canonical Signed Digit (CSD) algorithm. Chapter 4 describes simulation and synthesis tools necessary to simulate and synthesize the given hardwareusing Verilog Hardware Description Language .Chapter 5 presents the simulation results which are simulated using Vivado Design Suite simulator.Also the synthesis and implementation results are carried out using Nexys 4 DDR board,which is ready to use digital cir cuit development platform based on the latest Artix -7 Field Programmable Gate Array (FPGA) (XC7A100T-1CSG324C) from Xilinx. Finally,the results of the project work and conclusions are drawn.

CHAPTER 2

FLOATING POINT MULTIPLICATION

2.1 NUMBER SYSTEM

The number system is a system generally used for representing or expressing numbers. It is a mathematical notation which can be a combination of numbers and alphabets .There are various kinds of number systems like binary,decimal,octal,hexadecimal based on the base.The base is also called as “radix”.

2.1.1 FIXED POINT

The term 'fixed point' refers to the corresponding manner in which the numbers are represented.A fixed point decimal number system has a limited number of digits and a decimal point in a fixed location.

To define a fixed-point type conceptually, all we need are two criterions:

width of the number and binary point position within the number

We will use the notation fixed <w, b> where w stands for the number of bits used as a whole (the Width of a number), and b stands for the position of binary point counting from the Least Significant Bit(LSB) (counting from 0).

For example, fixed<8,3> denotes an 8-bit fixed point number, of which 3 right most bits are fractional. Therefore, the bit pattern:

represents a real number:

00010.110

= 1 * 2^1 + 1 * 2^-1 + 1 * 2^-2

= 2 + 0.5 + 0.25

= 2.75

Fixed point numbers are indeed an in depth relative to integer representation. The two only differs within the position of binary point. In fact, you would possibly even consider integer representation as a "special case" of fixed-point numbers, where the binary point is at position 0. All the arithmetic operations a computer can operate integer can therefore be applied to fixed point number also . The disadvantage of number , is than in fact the loss of range and precision. For example, in a fixed representation, our fractional part is only precise to a quantum of 0.5. We cannot represent number like 0.75. We can represent 0.75 with fixed, but then we lose range on the integer part.

2.1.2 FLOATING POINT

The floating point is the arithmetic using the representation of real numbers as an approximation in supporting trade-off between range and precision. For this reason, floating-point computation is usually found in systems which include very small and really large real numbers, which require fast processing times.

In computers the real numbers are represented in floating point (FP) format. Usually this suggests that the amount is split into exponent and fraction, which is additionally referred to as significand or mantissa. A scientific notation with a symbol , exponent, mantissa called floating point representation is employed for the important numbers. Its format is

Real number→mantissa×base^exponent

Where significand is an integer, base is an integer greater than or adequate to two, and exponent is additionally an integer. For example:

1.2345 = 12345 x 10^-4

2.1.3 SINGLE PRECISION FLOATING POINT MULTIPLICATION

Most modern computers use Institute of Electrical and Electronics Engineers (IEEE) 754 standard for representation of floating-point numbers. One of the most commonly used formats is the binary32 format of IEEE 754:

Sign bit-1 bit

Exponent-8 bits

Significand/mantissa-23 bits

Note that exponent is encoded using an offset-binary representation, which suggests it is often off by 127. So, if usually 10000000 in binary would be 128 in decimal, in single-precision the worth of exponent is:

exponent=128−offset=128−127=1

2.1.4 FLOATING POINT MULTIPLICATION

From the literature, it is observed that various multipliers such as Array multiplier, Vedic multiplier and Radix based multipliers are involved in multiplication.

In general, Floating Point Multiplication (FPM) of two numbers involves four steps:

• Non-signed multiplication of mantissas

• Normalization of the result

• Addition of the exponents, taking into account the bias.

• Calculation of the sign.

Let us consider two numbers

a=6.96875

b=−0.3418

Normalized values and biased exponents:

a=6.96875=1.7421875×2^2

b=−0.3418=−1.3672×2^−2

The exponents:

Exponent a=2

Exponent b=−2

The numbers in IEEE754 *binary32*:

a=01000000110111110000000000000000(binary32)

=10111110101011110000000001101001(binary32)

The mantissa could be rewritten as following totalling 24 bits per operand:

Mantissa a=1.101111100000000000000002

Mantissa b=1.010111100000000011010012

Their multiplication totals in 48 bits:

Mantissa=1.0011000011100010101101101110111000000000000000   002

Which has to be truncated to 24 bits:

mantissaa×b=1.001100001110001010110112=2.3819186687469482421875

10

The exponents 2 and -2 can easily be summed up so only last thing to do is to normalize fraction which means that the resulting number is:

a×b=−2.3819186687469482421875=−1.19095933437347412109375×21

Which could be written in IEEE 754 *binary32* format as:

a×b=01000000000110000111000101011011binary32

2.2 ADDERS

An adder is a digital circuit which performs addition of numbers. In many computers and other types of processors, adders are used in the ALU and also in other parts of the processors. Adders are used to calculate addresses, table indices and are also used to perform increment and decrement operations.The most common adders operate on binary numbers.

2.2.1 HALF ADDER(HA)

A combinational circuit that performs the addition of two single bits is called Half Adder. The **half adder** adds two binary digits *A* and *B*. It has two outputs, sum and carry. The carry signal represents an overflow into the next digit of a multi-digit addition. To design a simple half adder we use an XOR gate for sum and an AND gate for carry.

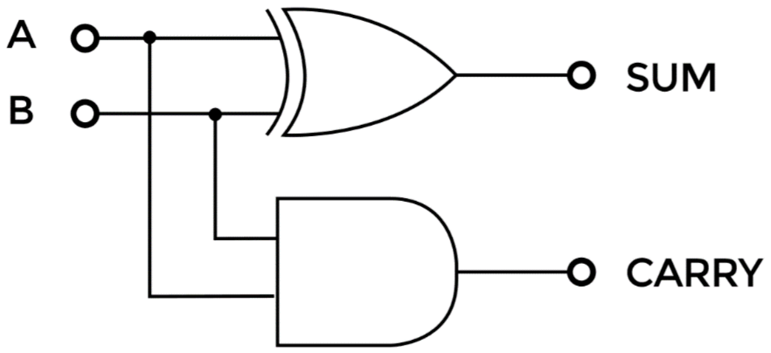


Fig 2.1 Half adder logic diagram

A	B	SUM	CARRY
0	0	0	0
0	1	1	0
1	0	1	0
1	1	1	1

Table 2.1 Half adder truth table

2.2.2 FULL ADDER(FA)

A combinational logic circuit that performs the addition of three single bits and produces two outputs is called Full Adder. A, B and Cin are the inputs. Sum and C-OUT are the outputs. In half adder we can add 2-bit binary numbers but we cannot add carry bit along with the two binary numbers. But in full adder we can carry in bit along with two binary numbers.

A	B	Cin	S	Cout

0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	0	1

Table 2.2 Full adder truth table

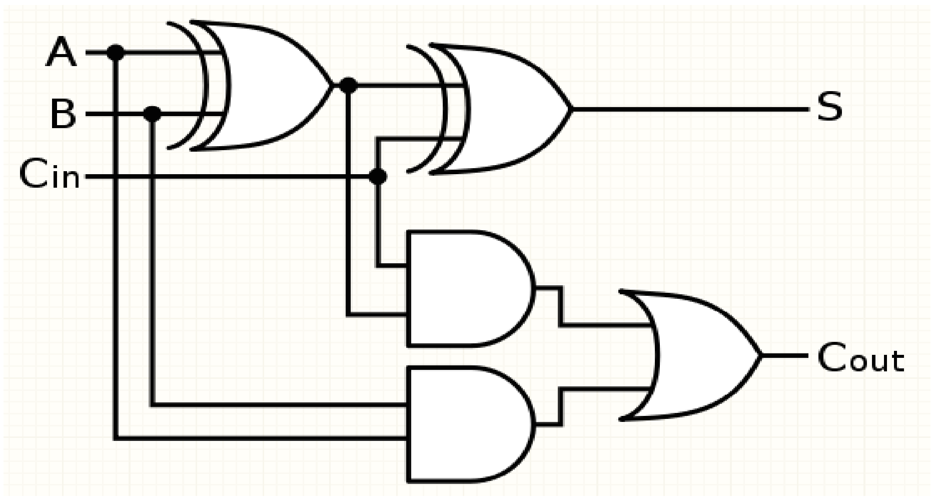


Fig 2.2 Full adder logic diagram

2.2.3 RIPPLE CARRY ADDER(RCA)

Ripple carry adder is a structure of multiple full adders is cascaded in a manner to gives the results of the addition of an n bit binary sequence and the carry will be generated at every full adder stage. These carry output at each full adder stage is forwarded to its next full adder and applied as a carry input to it. This process continues up to its last stage of adder circuit. So, each carry output bit is rippled to the next stage of a full adder and hence named as “RIPPLE CARRY ADDER”. The most important feature of it is to add the input bit sequences if the sequence is 4 bit or 5 bit or any.

There are various types in ripple-carry adders. They are:

- 4-bit ripple-carry adder
- 8-bit ripple-carry adder
- 16-bit ripple-carry adder

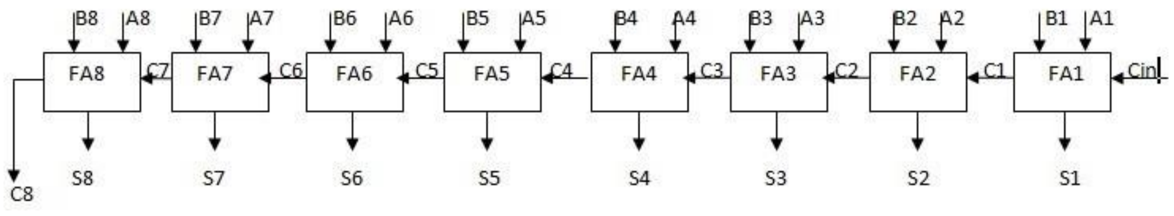
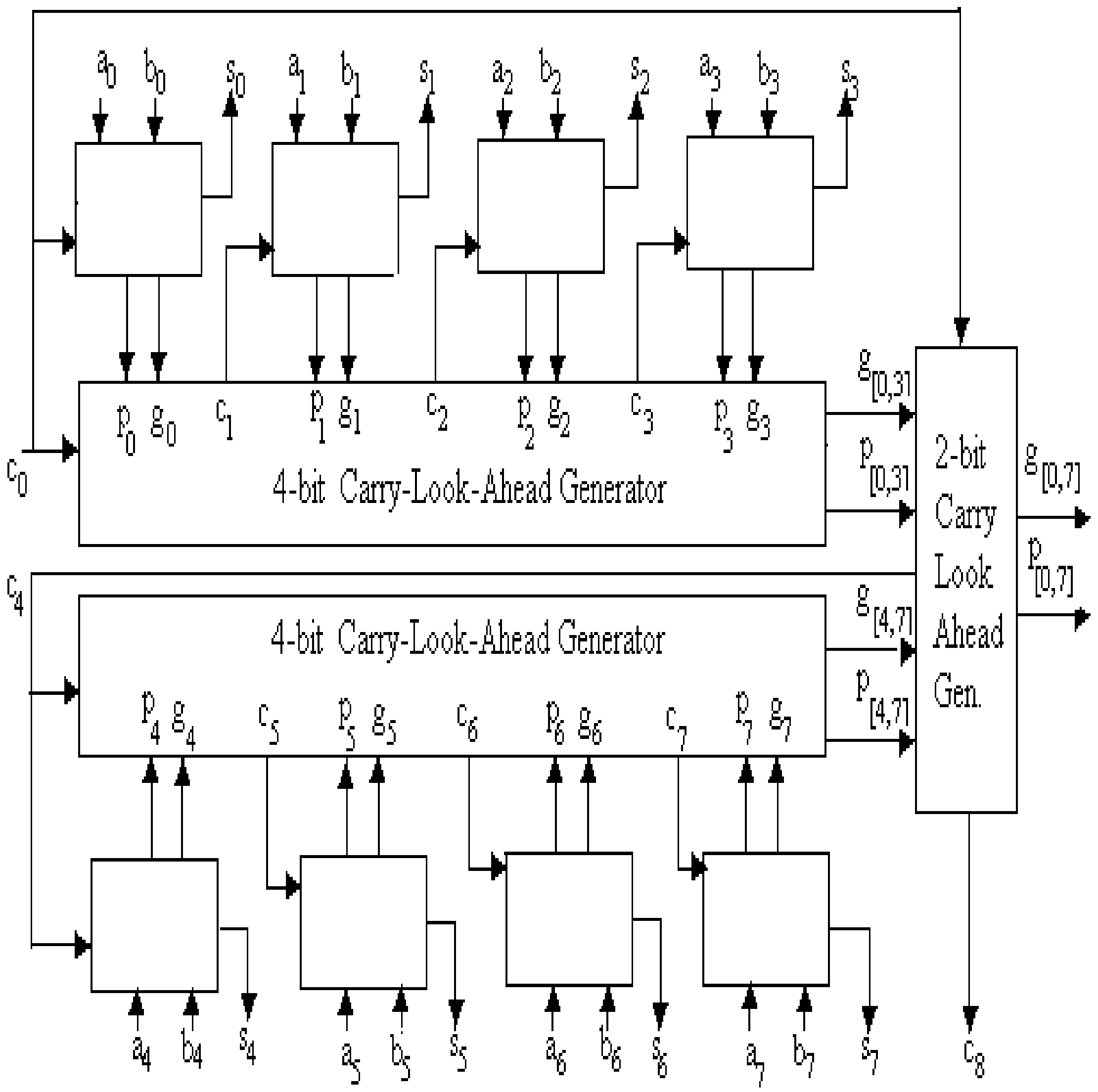


Fig 2.3 Ripple Carry adder logic diagram

2.2.4 CARRY LOOK AHEAD ADDER(CLA)

In ripple carry adders, for each adder block, the two bits that are to be added are available instantly. However, each adder block waits for the carry to arrive from its previous block. So, it is not possible to generate the sum and carry of any block until the input carry is known. The block waits for the block to produce its carry. So, there will be a considerable time delay which is carry propagation delay.

A carry look-ahead adder reduces the propagation delay by introducing more complex hardware. In this design, the ripple carry design is suitably transformed such that the carry logic over fixed groups of bits of the adder is reduced to two-level logic.



a_0 b_0 s_0 a_1 b_1 s_1 a_2 b_2 s_2 a_3 b_3 s_3
p_0 g_0 c_1 p_1 g_1 c_2 p_2 g_2 c_3 p_3 g_3
c_0
4-bit Carry-Look-Ahead Generator
$g_{[0,3]}$
$p_{[0,3]}$
2-bit Carry Look Ahead Gen.
$g_{[0,7]}$
$p_{[0,7]}$
c_4
4-bit Carry-Look-Ahead Generator
$g_{[4,7]}$
$p_{[4,7]}$
p_4 g_4 c_5 p_5 g_5 c_6 p_6 g_6 c_7 p_7 g_7
a_4 b_4 s_4 a_5 b_5 s_5 a_6 b_6 s_6 a_7 b_7 s_7 c_8

Fig 2.4 Carry look ahead adder

2.2.5 CARRY SAVE ADDER(CSA)

A carry save adder is a type of digital adder used in computer microarchitecture to compute the sum of three or more n-bit numbers in binary. It differs from other digital adders in that it outputs two numbers of the same dimensions as the inputs, one which is a sequence of partial sum bits and another which sequence of carry bits.

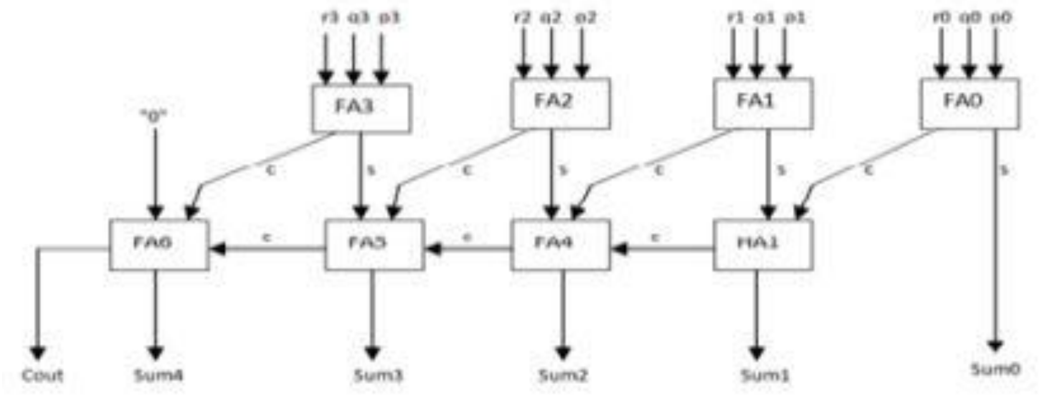


Fig 2.5 Carry save adder

2.2.6 CARRY- SELECT ADDER(CA)

In electronics, a carry-select adder is a particular way to implement an adder which is a logic element that computes the (n+1)-bit sum of two n-bit numbers.

The carry-select adder generally consists of two ripple carry adders and a multiplexer. Adding two n-bit numbers with a carry-select adder is done with two adders (therefore two ripple carry adders), in order to perform the calculation twice, one time with the assumption of the carry-in being zero and the other assuming it will be one. After the two results are calculated, the correct sum, as well as the correct carry-out, is then selected with the multiplexer once the correct carry-in is known.

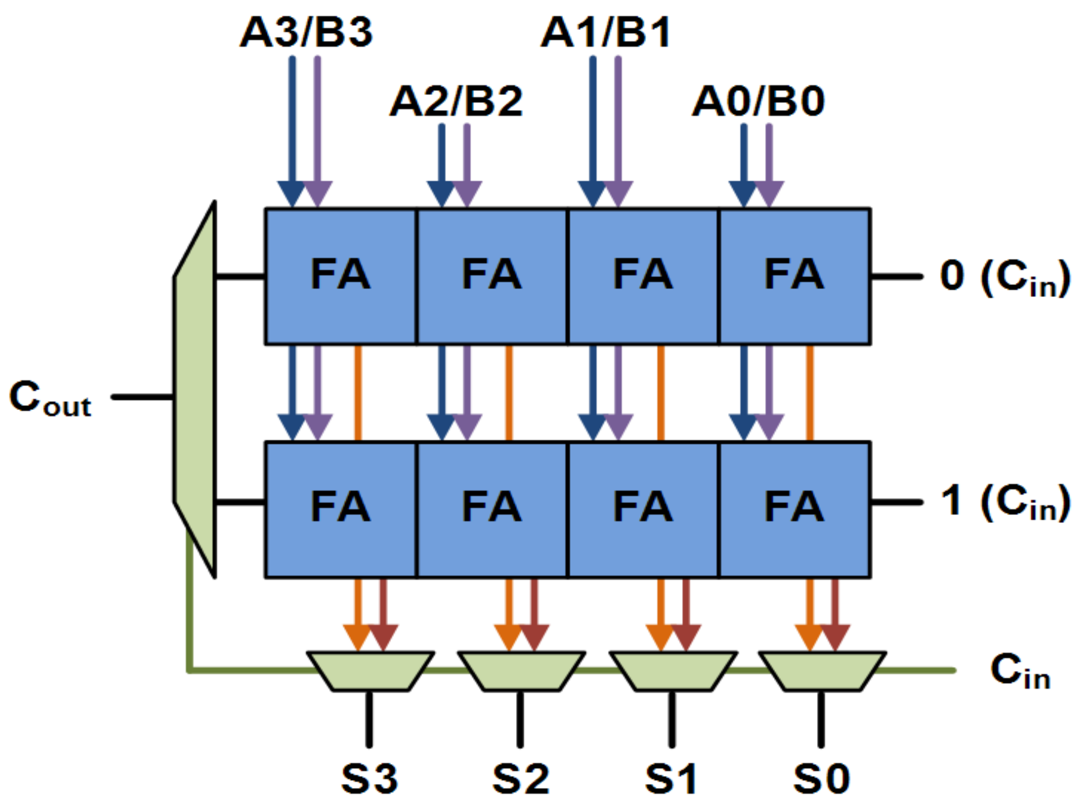


Fig 2.6 Carry select adder

Above is the basic building block of a carry-select adder, where the block size is 4. Two 4-bit ripple carry adders are multiplexed together, where the resulting carry and sum bits are selected by the carry-in. Since one ripple carry adder assumes a carry-in of 0, and the other assumes a carry-in of 1, selecting which adder had the correct assumption via the actual carry-in yields the desired result.

2.2.7 SPURIOUS POWER SUPPRESSION TECHNIQUE ADDER(SPST)

Adder is a circuit that is combinational and calculates the sum of three (full adder) or two (half adder) inputs. Full adder can be cascaded to produce n-stages of adder. This cascaded adder structure is called as parallel adder. The sum and carry of any stage cannot be calculated until the input carry occurs, this leads to a delay in the addition process. In order to overcome the delay, carry look ahead adder is proposed which is said to be a fast adder. To improve the speed of carry, look ahead adder, Spurious Power Suppression Technique (SPST) is used.

The SPST has a detection logic circuit find out if a transition in data bits of the result will occur in circuits, e.g., multipliers or adders. When a part of the data doesn't cause any change in the final result of the circuit, that portion of the data is latched to avoid unwanted transitions inside the processing units.

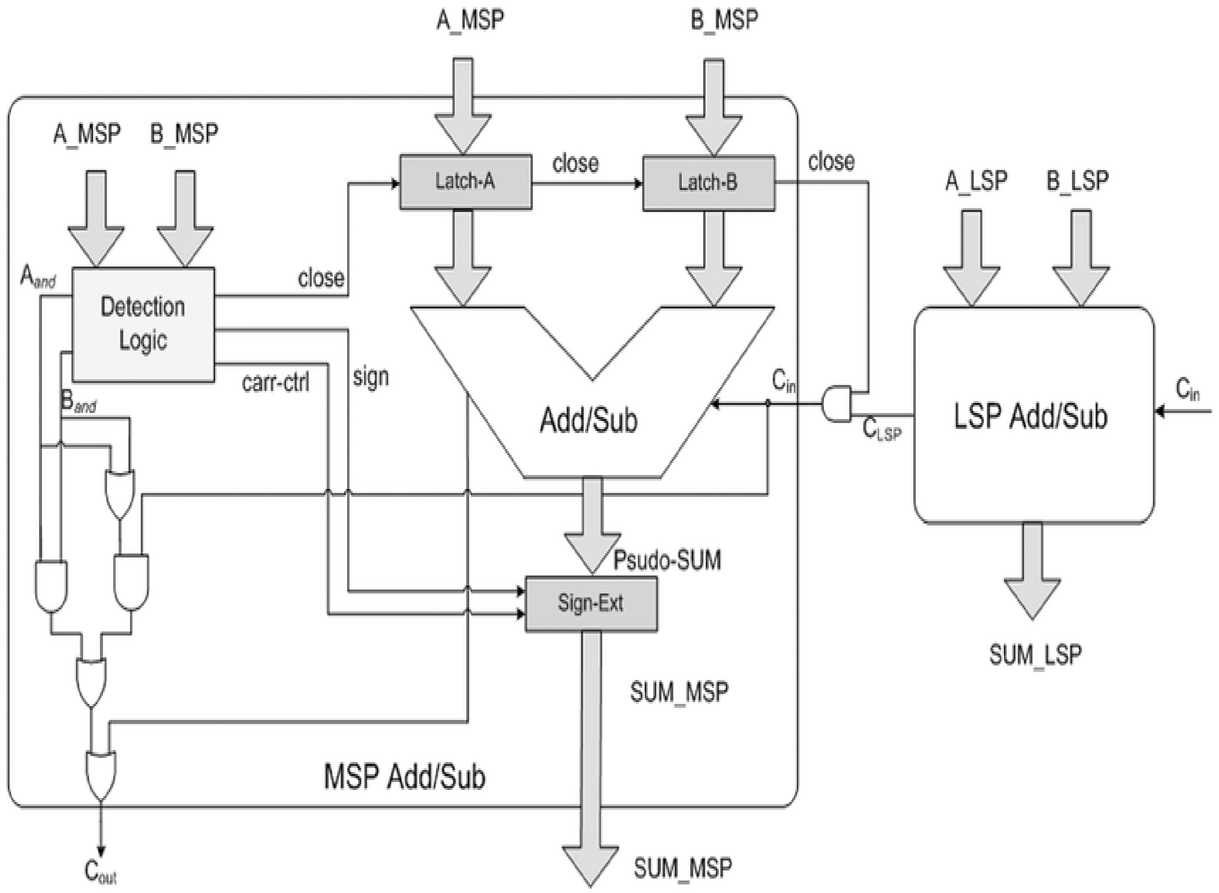


Fig 2.7 SPST adder

2.2.8 Comparision Between Different Types Of Adders:

S.No	Adder Type	No.Of AND Gates	No.Of EX-OR Gates	No.Of OR Gates
1	Half-Adder	1	1	-
2	Full Adder	2	2	1
3	8-Bit Ripple Carry Adder	16	16	8
4	8-Bit Carry – Look Ahead Adder	16	16	8
5	4-Bit Carry Save Adder	16	16	8
6	4-Bit Carry Select Adder	16	16	8

Table 2.3 Comparision between different types of Adders

2.3 MULTIPLIERS

Today’s processors require a very high-speed operation. Arithmetic operations such as addition, subtraction and multiplication are used in various digital circuits for high speed computation. Multiplication is the basic arithmetic operation in signal processing. All the signal and data processing operations like digital signal processing contain multiplication.

Speed is an essential parameter in a multiplication operation. It can be increased by reducing the number of steps in the computation process. In a digital system design, the parameters that determine the performance of the system are speed, power and area. As the multiplier requires the longest delay among the basic operational blocks in a digital system, the critical path is determined more by the multiplier. Also a multiplier consumes much area and dissipates more power. Hence, designing multipliers which offer a combination of the above parameters is better.

