IMPLEMENTATION OF AREA EFFICIENT 8-BIT MULTIPLIER USING HIGHER ORDER COMPRESSORS

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

A. HARI SIVA GANESH (318126512003) **B. BHANU PRAKASH (318126512007)**

B. SANTHOSH KUMAR (318126512006) **D. DURGA SANDEEP (318126512017)**

Under the guidance of

Dr.K.V.G.Srinivas

Assistant Professor

PRAGNANAM BRAHMA

ANITS

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)

(2021-2022)

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)

PRAGNANAM BRAHMA

ANITS

CERTIFICATE

This is to certify that the project report entitled **"IMPLEMENTATION OF 8-BIT AREA EFFICIENT MULTIPLIER USING HIGHER ORDER COMPRESSORS"** submitted by **A. Hari Siva Ganesh (318126512003), B. Santhosh Kumar (318126512006), B. Bhanu Prakash (318126512007), D. Durga Sandeep (318126512017)** in partial fulfilment 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.

[illegible]

Project Guide

Dr. K. V. G.Srinivas

Assistant Professor

Department of E.C.E

ANITS

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

[illegible]

Head of the Department

Dr. V. Rajalakshmi

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

A. HARI SIVA GANESH (318126512003),

B. SANTHOSH KUMAR (318126512006),

B. BHANU PRAKASH (318126512007),

D. DURGA SANDEEP (318126512017).

ABSTRACT

The accumulation of partial products is used to multiply binary bits. Full adders and half adders are commonly used for this. Various algorithms are utilized, including the Booth method, Ripple Carry, and Carry Save. The multiplier's performance in terms of latency, power, and area is determined by the number of stages used to sum the partial products. A higher order compressor-based area efficient 8x8 multiplier is proposed in this paper. To collect fragmentary products, we used the Wallace Tree structure. Carry save form is used in the structure. The carries generated in the adders are then passed on to the next stage, and so on, until the final stage. The final product is created using ripple carry form in the final stage. The Wallace Tree was chosen because it has a shorter critical path, resulting in reduced delay. Here we used two different methods to perform 8-bit multiplication and analysed both of them regarding area, power and some other parameters.

CONTENTS

ABSTRACT

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVATIONS

CHAPTER 1 INTRODUCTION **1**

1.1 Project Objective **1**

1.2 Project Outline **1**

CHAPTER 2 MULTIPLIERS **2**

2.1 Introduction to Multipliers **3**

2.2 Types of Multipliers **4**

CHAPTER 3 INTRODUCTION TO COMPRESSOR ADDERS **9**

3.1 What are compressor adders **9**

3.2 Types of compressor adders **9**

CHAPTER 4 OVERVIEW OF FPGA AND EDA SOFTWARE **14**

4.1 Introduction **14**

4.2 Overview of digilent basys-3 board **14**

4.2.1 Features of basys-3 **15**

4.2.2 Power supply **16**

4.2.3 FPGA configuration **18**

4.2.4 Memory **22**

4.2.5 Oscillators/Clocks **22**

4.2.6 USB-UART Bridge(serial port) **23**

4.2.7 USB-HID Host **23**

4.2.8 VGA Port **28**

4.2.9 Basic I/O 32

4.2.10 Pmod Ports 35

4.2.11 Built in Self-Test 38

4.3 Development flow 38

4.4 Overview of Verilog 40

4.5 Overview of EDA 46

CHAPTER 5 SIMULATION RESULTS 59

CONCLUSIONS 66

REFERENCES 67

LIST OF FIGURES

Fig 2.1 Serial /parallel Multiplier **3**

Fig 2.2 Shift and add multiplier **4**

Fig 2.3 Array multiplier **5**

Fig 2.4 Booth Multiplier **6**

Fig 2.5 Sequential multiplier **7**

Fig 2.6 Wallace tree multiplier **8**

Fig 3.2.1 circuit diagram of 4-2 compressor adder **10**

Fig 3.2.2 circuit diagram of 7-2 compressor adder **11**

Fig 3.2.3(a) circuit diagram of 5-3 compressor adder using gates **12**

Fig 3.2.3(b) modified circuit diagram of 5-3 compressor adder **12**

Fig 3.2.4 circuit diagram of 6-3 compressor adder **13**

Fig 3.2.5 circuit diagram of 7-3 compressor adder **13**

Fig 1 Basys-3 FPGA board with callouts **16**

Fig 2 Basys-3 Power circuit **17**

Fig 3 Basys-3 configuration options **19**

Fig 4 Basys-3 external memory **22**

Fig 6 Basys-3 FT2232HQ connections **23**

Fig 7 Basys-3 PK24 connections **24**

Fig 8 Ps/2 device to host timing diagram **24**

Fig 9 Keyboard scan codes **25**

Fig 10 Mouse data format **27**

Fig 11 VGA Port **28**

Fig 12 Colour CRT display **29**

Fig 13 VGA Horizontal synchronization **31**

Fig 14 Signal timings for a 640-pixel by 480 row display using a 25 MHz pixel clock and 60 Hz vertical refresh **32**

Fig 15 VGA display controller block diagram. **32**

Fig 16 General purpose I/O devices on the Basys-3 **33**

Fig 17 An un-illuminated seven-segment display and nine illumination patterns corresponding to decimal digits. **34**

Fig 18 Common anode circuit node **34**

Fig 19 Four digit scanning display controller timing diagram **35**

Fig 20 Pmod ports; front view as loaded on PCB **36**

Fig 21. Development flow **39**

Fig 22. CMOS inverter **43**

Fig 23. RTL diagram **59**

Fig 24. Synthesized schematic diagram **60**

Fig 25. Design synthesize status **60**

Fig 26**.** LUT table **60**

Fig 27. Logic and memory table **61**

Fig 28. Power report **61**

Fig 29. RTL diagram **62**

Fig 30**.** Synthesized schematic diagram **62**

Fig 31. Design synthesize status **63**

Fig 32. LUT table **63**

Fig 33. Power report **63**

Fig 34. Simulation output results **64**

LIST OF TABLES

Table 1. Basys3 components **16**

Table 2. Basys3 power supply **18**

Table 3. Keyboard commands **26**

Table 4. Microsoft Intellimouse-type extensions, commands, and actions. **27**

Table 5. Basys-3 Pmod pin assignment **36**

LIST OF ABBREVATIONS

DSP	Digital Signal Processing
FPM	Floating Point Multiplication
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
DDR	Double Data Rate
LUT	Look Up Tables
PLL	Phase-Locked Loop
HR	High Range
ODT	On-Die Terminations
PHY	Physical Layer
RTL	Register Transfer Level
EDA	Electronic Design Automation

CHAPTER 1

INTRODUCTION

Multiplication is a fundamental function in arithmetic operations, and operations like multiply and Accumulate (MAC) and inner product are among the most commonly used Computation Intensive Arithmetic Functions (CIAF) in DSP applications like convolution, Fast Fourier Transform (FFT), filtering, and in microprocessors' arithmetic and logic units. Because multiplication takes up the majority of the execution time in most DSP algorithms, a high-speed multiplier is required. Multiplication time is still the most important component in determining a DSP chip's instruction cycle time. As the number of computer and signal processing applications grows, so does the demand for high-speed processing. In many real-time signal and image processing applications, higher throughput arithmetic operations are required to attain the desired performance. Multiplication is one of the most important mathematical operations in such applications, and the creation of fast multiplier circuits has piqued attention for decades. For many applications, reducing the time delay and power consumption are critical criteria. Different multiplier architectures are presented in this paper.

1.1 PROJECT OBJECTIVE:

The main objective of this project is to design and implementation of area efficient multiplier with higher order compressors using Xilinx Vivado Software. The Synthesis and Implementation is done for different types of compressor adders using Xilinx Vivado. The performance is compared in terms of power, area and delay.

1.2 PROJECT OUTLINE:

- Project report is presented over the 6 remaining chapters.
- Chapter 2 Introduction to multipliers.
- Chapter 3 Introduction to compressor adders
- Chapter 4 Overview of fpga and eda software
- Chapter 5 Methodology
- Chapter 6 Results and conclusions

CHAPTER 2
INTRODUCTION TO MULTIPLIERS

2.1 MULTIPLIERS

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

The common multiplication method is “add and shift” algorithm. In parallel multipliers number of partial products to be added is the main parameter that determines the performance of the multiplier. To reduce the number of partial products to be added, Modified Booth algorithm is one of themost 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 seeadvantage of both algorithms in one multiplier.

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

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

2.2 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

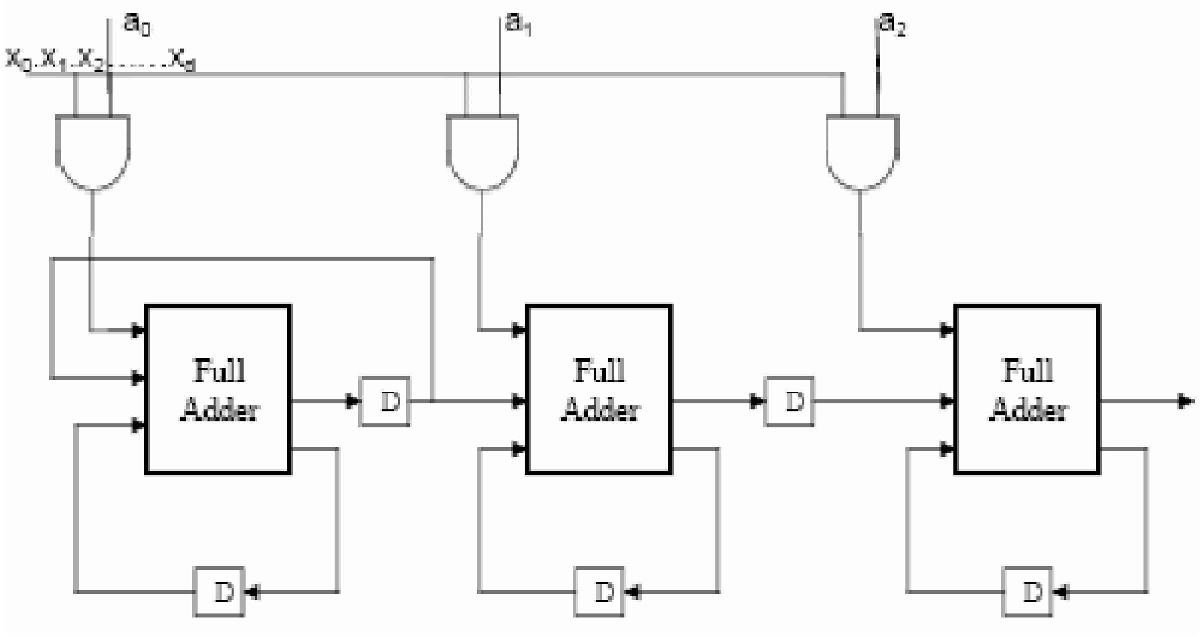


Fig 2.1 Serial /parallel Multiplier

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

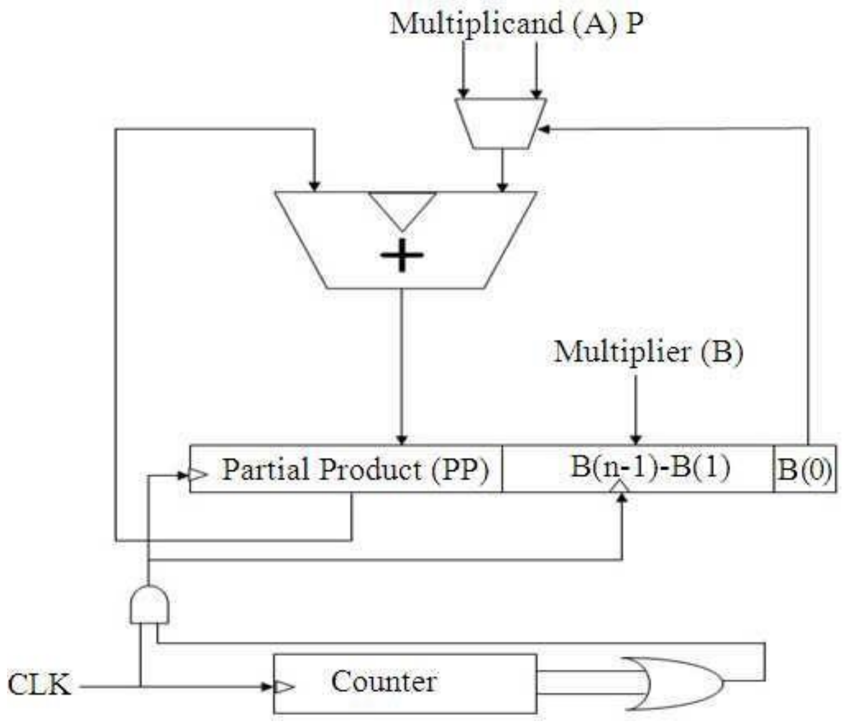


Fig 2.2 Shift and add multiplier

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 carrypropagate adder. N-1 adders are required where N is the multiplier length.