A multiplier is one of the key hardware blocks in many digital signal processing systems. Some of the DSP applications where a multiplier plays an important role include digital filters, digital communications and spectral analysis. Many current DSP applications are targeted at portable, battery-operated systems, in order that power dissipation becomes one among the first design constraints. As multipliers are complex circuits and have to operate at a high system clock rate, reducing the delay of a multiplier is an important a part of satisfying the overall design. This operation involves two major steps:Partial product generation and accumulation.

2.3.1 PARTIAL PRODUCTS

Formation of partial products

The basic algorithm for multiplication of two binary numbers, M(multiplier) and N(multiplicand) makes use of the distributive property of multiplication. That is, if M can be written as a sum of smaller numbers.

$$M.N = (M_0+M_1+\ldots\ldots\ldots +M_{m-1}).\ N = M_0N+M_1N+\ldots\ldots\ldots +M_{m-1}N$$

Also a multiplier consumes most of the area and dissipates more power. Hence designing multipliers which offer either of the following design targets – high speed, low power consumption, less area or even a combination of them is preffered.

2.3.2 TYPES OF MULTIPLIERS

a. Shift-and-Add Multiplication

In this method, we add a number with itself and rotate the other number each time and shift it by one bit to left along with the carry. If carry is present add those two numbers.

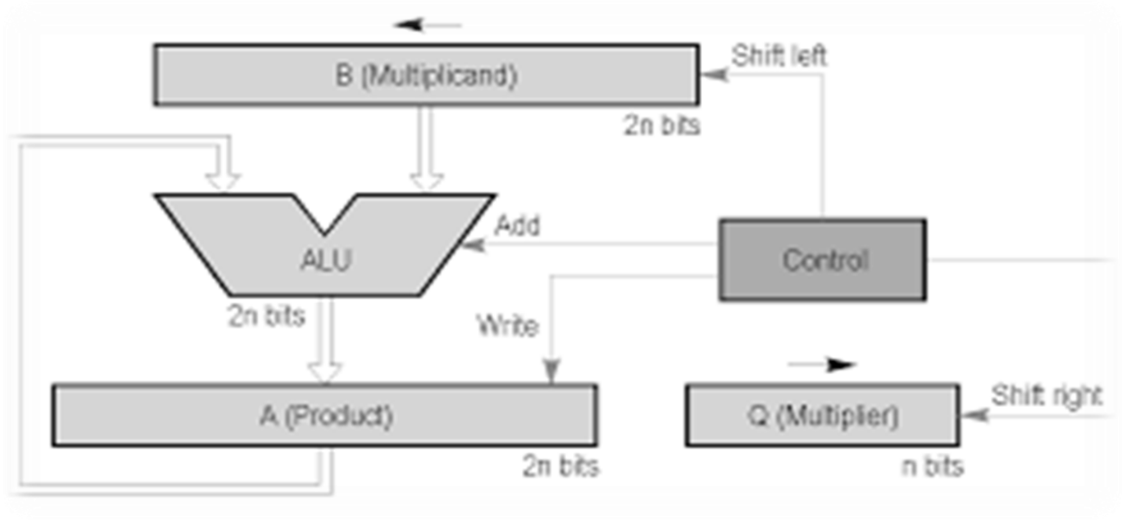


Fig 2.8 Shift- and add multiplier

This algorithm adds the multiplicand to itself by M times, where M denotes the multiplier. To multiply these two numbers this
algorithm takes the digits of the multiplier and places the intermediate product in the appropriate position to the left of earlier results.

b. Array Multiplier

An array multiplier may be a digital combinational circuit used for multiplying two binary numbers by employing an array of full adders and half adders. This array is employed for the nearly simultaneous addition of the varied product terms involved. To form the varied product terms, an array of AND gates is employed before the Adder array.

Checking the bits of the multiplier one at a time and forming partial products may be a serial operation that needs a sequence of add and shift micro-operations. The multiplication of two binary numbers are often through with one micro-operation by means of a combinational circuit that forms the merchandise bits all directly . This is a quick way of multiplying two numbers since all it takes is that the time for the signals to propagate through the gates that form the multiplication array. However, an array multiplier requires an outsized number of gates, and for this reason it had been not economical until the event of integrated circuits.

For implementation of array multiplier with a combinational circuit, consider the multiplication of two 2-bit numbers as shown in figure. The multiplicand bits are b1 and b0, the multiplier bits are a1 and a0, and therefore the product is:

Assume A = a1a0 and B= b1b0, the bits of the final product term P is written as:

1. P (0) = a0b0
2. P (1) =a1b0 + b1a0
3. P (2) = a1b1 + c1 where c1 is the carry generated during the addition for the P (1) term.
4. P (3) = c2 where c2 is the carry generated during the addition for the P (2) term.

For the above multiplication, an array of four AND gates are required to form the various product terms like a0b0 etc. and then an adder array is required to calculate the sums involving the varied product terms and carry combinations mentioned within the above equations so as to urge the final Product bits.

1. The first partial product is formed by multiplying a0 by b1, b0. The multiplication of two bits like a0 and b0 produces a 1 if both bits are 1; otherwise, it produces 0. This is just like an AND operation and may be implemented with an AND circuit .

2. The first partial product is formed by means of two AND gates.

3. The second partial product is formed by multiplying a1 by b1b0 and is shifted one position to the left.

4. The above two partial products are added with two half-adder (HA) circuits. Usually there are more bits within the partial products and it'll be necessary to use full-adders to supply the sum.

5. Note that the least significant bit of the product does not have to go through an adder since it is formed by the output of the first AND gate

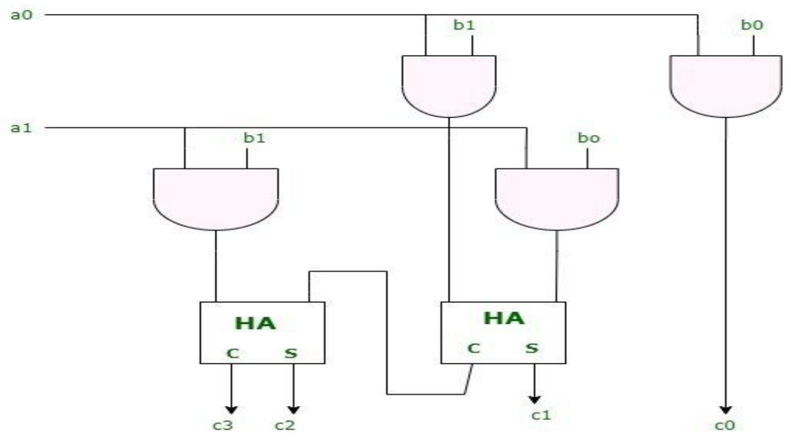


Fig 2.9 Array multiplier

c. Vedic Multiplier

The Vedic multiplication is predicated on 16 Vedic sutras which describes natural way of solving an entire range of mathematical problems. A simple digital multiplier (referred henceforth as Vedic multiplier) architecture supported the Urdhva Triyakbhyam (UT)(Vertically and Cross wise) Sutra is presented. This Sutra was traditionally utilized in ancient India for the multiplication of two decimal numbers in relatively less time.

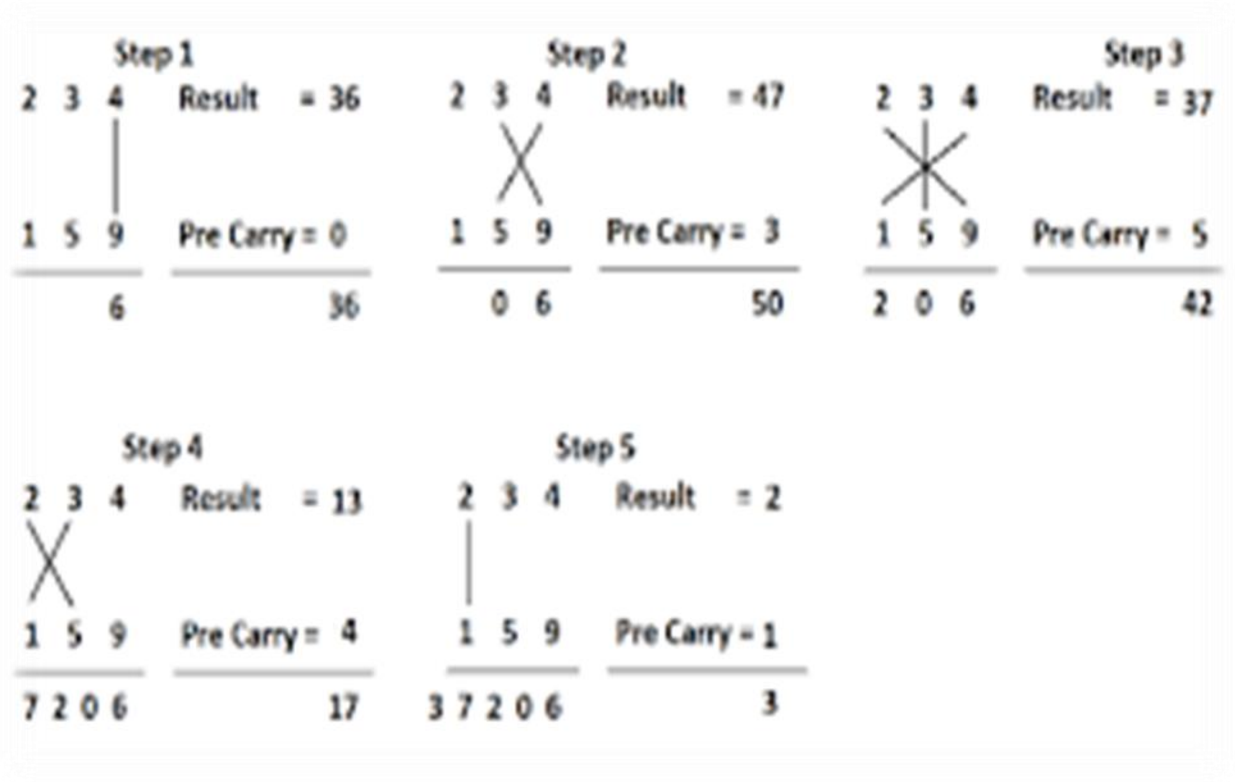


Fig 2.10 Vedic multiplication process

1. Vertical Multiplication of 1st digits of 2 numbers.
2. Crosswise Multiplication Addition of 1st 2 digits 2 numbers. (i.e. Crosswise Multiplication of 1st 2 digits and adding them.)
3. Crosswise Multiplication Addition of all 3 digits of both the numbers.
4. Crosswise Multiplication Addition of last 2 digits 2 numbers.
5. Vertical Multiplication of last digits 2 numbers.
6. For all steps, except 1st step, each compartment needs to have only 1 digit. If not then carry forward initial digits to previous compartment.

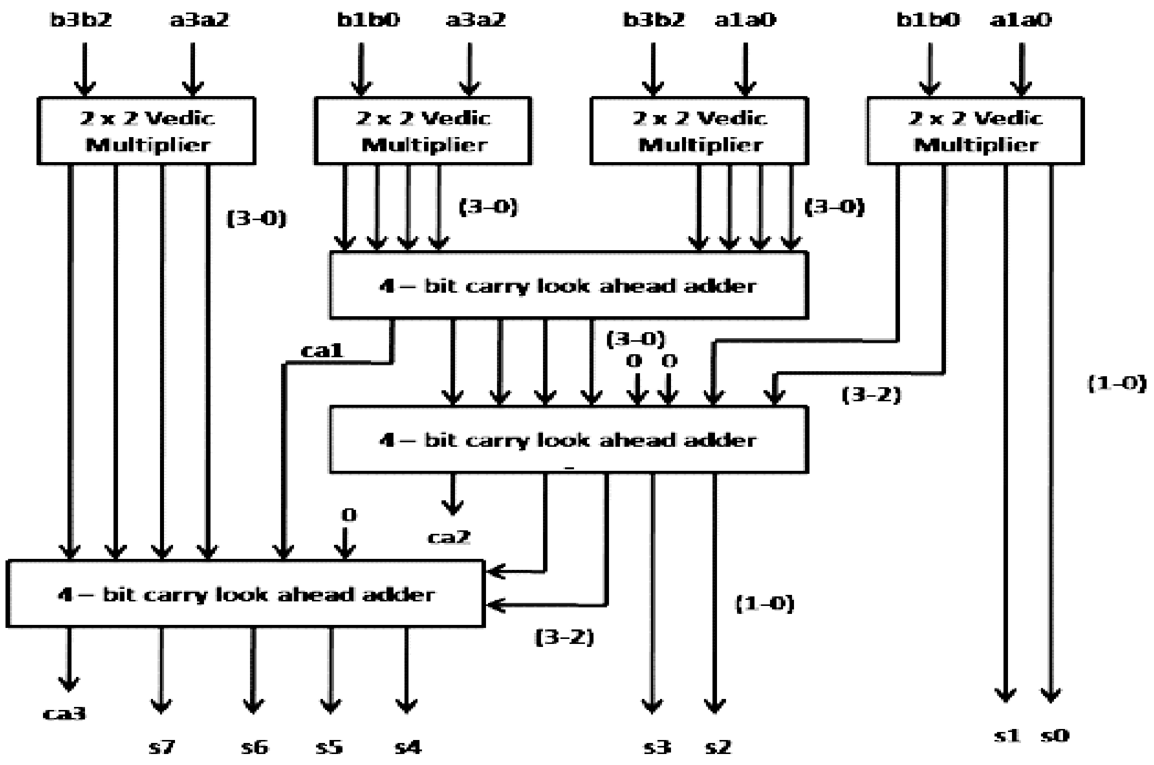


Fig 2.11 4*4 Vedic multiplier architecture (need to add some matter)

The multiplication operation in this Sutra is computed and the delay in output is reduced. The sutra is also known as ‘vertically and crosswise’. Initially this Sutra was used for only decimal numbers . However, it can also be used for binary numbers. The method how binary multiplication is completed using UT method for 2-bit, 3-bit and 4-bit numbers.

Since the partial products and their sums are calculated independently, the multiplier is independent of the clock of the processor. The delay is generated only due to the occurrence of carry from the partial products which is needed to be added in each next step’s partial product.

CHAPTER 3
CANONICAL SIGNED DIGIT

CSD representations have proven to be useful in implementating multipliers with reduced complexity,because cost of multiplication is a direct function of non zero bits in the multiplier and as CSD algorithm helps in reducing the number of these non zero bits it is very useful in implementing efficient multiplier.

The CSD format aims to scale back the amount of additives during multiplication. The CSD format features a ternary set as against a binary set in number representation. The symbols utilized in this format are {-1, 0, 1}. The goal is to group consecutive 1s and alter them to a ternary representation from binary representation. This is done ranging from the rightmost 1 and proceeding left until the last 1. So this never has adjacent non-zero bits. It is proved that the amount of operations never exceeds n/2 and on a mean it are often reduced to n/3.

3.1 Canonical Signed Digit Algorithm

Conversion from a binary representation to CSD representation requires the following steps:

- Observe where there are two consecutive 1's and convert the right most 1 into -1
- Add +1 to the next bits

The CSD presentation of a number is unique. The number of nonzero digits is minimal. There cannot be two consecutive non-zero digits for example, the binary value 1 1 1 1 1 (decimal number 31) could be written as the signed-digit value 1 0 0 0 0 -1 = $2^5 - 2^0$ =d31

CSD representation of integers 1, 2, … 10 is as follows:

B2	B1	B0	C3	C2	C1	C0
0	0	0	0	0	0	0
0	0	1	0	0	0	1
0	1	0	0	0	1	0
0	1	1	0	1	0	-1
1	0	0	0	1	0	0
1	0	1	0	1	0	1
1	1	0	1	0	-1	0
1	1	1	1	0	0	-1

Table 3.1 Binary to CSD conversion

3.2 Comparision of Binary multiplication Vs. CSD multiplication

Let us consider an example-14 multiplied with 15 .

Multiplicand => 1110

Multiplier=> 1111

Binary Multiplication

1110*1111

```
    1110
   1110x
  1110xx
 1110xxx
--------
11010010
```

CSD Multiplication

CSD code for multiplier-1000-1

1110*1000-1

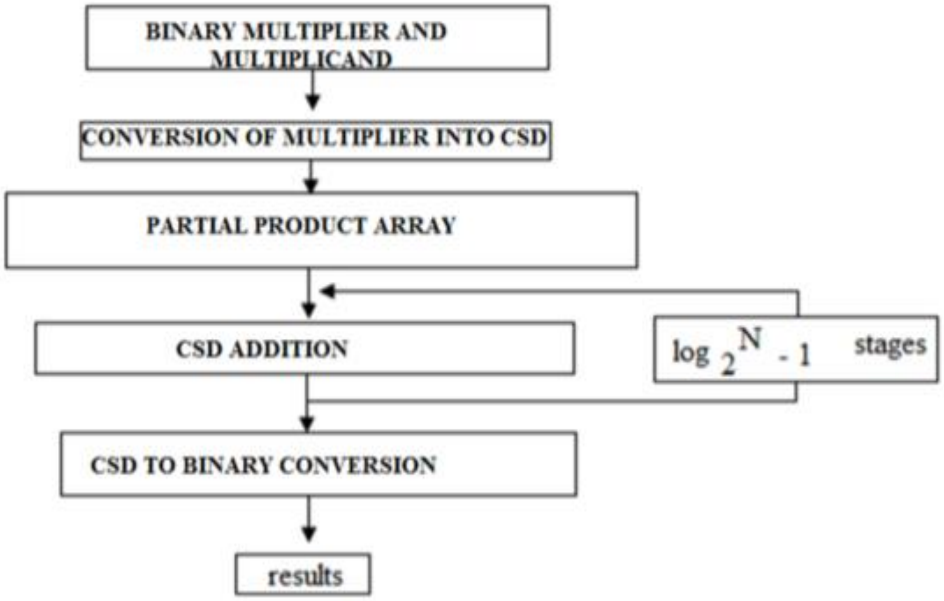
```
     -1-1-10
1110 x x x x
------------
1110-1-1-10
```

2^7+2^6+2^4+2^1=d210

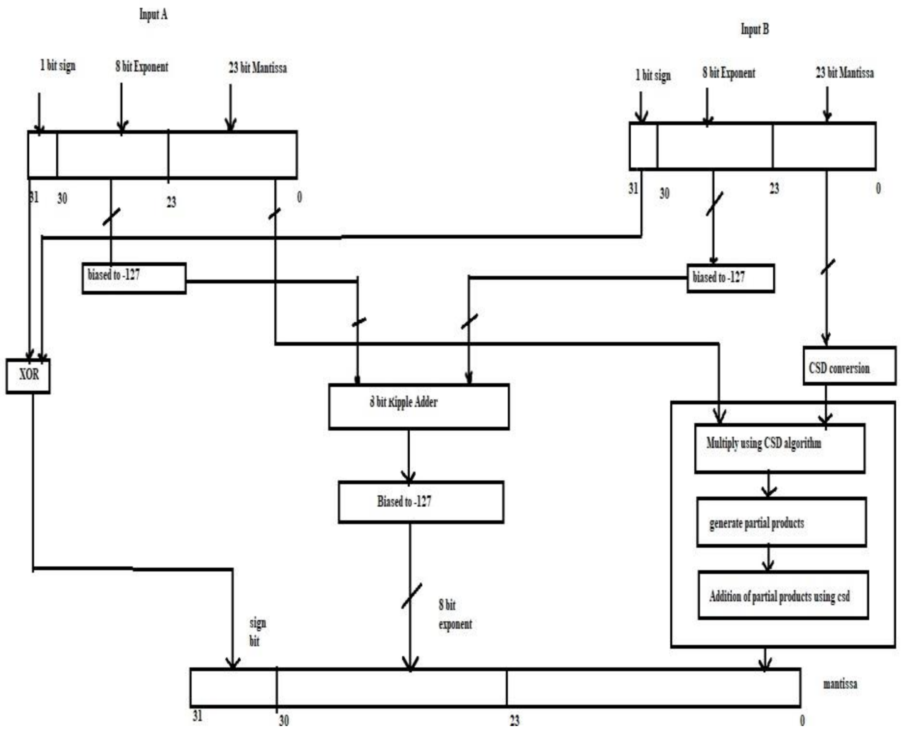
(2^7+2^6+2^5)-(2^3+2^2+2^1)=d210

Here when we observe binary multiplication the number of stages involved for multiplication are more when compared to that of the stages of CSD multiplications.It is clearly observed that the number of stages are decreased as the number of 1's of the multiplier are reduced and are converted into 0's using the CSD algorithm.Later after performing the multiplication in binary and CSD respectively the result is converted into decimal format. And when observed the results both are of same value in decimal format. That's the reason we are using CSD algorithm for efficient multiplication- as the number of stages or partial products terms are reduced the hardware requirement is also reduced correspondingly which leads to reduction in area utilization.

3.3 CSD FLOW



3.3 FPM using CSD



CHAPTER 4
OVERVIEW OF FPGA AND EDA SOFTWARE

4.1 INTRODUCTION

Developing a large FPGA –based system is an involved process that consists of many complex transformations and optimization algorithms. Software tools are needed to automate some of the tasks. We use Nexys4 Board which is a complete, ready-to- use digital circuit development platform based on the latest Artix-7 Field Programmable Gate Array (FPGA) (XC7A100T-ICSG324C) from Xilinx for synthesis and implementation, and use the Vivado Design Suite for simulation.

A Field Programmable Gate Array (FPGA) is a logic device that contains a two-dimensional add shift of generic logic cells and programmable switches. A logic cell can be programmed to perform a simple function, and a programmable switch can be customized to provide interconnection among the logic cells.

A custom design can be implemented by specifying the function of each logic cell and selectively setting the connection of each programmable switch. Once the design and synthesis arecompleted, we can use a simple adapter cable to download the desired logic cell and switch configuration to the FPGA device and obtain the custom circuit. Since this process can be done “in the field” rather than “in fabrication facility (fab),” the device is known as field programmable.

4.2 OVERVIEW OF DIGILENT NEXYS4 BOARD

The Nexys 4 DDR board is a complete, ready-to-use digital circuit development platform based on the latest Artix-7™ Field Programmable Gate Array (FPGA) from Xilinx®. With its large, high-capacity FPGA (Xilinx part number XC7A100T-1CSG324C), generous external memories, and collection of USB, Ethernet, and other ports, the Nexys4 DDR(Double Data Rate) can host designs ranging from introductory combinational circuits to powerful embedded processors. Several built-in peripherals, including an accelerometer, temperature sensor, MEMs digital microphone, a speaker amplifier, and several I/O devices allow the Nexys4 DDR to be used for a wide range of designs without needing any other components.

4.2.1 Features Of Nexys 4 DDR

- 15,850 logic slices, each with four 6-input LUTs(Look Up Tables) and 8 flip-flops
- 4,860 Kbits of fast block RAM
- Six clock management tiles, each with phase-locked loop (PLL)
- 240 DSP slices
- Internal clock speeds exceeding 450 MHz
- On-chip analog-to-digital converter (XADC)
- 16 user switches
- USB-UART Bridge
- 12-bit VGA output
- 3-axis accelerometer
- 128MiB DDR2
- Pmod for XADC signals
- 16 user LEDs
- Two tri-colour LEDs
- PWM audio output
- Temperature sensor
- Serial Flash
- Digilent USB-JTAG port for FPGA programming and communication
- Two 4-digit 7-segment displays
- Micro SD card connector
- PDM microphone
- 10/100 Ethernet PHY
- Four Pmod ports

- USB HID Host for mice, keyboards and memory sticks

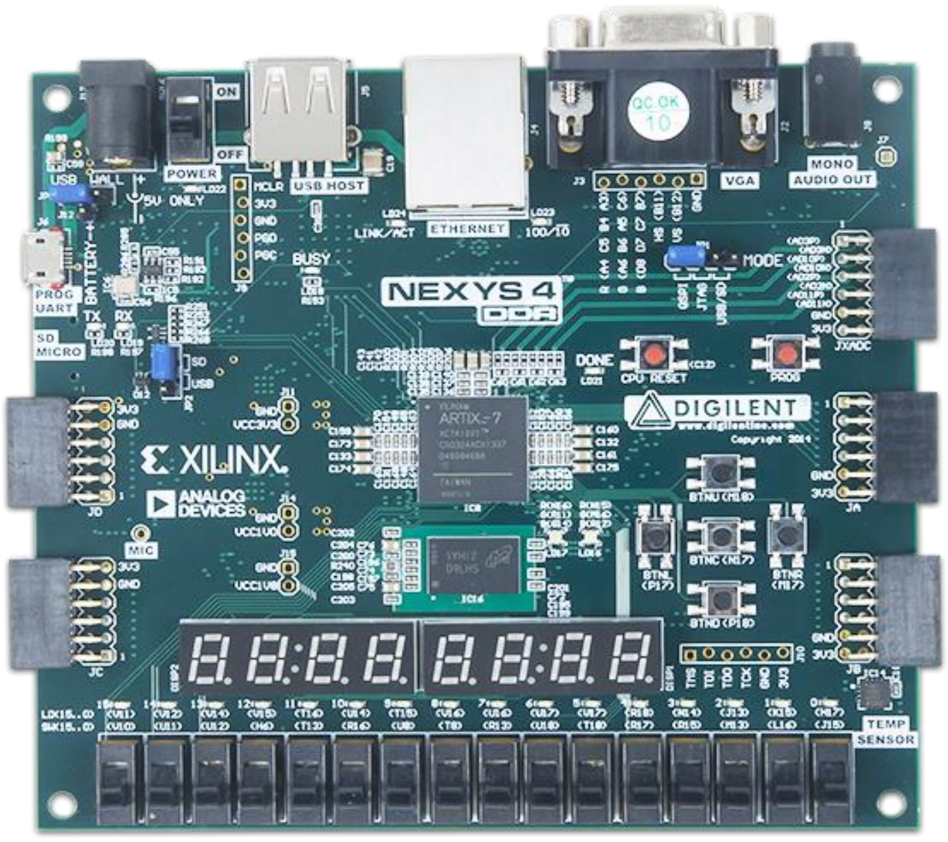
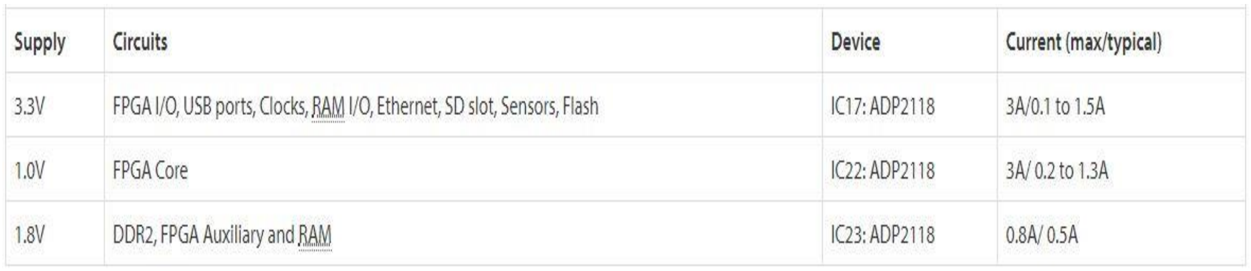


Fig 4.1 Nexys 4 DDR FPGA kit

4.2.2 Power Supplies

The Nexys4 DDR board can receive power from the Digilent USB-JTAG port (J6) or from an external power supply. Jumper JP3 (near the power jack) determines which source is used.

All Nexys4 DDR power supplies can be turned on and off by a single logic-level power switch (SW16). A power-good LED (LD22), driven by the “power good” output of the ADP2118 supply, indicates that the supplies are turned on and operating normally.



Supply	Circuits	Device	Current (max/typical)
3.3V	FPGA I/O, USB ports, Clocks, RAM I/O, Ethernet, SD slot, Sensors, Flash	IC17: ADP2118	3A/0.1 to 1.5A
1.0V	FPGA Core	IC22: ADP2118	3A/ 0.2 to 1.3A
1.8V	DDR2, FPGA Auxiliary and RAM	IC23: ADP2118	0.8A/ 0.5A

Table 4.1 Nexys 4 DDR power supplies

4.2.3 Memory

The Nexys4 DDR board contains two external memories: a 1Gib (128MiB) DDR2 SDRAM and a 128Mib (16MiB) non-volatile serial Flash device. The DDR2 modules are integrated on-board and connect to the FPGA using the industry standard interface.

4.2.4 DDR2

The Nexys4 DDR includes one Micron MT47H64M16HR-25:H DDR2 memory component, creating a single rank, 16-bit wide interface. It is routed to a 1.8V-powered HR (High Range) FPGA bank with 50 ohm controlled single-ended trace impedance. 50 ohm internal terminations in the FPGA are used to match the trace characteristics. Similarly, on the memory side, on-die terminations (ODT) are used for impedance matching.

For proper operation of the memory, a memory controller and physical layer (PHY) interface needs to be included in the FPGA design. There are two recommended ways to do that, which are outlined below and differ in complexity and design flexibility.

4.2.5 Quad-SPI Flash

FPGA configuration files can be written to the Quad-SPI Flash (Spansion part number S25FL128S), and mode settings are available to cause the FPGA to automatically read a configuration from this device at power on. An Artix-7 100T configuration file requires just less than four MiB (mebibyte) of memory, leaving about 77% of the flash device available for user data. Or, if the FPGA is getting configured from another source, the whole memory can be used for custom data.

4.2.6 Oscillators/Clocks

The Nexys4 DDR board includes a single 100 MHz crystal oscillator connected to pin E3 (E3 is a MRCC input on bank 35). The input clock can drive MMCMs or PLLs to generate clocks of various frequencies and with known phase relationships that may be needed throughout a design. Some rules restrict which MMCMs and PLLs may be driven by the 100 MHz input clock. For a full description of these rules and of the capabilities of the Artix-7 clocking resources, refer to the “7 Series FPGAs Clocking Resources User Guide” available from Xilinx.

Xilinx offers the Clocking Wizard IP core to help users generate the different clocks required for a specific design. This wizard will properly instantiate the needed MMCMs and PLLs based on the desired frequencies and phase relationships specified by the user. The wizard will then output an easy-to-use wrapper component around these clocking resources that can be inserted into the user’s design. The clocking wizard can be accessed from within the Project Navigator or Core Generator tools.

4.2.7 Pmod Ports

The Pmod ports are arranged in a 2×6 right-angle, and are 100-mil female connectors that mate with standard 2×6 pin headers. Each 12-pin Pmod port provides two 3.3V VCC signals (pins 6 and 12), two Ground signals (pins 5 and 11), and eight logic signals, as shown in Figure 20. The VCC and Ground pins can deliver up to 1A of current.

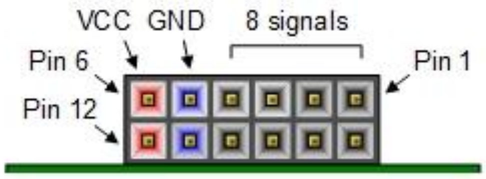


Fig 4.3 PMOD ports

4.3 DEVELOPMENT FLOW

The simplified development flow of an FPGA-based system is shown below. The left portion of the flow is the refinement and programming process, in which a system is transformed from an abstract textual HDL description to a device cell- level configuration and then downloaded to the FPGA device. The right portion is the validation process, which checks whether the system meets the functional specification and performance goals. The major steps in the flow are:

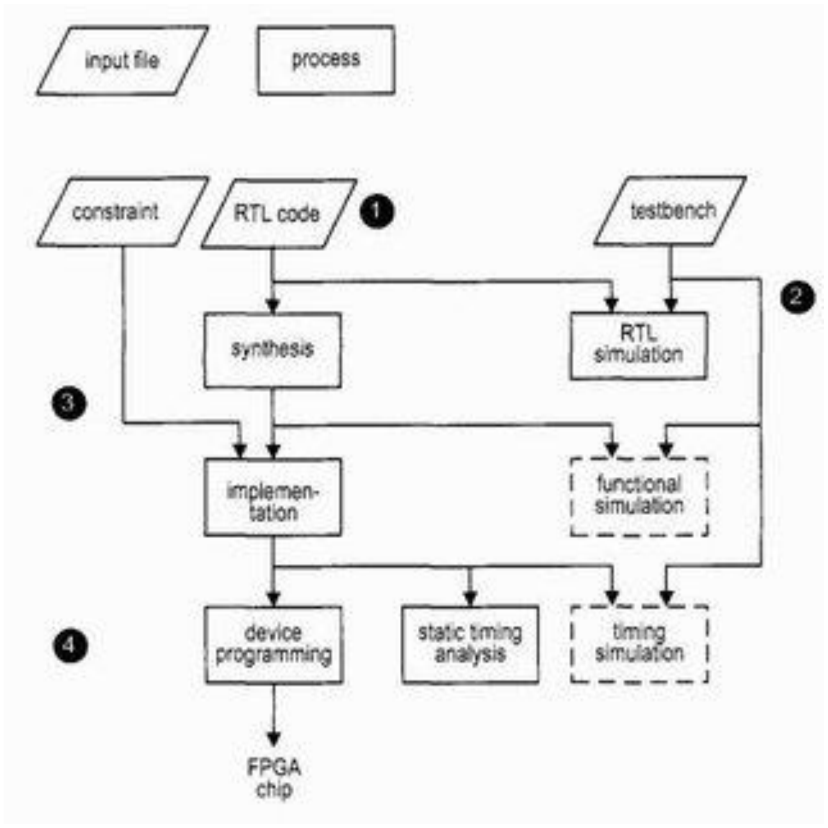


Fig 4.4 Development flow

a)Design Entry: Design the system and derive the HDL file(s). We may need to add a separate constraint file to specify certain implementation constraints.

b)RTL Simulation: Develop the test bench in HDL and perform RTL(Register Transfer Level) simulation. The RTL term reflects the fact that the HDL code is done at the register transfer level.

c)Synthesis: The synthesis process is generally known as logic synthesis, in which the software transforms the HDL constructs to generic level components, such as simple logic gates and FFs.

d)Implementation: The implementation process consists of three smaller processes: translate map, and place and route. The translate process merges multiple design files to a single netlist. The map process, which is generally known as technology mapping, maps the generic gates in the netlist to FPGA logic cells and IOBs. The place and route process, which is generally known as placement and routing, derives the physical layout inside the FPGA chip. It places the cells in physical locations and determines routes to connect various signals. In the Xilinx flow, static timing analysis, whichdetermines various timing parameters, such as maximal propagation delay and maximal clock frequency, is performed at the end of the implementation process.

e)Generate and download the programming file: In this process, a configuration file is generated according to the final netlist. This file is downloaded to an FPGA device serially to configure the logic cells and switches. The physical circuit can be verified accordingly.

The optional functional simulation can be performed after synthesis, and the optional timing simulation can be performed after implementation. Functional simulation uses a synthesized netlist to replace the RTL description and checks the correctness of the synthesis process. Timing simulation uses the final netlist, along with detailed timing data, to perform simulation. Because of the complexity of the netlist, functional and timing simulation may require a significant amount of time. If we follow good design and coding practices, the HDL code will be synthesized and implemented correctly. We only need to use RTL simulation to check the correctness of the HDL code and use static timing analysis to examine the relevant timing information. Both functional and timing simulations can be omitted from the development flow.

4.4 Overview of VERILOG

Verilog, standardized as IEEE 1364, is a hardware description language (HDL) used to model electronics systems. It is most commonly used in the design and verification of digital circuits at the regular – transfer level of abstraction. It is also used in the verification of analog circuits and mixed signal circuits HDL’s allows the design to be simulated earlier in the design cycle in order to correct errors or experiment with different architectures. Designs described in HDL are technology independent, easy to design and debug, and are usually more readable than schematics, particularly for large circuits. Verilog can be used to describe designs at four levels of abstraction:

Algorithmic level (much like a code if, case and loop statements).

- Register transfer level (RTL uses registers connected by Boolean equations
- Gate level (interconnected AND, NOR etc.).
- Switch level (the switches are MOS transistors inside gates).

A Verilog design consists of a hierarchy of modules. Modules encapsulate design hierarchy, and communicate with other modules through a set of declared input, output and bidirectional ports. Internally, a module can contain any combination of the following: net/variable declarations (wire, reg, integer etc), concurrent and sequential statement blocks and instances of other modules (sub – hierarchies).

4.4.1 Features of Verilog HDL

Verilog HDL offers many useful features for hardware design

- Verilog (verify logic) HDL is a general-purpose hardware description language that is easy to learn and use. It is similar in syntax to the C programming language. Designers with C programming experience will find it easy to learn Verilog HDL.
- Verilog HDL allows different levels of abstraction to be mixed in the same model. Thus, a designer can define a hardware model in terms of switches, gates, RTL or behavioral code. Also, a designer needs to learn only one language for stimulus and hierarchical design.
- Most popular logic synthesis tools support Verilog HDL. This makes it the language of choice for designers.
- All fabrication vendors provide Verilog HDL libraries for post logic synthesis simulation. Thus, designing a chip in Verilog HDL allows the widest choice of vendors.

- The programming language interface (PLI) is a powerful feature that allows the user to custom C code to interact with the internal data structures of Verilog. Designers can customize a Verilog HDL simulator to their need with the Programming language interface (PLI).

4.4.2 Module Declaration

A module is the principal design entity in Verilog. The first line of a module declaration specifies the name and port list (arguments). The next few lines specifies the I/O type (input, output or inout) and width of each port. The default port width is 1 bit. Then the port variables must be declared wire, reg. The default is wire. Typically, inputs are wire since their data is latched outside the module. Outputs are type reg if their signals were stored inside an always or initial block.

Syntax

```
Module model_name(port_list)
Input[msb:lsb] input_port_list;
Output[msb:lsb] output_port_list;
inout[msb:lsb] inout_port_list;
…………Statements…………….
endmodule
```

Example

```
module add_sub (add, in1, in2, out);
input add;
input [7:0]in, in2;
wire in1, in2;
output [7:0]out;
reg out;
…………..Statements…………
```

endmodule

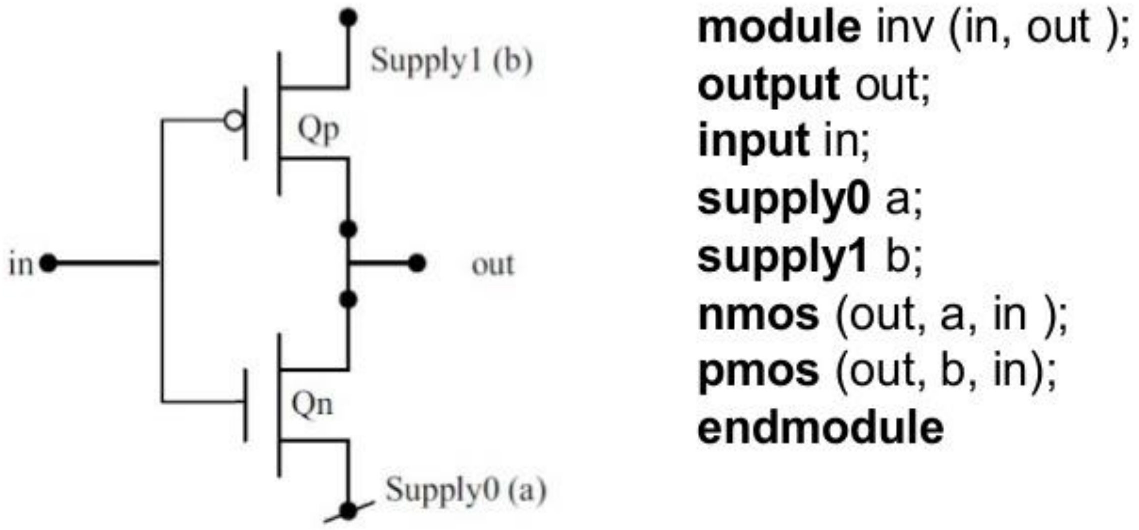
Verilog has four levels of modeling:

1) The Switch level modeling.
2) Gate level modeling.
3) The Data flow level.
4) The Behavioral level.

1) Switch level modeling

Switch level modeling forms the basic level of modeling digital circuits. The switches are available as a primitives in Verilog; they are central to design description at this level. Basic gates can be defined in terms of such switches. Switch-level modeling is a recently developed design and analysis methodology for MOS VLSI circuits. At the switch level, important features of MOS circuits can be directly modeled using a moderate number of discrete parameters, including switch states, resistance, capacitance, and bidirectional signals. Switch-level models, provide more accurate behavioral and structural information than gate-level logical models, while avoiding the high computational cost associated with analog electrical models. It provides a level of abstraction between the logic and analog-transistor levels of abstraction, describing the interconnection of transmission gates which are abstractions of individual mos and cmos transistors. The design of MOS technology electronic circuits requires functionallevel simulations, as well as switch and circuit-level simulations. Functional simulations are necessary to understand the behaviour independently of the implementation details.

CMOS inverter:



```
module inv (in, out );
output out;
input in;
supply0 a;
supply1 b;
nmos (out, a, in );
pmos (out, b, in);
endmodule
```

Fig 4.5 CMOS inverter

2) Gate level modeling

Modeling done at this level is usually called gate level modeling as it involves gates and has a one to one relation between a hardware schematic and the Verilog code. Verilog supports a few basic logic gates known as primitives as they can be instantiated like modules since they are already predefined. In general, gate-level modeling is used for implementing lowest level modules in a design like, full-adder, multiplexers, etc. Verilog HDL has gate primitives for all basic gates. Gate primitives are predefined in Verilog, which are ready to use.Multiple input gate primitives include and, nand, or, nor, xorand xnor. Designer should know the gate level diagram of the design.

Example

```
module or_gate (out, a, b, c, d);
input a, b, c, d;
wire x, y;
output out;
or or1(x, a, b);
or or2(y, c, d);
or orfinal(out, x, y);
endmodule
```

3) Data flow level modelling

Dataflow modeling provides the means of describing combinational circuits by their function rather than by their gate structure. Dataflow modeling uses a number of operators that act on operands to produce the desired results. Verilog HDL provides about 30 operator types.

Dataflow modeling uses continuous assignments and the keyword assign. A continuous assignment is a statement that assigns a value to a net. The datatype net is used in Verilog HDL to represent a physical connection between circuit elements. The value assigned to the net is specified by an expression that uses operands and operators. As an example, assuming that the variables were declared,a2-to-1 multiplexer with data inputs A and B, select input S, and output Y is described with continuous assignment.

```
 assign Y= (A & S) | (B & S)
```

Example

The dataflow description of a 2-to-4-line decoder is shown in HDL below. The circuit is defined with four continuous assignment statements using Boolean expressions, one for each output. // Dataflow description of 2-to-4 line decoder with enable input (E) module decoder_df (A,B,E,D);

```
 input A,B,E;
 output [3,:0] D;
 assign D[3] =~(~A & ~B & ~E);
 assign D[2] =~(~A & B & ~E);
 assign D[1] =~( A & ~B & ~E);
 assign D[0] =~( A & B & ~E);
endmodule
```

4) Behavioral level modelling

This is the highest level of abstraction provided by Verilog HDL. A module can be implemented in terms of the desired design algorithm without concern for the hardware implementation details. It specifies the circuit in terms of its expected behavior.It specifies the circuit in terms of its expected behaviour.It is the closest to a natural language description of the circuit functionality, but also the most difficult to synthesize.

All that a designer need is the algorithm of the design, which is the basic information for any design. This level simulates the behavioural level of the circuits and development rate in this level is highest. Although the development rate in this abstraction level is high there are some drawbacks such as the delay modeling is not possible. In practice any circuit is first implemented in this level to understand the theoretical possibility of the circuit and then it is implemented in at a lower level to analyse the practical aspects. This level of Verilog has blocking and non-blocking assignment. Blocking assignment is sequential nature and using the combination of both assignments, complex sequential can be modeled with ease.

The abstraction in this modeling is as simple as writing the logic in C language. This is a very powerful abstraction technique. All that a designer needs are the algorithm of the design, which is the basic information for any design. This level is very important because with the increasing complexity of digital design, it has become vitally important to make wise design decisions early in a project. Designers need to be able to evaluate the trade-offs of various architectures and algorithms before they decide on the optimum architecture and algorithm to implement in hardware. Only after the high-level architecture and algorithm are finalized, designers start focusing on building the digital circuit to implement the algorithm. Most of the behavioural modeling is done using two important constructs: initial and always. All the other behavioural statements appear only inside these two structured procedure constructs.

Initial construct

The statements which come under the initial construct constitute the initial block. The initial block is executed only once in the simulation, at time 0. If there are more than one initial blocks, than all the initial blocks are executed concurrently. The initial construct is used as follows:

```
Initial
Begin
Reset = 1'b0;
clk = 1'b1;
end
```

In the first initial block there is more than one statements hence they are written between begin and end. If there is only one statement then there is no need to put begin and end.

Always construct

The statements which come under the always construct constitute thealways block. The always block starts at time 0, and keepsonexecutingall the simulation time. It works like an infinite loop. It is generally usedto model a functionality that is continuously repeated.

```
always
#5 clk = ~clk;
initial
clk = 1'b0;
```

The above code generates a clock signal clk, with a time period of 10units. The initial blocks initiates the clk value to 0 at time 0. Then afterevery 5 units of time it is toggled, hence we get a time period of 10units. This is the general way to generate a clock signal for use in testbenches.

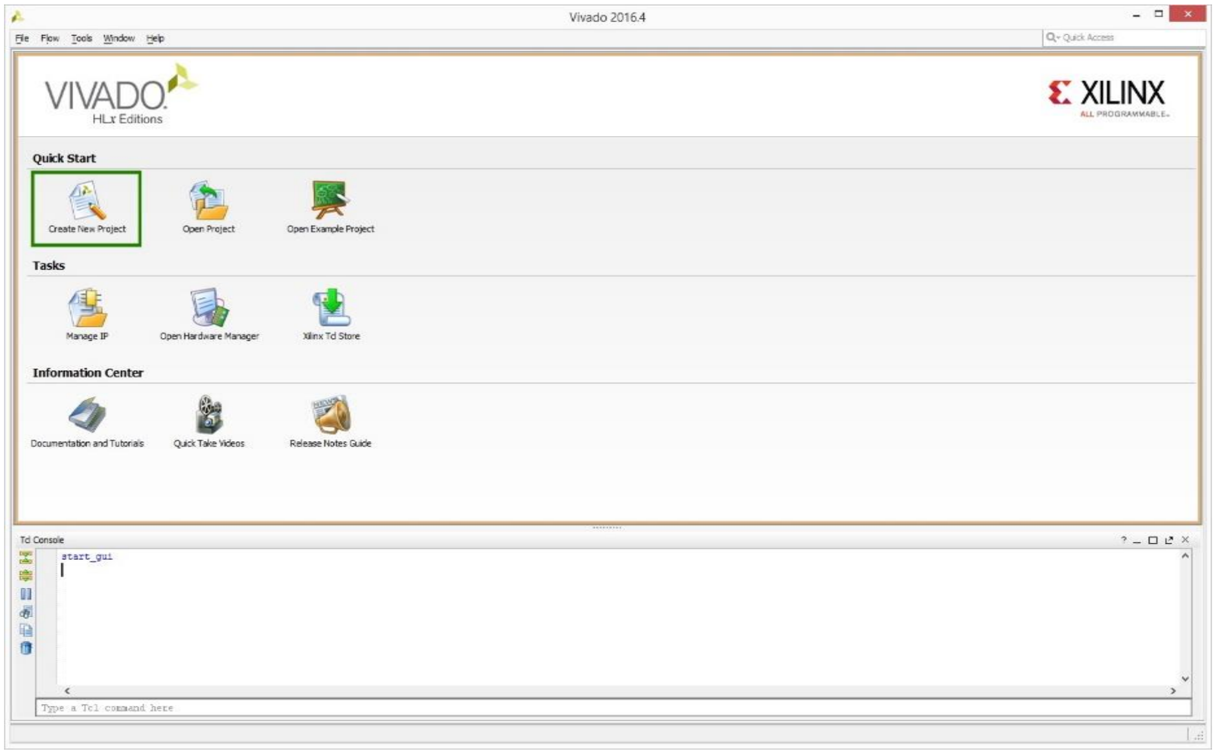
4.5 Overview of EDA (Electronic Design Automation) software

Vivado Design Suite is a software suite produced by Xilinx for synthesis and analysis of HDL designs, superseding Xilinx ISE with additional features for system on a chip development and high level synthesis.

Setting up a vivado project involves the following steps-

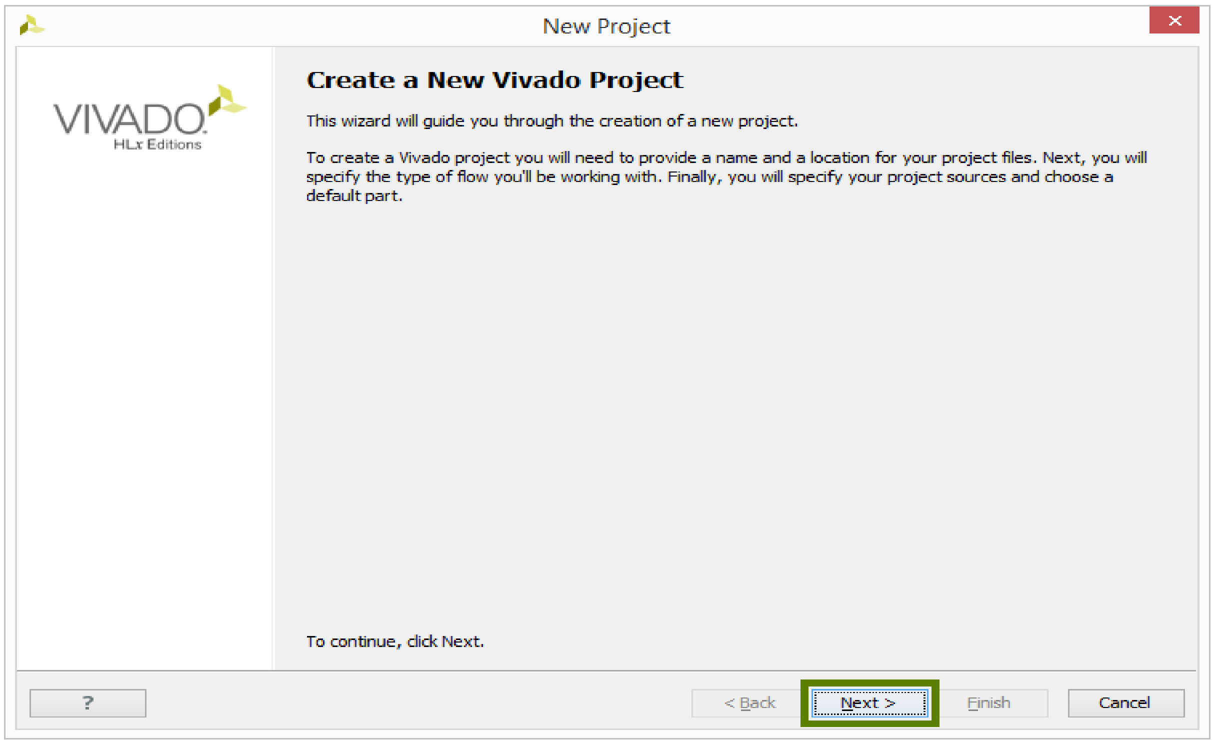
Creating a New Project

After launching Vivado, from thestartup page click the "Create New Project" icon. Alternatively, you can select **File-> New project**.



```
start_gui
```

The New Project wizard will launch, click the "Next>" button to proceed.