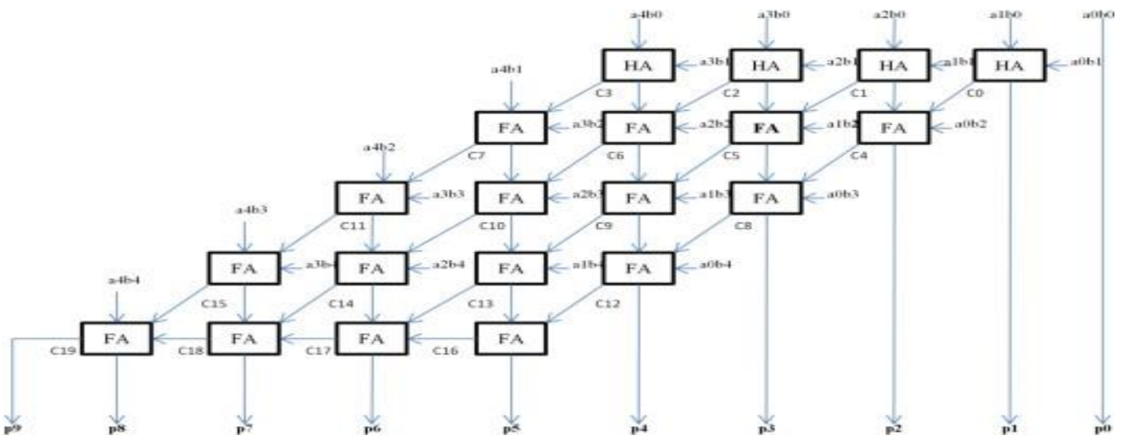


Fig 2.3 Array multiplier

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 casethe 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 rangeof numbers to be represented, a higher representation radix leads to fewer digits. Since a k-bitbinary number can be interpreted as K/2-digit radix-4 number, a K/3-digit radix-8 number, andso on, it can deal with more than one bit of the multiplier in each cycle by using high radix multiplication.

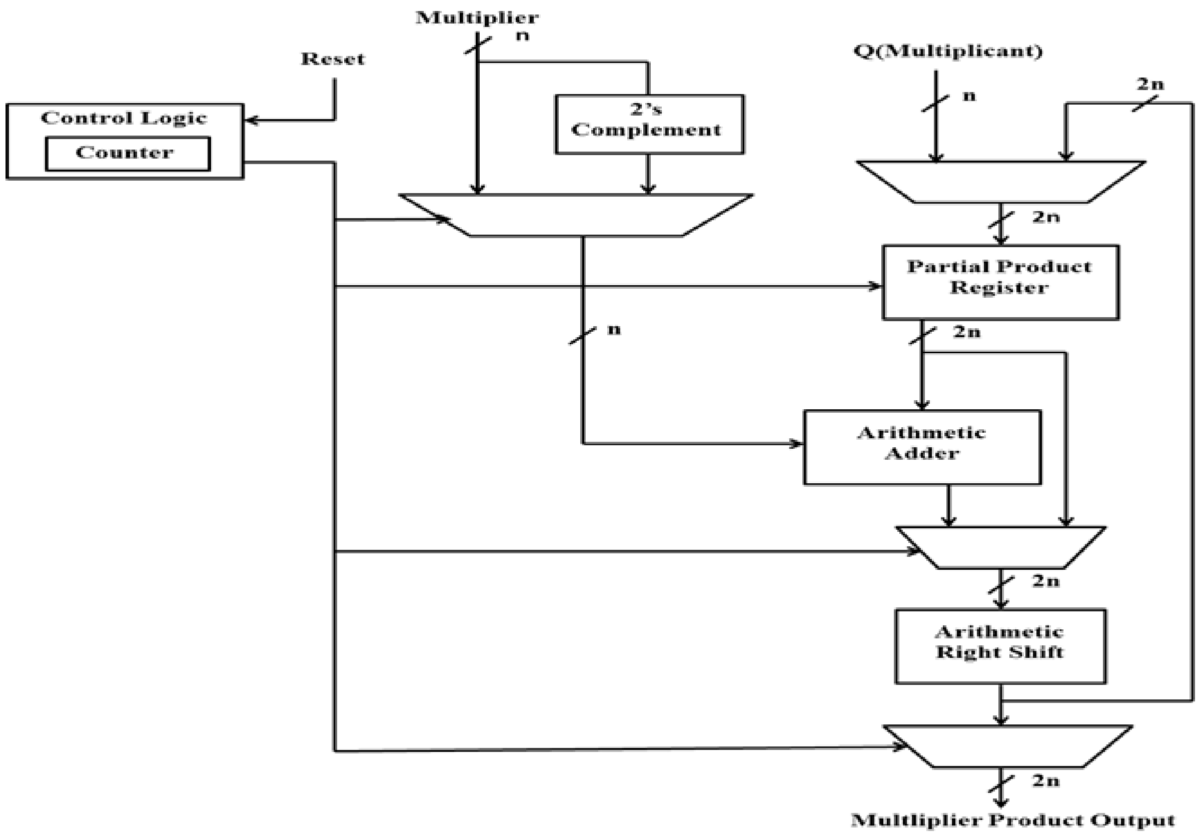


Fig 2.4 Booth Multiplier

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

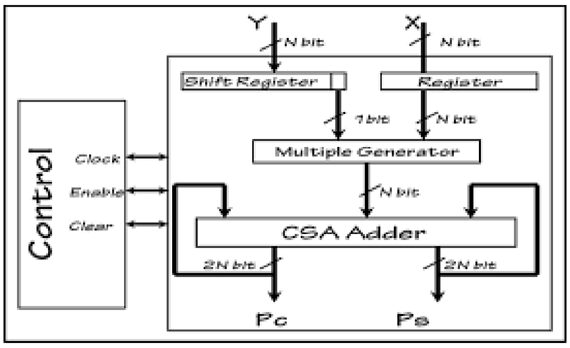


Fig 2.5 Sequential multiplier

Wallace tree Multiplier: 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 layerhas O (1) propagation delay. As making the partial products is O (1) and the final addition is O (log n), the multiplication is only, not much slower than addition (however, much more expensive in the gate count). Naively adding partial products with regular adders would require O ($\log^2$ n) time.

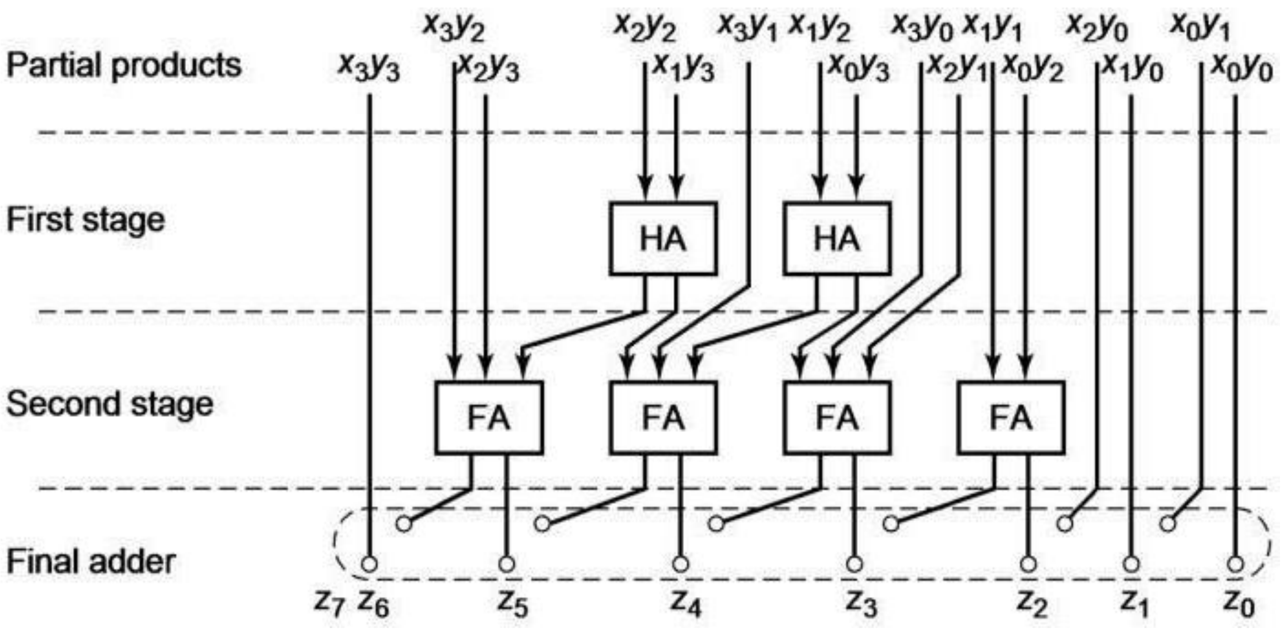


Fig 2.6 Wallace tree multiplier

CHAPTER 3
INTRODUCTION TO COMPRESSOR ADDERS

3.1 COMPRESSOR ADDERS

Compressor adders are basic circuits which add bits more than four at a time to give better delay results over the combinational circuits of half and full adders. The symbolic representation of compressor architecture is N − r where 'N' represents the number of the bits that are fed and 'r' represents the total count of the 1s present in N bits. It actually reduces the gate counts and delay in comparison to adder circuits and that is why named as compressor. A large part of research has been done in improving the circuits of lower compressors. Along with this, higher compressors are also implemented to add higher number of bits. The main compressor architectures which are used widely are 4-2,7-2, 5-3, 10-4, 15-4 and also 4-3,6-3,7-3,8-4.

3.2 TYPES OF COMPRESSOR ADDERS

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

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

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

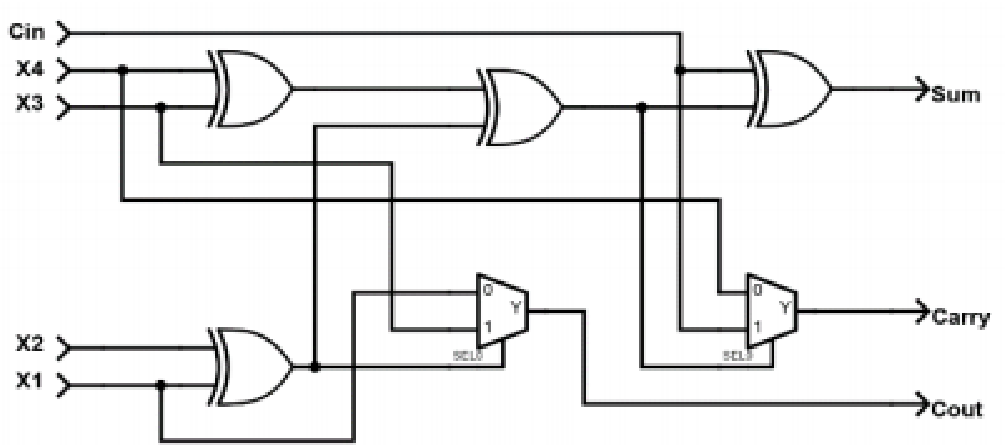


Fig 3.2.1 circuit diagram of 4-2 compressor adder

5-3 COMPRESSOR ADDER: A 5-3 compressor adder is a logical circuit in which maximum five bits can be added at the same time and three bits resultant of maximum value 101 is obtained. The circuit uses three 4:1 multiplexer. This multiplexer allows only one output to be high at a time and this property makes the multiplier fast and low power consuming circuit [13–15]. The circuit is reorganized in such a way that only 3 XOR operations are used instead of 5 XOR operations (in case of conventional 5-3 compressor) and other two inputs (X3 and X4) acts as a control signal. The conventional and the modified 5-3 compressor adder circuit are shown in

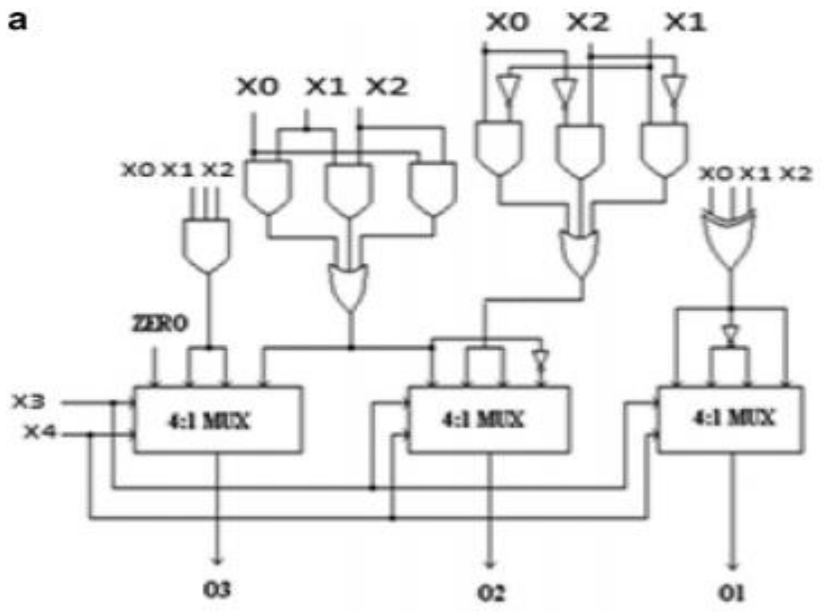


Fig 3.2.3(a) circuit diagram of 5-3 compressor adder using gates

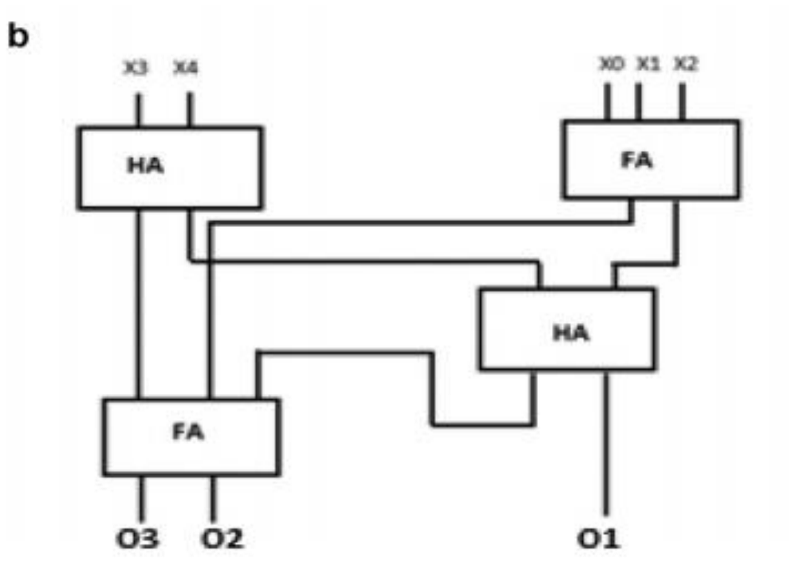


Fig 3.2.3(b) modified circuit diagram of 5-3 compressor adder

6-3 COMPRESSOR ADDER:

A combinational logic circuit of 6:3 compressor accepts six inputs and produces three outputs. The six input bits are summed up to produce the three bit output. In this three full adders and one half adder is used. The six input bits are given to two full adders which results in four outputs. These four output bits are fed to full adder and half adder as shown in the architecture and produces three outputs which are the final output bits of the result. The maximum output for a 6:3 compressor is 110.

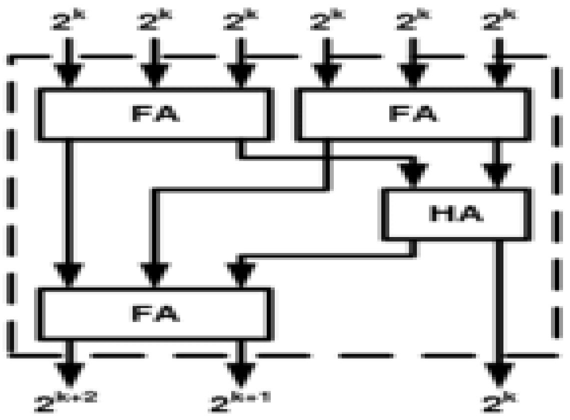


Fig 3.2.4 circuit diagram of 6-3 compressor adder

7-3 COMPRESSOR ADDER:

Combinational logic circuit of 7:3 compressor accepts seven inputs and produces three outputs. The seven input bits are summed up to produce the three bit cc output. In this, four full adders are used. The seven input bits are given to three full adders as shown in the figure. The first six bits are given to two full adders and the last bit is fed to another full adder. Finally the output bits are fetched as shown.

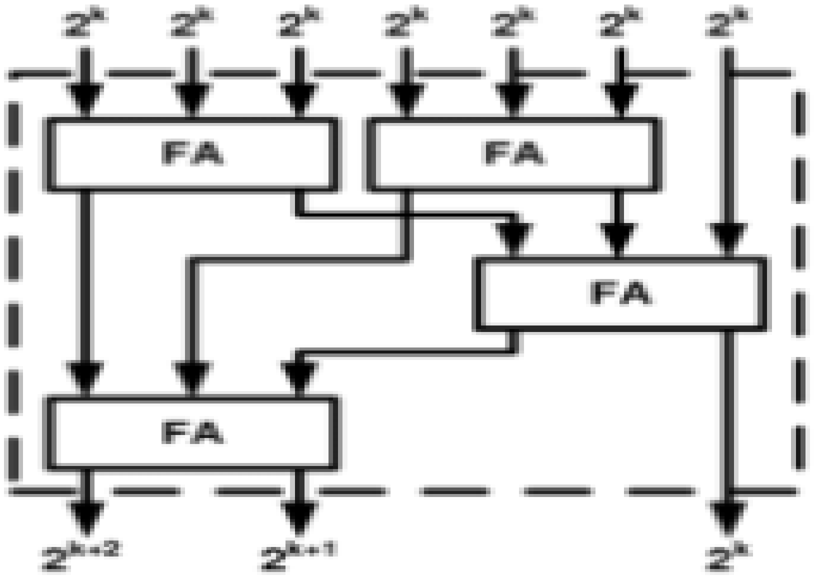


Fig 3.2.5 circuit diagram of 7-3 compressor adder

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 are completed, 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 BASYS3 BOARD

The Basys-3 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 high-capacity FPGA (Xilinx part number XC7A35T- 1CPG236C), low overall cost, and collection of USB, VGA, and other ports, the Basys-3 can host designs ranging from introductory combinational circuits to complex sequential circuits like embedded processors and controllers. It includes enough switches, LEDs, and other I/O devices to allow a large number of designs to be completed without the need for any additional hardware, and enough uncommitted FPGA I/O pins to allow designs to be expanded using Digilent Pmods or other custom boards and circuits.

4.2.1 Features of Basys-3

- 33,280 logic cells in 5200 slices (each slice contains four 6-input LUTs and 8 flip-flops)
- 1,800 Kbits of fast block RAM
- Five clock management tiles, each with a phase-locked loop(PLL)
- 90 DSP slices
- Internal clock speeds exceeding 450MHz
- On-chip analog-to-digital converter(XADC)
- 16 user switches
- 16 user LEDs
- 5 user push buttons
- 4-digit7-segmentdisplay
- Three Pmod ports
- Pmod for XADC signals
- 12-bit VGA output
- USB-UART Bridge
- Serial Flash
- Digilent USB-JTAG port for FPGA programming and communication
- USB HID Host for mice, keyboards and memory sticks

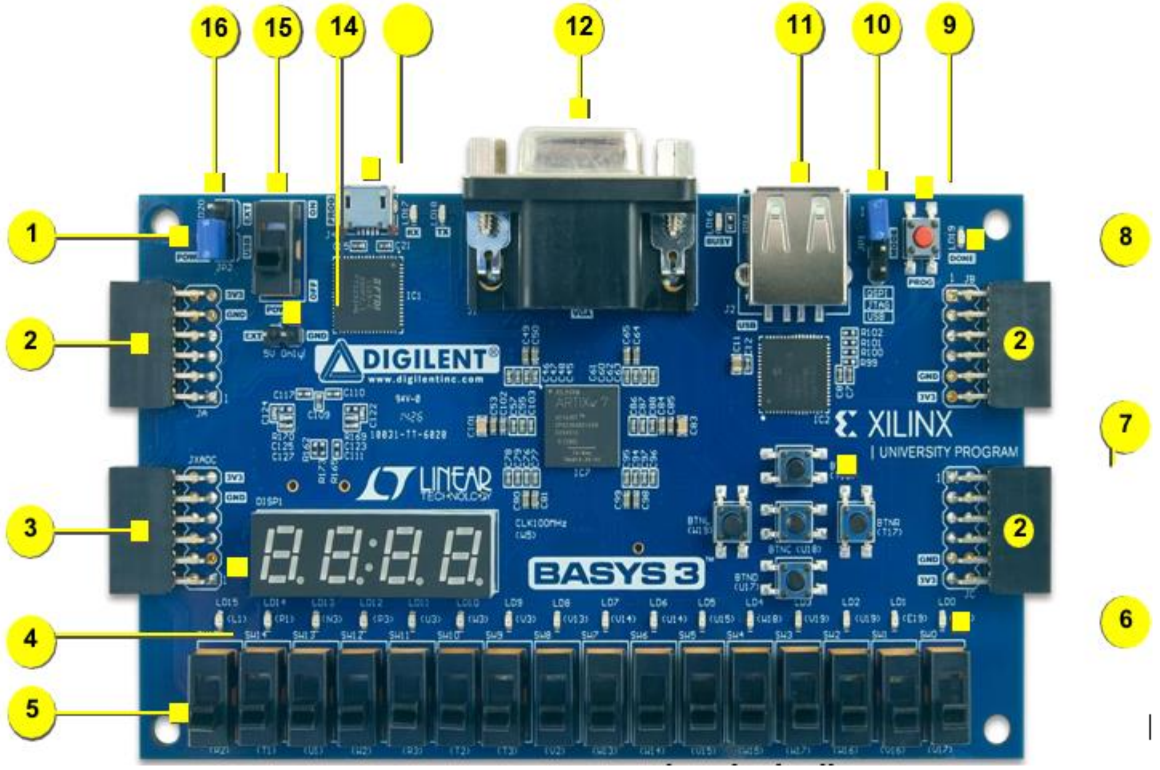


Fig1. Basys-3 FPGA board with callouts.