New Project

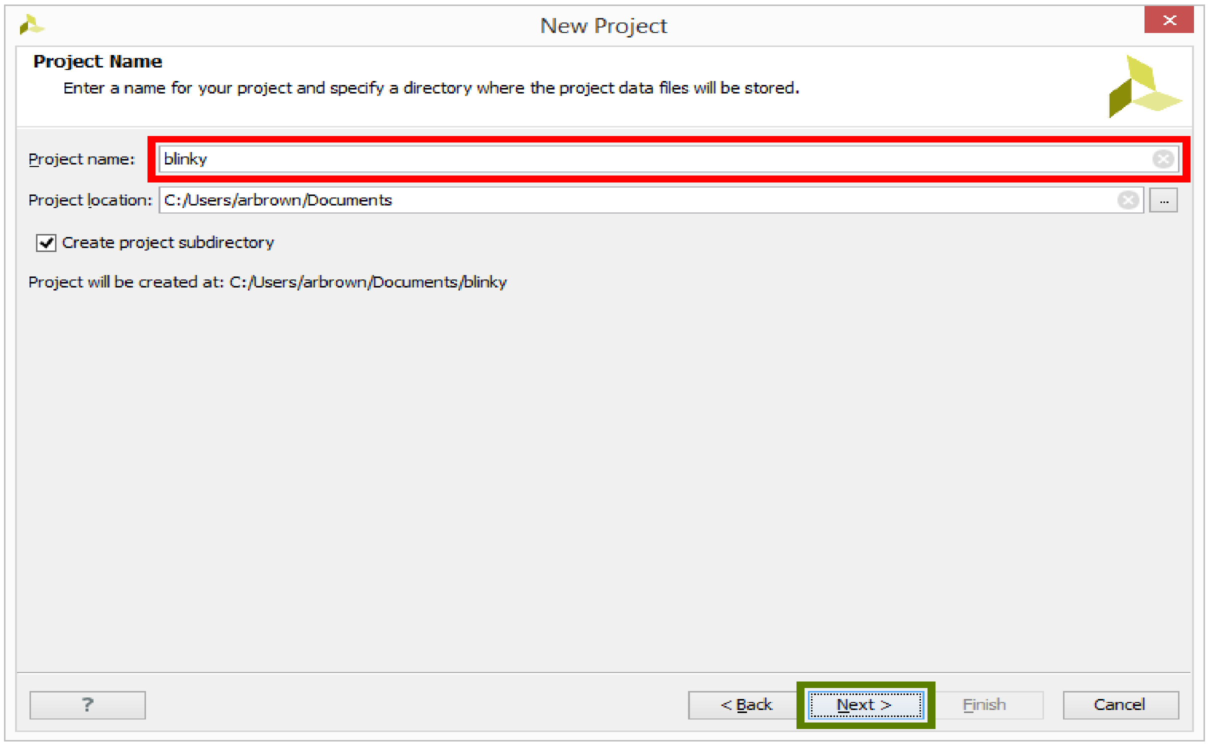
Create a New Vivado Project

This wizard will guide you through the creation of a new project.

To create a Vivado project you will need to provide a name and a location for your project files. Next, you will specify the type of flow you'll be working with. Finally, you will specify your project sources and choose a default part.

To continue, click Next.

Enter a project name and select a project location. Make certain there are NO SPACES in either! It's not a bad idea to only use letters, numbers, and underscores as well. If necessarysimply create a new directory for your Xilinx Vivado projects in your root drive (e.g. C:/Vivado). You will likely always want to select the "create project sub-directory" check-box as well. This keeps the things neatly organized with a directory for each project and helps avoid problems. Click the "Next>" button to proceed.


New Project

Project Name

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

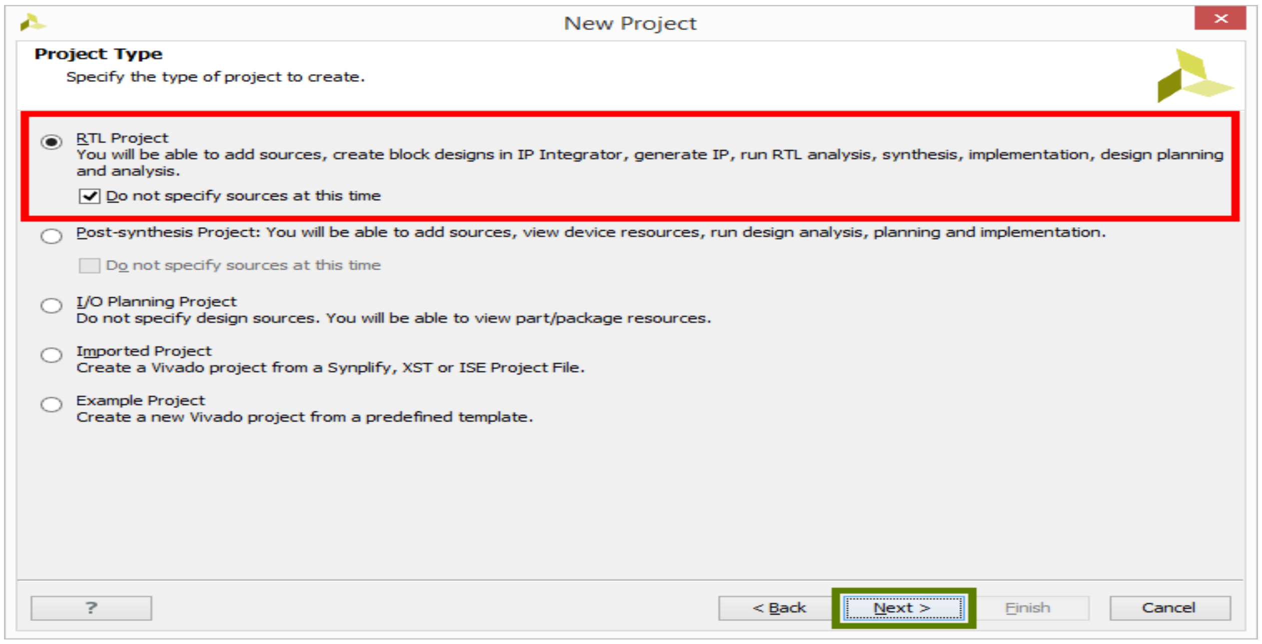
Project name: blinky

Project location: C:/Users/arbrown/Documents

☑ Create project subdirectory

Project will be created at: C:/Users/arbrown/Documents/blinky

Select the "RTL Project" radial and select "Do not specify sources at this time" check - box. If you don't select the check-box the wizard will take you through some additional steps to optionally add pre existing items such as VERILOG or Verilog source files, Vivado IP blocks, and .XDC constraint file for device pin amd timing configuration. For this first project you will add necessary items later. Click the "Next>" button to proceed.


New Project

Project Type

Specify the type of project to create.

(●) RTL Project
You will be able to add sources, create block designs in IP Integrator, generate IP, run RTL analysis, synthesis, implementation, design planning and analysis.
[✓] Do not specify sources at this time

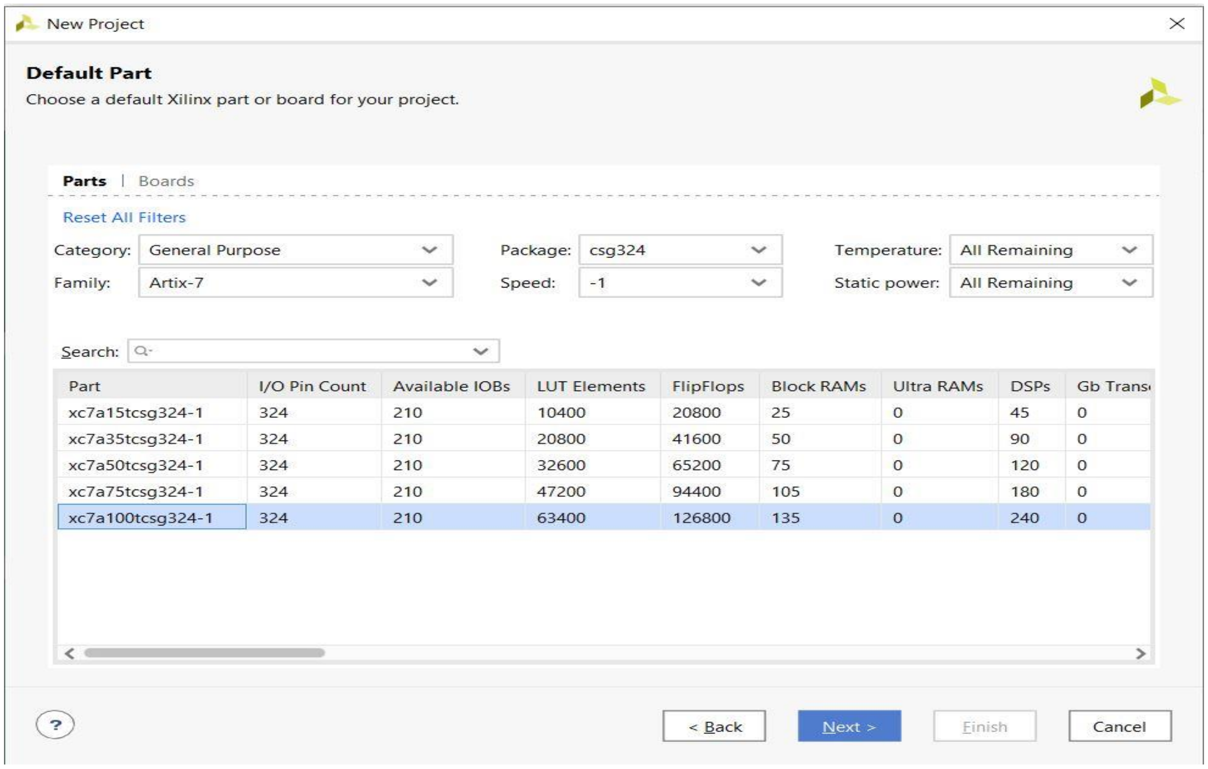
( ) Post-synthesis Project: You will be able to add sources, view device resources, run design analysis, planning and implementation.
[ ] Do not specify sources at this time

( ) I/O Planning Project
Do not specify design sources. You will be able to view part/package resources.

( ) Imported Project
Create a Vivado project from a Synplify, XST or ISE Project File.

( ) Example Project
Create a new Vivado project from a predefined template.

In case of Nexys 4 it's Artix-7 chip that's on the board, and we filter the specifications as package -csg324, speed grade=(-1) shown will help you get to the correct device that's highlighted. Once you select the correct device click the "Next>" button to proceed.

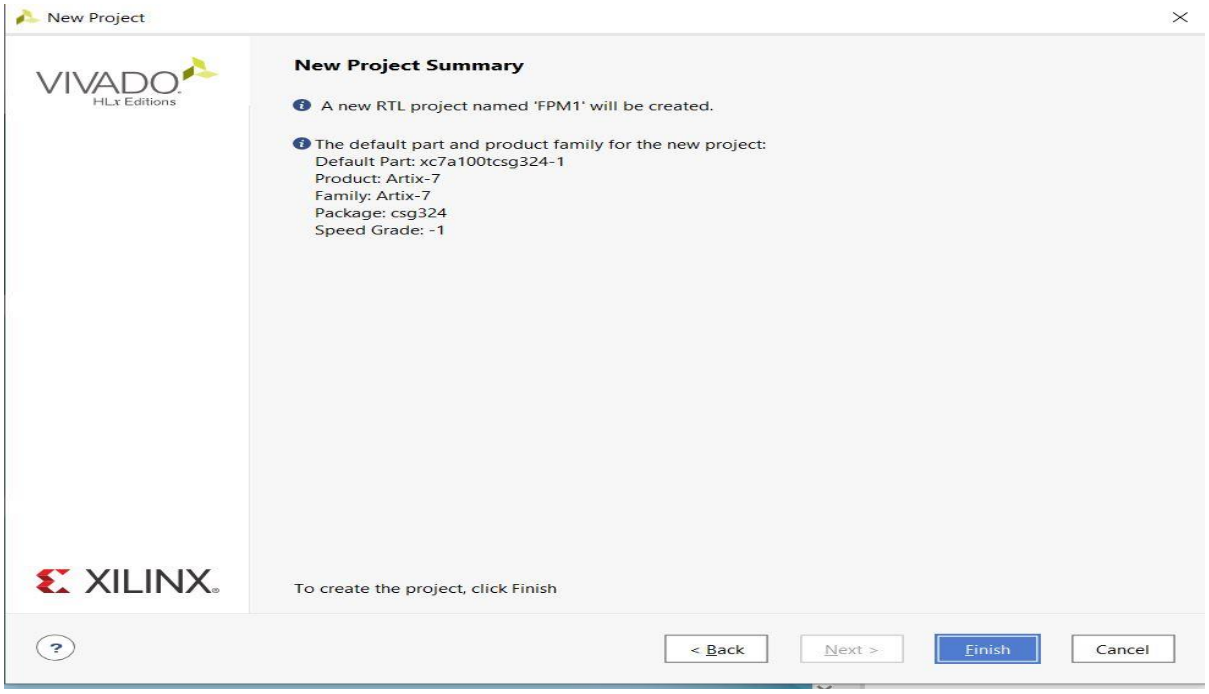


Default Part

Choose a default Xilinx part or board for your project.

Category: General Purpose
Family: Artix-7
Package: csg324
Speed: -1
Temperature: All Remaining
Static power: All Remaining

Part	I/O Pin Count	Available IOBs	LUT Elements	FlipFlops	Block RAMs	Ultra RAMs	DSPs	Gb Trans
xc7a15tcsg324-1	324	210	10400	20800	25	0	45	0
xc7a35tcsg324-1	324	210	20800	41600	50	0	90	0
xc7a50tcsg324-1	324	210	32600	65200	75	0	120	0
xc7a75tcsg324-1	324	210	47200	94400	105	0	180	0
xc7a100tcsg324-1	324	210	63400	126800	135	0	240	0



New Project Summary

A new RTL project named 'FPM1' will be created.

The default part and product family for the new project:
Default Part: xc7a100tcsg324-1
Product: Artix-7
Family: Artix-7
Package: csg324
Speed Grade: -1

To create the project, click Finish

Click the “Finish” button and Vivado will proceed to create your project as specified.

STEPS FOR DESIGN ENTRY/ IMPLEMENTATION

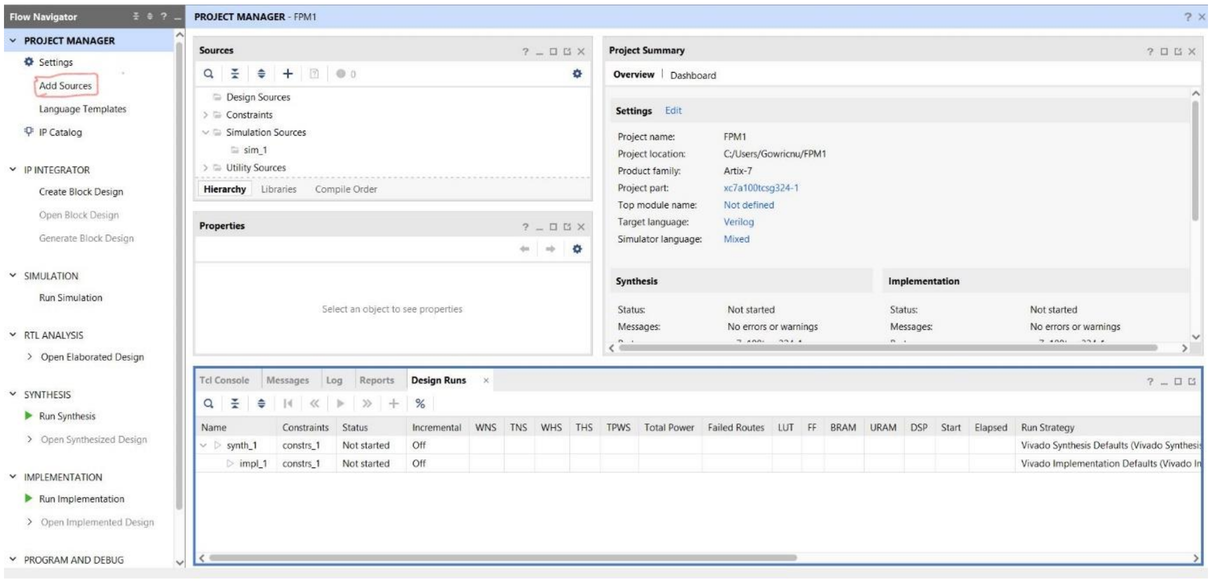
Working through the basic project flow

The Vivado project window contains a lot of information,and the information displayed can change depending on what part of the design currently have open as you work through the steps of your project. Keep this in mind as you work through this guide, because if you don't see a specific sub window or sub window tab it's possible you are'nt in the correct part of the design.

The "Flow Navigator" on the left side of the screen has all the major project phases organised from top to bottom in their natural chronological order. You begin in the "project manager"portion of the flow and the header at the top of the screen next to the flow navigator reflects this.This header and the corresponded highlighted section in the flow navigator will tell you which phase of the design you have opened.

Project Manager

Now click on "Add sources" under the project manager phase of the flow navigator.



Settings Edit

Project name:	FPM1
Project location:	C:/Users/Gowricnu/FPM1
Product family:	Artix-7
Project part:	xc7a100tcsg324-1
Top module name:	Not defined
Target language:	Verilog
Simulator language:	Mixed

Synthesis

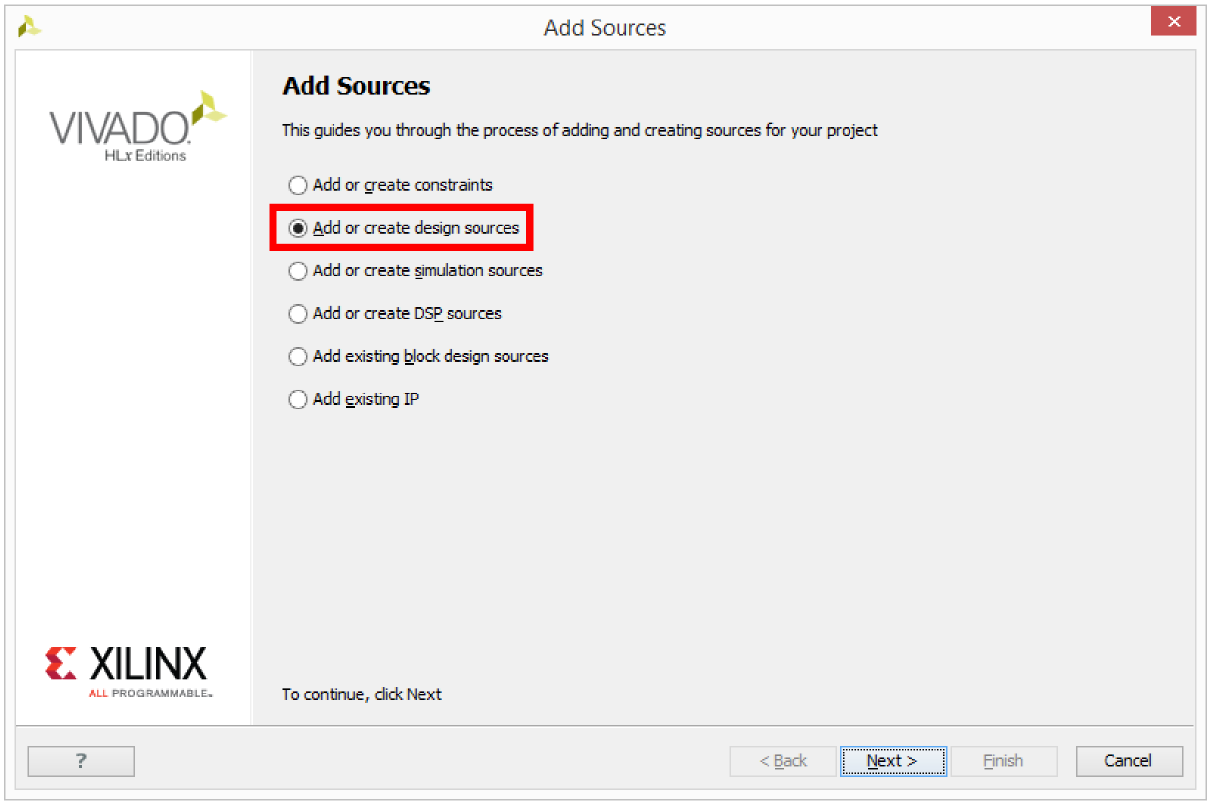
Status:	Not started
Messages:	No errors or warnings

Implementation

Status:	Not started
Messages:	No errors or warnings

Name	Constraints	Status	Incremental	WNS	TNS	WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF	BRAM	URAM	DSP	Start	Elapsed	Run Strategy
synth_1	constrs_1	Not started	Off															Vivado Synthesis Defaults (Vivado Synthesis
impl_1	constrs_1	Not started	Off															Vivado Implementation Defaults (Vivado In

Select the "Add or create design sources" radial and then click the "Next" button.


Add Sources

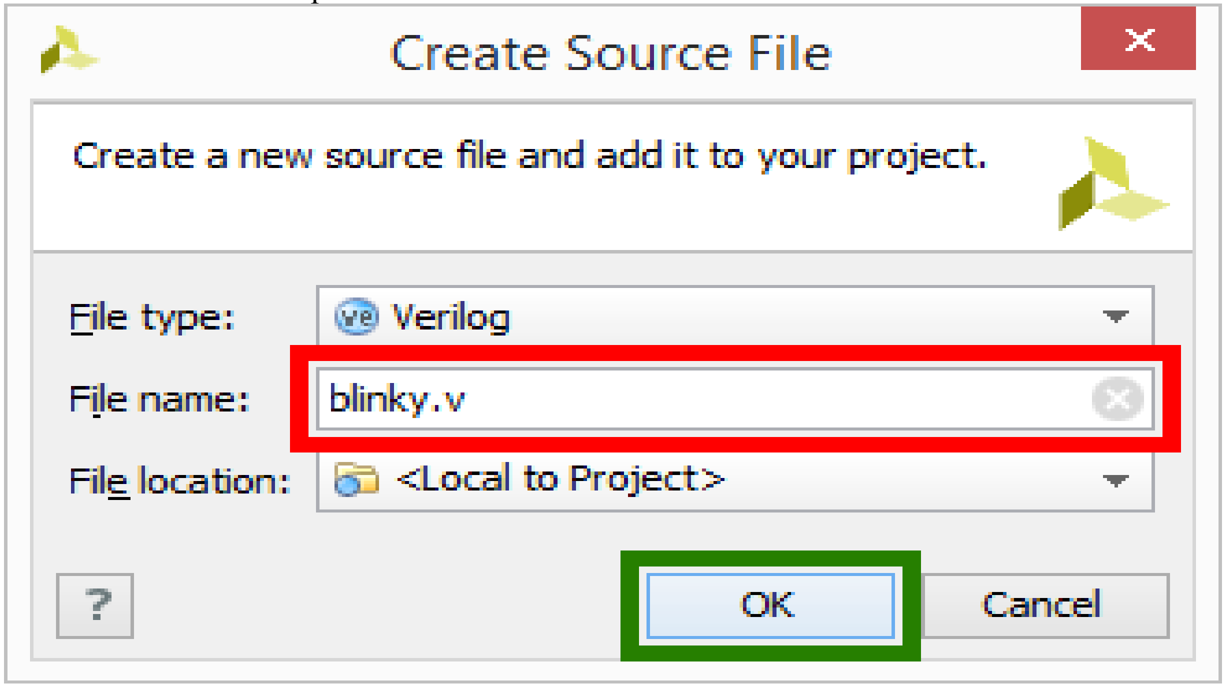
Add Sources

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 existing block design sources
- ( ) Add existing IP

To continue, click Next

Click the “Create file”button or click the Green “+”symbol in the upper left corner and select the “create file” option.


Create Source File

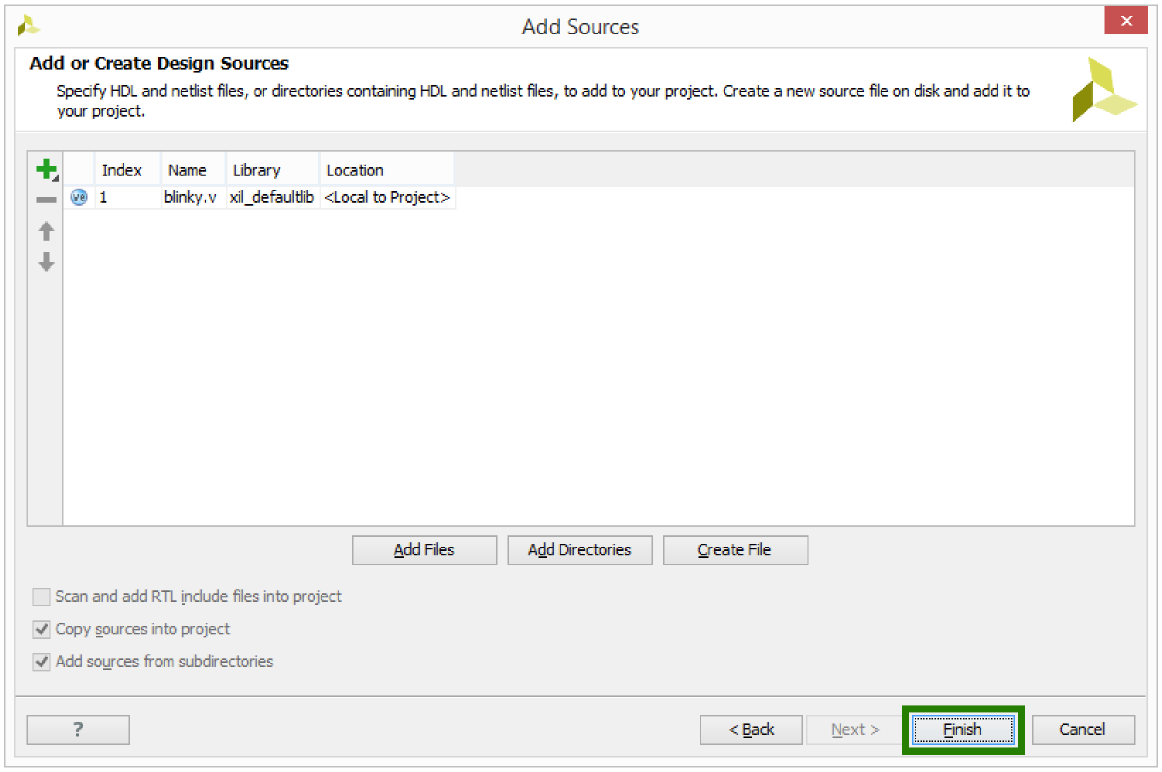
Create a new source file and add it to your project.