Table 1. Basys-3 components

Callout	Component Description	Callout	Component Description
1	Power good LED	9	FPGA configuration reset button
2	Pmod port(s)	10	Programming mode jumper
3	Analog signal Pmod port (XADC)	11	USB host connector
4	Four digit 7-segment display	12	VGA connector
5	Slide switches (16)	13	Shared UART/ JTAG USB port
6	LEDs (16)	14	External power connector
7	Pushbuttons (5)	15	Power Switch
8	FPGA programming done LED	16	Power Select Jumper

4.2.2. Power Supplies

The Basys-3 board can receive power from the Digilent USB-JTAG port (J4) or from a 5V external power supply.Jumper JP3 (near the power switch) determines which source is used.

All Basys-3 power supplies can be turned on and off by a single logic-level power switch (SW16). A power-good LED(LD20), driven by the "power good" output of the LTC3633 supply, indicates that the supplies are turned on and operating normally. An overview of the Basys-3 power circuit is shown in Fig. 2.

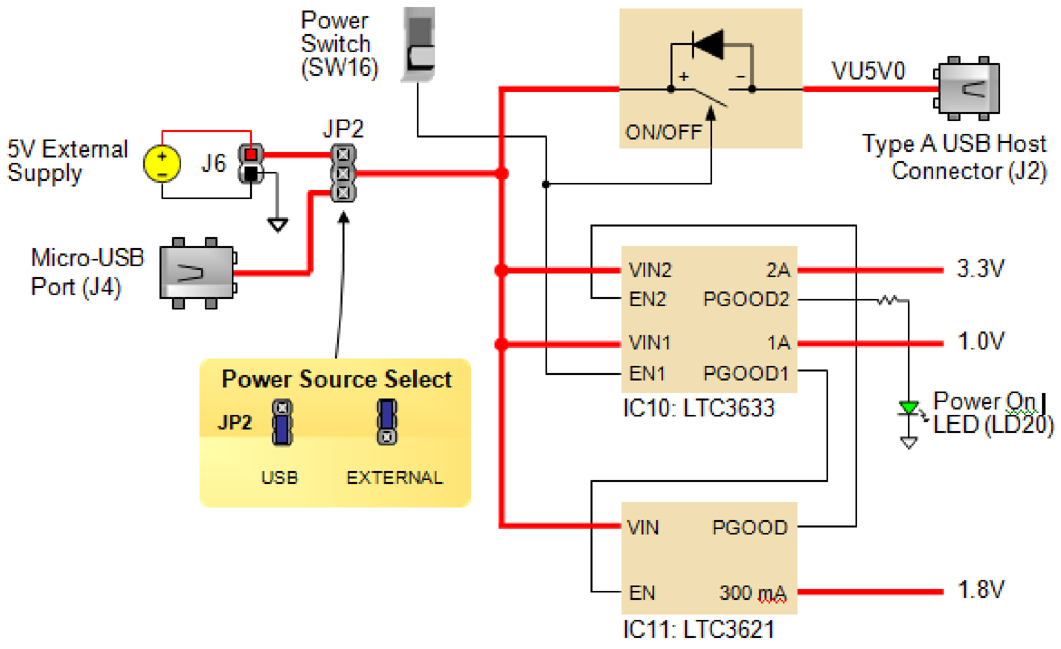


Fig2. Basys-3 power circuit.

The USB port can deliver enough power for the vast majority of designs. A few demanding applications, including any that drive multiple peripheral boards, might require more power than the USB port can provide. Also, some applications may need to run without being connected to a PC's USB port. In these instances an external power supply or battery pack can be used.

An external power supply can be used by plugging into the external power header (J6) and setting jumper JP2 to"EXT". The supply must deliver 4.5VDC to 5.5VDC and at least 1A of current (i.e., at least 5W of power). Many suitable supplies can be purchased through Digi-Key or other catalog vendors.

An external battery pack can be used by connecting the battery's positive terminal to the "EXT" pin of J6 and the negative terminal to the "GND" pin of J6. The power provided to USB devices that are connected to Host connectorJ2 is not regulated. Therefore, it is necessary to limit the maximum voltage of an external battery pack to 5.5V DC. The minimum voltage of the battery pack depends on the application; if the USB Host function (J2) is used, at least 4.6V needs to be provided. In other cases, the minimum voltage is 3.6V.

Voltage regulator circuits from Linear Technology create the required 3.3V, 1.8V, and 1.0V supplies from the mainpower input. Table 2 provides additional information (typical currents depend strongly on FPGA configuration andthe values provided are typical of medium size/speed designs).

Table 2. Basys-3 power supplies.

Supply	Circuits	Device	Current (max/typical)
3.3V	FPGA I/O, USB ports, Clocks, Flash, PMODs	IC10: LTC3633	2A/0.1 to 1.5A
1.0V	FPGA Core	IC10: LTC3633	2A/ 0.2 to 1.3A
1.8V	FPGA Auxiliary and Ram	IC11: LTC3621	300mA/ 0.05 to 0.15A

4.2.3 FPGA Configuration

After power-on, the Artix-7 FPGA must be configured (or programmed) before it can perform any functions. Youcan configure the FPGA in one of three ways:

A PC can use the Digilent USB-JTAG circuitry (portJ4, labeled "PROG") to program the FPGA any time thepower is on.

A file stored in the nonvolatile serial (SPI) flash device can be transferred to the FPGA using the SPI port.

A programming file can be transferred from a USB memory stick attached to the USB HID port.

Figure 3 shows the different options available for configuring the FPGA. An on-board "mode" jumper (JP1) selectsbetween the programming modes.

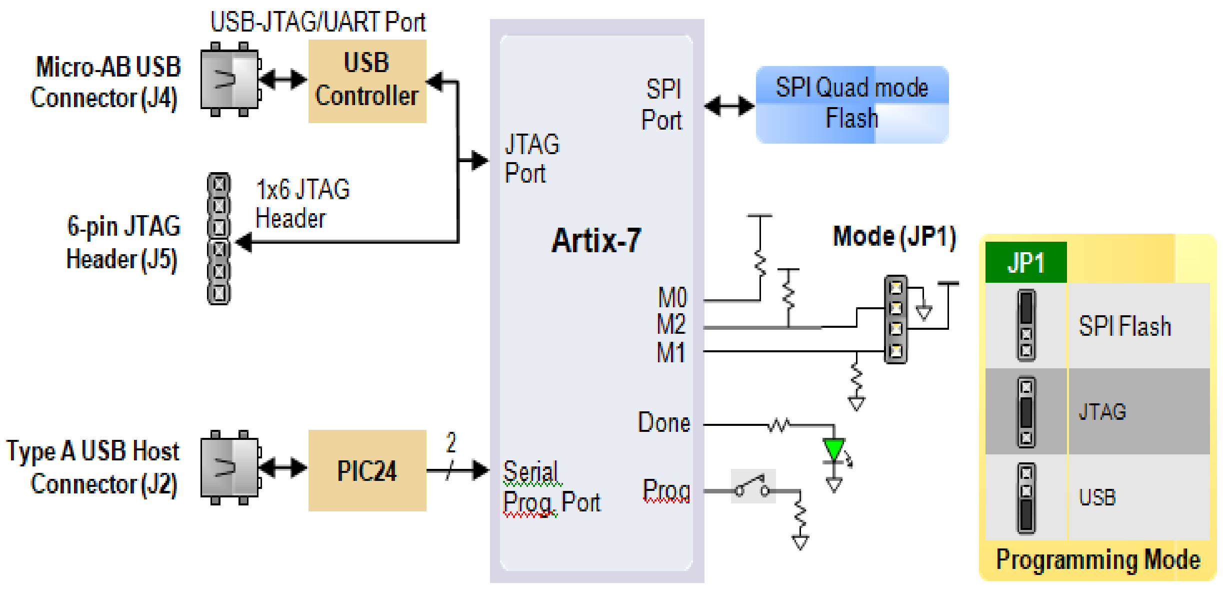


Fig3. Basys-3 configuration options

The FPGA configuration data is stored in files called bit streams that have the .bit file extension. The Vivadosoftware from Xilinx can create bit streams from VHDL, Verilog®, or schematic-based source files.

Bit stream are stored in SRAM-based memory cells within the FPGA. This data defines the FPGA's logic functionsand circuit connections, and it remains valid until it is erased by removing board power, by pressing the reset button attached to the PROG input, or by writing a new configuration file using the JTAG port.

An Artix-7 35T bit stream is typically 17,536,096 bits and can take a long time to transfer. The time it takes to program the Basys-3 can be decreased by compressing the bit stream before programming, and then allowing the FPGA to decompress the Bit stream itself during configuration. Depending on design complexity, compression ratiosof 10x can be achieved. Bit stream compression can be enabled within the Xilinx Tools (Vivado) to occur during generation. For instructions on how to do this, consult the Xilinx documentation for the toolset being used.

After being successfully programmed, the FPGA will cause the "DONE" LED to illuminate. Pressing the "PROG" button at any time will reset the configuration memory in the FPGA. After being reset, the FPGA will immediatelyattempt to reprogram itself from whatever method has been selected by the programming mode jumper.

The following sections provide greater detail about programming the Basys-3 using the different methods available.

4.2.3.1 JTAG Programming

The Xilinx Tools typically communicate with FPGAs using the Test Access Port and Boundary-Scan Architecture, commonly referred to as JTAG. During JTAG programming, a .bit file is transferred from the PC to the FPGA using the onboard Digilent USB-JTAG circuitry (port J4) or an external JTAG programmer, such as the Digilent JTAG-HS2 attached to port J5 (located below port JA). You can perform JTAG programming any time after the Basys-3 has been powered on regardless of what the mode jumper (JP1) is set to. If the FPGA is already configured, then the existing configuration is overwritten with the Bit stream being transmitted over JTAG. Setting the mode jumper to the JTAG setting (seen in Fig. 3) is useful to prevent the FPGA from being configured from any other Bit stream source until a JTAG programming occurs.

Programming the Basys-3 with an uncompressed Bit stream using the on-board USB_JTAG circuitry usually takes around five seconds. JTAG programming can be done using the hardware server in Vivado. The demonstration project available at digilentinc.com provides an in-depth tutorial on how to program your board.

4.2.3.2 JTAG Programming

When programming a nonvolatile flash device, a Bit stream file is transferred to the flash in a two-step process. First, the FPGA is programmed with a circuit that can program flash devices, and then data is transferred to the flash device via the FPGA circuit (this complexity is hidden from the user by the Xilinx Tools). After the flash device has been programmed, it can automatically configure the FPGA at a subsequent power-on or reset event as determined by the mode jumper setting (see Fig. 3). Programming files stored in the flash device will remain until they are overwritten, regardless of power-cycle events. Programming the flash can take as long as one or two minutes, which is mostly due to the lengthy erase process inherent to the memory technology. Once written, however, FPGA configuration can be very fast – less than a second. Bit stream compression, SPI bus width, and configuration rate are factors controlled by the Xilinx Tools that can affect configuration speed.

Quad-SPI programming can be performed using Vivado.

4.2.3.3 USB Host Programming

You can program the FPGA from a pen drive attached to the USB-HID port (J2) by doing the following:

1. Format the storage device (Pen drive) with a FAT32 file system.
2. Place a single bit configuration file in the root directory of the storage device.
3. Attach the storage device to the Basys-3.
4. Set the JP1 Programming Mode jumper on the Basys-3 to "USB".
5. Push the PROG button or power-cycle the Basys-3.

The FPGA will automatically be configured with the .bit file on the selected storage device. Any .bit files that arenot built for the proper Artix-7 device will be rejected by the FPGA.

The Auxiliary Function Status, or "BUSY" LED (LD16), gives visual feedback on the state of the configuration processwhen the FPGA is not yet programmed:

- When steadily lit, the auxiliary microcontroller is either booting up or currently reading the configurationmedium (pen drive) and downloading a Bit stream to the FPGA.
- A slow pulse means the microcontroller is waiting for a configuration medium to be plugged in.
- In case of an error during configuration, the LED will blink rapidly.

When the FPGA has been successfully configured, the behavior of the LED is application-specific. For example, if aUSB keyboard is plugged in, a rapid blink will signal the receipt of an HID input report from the keyboard.

4.2.4. Memory

The Basys-3 board contains a 32Mbit non-volatile serial Flash device, which is attached to the Artix-7 FPGA using a dedicated quad-mode (x4) SPI bus. The connections and pin assignments between the FPGA and the serial flash device are shown in Fig. 4.

FPGA configuration files can be written to the Quad SPI Flash (Spansion part number S25FL032), and mode settings are available to cause the FPGA to automatically read a configuration from this device at power on. An Artix-7 35T configuration file requires just over two Mbytes of memory, leaving approximately 48% of the flash device available for user data.

NOTE: Refer to the manufacturer's data sheets and the reference designs posted on Digilent's website for more information about the memory devices.

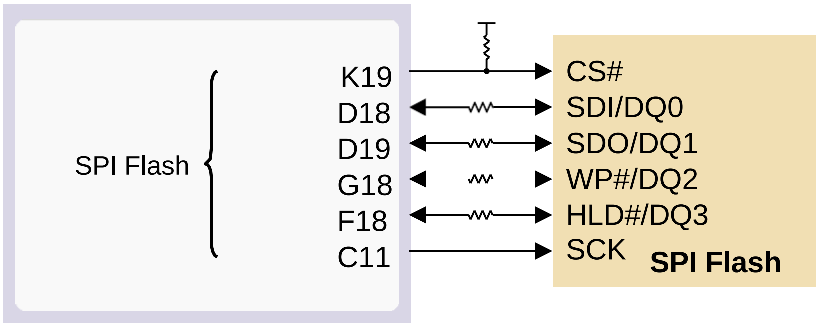


Fig4. Basys-3 external memory

4.2.5. Oscillators/Clocks

The Basys-3 board includes a single 100 MHz oscillator connected to pin W5 (W5 is a MRCC input on bank 34). 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 LogiCORE™ Clocking Wizard IP to help users generate the different clocks required for a specific design. This wizard properly instantiates 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 IP Catalog, which can be found under the Project Manager section of the Flow Navigator in Vivado.

4.2.6. USB-UART Bridge (Serial Port)

The Basys-3 includes an FTDI FT2232HQ USB-UART bridge (attached to connector J4) that allows you to use PC applications to communicate with the board using standard Windows COM port commands. Free USB-COM port drivers, available from www.ftdichip.com under the "Virtual Com Port" or VCP heading, convert USB packets to UART/serial port data. Serial port data is exchanged with the FPGA using a two-wire serial port (TXD/RXD). After the drivers are installed, I/O commands can be used from the PC directed to the COM port to produce serial data traffic on the B18 and A18 FPGA pins.

Two on-board status LEDs provide visual feedback on traffic flowing through the port: the transmit LED (LD18) and the receive LED (LD17). Signal names that imply direction are from the point-of-view of the DTE (Data Terminal Equipment), in this case the PC.

The FT2232HQ is also used as the controller for the Digilent USB-JTAG circuitry, but the USB-UART and USB-JTAG functions behave entirely independent of one another. Programmers interested in using the UART functionality of the FT2232 within their design do not need to worry about the JTAG circuitry interfering with the UART data transfers, and vice-versa. The combination of these two features into a single device allows the Basys-3 to be programmed, communicated with via UART, and powered from a computer attached with a single Micro USB cable. The connections between the FT2232HQ and the Artix-7 are shown in Fig. 6.

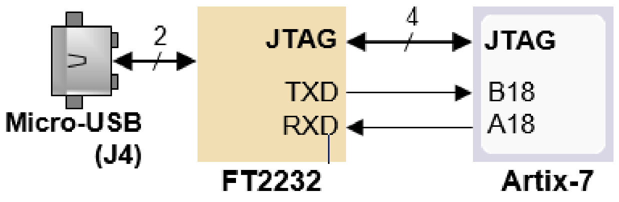


Fig6. Basys-3 FT2232HQ Connection

4.2.7.USB HID Host

The Auxiliary Function microcontroller (Microchip PIC24FJ128) provides the Basys-3 with USB HID host capability. After power-up, the microcontroller is in configuration mode, either downloading a Bit stream to the FPGA or waiting for it to be programmed from other sources. Once the FPGA is programmed, the microcontroller switches to application mode, which in this case is USB HID Host mode. Firmware in the microcontroller can drive a mouse or a keyboard attached to the type A USB connector at J2 labeled "USB." Hub support is not currently available, so only a single mouse or a single keyboard can be used. The PIC24 drives several signals into the FPGA – two are used to implement a standard PS/2 interface for communication with a mouse or keyboard, and the others are connected to the FPGA's two-wire serial programming port, so the FPGA can be programmed from a file stored on a USB pen drive.

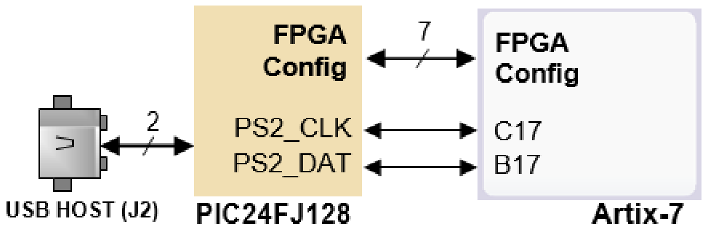
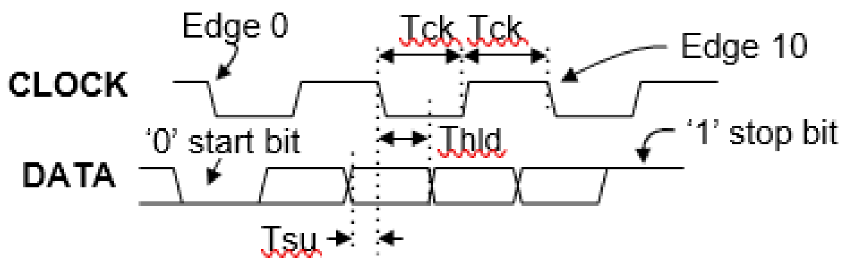


Fig7. Basys-3 PIC24 connections.

4.2.7.1 HID CONTROLLER

The Auxiliary Function microcontroller hides the USB HID protocol from the FPGA and emulates an old-style PS/2 bus. The microcontroller behaves just like a PS/2 keyboard or mouse would. This means new designs can re-use existing PS/2 IP cores. Mice and keyboards that use the PS/2 protocol use a two-wire serial bus (clock and data) to communicate with a host. On the Basys-3, the microcontroller emulates a PS/2 device while the FPGA plays the role of the host. Both the mouse and the keyboard use 11-bit words that include a start bit, data byte (LSB first), odd parity, and stop bit, but the data packets are organized differently, and the keyboard interface allows bi- directional data transfers (so the host device can illuminate state LEDs on the keyboard). Bus timings are shown in Fig. 8.



Symbol	Parameter	Min	Max
T_{CK}	Clock time	30us	50us
T_{SU}	Data-to-clock setup time	5us	25us
T_{HLD}	Clock-to-data hold time	5us	25us

Fig8. PS/2 device-to host timing diagram.

The clock and data signals are only driven when data transfers occur; otherwise they are held in the idle state at logic '1.' This requires that when the PS/2 signals are used in a design, internal pull-ups must be enabled in the FPGA on the data and clock pins. The clock signal is normally driven by the device, but may be held low by the hostin special cases. The timings define signal requirements for mouse-to-host communications and bi-directional keyboard communications. A PS/2 interface circuit can be implemented in the FPGA to create a keyboard or mouse interface.

When a keyboard or mouse is connected to the Basys-3, a "self-test passed" command (0xAA) is sent to the host. After this, commands may be issued to the device. Since both the keyboard and the mouse use the same PS/2 port, one can tell the type of device connected using the device ID. This ID can be read by issuing a Read ID command (0xF2). Also, a mouse sends its ID (0x00) right after the "self-test passed" command, which distinguishesit from a keyboard.

4.2.7.2 Keyboard

The keyboard uses open-collector drivers so the keyboard, or an attached host device, can drive the two-wire bus(if the host device will not send data to the keyboard, then the host can use input-only ports).

PS/2-style keyboards use scan codes to communicate key press data. Each key is assigned a code that is sent whenever the key is pressed. If the key is held down, the scan code will be sent repeatedly about once every 100ms. When a key is released, an F0 key-up code is sent, followed by the scan code of the released key. If a key can be shifted to produce a new character (like a capital letter), then a shift character is sent in addition to the scan code and the host must determine which ASCII character to use. Some keys, called extended keys, send an E0 ahead of the scan code (and they may send more than one scan code). When an extended key is released, an E0 F0key-up code is sent, followed by the scan code. Scan codes for most keys are shown in Fig. 9.

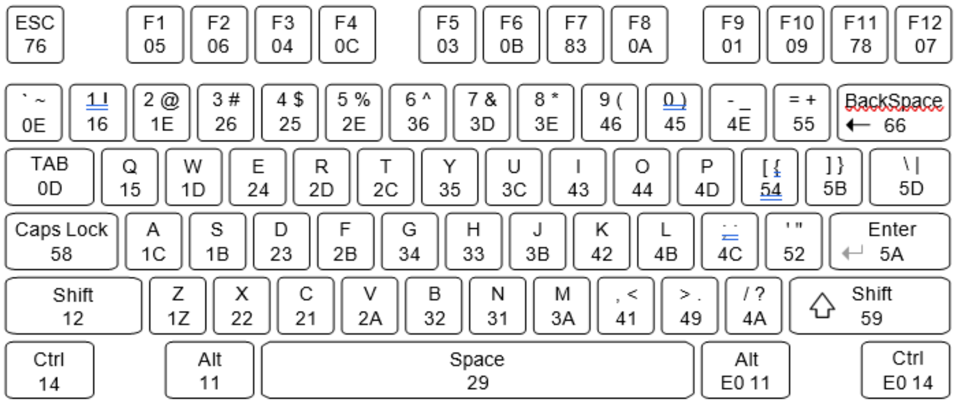


Fig 9. Keyboard scan codes.

A host device can also send data to the keyboard. Table 3 shows a list of some common commands a host mightsend.

The keyboard can send data to the host only when both the data and clock lines are high (or idle). Because the host is the bus master, the keyboard must check to see whether the host is sending data before driving the bus. To facilitate this, the clock line is used as a "clear to send" signal. If the host drives the clock line low, the keyboard must not send any data until the clock is released. The keyboard sends data to the host in 11-bit words that contain a '0' start bit, followed by 8-bits of scan code (LSB first), followed by an odd parity bit, and terminated witha '1' stop bit. The keyboard generates 11 clock transitions (at 20 to 30 KHz) when the data is sent, and data is valid on the falling edge of the clock.

Table 3. Keyboard commands.

Command	Action
ED	Set Num Lock, Caps Lock, and Scroll Lock LEDs. Keyboard returns FA after receiving ED, then host sends a byte to set LED status: bit 0 sets Scroll Lock, bit 1 sets Num Lock, and bit 2 setsCaps lock. Bits 3 to 7 are ignored.
EE	Echo (test). Keyboard returns EE after receiving EE.
F3	Set scan code repeat rate. Keyboard returns F3 on receiving FA, then host sends second byte to set the repeat rate.
FE	Resend. FE directs keyboard to re-send most recent scan code.
FF	Reset. Resets the keyboard.