File type: Verilog

File name: blinky.v

File location: <Local to Project>

Make sure the options shown are selected in the “create source file”pop up, and for the sake of following enter “blinky” for the file name. Click the ok button when finished. Click the “finish button” and Vivado will then bring up the define module name.



Add or Create Design Sources

Specify HDL and netlist files, or directories containing HDL and netlist files, to add to your project. Create a new source file on disk and add it to your project.

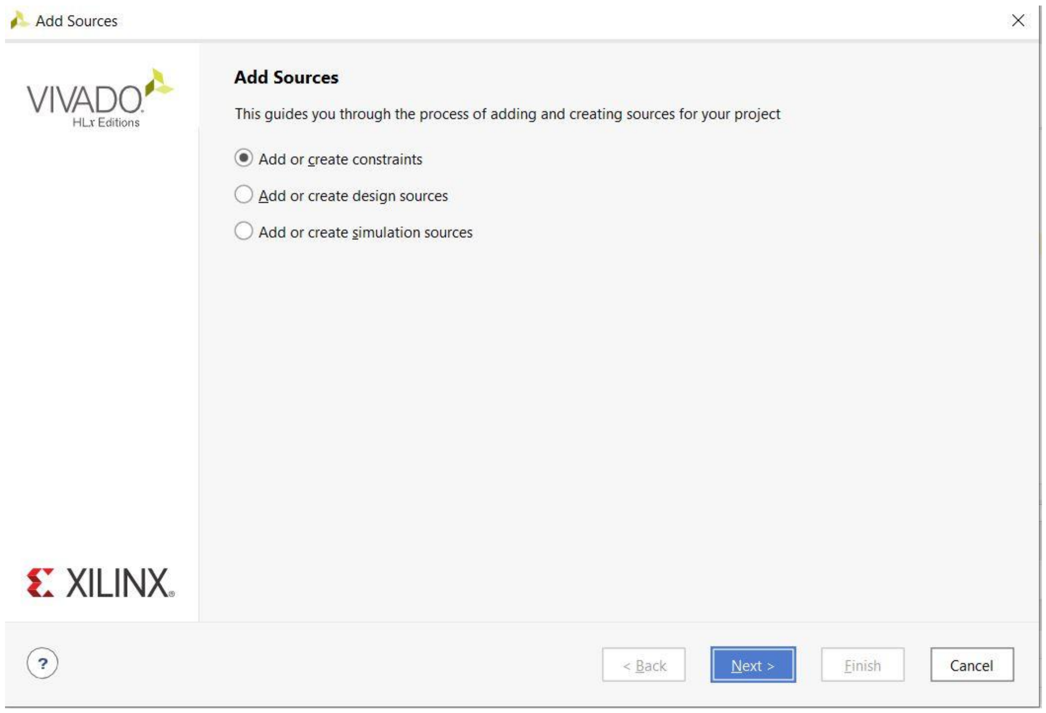
Index	Name	Library	Location
1	blinky.v	xil_defaultlib	<Local to Project>

☐ Scan and add RTL include files into project
☑ Copy sources into project
☑ Add sources from subdirectories

You can use the “define module” window to automatically write some of the verilog code. Additional “I/O definitions” can be added by either clicking the green “+”symbol in the upper left or by simply clicking on the next empty line.
Note that if you would rather write your own code from scratch you can just simply click the “cancel”button and Vivado will create a completely blank Verilog source file inside your project.If you click the “OK” button without defining any “I/O definitions” Vivado will still write the basic Verilog code structure but the port definition will be empty and commented and you can write down the required remaining program.

STEPS FOR CONSTRAINTS FILE CREATION

Click on “Add sources” under the project manager phase of the Flow navigator.
Select the “Add or create constraints” radialand then click the “Next>”button

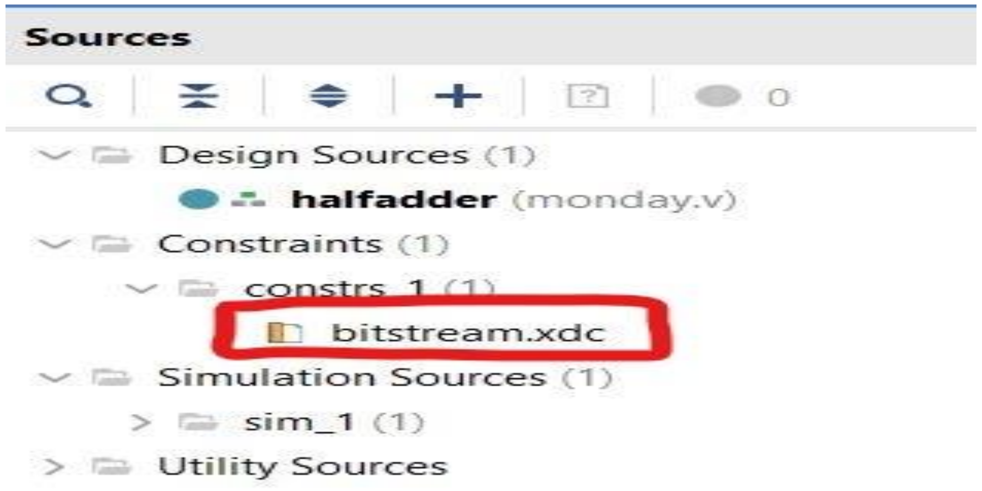

Add Sources

Add Sources

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

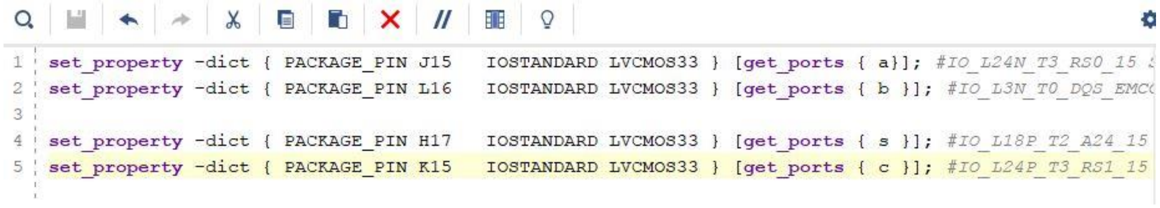
[✓ Add or create constraints]
Add or create design sources
Add or create simulation sources

Then appears a popup asking for file name of the constraint file. Assign name to the constraint file and make sure **that there are no spaces!** and then click “OK”.


Sources

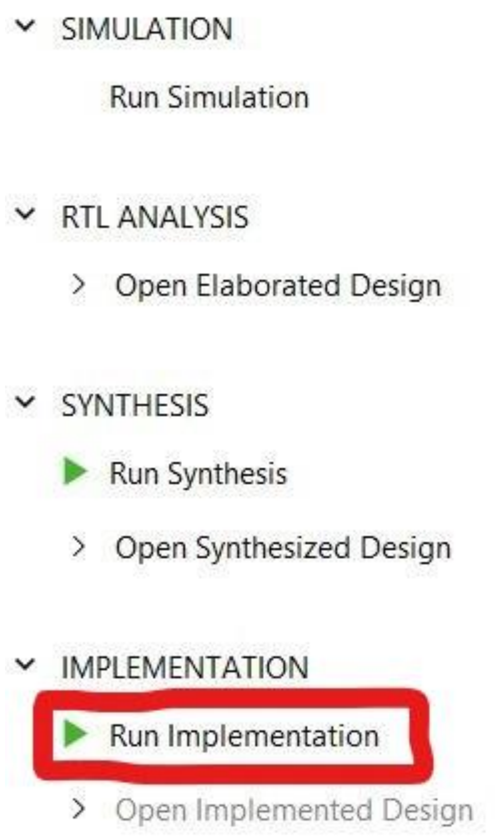
Design Sources (1)
halfadder (monday.v)
Constraints (1)
constrs_1 (1)
bitstream.xdc
Simulation Sources (1)
sim_1 (1)
Utility Sources

Write the bit file according to the requirement.The required memory locations are mapped to the program variables.

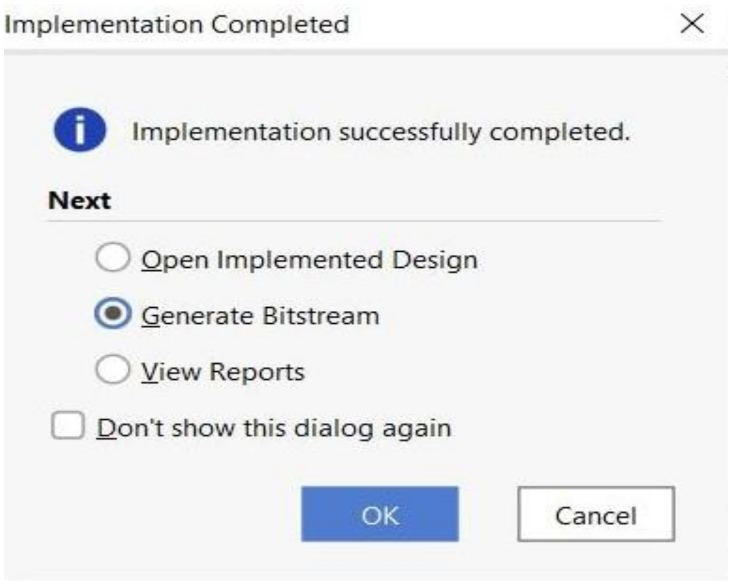


```
set_property -dict { PACKAGE_PIN J15   IOSTANDARD LVCMOS33 } [get_ports { a}]; #IO_L24N_T3_RS0_15
set_property -dict { PACKAGE_PIN L16   IOSTANDARD LVCMOS33 } [get_ports { b }]; #IO_L3N_T0_DQS_EMC

set_property -dict { PACKAGE_PIN H17   IOSTANDARD LVCMOS33 } [get_ports { s }]; #IO_L18P_T2_A24_15
set_property -dict { PACKAGE_PIN K15   IOSTANDARD LVCMOS33 } [get_ports { c }]; #IO_L24P_T3_RS1_15
```

The above written bit file is for halfadder. The input variables a and b are mapped to input switch locations and the output variables s,c are mapped to output LED locations.In this way a bit file can be written according to our program requirements. We can also access locations other than switches and LED's like LCD display,clock,buttons and Pmod Headers. The Pmod Headers are used for external interfacing.
After writing bit file click on "Run Implementation"



After successfully completing the implementation a pop up appears as follows


Implementation Completed

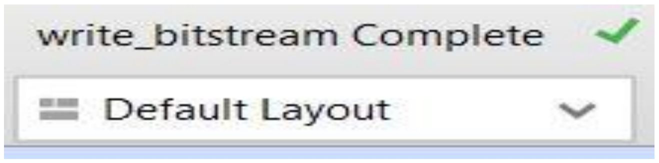
Implementation successfully completed.

Next

( ) Open Implemented Design
(●) Generate Bitstream
( ) View Reports

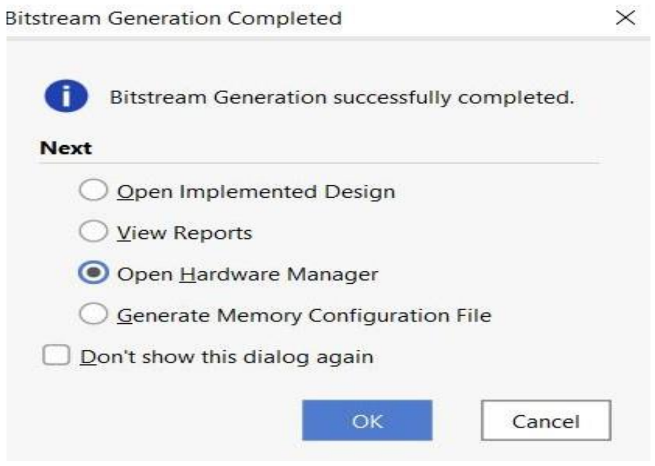
☐ Don't show this dialog again

Click on “Generate Bitstream” and then “OK”.


write_bitstream Complete ✓

This is an indication for completion of the generation of the bitstream which appears on top right corner.

Now we have to connect to hardware device to target the device


Bitstream Generation Completed

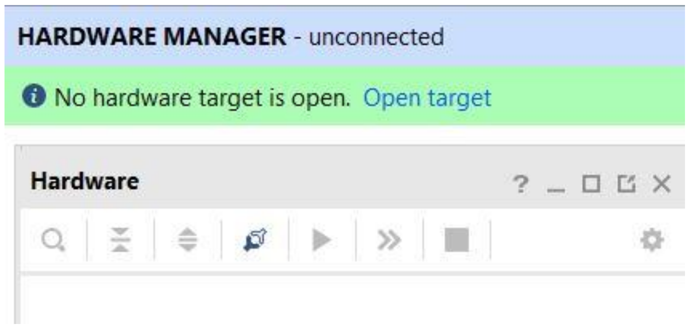
Bitstream Generation successfully completed.

Next

( ) Open Implemented Design
( ) View Reports
(●) Open Hardware Manager
( ) Generate Memory Configuration File

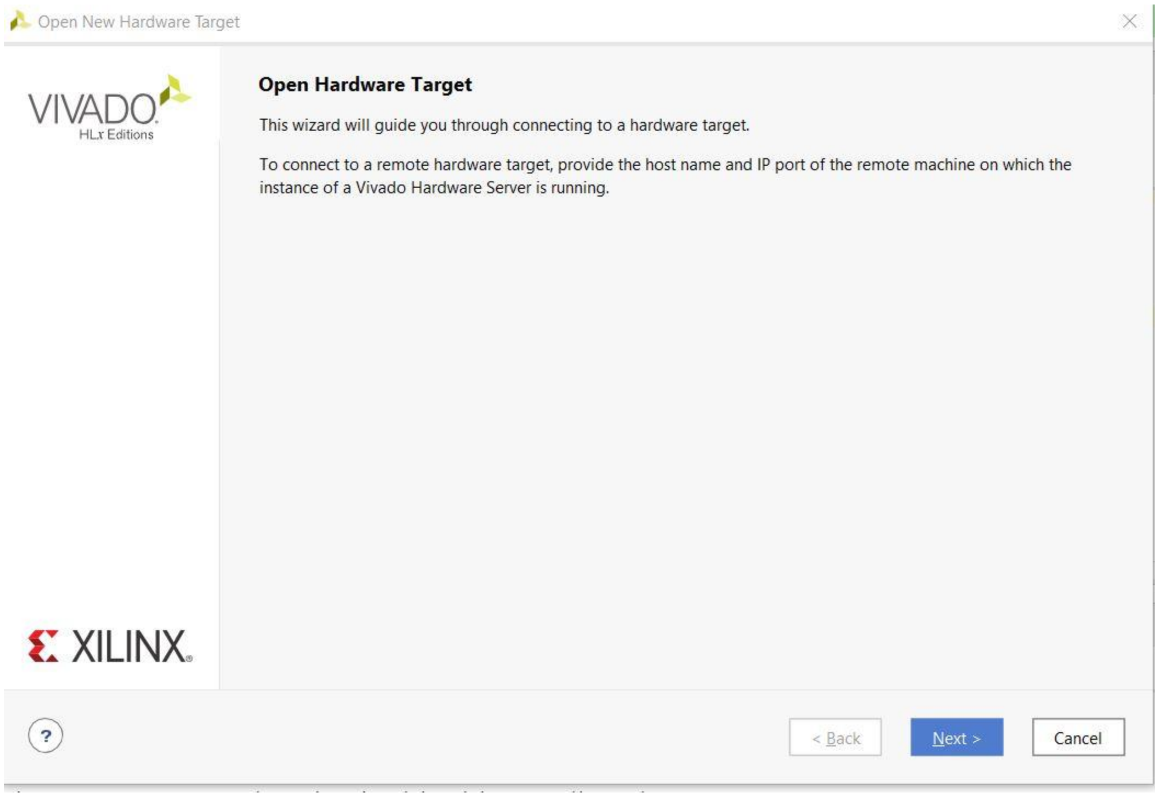
☐ Don't show this dialog again

Click on “Open Hardware Manager” and then “OK”


HARDWARE MANAGER - unconnected

No hardware target is open. Open target

Click on “Open Target” to target a device.The Open New Hardware Target wizard will launch,click the “Next>” button to proceed.

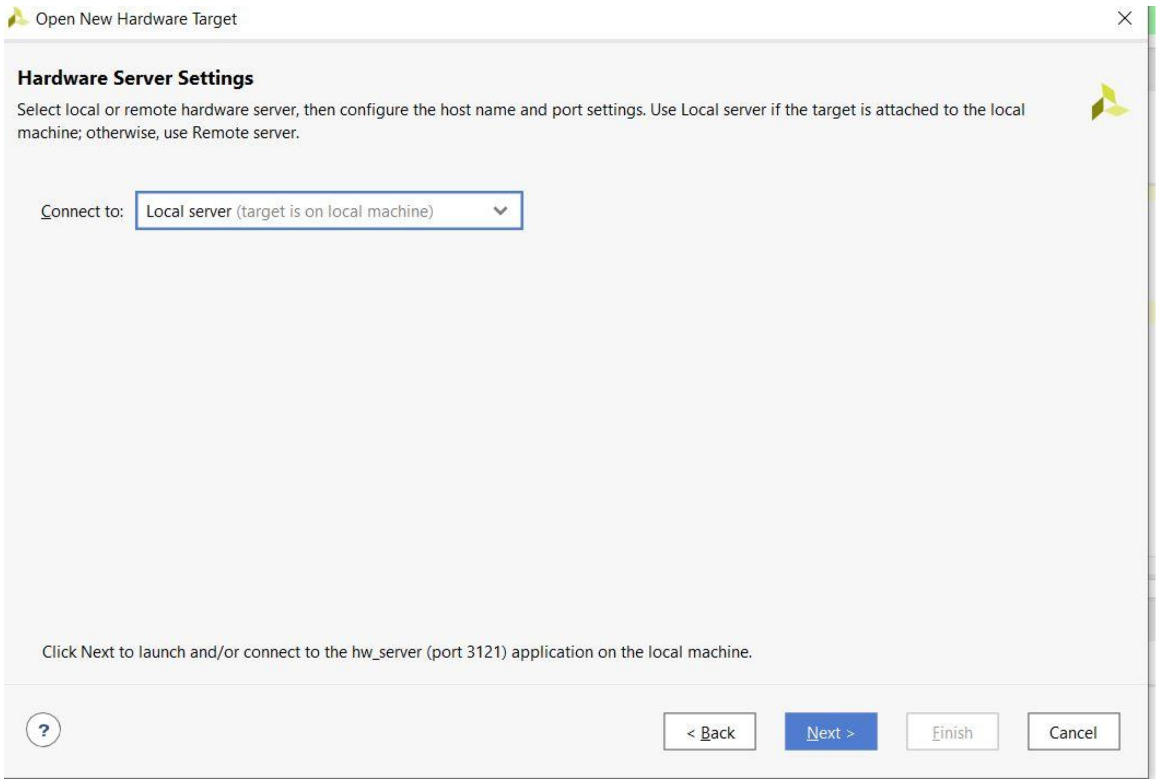

Open New Hardware Target

Open Hardware Target

This wizard will guide you through connecting to a hardware target.

To connect to a remote hardware target, provide the host name and IP port of the remote machine on which the instance of a Vivado Hardware Server is running.

Select the “local server(target is on local machine)” from the drop down if it is’nt already,and then click the “Next>”button to proceed. Vivado will work for a moment to find any valid target devices connected to your local machine.



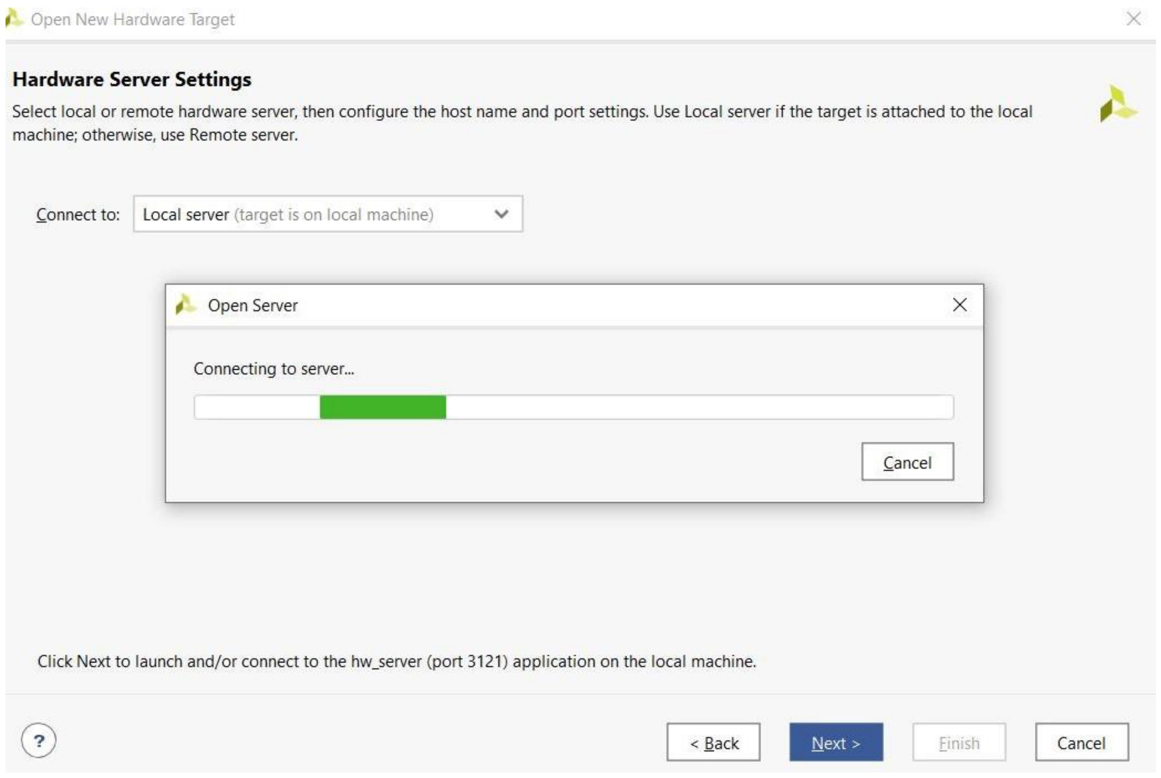
Open New Hardware Target

Hardware Server Settings

Select local or remote hardware server, then configure the host name and port settings. Use Local server if the target is attached to the local machine; otherwise, use Remote server.

Connect to: Local server (target is on local machine)

Click Next to launch and/or connect to the hw_server (port 3121) application on the local machine.



Open New Hardware Target

Hardware Server Settings

Select local or remote hardware server, then configure the host name and port settings. Use Local server if the target is attached to the local machine; otherwise, use Remote server.

Connect to: Local server (target is on local machine)

Open Server

Connecting to server...

Click Next to launch and/or connect to the hw_server (port 3121) application on the local machine.

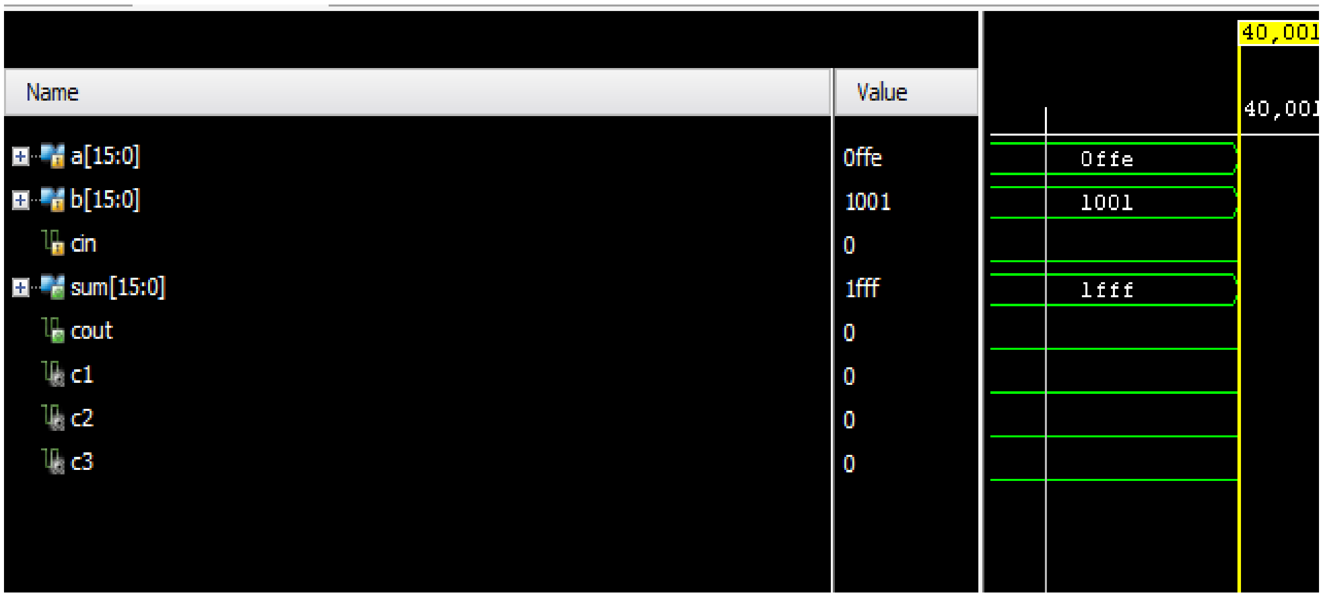
Select your specific Hardware device. Click the “Finish”button and Vivado will attempt to connect to your specified hardware.Now click “Program device” under the program and debug phase of the Flow Navigator and then your specific device from the menu that appears.