4.2.7.3 MOUSE

Once entered in stream mode and data reporting has been enabled, the mouse outputs a clock and data signal when it is moved. Otherwise, these signals remain at logic '1.' Each time the mouse is moved, three 11-bit words are sent from the mouse to the host device, as shown in Fig. 10. Each of the 11-bit words contains a '0' start bit, followed by 8 bits of data (LSB first), followed by an odd parity bit, and terminated with a '1' stop bit. Thus, each data transmission contains 33 bits, where bits 0, 11, and 22 are '0' start bits, and bits 11, 21, and 33 are '1' stop bits. The three 8-bit data fields contain movement data as shown in the Fig. 10. Data is valid at the falling edge ofthe clock, and the clock period is 20 to 30 KHz.

The mouse assumes a relative coordinate system wherein moving the mouse to the right generates a positive number in the X field, and moving to the left generates a negative number. Likewise, moving the mouse up generates a positive number in the Y field, and moving down represents a negative number (the XS and YS bits in the status byte are the sign bits – a '1' indicates a negative number). The magnitude of the X and Y numbers represent the rate of mouse movement; the larger the number, the faster the mouse is moving (the XV and YV bits in the status byte are movement overflow indicators – a '1' means overflow has occurred). If the mouse moves continuously, the 33-bit transmissions are repeated every 50ms or so. The L and R fields in the status byte indicate Left and Right button presses (a '1' indicates that the button is being pressed).

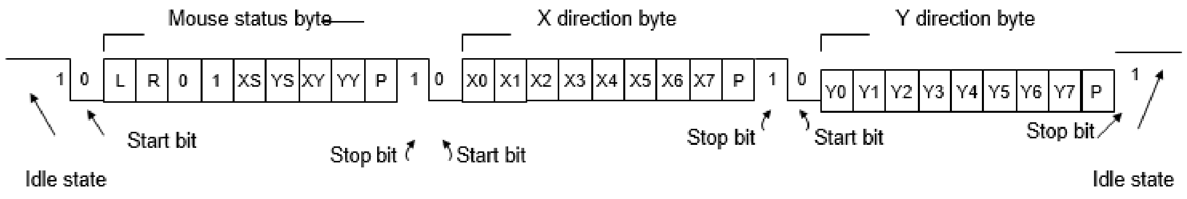


Fig 10. Mouse data format.

The microcontroller also supports Microsoft® IntelliMouse®-type extensions for reporting back a third axis representing the mouse wheel, as shown in Table 4.

Table 4. Microsoft Intelli mouse-type extensions, commands, and actions.

Command	Action
EA	Set stream mode. The mouse responds with "acknowledge" (0xFA) then resets its movement counters and enters stream mode.
F4	Enable data reporting. The mouse responds with "acknowledge" (0xFA) then enables data reporting and resets its movement counters. This command only affects behavior in stream mode. Once issued, mouse movement will automatically generate a data packet.
F5	Disable data reporting. The mouse responds with "acknowledge" (0xFA) then disables data reporting and resets its movement counters.
F3	Set mouse sample rate. The mouse responds with "acknowledge" (0xFA) then reads one more byte from the host. This byte is then saved as the new sample rate, and a new "acknowledge" packet is issued.
FE	Resend. FE directs mouse to re-send last packet.
FF	Reset. The mouse responds with "acknowledge" (0xFA) then enters reset mode.

4.2.8 VGA PORT

NOTE: A helpful way to understand the way that VGA signals are transmitted is to understand the method of which CRT (Cathode Ray Tubes) function for displaying images. Although the technology may seem outdated, it is from this legacy that many of the signal names and timings have originated.

The Basys-3 board uses 14 FPGA signals to create a VGA port with 4-bits per colour and the two standard sync signals (HS – Horizontal Sync, and VS – Vertical Sync). The colour signals use resistor-divider circuits that work in conjunction with the 75 ohm termination resistance of the VGA display to create 16 signal levels each on the red, green, and blue VGA signals. This circuit, shown in Fig. 11, produces video colour signals that proceed in equal increments between 0V (fully off) and 0.7V (fully on). Using this circuit, 4096 different colours can be displayed, one for each unique 12-bit pattern. A video controller circuit must be created in the FPGA to drive the sync and colour signals with the correct timing in order to produce a working display system.

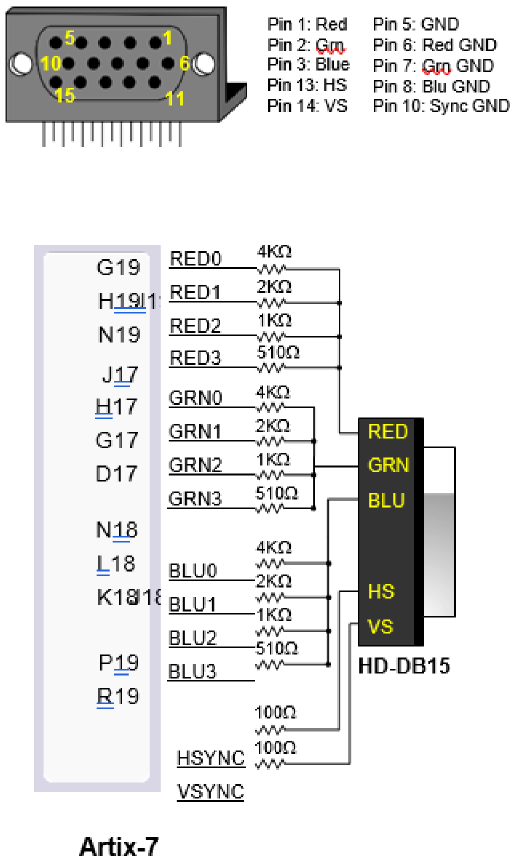


Fig 11.VGA port

4.2.8.1 VGA SYSTEM TIMING

VGA signal timings are specified, published, copyrighted, and sold by the VESA® organization (www.vesa.org). The following VGA system timing information is provided as an example of how a VGA monitor might be driven in 640 by 480 mode.

NOTE: For more precise information, or for information on other VGA frequencies, refer to documentation available at the VESA website.

CRT-based VGA displays use amplitude-modulated moving electron beams (or cathode rays) to display information on a phosphor-coated screen. LCD displays use an array of switches that can impose a voltage across a small amount of liquid crystal, thereby changing light permittivity through the crystal on a pixel-by-pixel basis. Although the following description is limited to CRT displays, LCD displays have evolved to use the same signal timings as CRT displays (so the "signals" discussion below pertains to both CRTs and LCDs). Color CRT displays use three electron beams (one for red, one for blue, and one for green) to energize the phosphor that coats the inner side of the display end of a cathode ray tube (see Fig. 12).

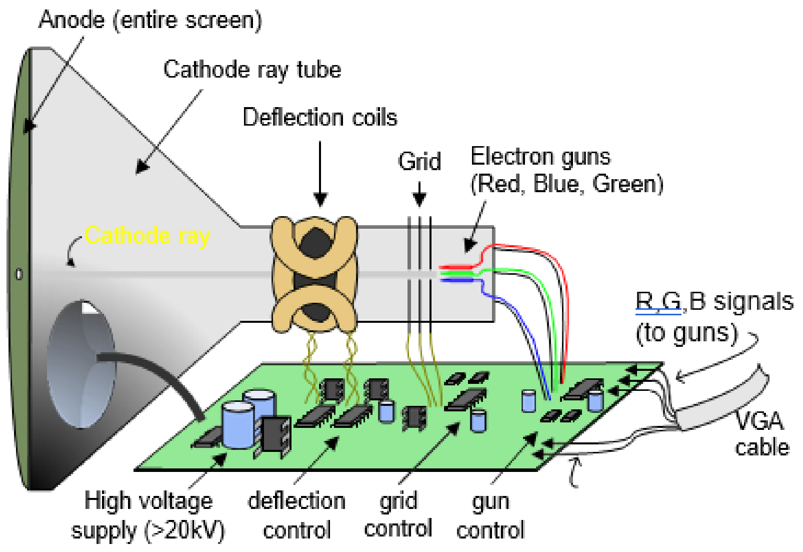


Fig 12. Colour CRT display.

Electron beams emanate from "electron guns" which are finely-pointed heated cathodes placed in close proximity to a positively charged annular plate called a "grid." The electrostatic force imposed by the grid pulls rays of energized electrons from the cathodes, and those rays are fed by the current that flows into the cathodes. These particle rays are initially accelerated towards the grid, but they soon fall under the influence of the much larger electrostatic force that results from the entire phosphor-coated display surface of the CRT being charged to 20kV (or more). The rays are focused to a fine beam as they pass through the center of the grids, and then they accelerate to impact on the phosphor-coated display surface. The phosphor surface glows brightly at the impact point, and it continues to glow for several hundred microseconds after the beam is removed. The larger the current fed into the cathode, the brighter the phosphor will glow.

Between the grid and the display surface, the beam passes through the neck of the CRT where two coils of wire produce orthogonal electromagnetic fields. Because cathode rays are composed of charged particles (electrons), they can be deflected by these magnetic fields. Current waveforms are passed through the coils to produce magnetic fields that interact with the cathode rays and cause them to transverse the display surface in a "raster" pattern, horizontally from left to right and vertically from top to bottom, as shown in Fig. 13. As the cathode ray moves over the surface of the display, the current sent to the electron guns can be increased or decreased to change the brightness of the display at the cathode ray impact point.

Information is only displayed when the beam is moving in the "forward" direction (left to right and top to bottom), and not during the time the beam is reset back to the left or top edge of the display. Much of the potential display time is therefore lost in "blanking" periods when the beam is reset and stabilized to begin a new horizontal or vertical display pass. The size of the beams, the frequency at which the beam can be traced across the display, and the frequency at which the electron beam can be modulated determine the display resolution.

Modern VGA displays can accommodate different resolutions, and a VGA controller circuit dictates the resolution by producing timing signals to control the raster patterns. The controller must produce synchronizing pulses at 3.3V (or 5V) to set the frequency at which current flows through the deflection coils, and it must ensure that video data is applied to the electron guns at the correct time. Raster video displays define a number of "rows" that corresponds to the number of horizontal passes the cathode makes over the display area, and a number of "columns" that corresponds to an area on each row that is assigned to one "picture element" or pixel. Typical displays use from 240 to 1200 rows and from 320 to 1600 columns. The overall size of a display and the number of rows and columns determines the size of each pixel.

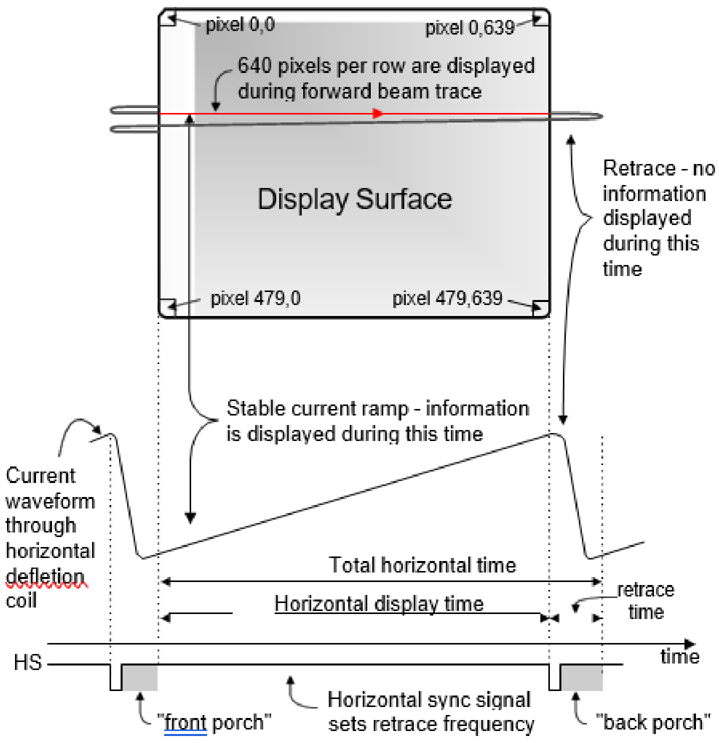
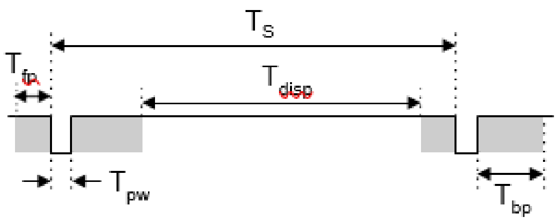

pixel 0,0
pixel 0,639
640 pixels per row are displayed during forward beam trace
Display Surface
Retrace - no information displayed during this time
pixel 479,0
pixel 479,639
Stable current ramp - information is displayed during this time
Current waveform through horizontal defletion coil
Total horizontal time
Horizontal display time
retrace time
HS
time
"front porch"
Horizontal sync signal sets retrace frequency
"back porch"

Fig 13. VGA horizontal synchronization.