After sometime the device will be programmed and the required outputs(LED’s) can be obtained by varying the inputs(switches)

CHAPTER 5

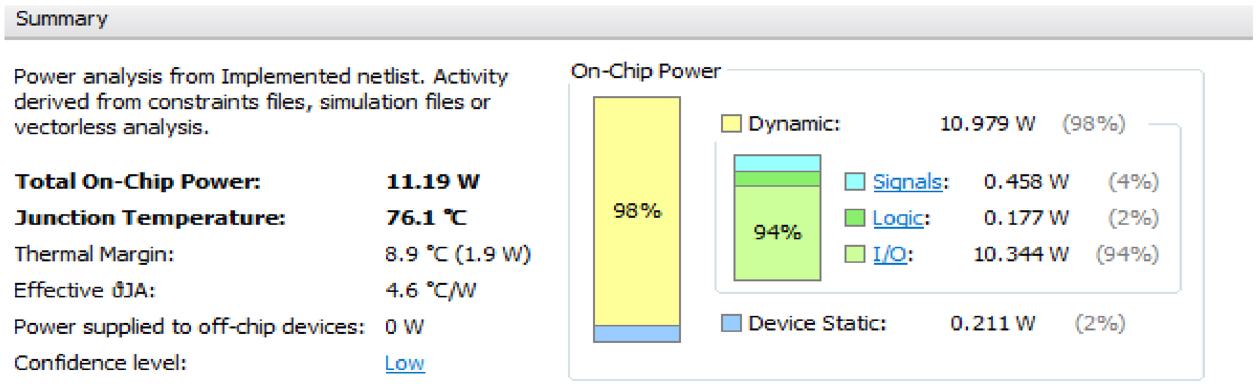
RESULTS AND CONCLUSIONS

In this chapter various adders like carry look ahead adder,ripple carry adder and SPST adder and various multipliers like vedic multipler,array multiplier and also combination of adder and multiplier like vedic spst,Floating Point Multiplication(FPM) were simulated using Xilinx vivado 2016.1 design suite and the results are presented.Synthesis andimplementation is done using nexys 4 ddr board based on the artix 7field programmable gate array(fpga) from Xilinx.Also the performance comparision multipliers in terms of power are presented.



```
Name        Value
a[15:0]     0ffe
b[15:0]     1001
cin         0
sum[15:0]   1fff
cout        0
c1          0
c2          0
c3          0
```

Fig 5.1(a) CLA Adder Simulation Report



Summary

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

Total On-Chip Power:	**11.19 W**
Junction Temperature:	**76.1 ℃**
Thermal Margin:	8.9 ℃ (1.9 W)
Effective ϑJA:	4.6 ℃/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

On-Chip Power

Dynamic:	10.979 W	(98%)
Signals:	0.458 W	(4%)
Logic:	0.177 W	(2%)
I/O:	10.344 W	(94%)
Device Static:	0.211 W	(2%)

Fig5.1(b) CLA Adder Power Report

```
+-------------------------+------+-------+-----------+-------+
|        Site Type        | Used | Fixed | Available | Util% |
+-------------------------+------+-------+-----------+-------+
| Slice LUTs              |   27 |     0 |     63400 |  0.04 |
|   LUT as Logic          |   27 |     0 |     63400 |  0.04 |
|   LUT as Memory         |    0 |     0 |     19000 |  0.00 |
| Slice Registers         |    0 |     0 |    126800 |  0.00 |
|   Register as Flip Flop |    0 |     0 |    126800 |  0.00 |
|   Register as Latch     |    0 |     0 |    126800 |  0.00 |
| F7 Muxes                |    0 |     0 |     31700 |  0.00 |
| F8 Muxes                |    0 |     0 |     15850 |  0.00 |
+-------------------------+------+-------+-----------+-------+
```

Fig5.1(c) CLA Adder Synthesis Report

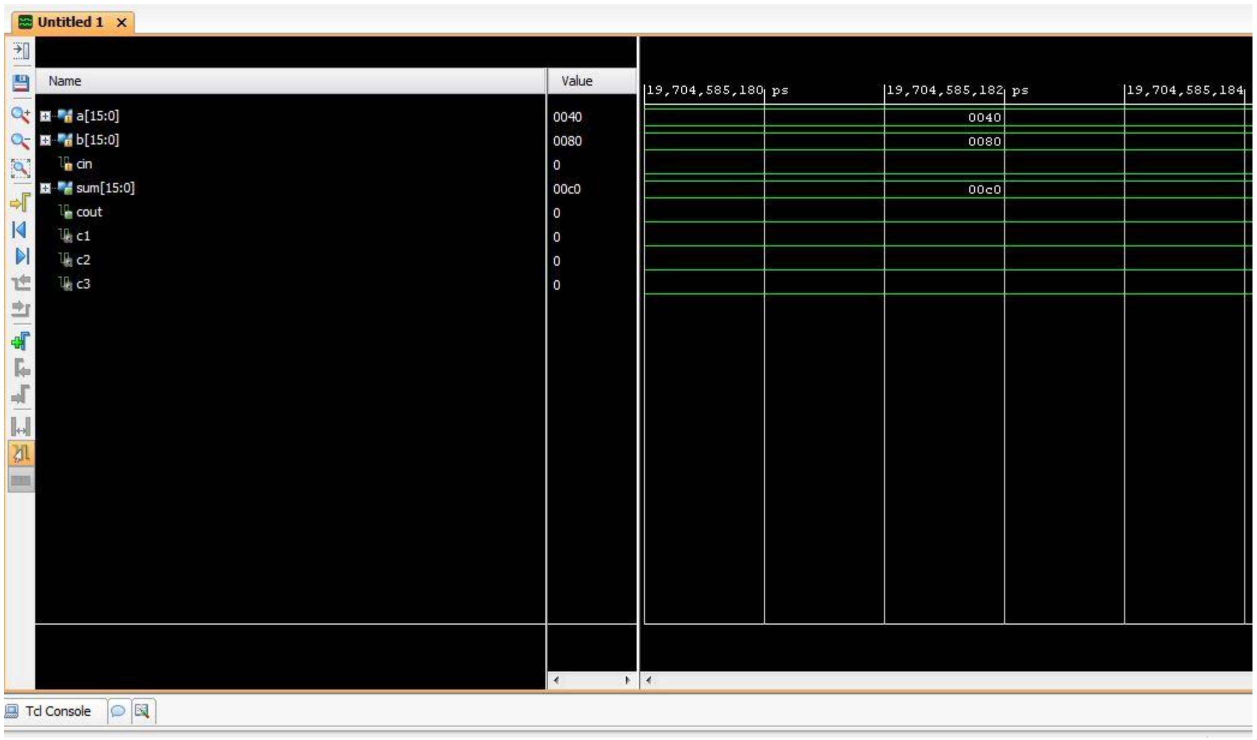
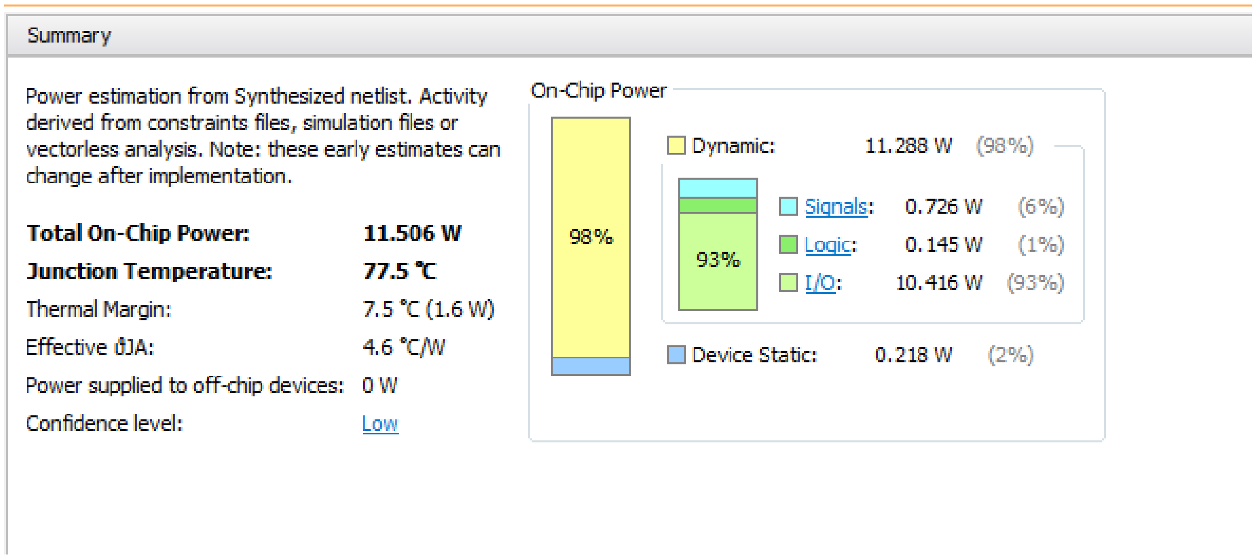


Fig 5.2(a) RCA Simulation Report


Summary

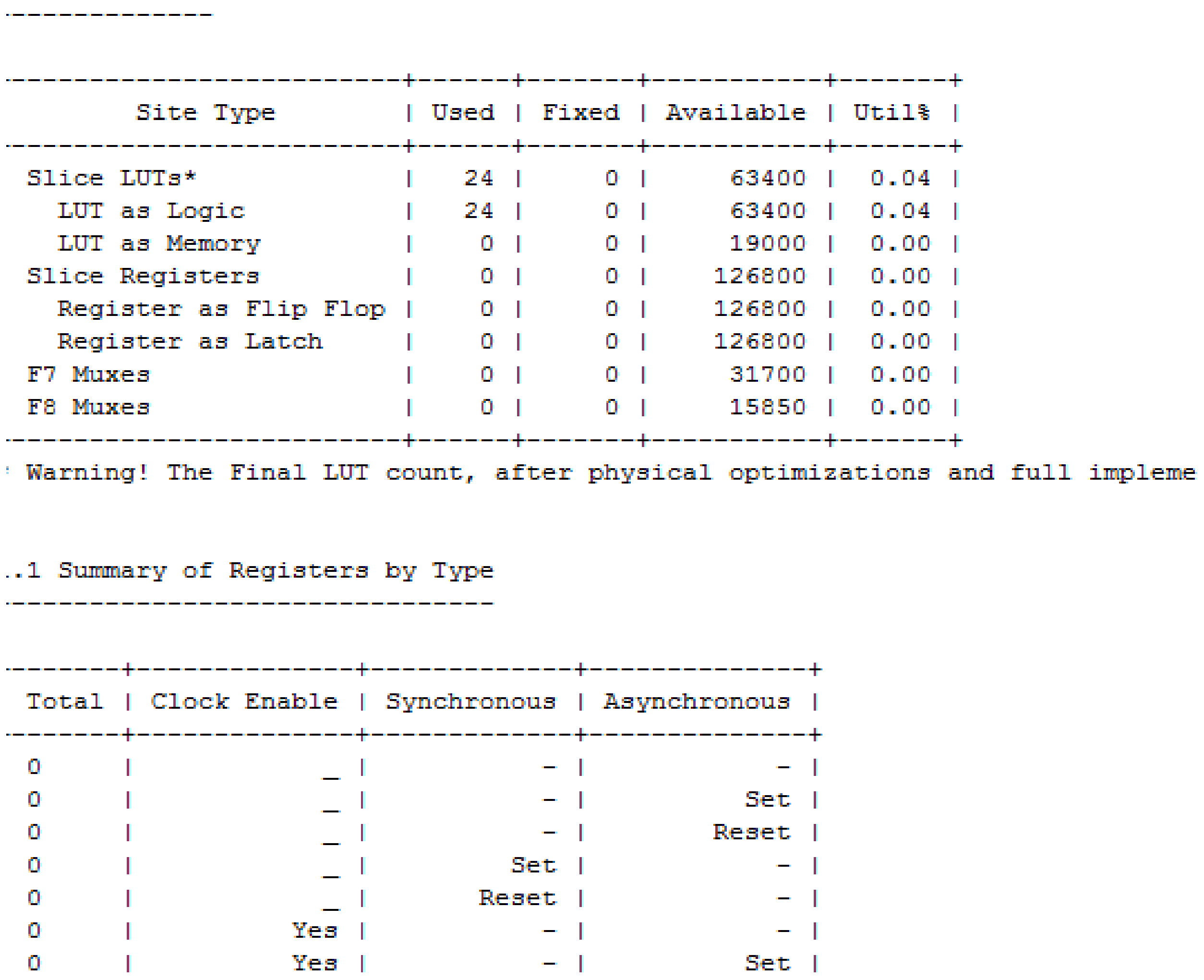
Power estimation from Synthesized netlist. Activity derived from constraints files, simulation files or vectorless analysis. Note: these early estimates can change after implementation.

Total On-Chip Power:	**11.506 W**
Junction Temperature:	**77.5 ℃**
Thermal Margin:	7.5 ℃ (1.6 W)
Effective ϑJA:	4.6 ℃/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

On-Chip Power

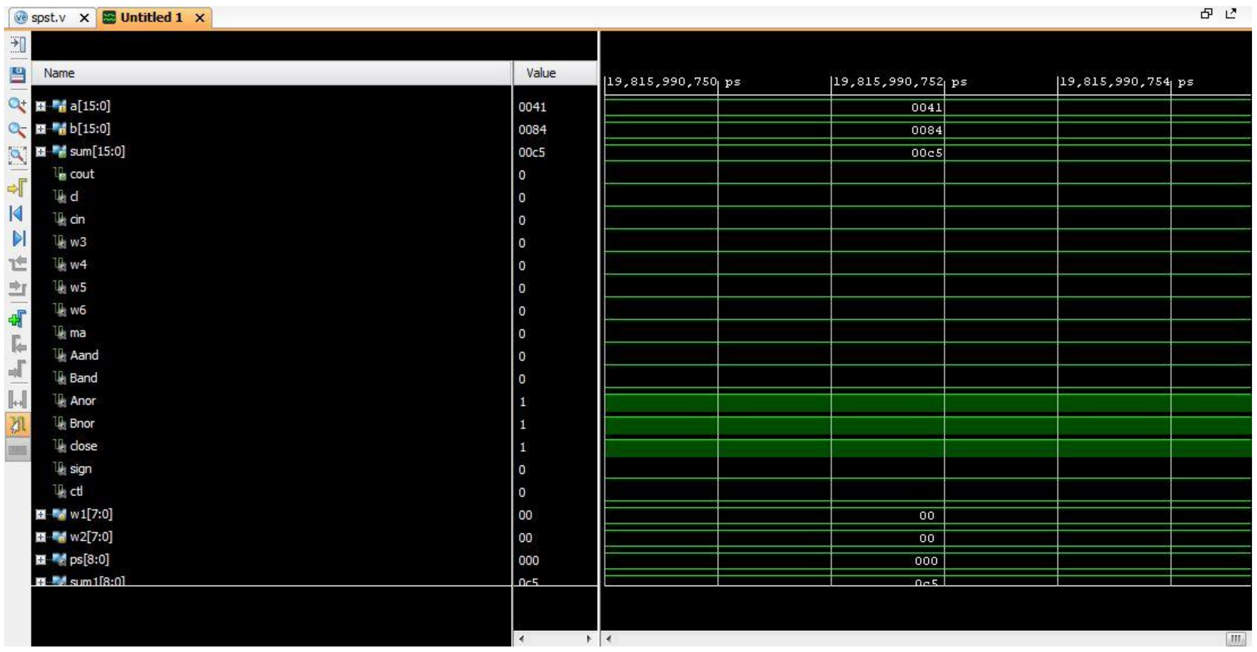
Dynamic:	11.288 W	(98%)
Signals:	0.726 W	(6%)
Logic:	0.145 W	(1%)
I/O:	10.416 W	(93%)
Device Static:	0.218 W	(2%)

Fig 5.2(b) RCA Power Report



```
--------------

+-------------------------+------+-------+-----------+-------+
|        Site Type        | Used | Fixed | Available | Util% |
+-------------------------+------+-------+-----------+-------+
| Slice LUTs*             |   24 |     0 |     63400 |  0.04 |
|   LUT as Logic          |   24 |     0 |     63400 |  0.04 |
|   LUT as Memory         |    0 |     0 |     19000 |  0.00 |
| Slice Registers         |    0 |     0 |    126800 |  0.00 |
|   Register as Flip Flop |    0 |     0 |    126800 |  0.00 |
|   Register as Latch     |    0 |     0 |    126800 |  0.00 |
| F7 Muxes                |    0 |     0 |     31700 |  0.00 |
| F8 Muxes                |    0 |     0 |     15850 |  0.00 |
+-------------------------+------+-------+-----------+-------+
* Warning! The Final LUT count, after physical optimizations and full impleme


..1 Summary of Registers by Type
--------------------------------

+-------+--------------+-------------+--------------+
| Total | Clock Enable | Synchronous | Asynchronous |
+-------+--------------+-------------+--------------+
| 0     |            _ |           - |            - |
| 0     |            _ |           - |          Set |
| 0     |            _ |           - |        Reset |
| 0     |            _ |         Set |            - |
| 0     |            _ |       Reset |            - |
| 0     |          Yes |           - |            - |
| 0     |          Yes |           - |          Set |
```

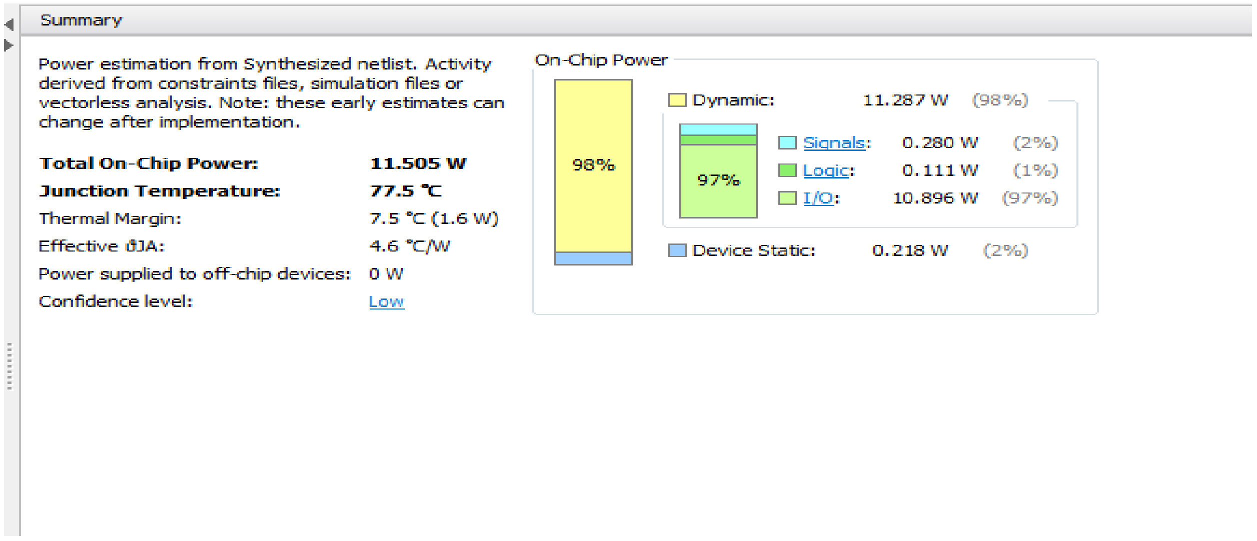
Fig 5.2(b) RCA Synthesis Report



Name	Value
a[15:0]	0041
b[15:0]	0084
sum[15:0]	00c5
cout	0
d	0
cin	0
w3	0
w4	0
w5	0
w6	0
ma	0
Aand	0
Band	0
Anor	1
Bnor	1
close	1
sign	0
ctl	0
w1[7:0]	00
w2[7:0]	00
ps[8:0]	000
sum1[8:0]	0c5

```
19,815,990,750 ps        19,815,990,752 ps        19,815,990,754 ps
```

Fig 5.3 (a) SPST Adder Simulation Report



Summary

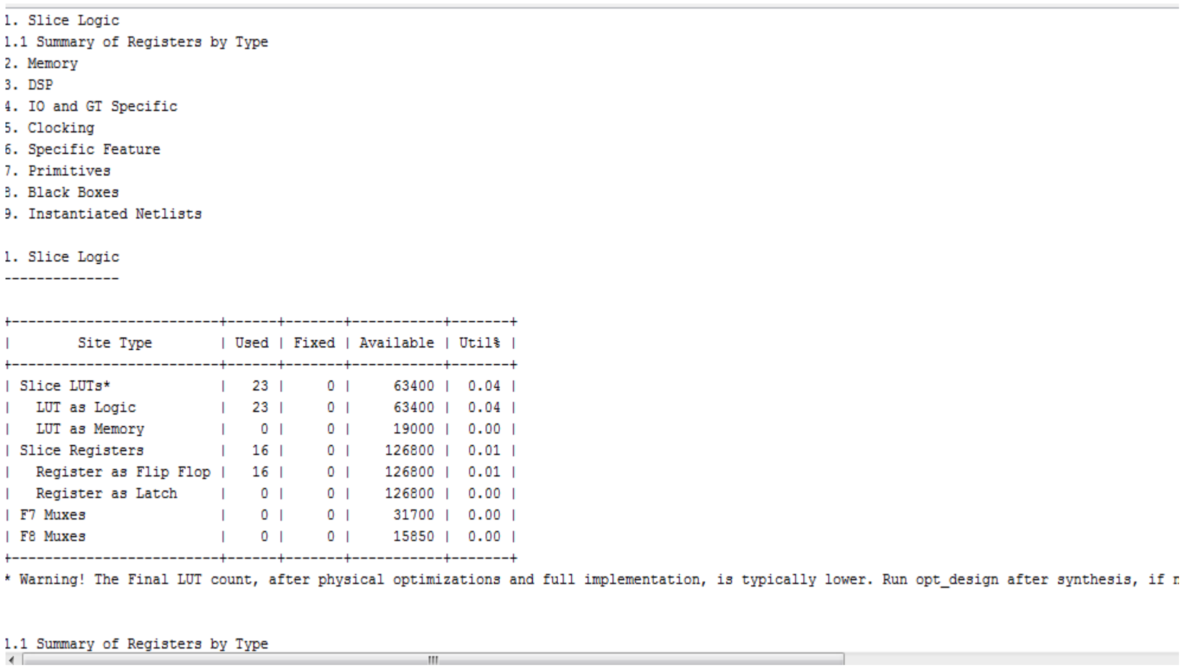
Power estimation from Synthesized netlist. Activity derived from constraints files, simulation files or vectorless analysis. Note: these early estimates can change after implementation.

Total On-Chip Power:	**11.505 W**
Junction Temperature:	**77.5 ℃**
Thermal Margin:	7.5 ℃ (1.6 W)
Effective ϑJA:	4.6 ℃/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

On-Chip Power

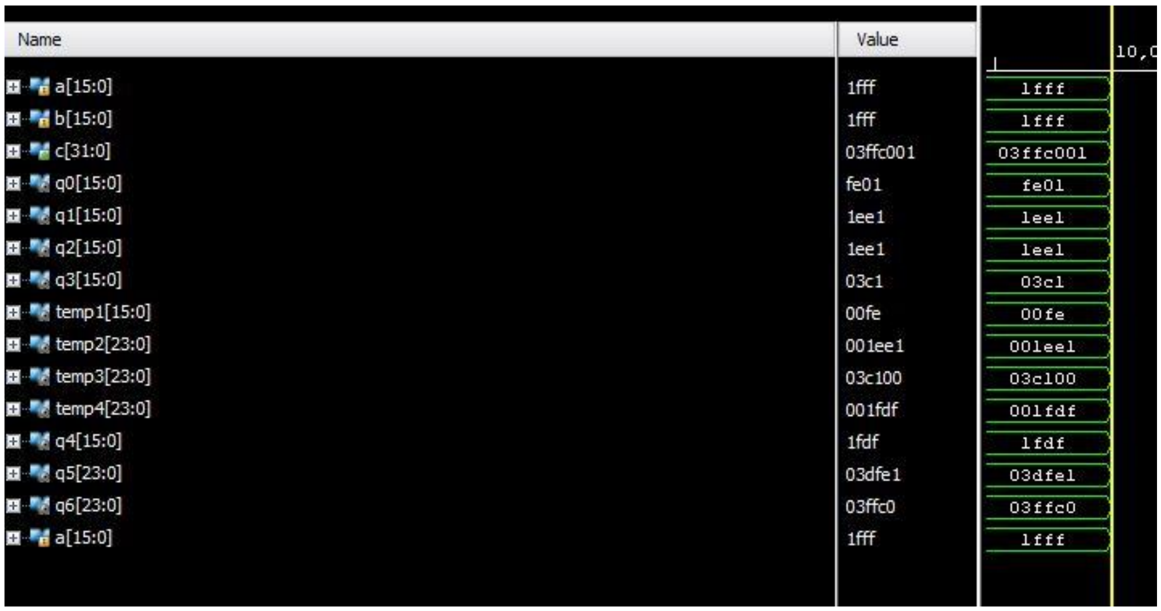
Dynamic:	11.287 W	(98%)
Signals:	0.280 W	(2%)
Logic:	0.111 W	(1%)
I/O:	10.896 W	(97%)
Device Static:	0.218 W	(2%)

Fig 5.3(b) SPST Power Report



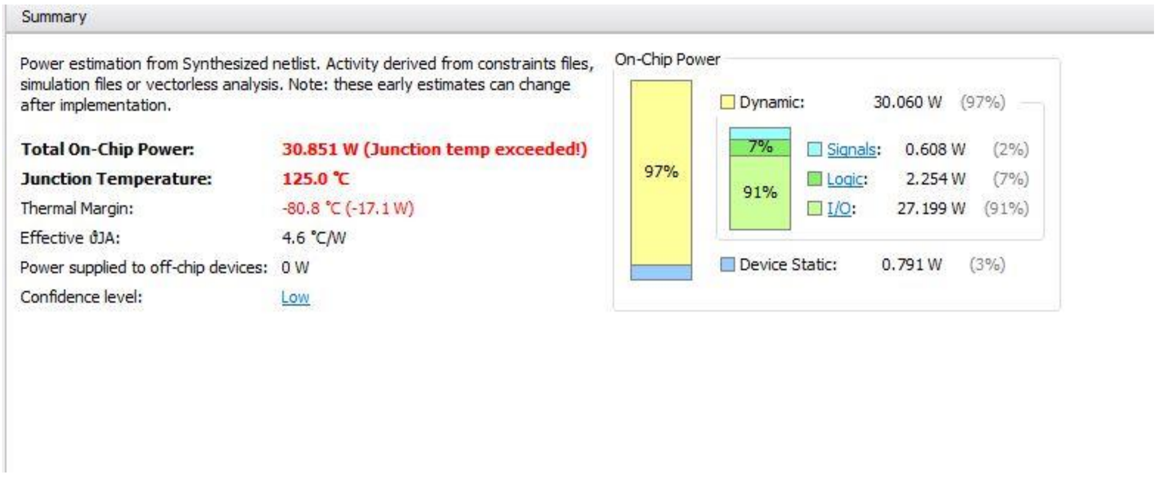
```
1. Slice Logic
1.1 Summary of Registers by Type
2. Memory
3. DSP
4. IO and GT Specific
5. Clocking
6. Specific Feature
7. Primitives
8. Black Boxes
9. Instantiated Netlists

1. Slice Logic
--------------

+-------------------------+------+-------+-----------+-------+
|        Site Type        | Used | Fixed | Available | Util% |
+-------------------------+------+-------+-----------+-------+
| Slice LUTs*             |   23 |     0 |     63400 |  0.04 |
|   LUT as Logic          |   23 |     0 |     63400 |  0.04 |
|   LUT as Memory         |    0 |     0 |     19000 |  0.00 |
| Slice Registers         |   16 |     0 |    126800 |  0.01 |
|   Register as Flip Flop |   16 |     0 |    126800 |  0.01 |
|   Register as Latch     |    0 |     0 |    126800 |  0.00 |
| F7 Muxes                |    0 |     0 |     31700 |  0.00 |
| F8 Muxes                |    0 |     0 |     15850 |  0.00 |
+-------------------------+------+-------+-----------+-------+
* Warning! The Final LUT count, after physical optimizations and full implementation, is typically lower. Run opt_design after synthesis, if r


1.1 Summary of Registers by Type
```

Fig 5.3(c) SPST Synthesis Report



Name	Value	
a[15:0]	1fff	1fff
b[15:0]	1fff	1fff
c[31:0]	03ffc001	03ffc001
q0[15:0]	fe01	fe01
q1[15:0]	1ee1	1ee1
q2[15:0]	1ee1	1ee1
q3[15:0]	03c1	03c1
temp1[15:0]	00fe	00fe
temp2[23:0]	001ee1	001ee1
temp3[23:0]	03c100	03c100
temp4[23:0]	001fdf	001fdf
q4[15:0]	1fdf	1fdf
q5[23:0]	03dfe1	03dfe1
q6[23:0]	03ffc0	03ffc0
a[15:0]	1fff	1fff

Fig 5.4(a) Vedic Multiplier Simulation Report



Summary

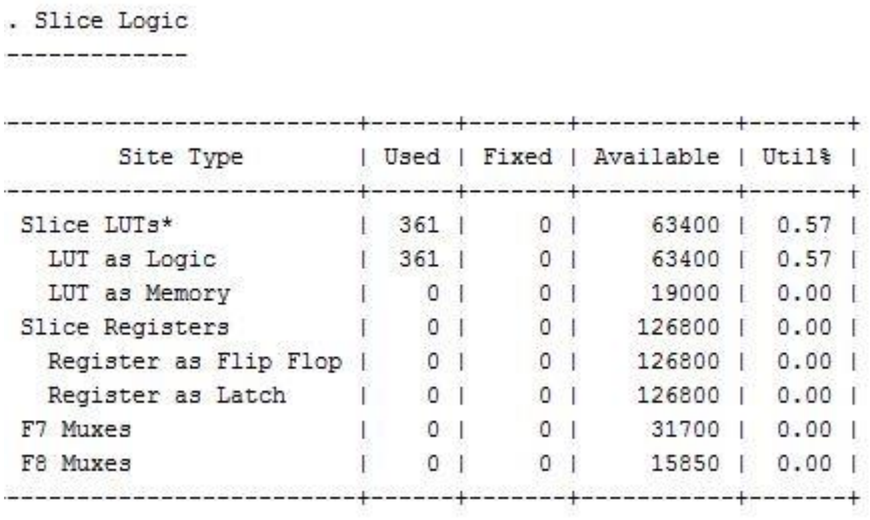
Power estimation from Synthesized netlist. Activity derived from constraints files, simulation files or vectorless analysis. Note: these early estimates can change after implementation.

Total On-Chip Power:	**30.851 W (Junction temp exceeded!)**
Junction Temperature:	**125.0 ℃**
Thermal Margin:	-80.8 ℃ (-17.1 W)
Effective ϑJA:	4.6 ℃/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

On-Chip Power

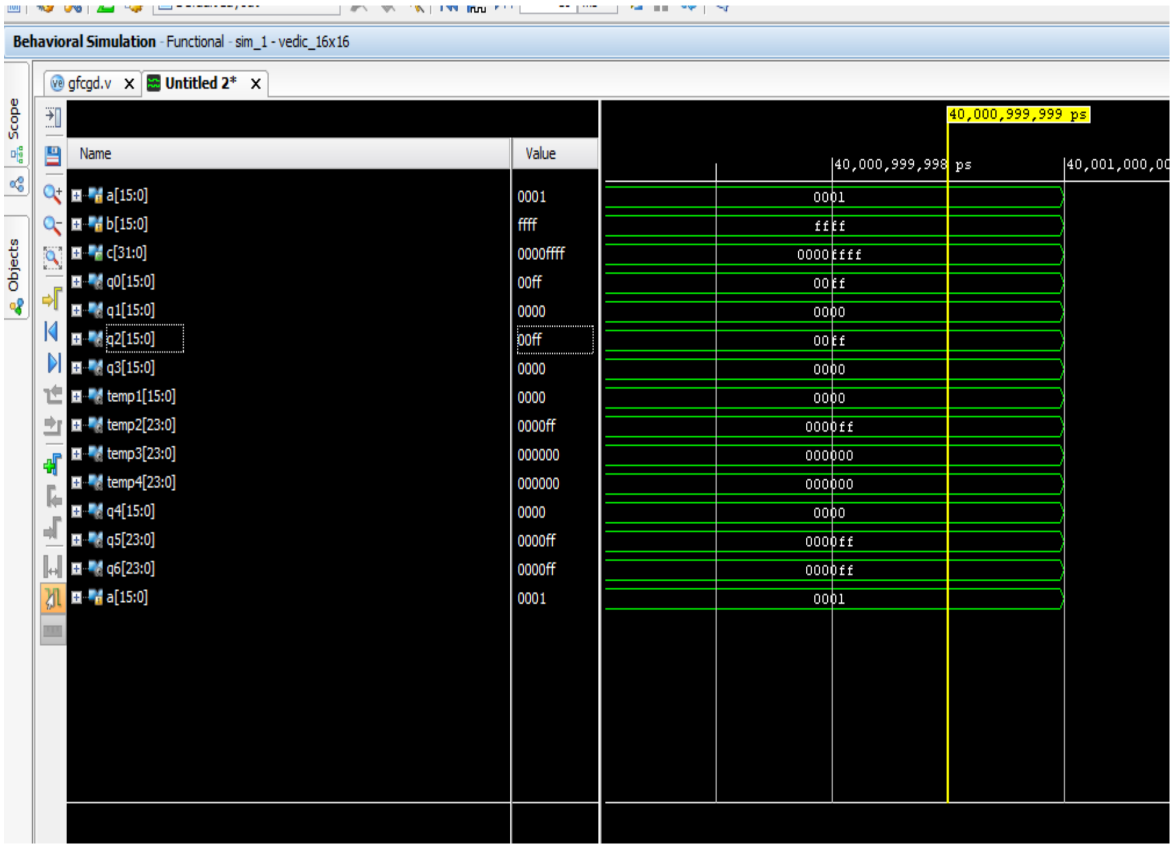
Dynamic:	30.060 W	(97%)
Signals:	0.608 W	(2%)
Logic:	2.254 W	(7%)
I/O:	27.199 W	(91%)
Device Static:	0.791 W	(3%)

Fig 5.4(b) Vedic Multiplier Power Report



```
. Slice Logic
-------------

+-------------------------+------+-------+-----------+-------+
|        Site Type        | Used | Fixed | Available | Util% |
+-------------------------+------+-------+-----------+-------+
| Slice LUTs*             |  361 |     0 |     63400 |  0.57 |
|   LUT as Logic          |  361 |     0 |     63400 |  0.57 |
|   LUT as Memory         |    0 |     0 |     19000 |  0.00 |
| Slice Registers         |    0 |     0 |    126800 |  0.00 |
|   Register as Flip Flop |    0 |     0 |    126800 |  0.00 |
|   Register as Latch     |    0 |     0 |    126800 |  0.00 |
| F7 Muxes                |    0 |     0 |     31700 |  0.00 |
| F8 Muxes                |    0 |     0 |     15850 |  0.00 |
+-------------------------+------+-------+-----------+-------+
```

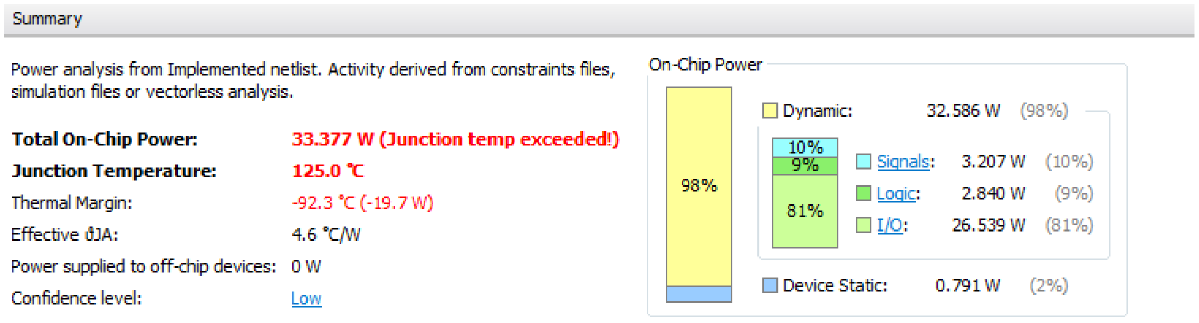
Fig 5.4(c) Vedic Multiplier Synthesis Report



Behavioral Simulation - Functional - sim_1 - vedic_16x16

Name	Value
a[15:0]	0001
b[15:0]	ffff
c[31:0]	0000ffff
q0[15:0]	00ff
q1[15:0]	0000
q2[15:0]	00ff
q3[15:0]	0000
temp1[15:0]	0000
temp2[23:0]	0000ff
temp3[23:0]	000000
temp4[23:0]	000000
q4[15:0]	0000
q5[23:0]	0000ff
q6[23:0]	0000ff
a[15:0]	0001

Fig 5.5(a) Vedic Multiplier Using SPST Simulation Report



Summary

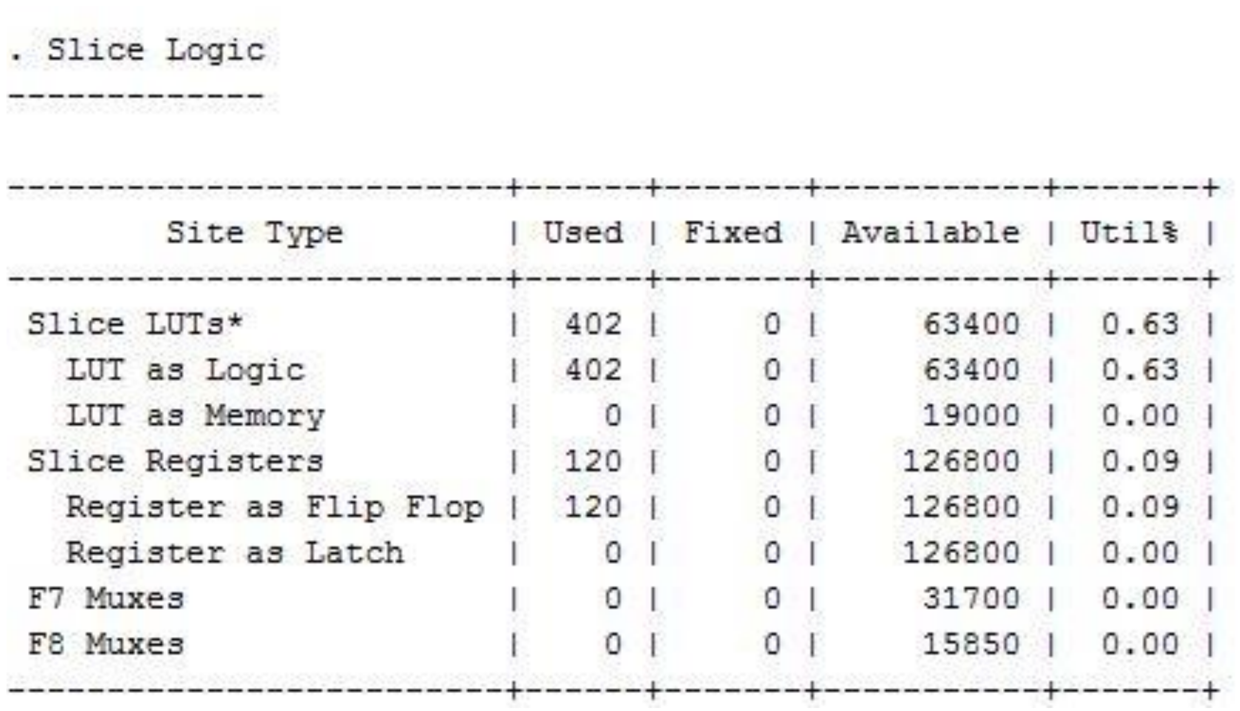
Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**33.377 W (Junction temp exceeded!)**
Junction Temperature:	**125.0 ℃**
Thermal Margin:	-92.3 ℃ (-19.7 W)
Effective ϑJA:	4.6 ℃/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

On-Chip Power

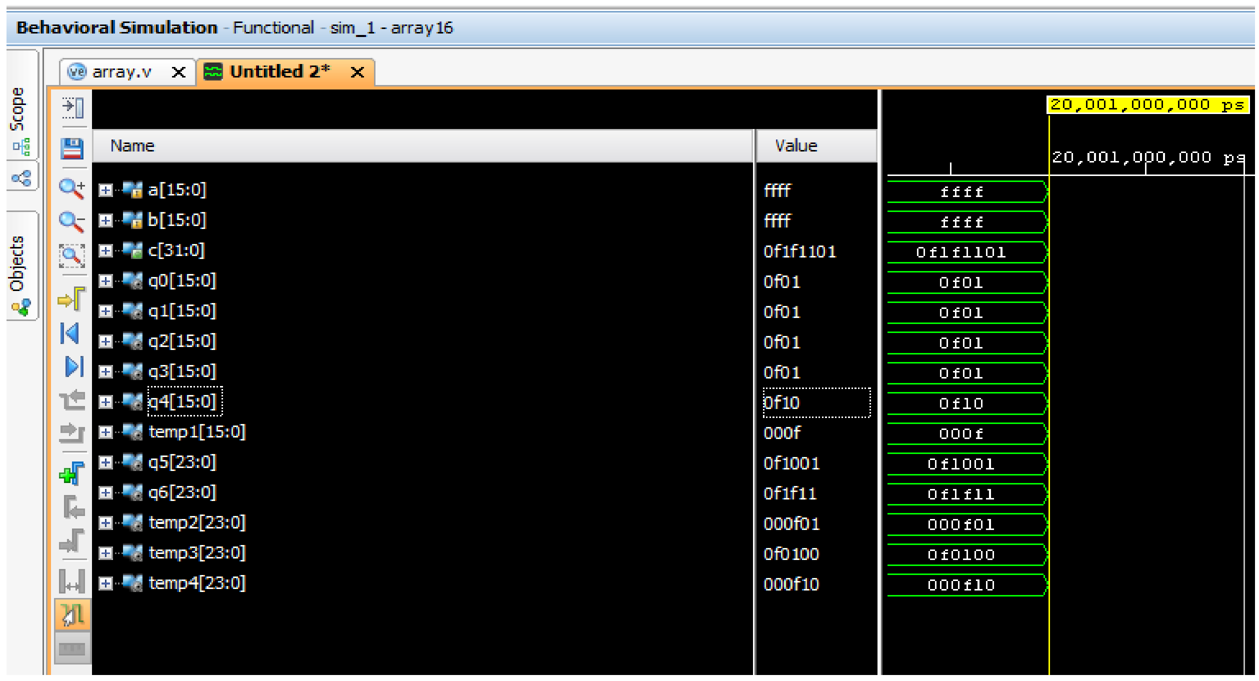
Dynamic:	32.586 W	(98%)
Signals:	3.207 W	(10%)
Logic:	2.840 W	(9%)
I/O:	26.539 W	(81%)
Device Static:	0.791 W	(2%)

Fig 5.5(b) Vedic Using SPST Synthesis Report



```
. Slice Logic
-------------

+-------------------------+------+-------+-----------+-------+
|        Site Type        | Used | Fixed | Available | Util% |
+-------------------------+------+-------+-----------+-------+
| Slice LUTs*             |  402 |     0 |     63400 |  0.63 |
|   LUT as Logic          |  402 |     0 |     63400 |  0.63 |
|   LUT as Memory         |    0 |     0 |     19000 |  0.00 |
| Slice Registers         |  120 |     0 |    126800 |  0.09 |
|   Register as Flip Flop |  120 |     0 |    126800 |  0.09 |
|   Register as Latch     |    0 |     0 |    126800 |  0.00 |
| F7 Muxes                |    0 |     0 |     31700 |  0.00 |
| F8 Muxes                |    0 |     0 |     15850 |  0.00 |
+-------------------------+------+-------+-----------+-------+
```

Fig 5.5(c) VEDIC MULTIPLIER USING SPST POWER Report



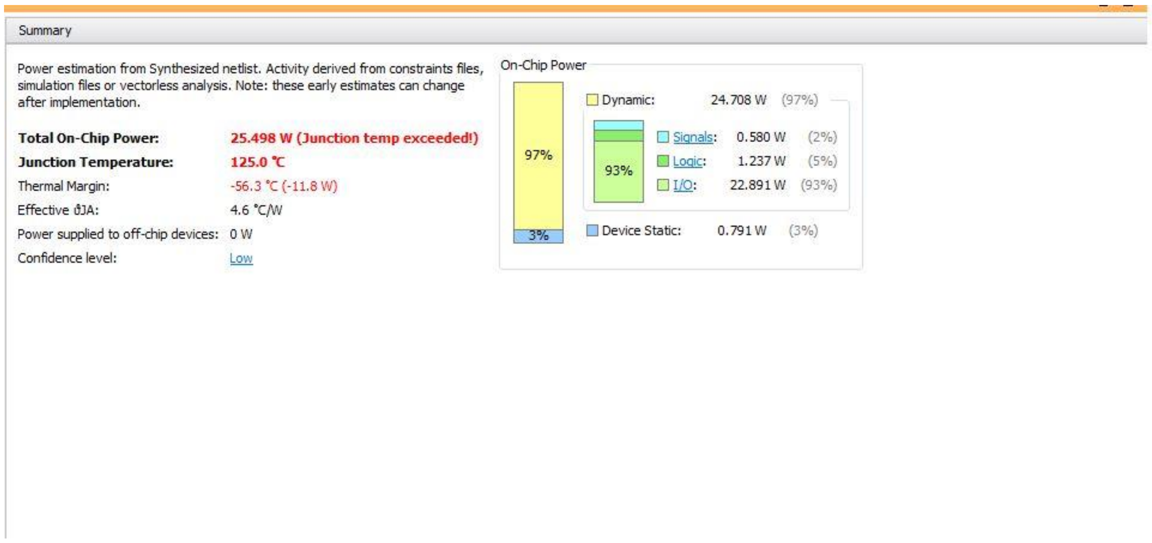
Behavioral Simulation - Functional - sim_1 - array16

array.v | Untitled 2*

20,001,000,000 ps

Name	Value	20,001,000,000 ps
a[15:0]	ffff	ffff
b[15:0]	ffff	ffff
c[31:0]	0f1f1101	0f1f1101
q0[15:0]	0f01	0f01
q1[15:0]	0f01	0f01
q2[15:0]	0f01	0f01
q3[15:0]	0f01	0f01
q4[15:0]	0f10	0f10
temp1[15:0]	000f	000f
q5[23:0]	0f1001	0f1001
q6[23:0]	0f1f11	0f1f11
temp2[23:0]	000f01	000f01
temp3[23:0]	0f0100	0f0100
temp4[23:0]	000f10	000f10

Fig 5.6(a) Array Multiplier Simulation Report


Summary

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

Total On-Chip Power:	**25.498 W (Junction temp exceeded!)**
Junction Temperature:	**125.0 °C**
Thermal Margin:	-56.3 °C (-11.8 W)
Effective ϑJA:	4.6 °C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

On-Chip Power

Dynamic:	24.708 W	(97%)
Signals:	0.580 W	(2%)
Logic:	1.237 W	(5%)
I/O:	22.891 W	(93%)
Device Static:	0.791 W	(3%)

Fig 5.6 (b) Array Multiplier Power Report

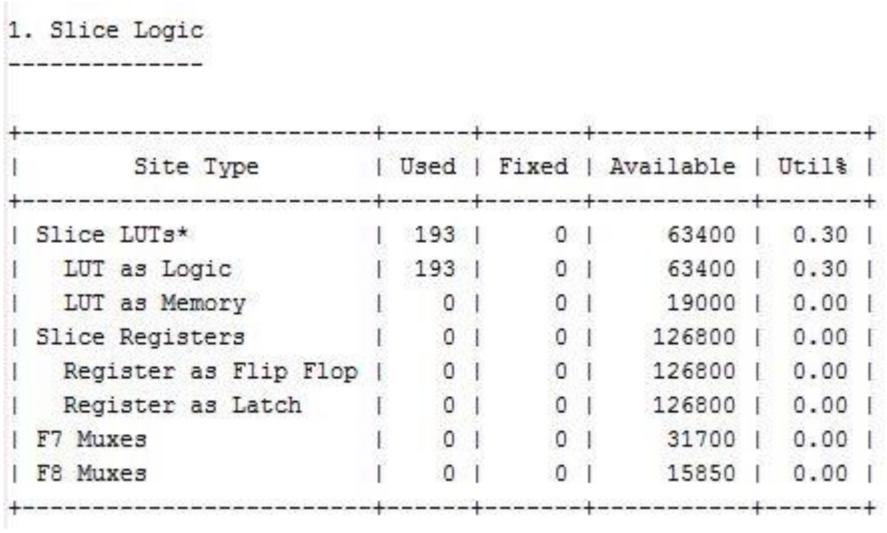
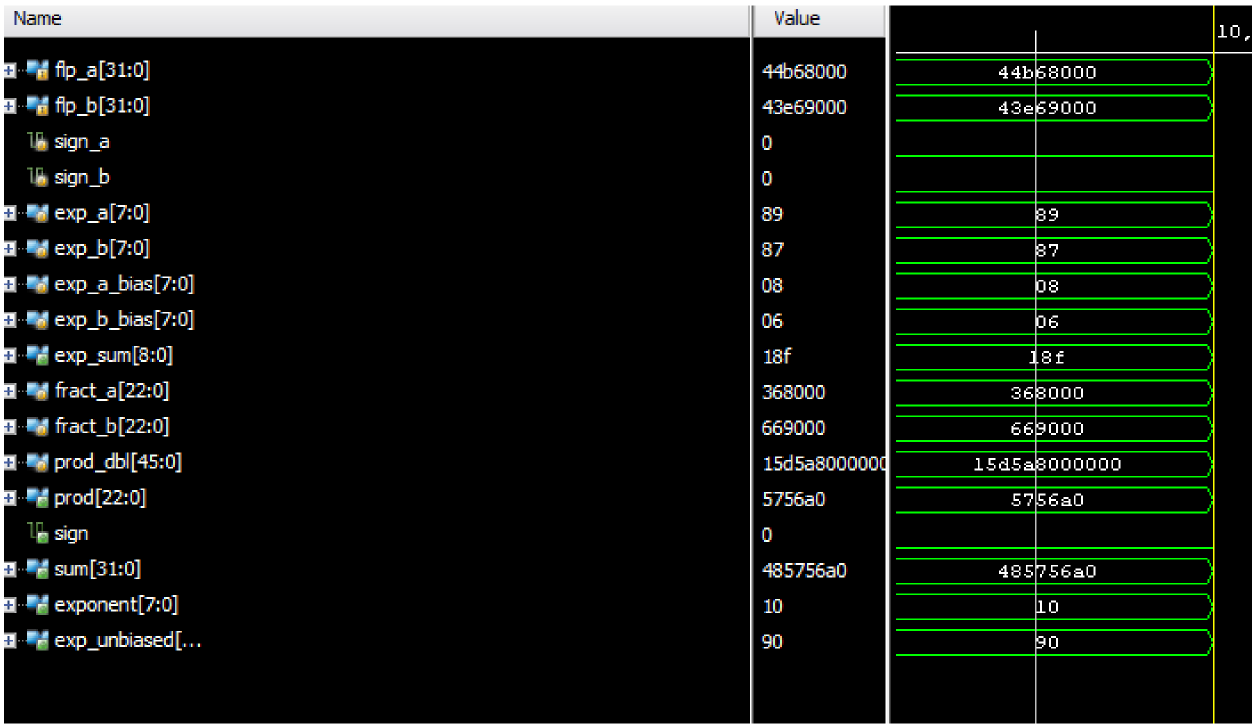
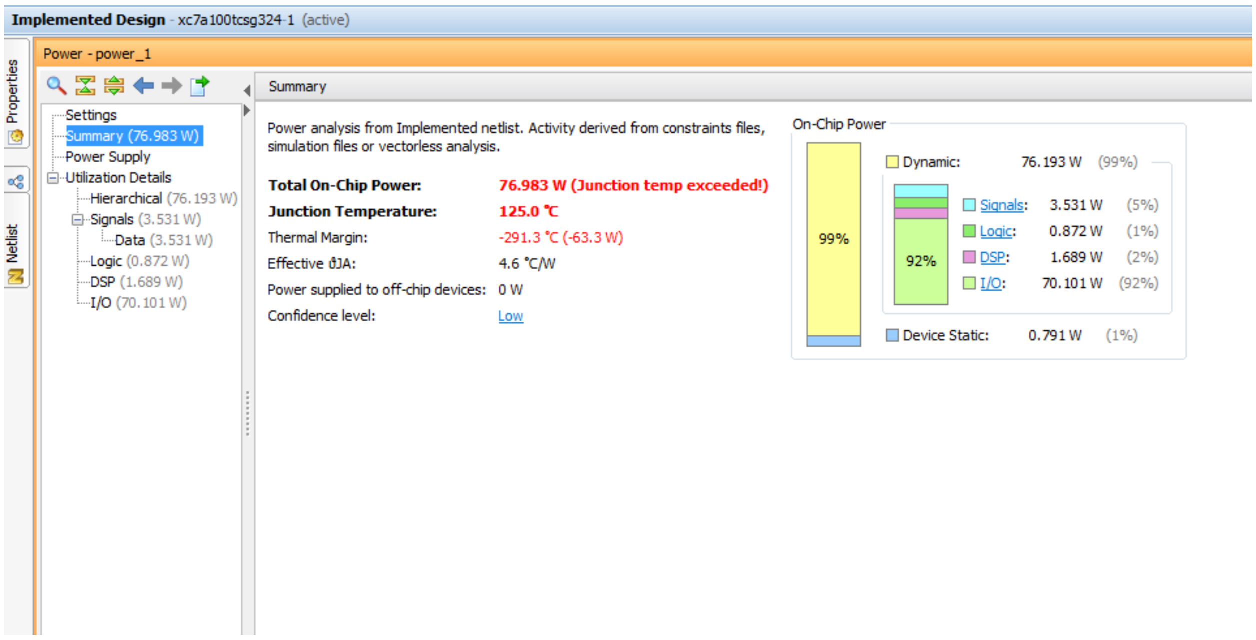

```
1. Slice Logic
--------------

+-------------------------+------+-------+-----------+-------+
|        Site Type        | Used | Fixed | Available | Util% |
+-------------------------+------+-------+-----------+-------+
| Slice LUTs*             |  193 |     0 |     63400 |  0.30 |
|   LUT as Logic          |  193 |     0 |     63400 |  0.30 |
|   LUT as Memory         |    0 |     0 |     19000 |  0.00 |
| Slice Registers         |    0 |     0 |    126800 |  0.00 |
|   Register as Flip Flop |    0 |     0 |    126800 |  0.00 |
|   Register as Latch     |    0 |     0 |    126800 |  0.00 |
| F7 Muxes                |    0 |     0 |     31700 |  0.00 |
| F8 Muxes                |    0 |     0 |     15850 |  0.00 |
+-------------------------+------+-------+-----------+-------+
```