Video data typically comes from a video refresh memory; with one or more bytes assigned to each pixel location (the Basys-3 uses 12-bits per pixel). The controller must index into video memory as the beams move across the display, and retrieve and apply video data to the display at precisely the time the electron beam is moving across a given pixel.

A VGA controller circuit must generate the HS and VS timings signals and coordinate the delivery of video data based on the pixel clock. The pixel clock defines the time available to display one pixel of information. The VS signal defines the "refresh" frequency of the display, or the frequency at which all information on the display is redrawn. The minimum refresh frequency is a function of the display's phosphor and electron beam intensity, with practical refresh frequencies falling in the 50Hz to 120Hz range. The number of lines to be displayed at a given refresh frequency defines the horizontal "retrace" frequency. For a 640-pixel by 480-row display using a 25 MHz pixel clock and 60 +/-1Hz refresh, the signal timings shown in Fig. 14 can be derived. Timings for sync pulse width and front and back porch intervals (porch intervals are the pre- and post-sync pulse times during which information cannot be displayed) are based on observations taken from actual VGA displays.



Symbol	Parameter	Vertical Sync			Horiz. Sync	
		Time	Clocks	Lines	Time	Clks
T_S	Sync pulse	16.7ms	416,800	521	32 us	800
T_{disp}	Display time	15.36ms	384,000	480	25.6 us	640
T_{pw}	Pulse width	64 us	1,600	2	3.84 us	96
T_{fp}	Front porch	320 us	8,000	10	640 ns	16
T_{bp}	Back porch	928 us	23,200	29	1.92 us	48

Fig 14. Signal timings for a 640-pixel by 480 row display using a 25 MHz pixel clock and 60 Hz vertical refresh.

A VGA controller circuit, such as the one diagramed in Fig. 15, decodes the output of a horizontal-sync counter driven by the pixel clock to generate HS signal timings. You can use this counter to locate any pixel location on a given row. Likewise, the output of a vertical-sync counter that increments with each HS pulse can be used to generate VS signal timings, and you can use this counter to locate any given row. These two continually running counters can be used to form an address into video RAM. No time relationship between the onset of the HS pulse and the onset of the VS pulse is specified, so you can arrange the counters to easily form video RAM addresses, or to minimize decoding logic for sync pulse generation.

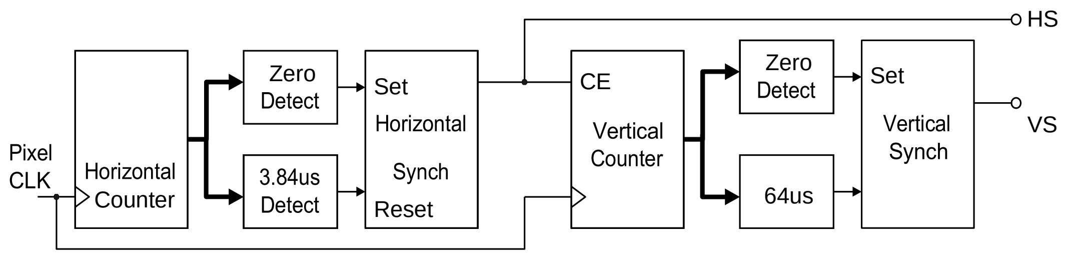


Fig 15. VGA display controller block diagram.

4.2.9. BASIC I/O

The Basys-3 board includes sixteen slide switches, five push buttons, sixteen individual LEDs, and a four-digit seven- segment display, as shown in Fig.16. The pushbuttons and slide switches are connected to the FPGA via series resistors to prevent damage from inadvertent short circuits (a short circuit could occur if an FPGA pin assigned to a pushbutton or slide switch was inadvertently defined as an output). The five pushbuttons, arranged in a plus-sign configuration, are "momentary" switches that normally generate a low output when they are at rest, and a high output only when they are pressed. Slide switches generate constant high or low inputs depending on their position.

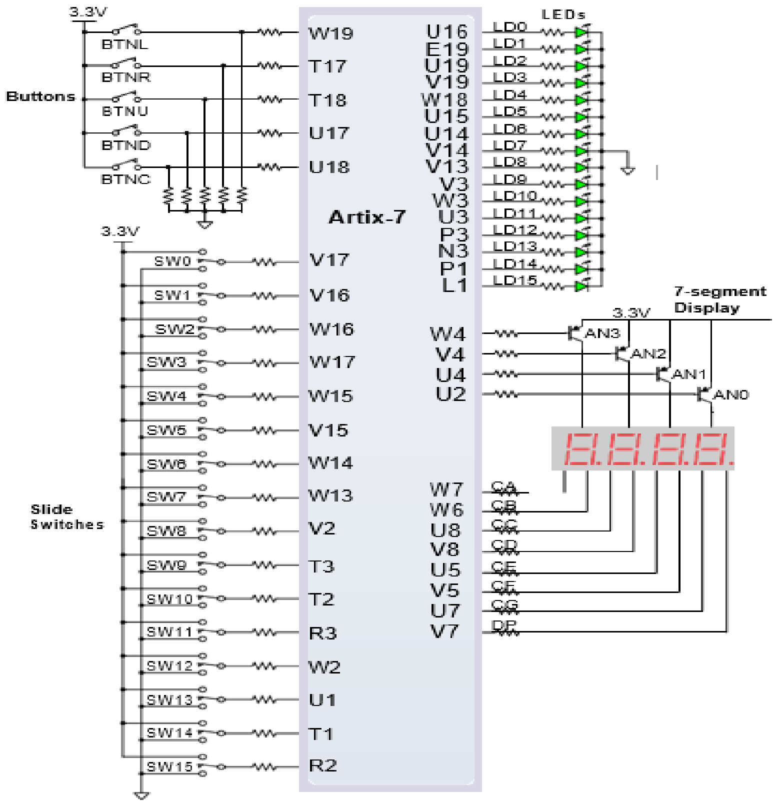


Fig 16. General purpose I/O devices on the Basys-3.

The sixteen individual high-efficiency LEDs are anode-connected to the FPGA via 330 ohm resistors, so they will turn on when a logic high voltage is applied to their respective I/O pin. Additional LEDs, which are not user- accessible, indicate power-on, FPGA programming status, and USB port status.

4.2.9.1 Seven-Segment Display

The Basys-3 board contains one four-digit common anode seven-segment LED display. Each of the four digits is composed of seven segments arranged in a "figure 8" pattern, with an LED embedded in each segment. Segment LEDs can be individually illuminated, so any one of 128 patterns can be displayed on a digit by illuminating certain LED segments and leaving the others dark, as shown in Fig. 17. Of these 128 possible patterns, the ten corresponding to the decimal digits are the most useful.

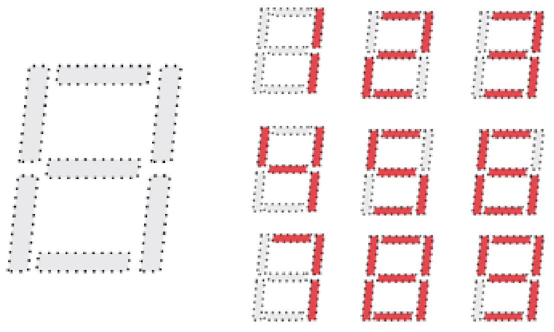


Fig 17. An un-illuminated seven-segment display and nine illumination patterns corresponding to decimal digits.

The anodes of the seven LEDs forming each digit are tied together into one "common anode" circuit node, but the LED cathodes remain separate, as shown in Fig. 18. The common anode signals are available as four "digit enable" input signals to the 4-digit display. The cathodes of similar segments on all four displays are connected into seven circuit nodes labeled CA through CG (for example, the four "D" cathodes from the four digits are grouped together into a single circuit node called "CD"). These seven cathode signals are available as inputs to the 4-digit display.

This signal connection scheme creates a multiplexed display, where the cathode signals are common to all digits but they can only illuminate the segments of the digit whose corresponding anode signal is asserted.

To illuminate a segment, the anode should be driven high while the cathode is driven low. However, since the Basys-3 uses transistors to drive enough current into the common anode point, the anode enables are inverted. Therefore, both the AN0..3 and the CA..G/DP signals are driven low when active.

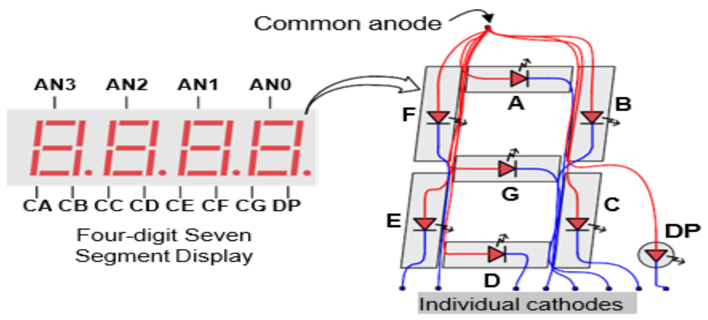


Fig 18. Common anode circuit node.

A scanning display controller circuit can be used to show a four-digit number on this display. This circuit drives the anode signals and corresponding cathode patterns of each digit in a repeating, continuous succession at an update rate that is faster than the human eye can detect. Each digit is illuminated just one-fourth of the time, but because the eye cannot perceive the darkening of a digit before it is illuminated again, the digit appears continuously illuminated. If the update, or "refresh", rate is slowed to around 45Hz, a flicker can be noticed in the display.

For each of the four digits to appear bright and continuously illuminated, all four digits should be driven once every 1 to 16ms, for a refresh frequency of about 1 KHz to 60Hz. For example, in a 62.5Hz refresh scheme, the entire display would be refreshed once every 16ms, and each digit would be illuminated for 1/4 of the refresh cycle, or 4ms. The controller must drive the cathodes low with the correct pattern when the corresponding anode signal is driven high. To illustrate the process, if AN0 is asserted while CB and CC are asserted, then a "1" will be displayed in digit position 1. Then, if AN1 is asserted while CA, CB, and CC are asserted, a "7" will be displayed in digit position 2. If AN0, CB, and CC are driven for 4ms, and then AN1, CA, CB, and CC are driven for 4ms in an endless succession, the display will show "71" in the first two digits. An example timing diagram for a four-digit controller is shown in Fig. 19.