Fig 5.6(c) Array Multiplier Synthesis Report



```
Name                Value
flp_a[31:0]         44b68000        44b68000
flp_b[31:0]         43e69000        43e69000
sign_a              0
sign_b              0
exp_a[7:0]          89              89
exp_b[7:0]          87              87
exp_a_bias[7:0]     08              08
exp_b_bias[7:0]     06              06
exp_sum[8:0]        18f             18f
fract_a[22:0]       368000          368000
fract_b[22:0]       669000          669000
prod_dbl[45:0]      15d5a8000000    15d5a8000000
prod[22:0]          5756a0          5756a0
sign                0
sum[31:0]           485756a0        485756a0
exponent[7:0]       10              10
exp_unbiased[...    90              90
```

Fig 5.7(a) FPM Simulation Report



Implemented Design - xc7a100tcsg324-1 (active)

Power - power_1

- Settings
- Summary (76.983 W)
- Power Supply
- Utilization Details
  - Hierarchical (76.193 W)
  - Signals (3.531 W)
    - Data (3.531 W)
  - Logic (0.872 W)
  - DSP (1.689 W)
  - I/O (70.101 W)

Summary

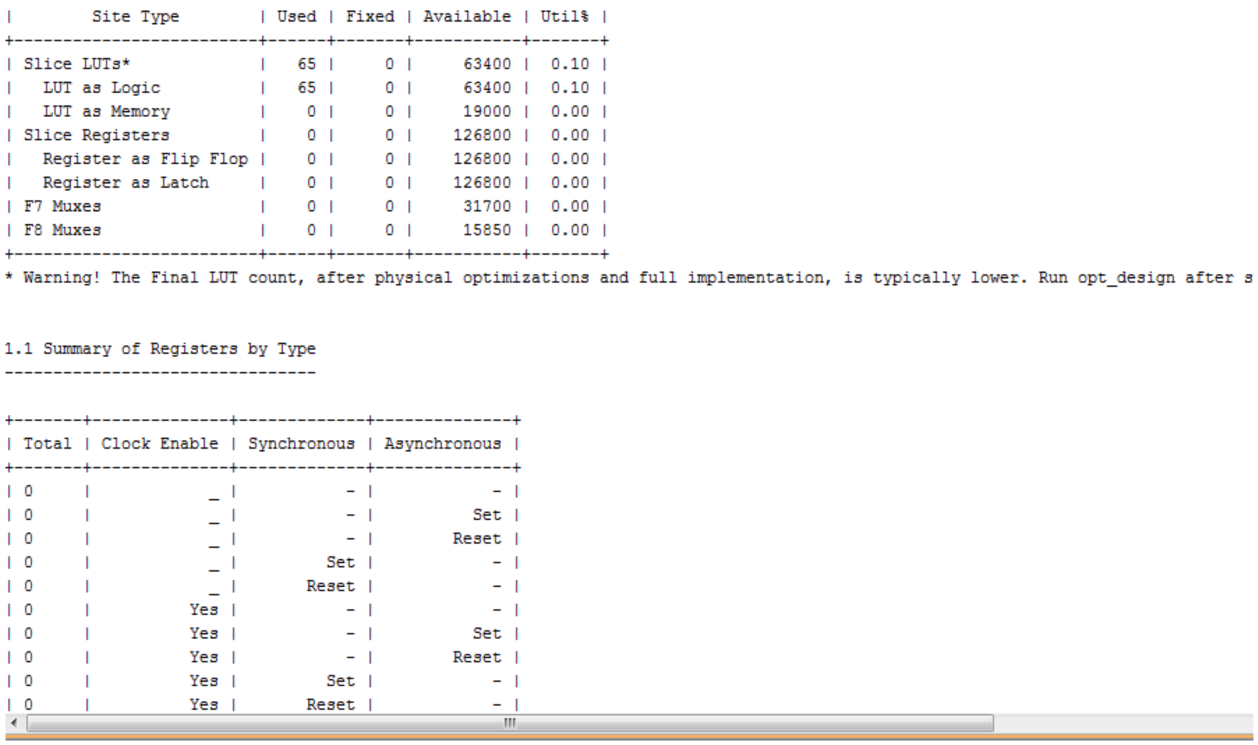
Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**76.983 W (Junction temp exceeded!)**
Junction Temperature:	**125.0 °C**
Thermal Margin:	-291.3 °C (-63.3 W)
Effective ϑJA:	4.6 °C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

On-Chip Power

Dynamic:	76.193 W	(99%)
Signals:	3.531 W	(5%)
Logic:	0.872 W	(1%)
DSP:	1.689 W	(2%)
I/O:	70.101 W	(92%)
Device Static:	0.791 W	(1%)

Fig 5.7(b) FPM Power Report



```
|        Site Type        | Used | Fixed | Available | Util% |
+-------------------------+------+-------+-----------+-------+
| Slice LUTs*             |   65 |     0 |     63400 |  0.10 |
|   LUT as Logic          |   65 |     0 |     63400 |  0.10 |
|   LUT as Memory         |    0 |     0 |     19000 |  0.00 |
| Slice Registers         |    0 |     0 |    126800 |  0.00 |
|   Register as Flip Flop |    0 |     0 |    126800 |  0.00 |
|   Register as Latch     |    0 |     0 |    126800 |  0.00 |
| F7 Muxes                |    0 |     0 |     31700 |  0.00 |
| F8 Muxes                |    0 |     0 |     15850 |  0.00 |
+-------------------------+------+-------+-----------+-------+
* Warning! The Final LUT count, after physical optimizations and full implementation, is typically lower. Run opt_design after s


1.1 Summary of Registers by Type
--------------------------------

+-------+--------------+-------------+--------------+
| Total | Clock Enable | Synchronous | Asynchronous |
+-------+--------------+-------------+--------------+
| 0     |            _ |           - |            - |
| 0     |            _ |           - |          Set |
| 0     |            _ |           - |        Reset |
| 0     |            _ |         Set |            - |
| 0     |            _ |       Reset |            - |
| 0     |          Yes |           - |            - |
| 0     |          Yes |           - |          Set |
| 0     |          Yes |           - |        Reset |
| 0     |          Yes |         Set |            - |
| 0     |          Yes |       Reset |            - |
```

Fig 5.7(c) FPM Synthesis Report

IMPLEMENTATION RESULTS

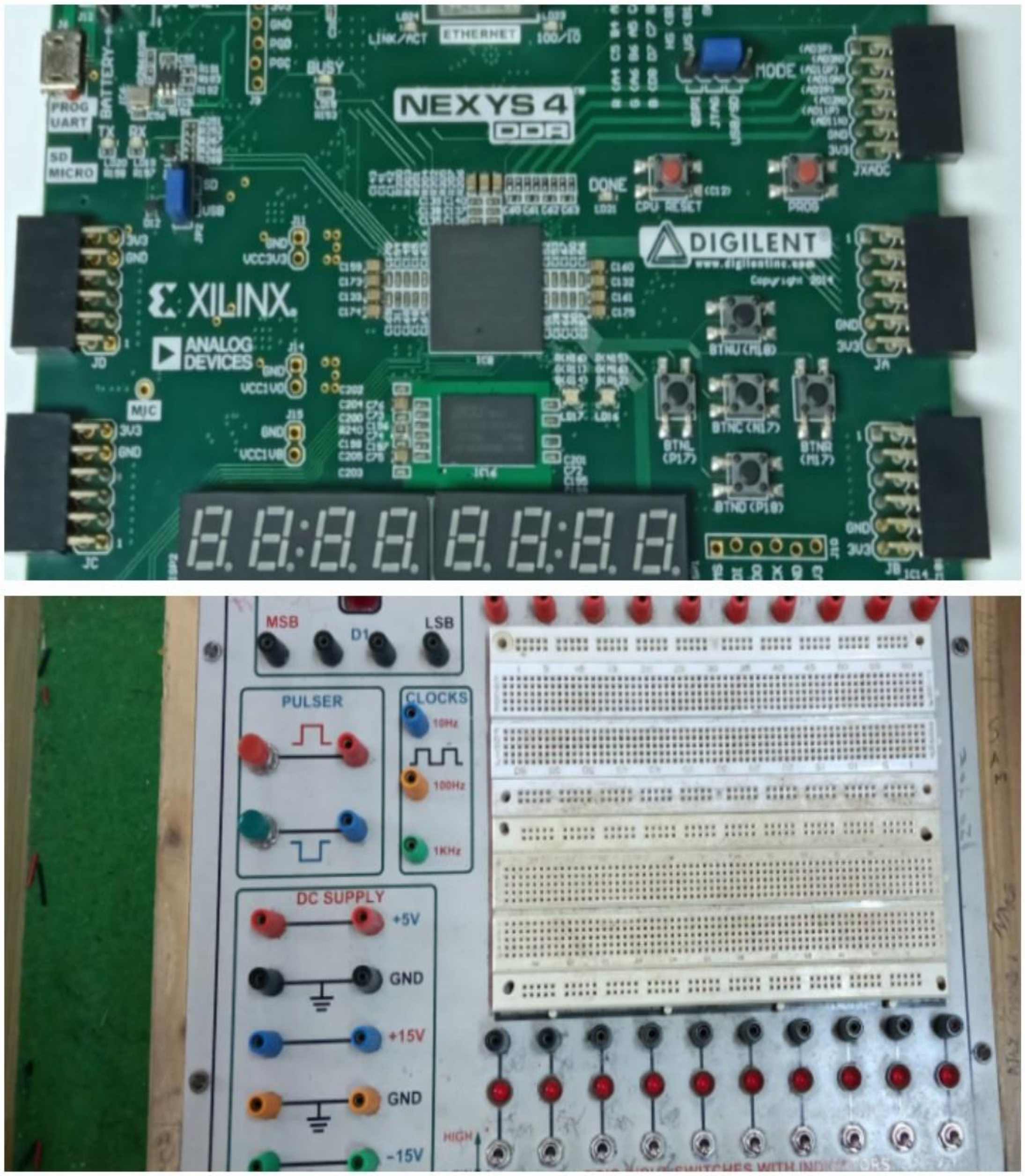


Fig 5.8 FPGA Trainer Kit and DAC Trainer Kit

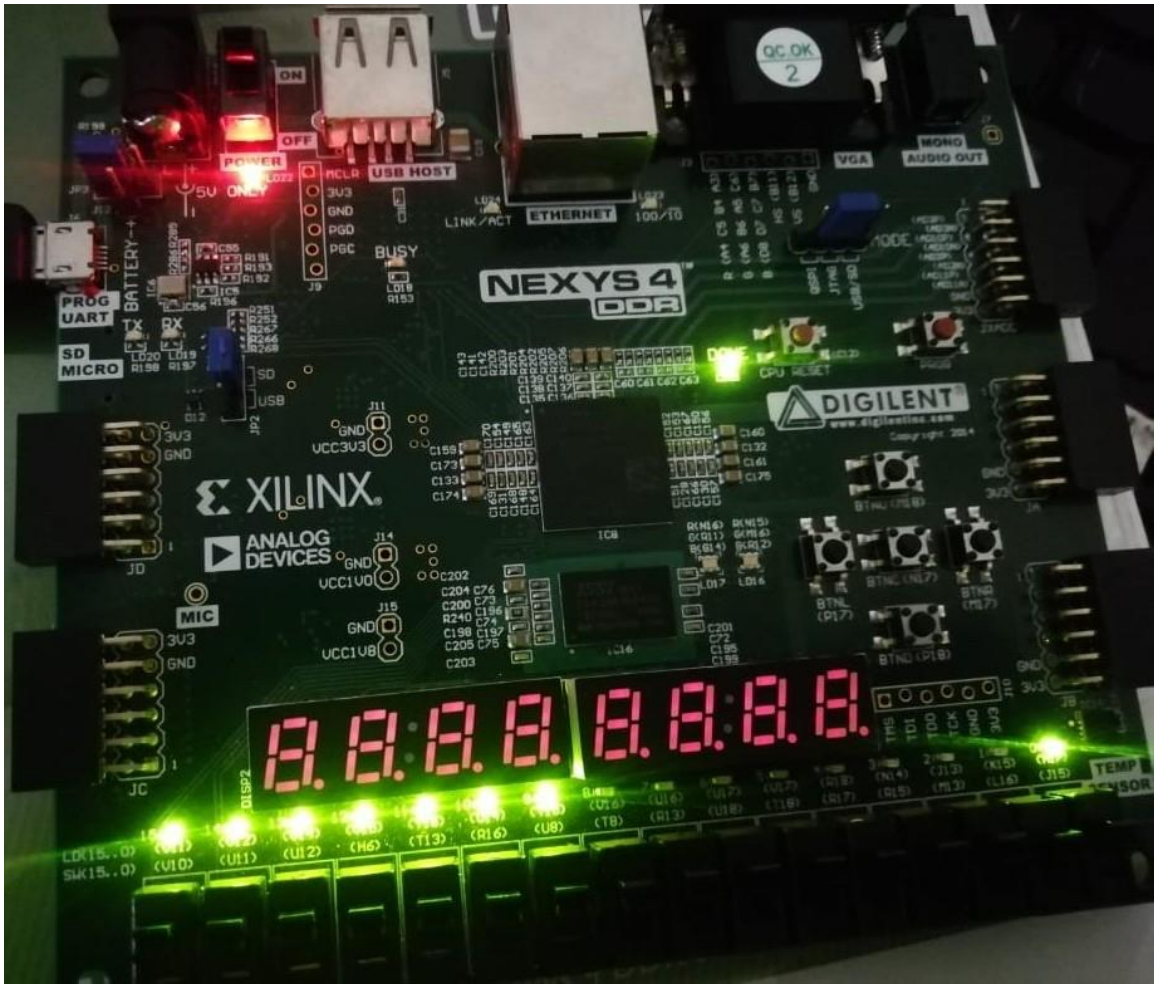


Fig 5.9 Implementation of 4 Bit Multiplication

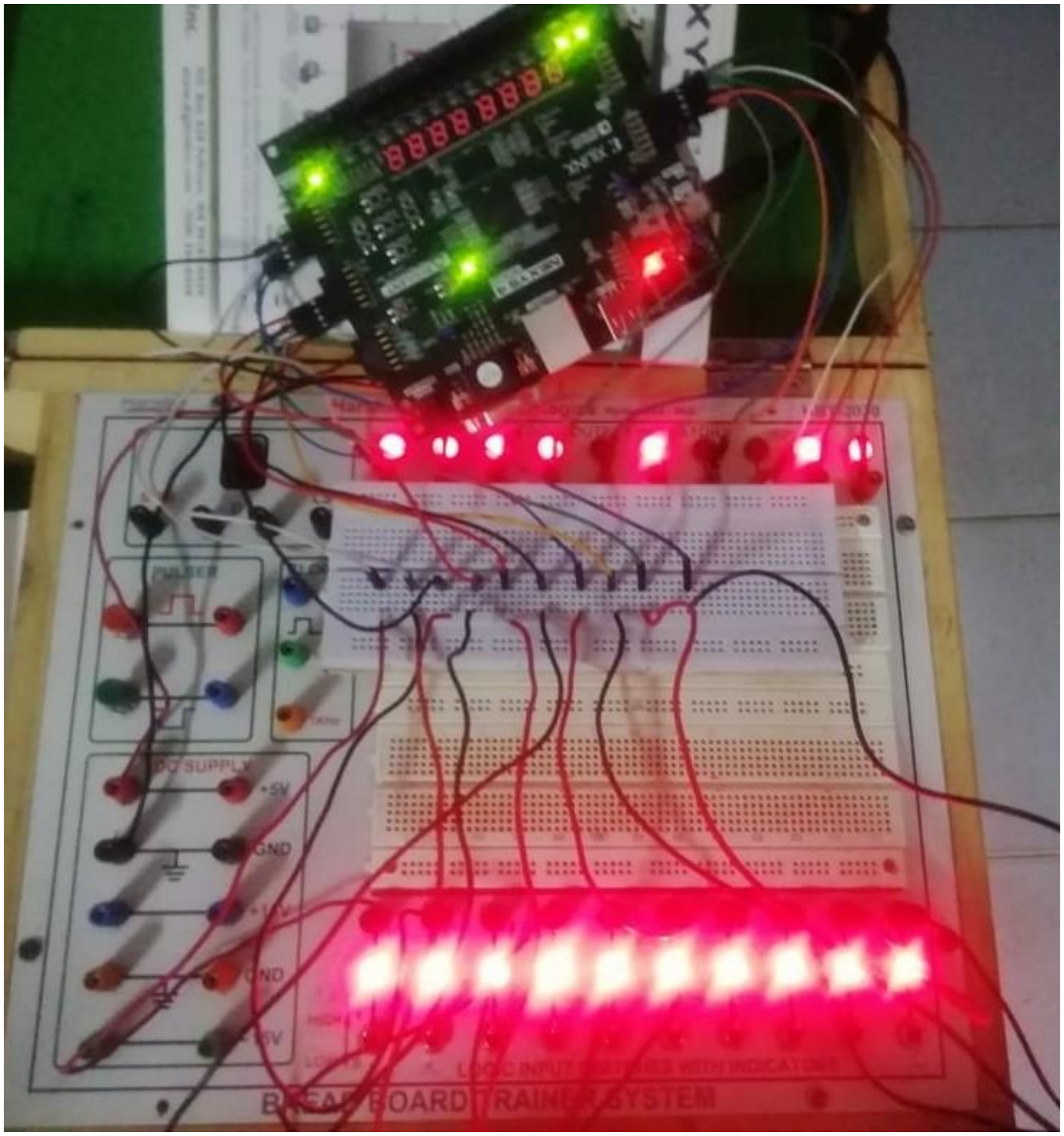

BREAD BOARD TRAINER SYSTEM
DC SUPPLY
PULSER
+5V
GND
LOGIC INPUT SWITCHES WITH INDICATORS

FIG 5.10 Implementation of 13-Bit Multiplication

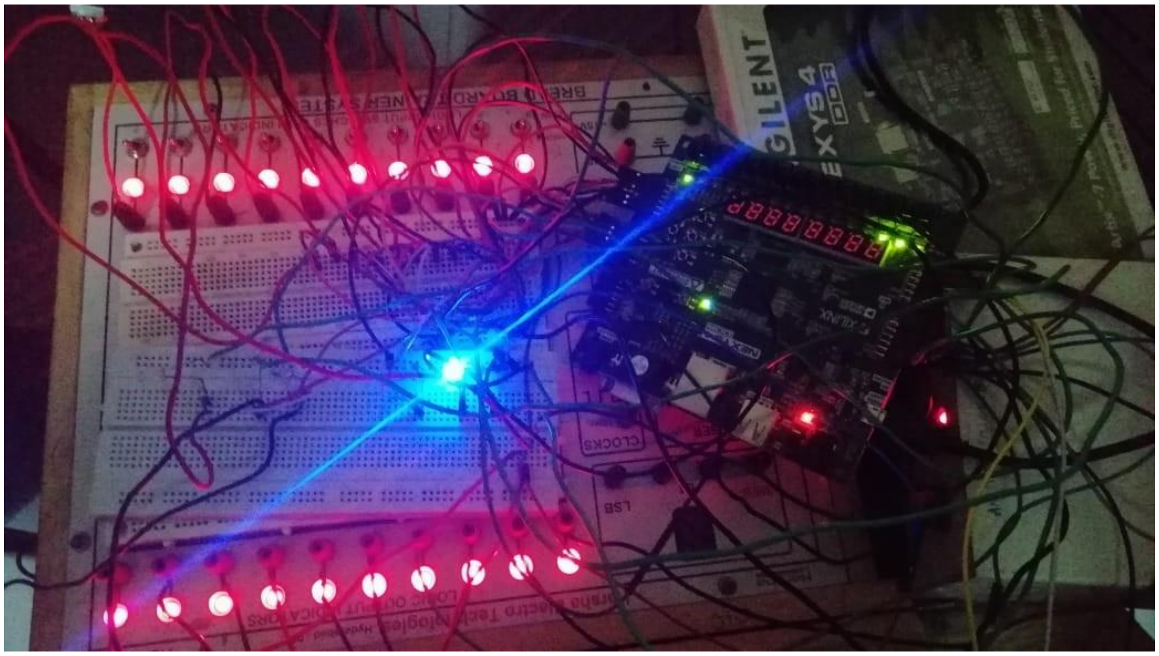


Fig 5.11 Implementation of 16-Bit Multiplication

CONCLUSION

In digital signal processing multiplication is a key operation which determines the overall performance of the multiplier. Using Floating point representation in multiplication makes the operation accurate than using normal multiplication. In this project we have performed floating point multiplication and analysed the power and synthesis results. We have implemented various multipliers like Vedic multiplier, array multiplier etc and compared their performance characteristics.

References

1. Jaiswal M. K., Hayden K.H.: “DSP48E Efficient Floating Point Multiplier Architectures on FPGA”, international conference on VLSI Design and Embedded Systems. pp.455-460,2017
2. Yogita B., Madhu Ch.: “A novel high-speed approach for 16x16 Vedic multiplication with Compressor adders”, J. Computers and Electrical Engineering. vol. 49, pp.39-49, 2016
3. Jaiswal M. K., Cheung R.C.C.: “VLSI Implementation of double-precision floating point multiplier using Karatsuba Technique”, J. Circuits, Systems, and Signal Processing, vol. 32, pp. 15- 27,2013
4. Aliparast P., 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”, J. Analog Integrated Circuits and Signal Processing, vol. 66, pp. 235-243,2011
5. Shiann-Rong K.: “Variable Latency Floating Point Multipliers for Low-PowerApplications”, IEEE transactions on very large scale integration (VLSI) systems, vol. 18, 2010.
6. Chang C.H., et.al.:“ Ultra low-voltage Low-power CMOS 4-2 and 5-2 Compressors for Fast Arithmetic Circuits,” Circuits and Systems I:Regular Papers, IEEE Transactions on vol.51,pp.1985- 97,2004.
7. Swapna E.: “A Spurious Power Suppression technique for a low power multiplier”, International Journal of Engineering Research Technology(IETE), vol. 2, pp.1-5,2013
8. Ramalatha M., Deena Dayalan K., Dharani K., Deborah Priya S.: “High Speed Energy Efficient ALU Design using vedic Multiplication Techniques, ACTEA200