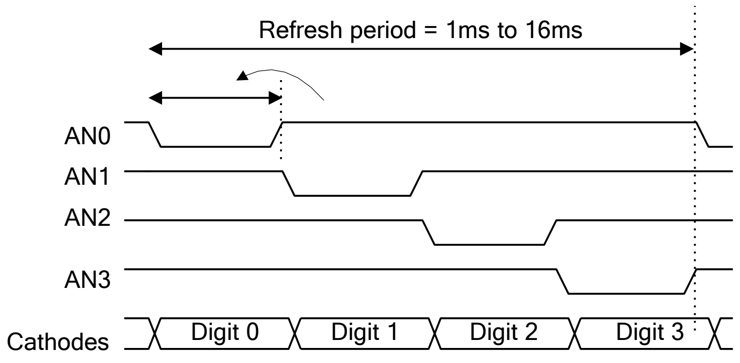


Fig 19. Four digit scanning display controller timing diagram.

4.2.10. PMOD PORTS

The Pmod ports are arranged in a 2x6 right-angle, and are 100-mil female connectors that mate with standard 2x6pin 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 Fig. 20. The VCC and Ground pins can deliver up to 1A of current. Pmod data signals are not matched pairs, and they are routed using best-available tracks without impedance control or delay matching. Pin assignments for the Pmod I/O connected to the FPGA are shown in Table 5.

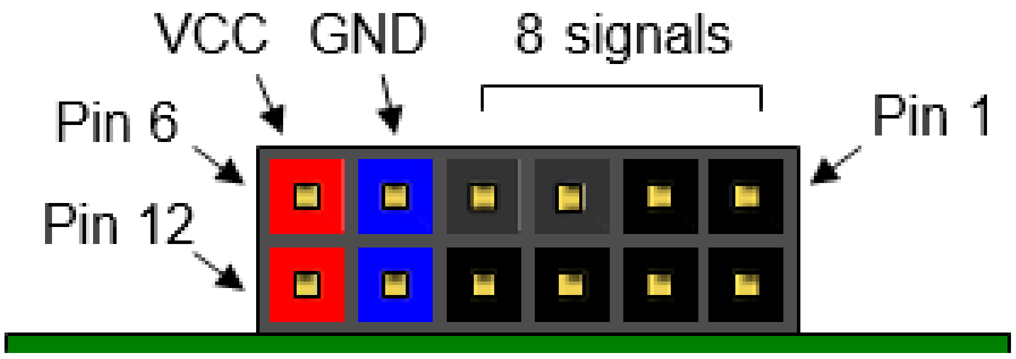
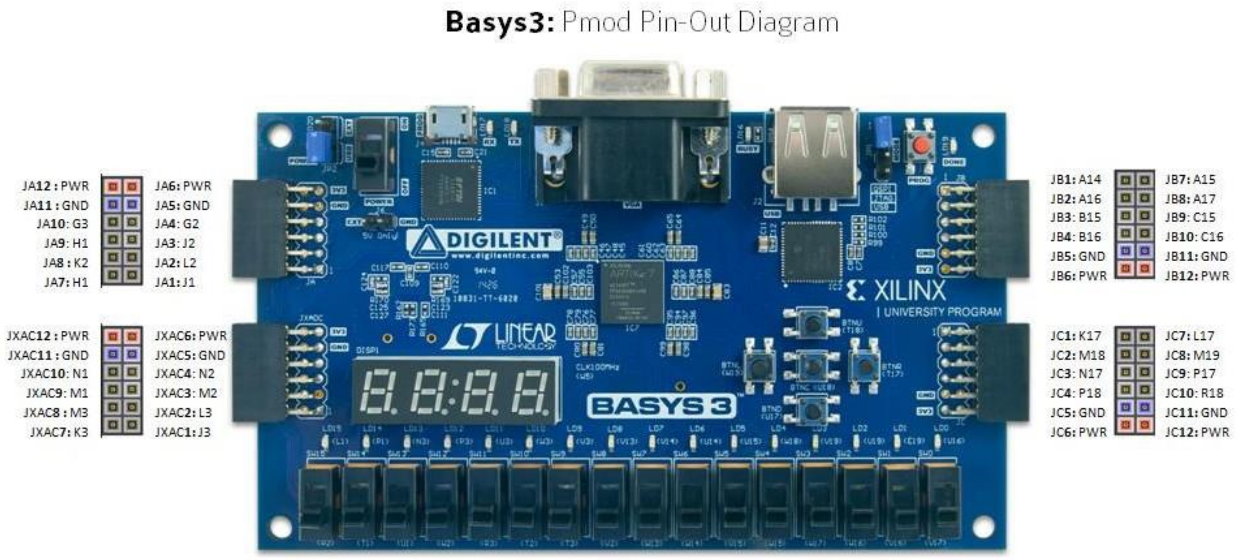


Fig 20. Pmod ports; front view as loaded on PCB.

Table 5. Basys-3 Pmod pin assignment.

Pmod JA	Pmod JB	Pmod JC	Pmod XDAC
JA1: J1	JB1: A14	JC1: K17	JXADC1: J3
JA2: L2	JB2: A16	JC2: M18	JXADC2: L3
JA3: J2	JB3: B15	JC3: N17	JXADC3: M2
JA4: G2	JB4: B16	JC4: P18	JXADC4: N2
JA7: H1	JB7: A15	JC7: L17	JXADC7: K3
JA8: K2	JB8: A17	JC8: M19	JXADC8: M3
JA9: H2	JB9: C15	JC9: P17	JXADC9: M1
JA10: G3	JB10: C16	JC10: R18	JXADC10: N1

Digilent produces a large collection of Pmod accessory boards that can attach to the Pmod expansion ports to addready-made functions like A/Ds, D/As, motor drivers, sensors, and other functions. See www.digilentinc.com for more information.



4.2.10.1 DUAL ANALOG/DIGITAL PMOD

The on-board Pmod expansion port, labeled "JXADC", is wired to the auxiliary analog input pins of the FPGA. Depending on the configuration, this connector can be used to input differential analog signals to the analog-to-digital converter inside the Artix-7 (XADC). Any or all pairs in the connector can be configured either as analog input or digital input-output.

The Dual Analog/Digital Pmod on the Basys-3 differs from the rest in the routing of its traces. The eight data signalsare grouped into four pairs, with the pairs routed closely coupled for better analog noise immunity. Furthermore, each pair has a partially loaded anti-alias filter laid out on the PCB. The filter does not have capacitors C33-C36. In designs where such filters are desired, the capacitors can be manually loaded by the user.

NOTE: The coupled routing and the anti-alias filters might limit the data speeds when used for digital signals. The XADC core within the Artix-7 is a dual channel 12-bit analog-to-digital converter capable of operating at 1

MSPS. Either channel can be driven by any of the auxiliary analog input pairs connected to the JXADC header. The XADC core is controlled and accessed from a user design via the Dynamic Reconfiguration Port (DRP). The DRP alsoprovides access to voltage monitors that are present on each of the FPGA's power rails, and a temperature sensor that is internal to the FPGA. For more information on using the XADC core, refer to the Xilinx document titled "7 Series FPGAs and Zynq-7000 All Programmable SoC XADC Dual 12-Bit 1 MSPS Analog-to-Digital Converter."

4.2.11. BUILT IN SELF TEST

A demonstration configuration is loaded into the SPI Flash device on the Basys-3 board during manufacturing. The source code and prebuilt Bit stream for this design are available for download from the Digilent website. If the demo configuration is present in the SPI Flash device and the Basys3 board is powered on in SPI mode, the demo project will allow basic hardware verification. Here is an overview of how this demo drives the different onboard components:

- The user LEDs are illuminated when the corresponding user switch is placed in the on position.
- The VGA port displays feedback from a USB Mouse.
- Connecting a mouse to the USB-HID Mouse port will allow the pointer on the VGA display to be controlled.
- On power-up, each digit of the seven-segment display will display a counter output from 0-9 that increments once a second.
- Pressing BTNU, BTNL, BTNR, or BTND will cause a digit of the seven-segment display to go blank.
- Pressing BTNC will reset the design.
- On power-up, a welcome message is sent over the UART. Also, every time a button is pressed a message is sent. The UART can be connected to using a terminal program with 9600 Baud, 8 data bits, 1 stop bit, and no parity.

All Basys3 boards are 100% tested during the manufacturing process. If any device on the Basys3 board fails test or is not responding properly, it is likely that damage occurred during transport or during use. Typical damage includes stressed solder joints and contaminants in switches and buttons resulting in intermittent failures. Stressed solder joints can be repaired by reheating and reflowing solder and contaminants can be cleaned with off-the-shelf electronics cleaning products. If a board fails test within the warranty period, it will be replaced at no cost. Contact Digilent for more details.

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:

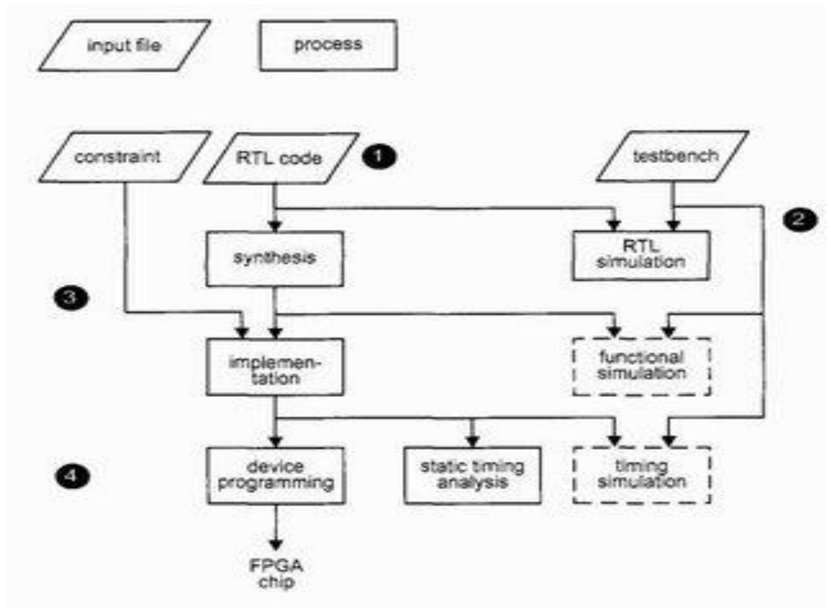


Fig 21. Development flow

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, which determines 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
```

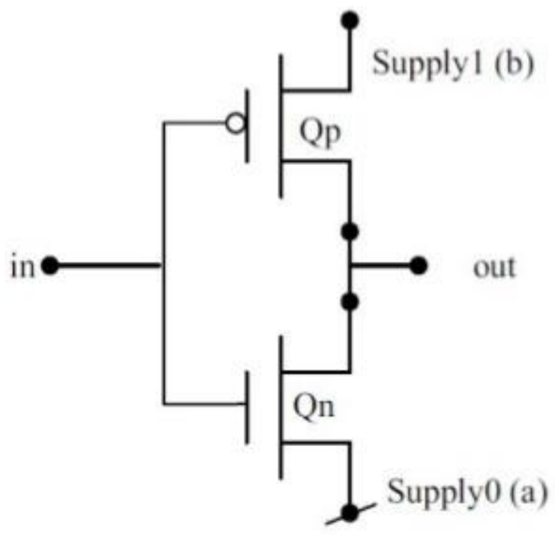
Verilog has four levels of modelling:

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

1) Switch level modelling

Switch level modelling forms the basic level of modelling 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 modelling 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 behavioural 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 functional level 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 22. CMOS inverter

2) Gate level modelling

Modelling done at this level is usually called gate level modelling 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 modelling 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, xor and 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 modelling provides the means of describing combinational circuits by their function rather than by their gate structure. Dataflow modelling uses a number of operators that act on operands to produce the desired results. Verilog HDL provides about 30 operator types.

Dataflow modelling 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) Behavioural 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 behaviour. 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 modelling 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 modelled with ease.

The abstraction in this modelling 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 modelling 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 the always block. The always block starts at time 0, and keeps on executing all the simulation time. It works like an infinite loop. It is generally used to 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 after every 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.

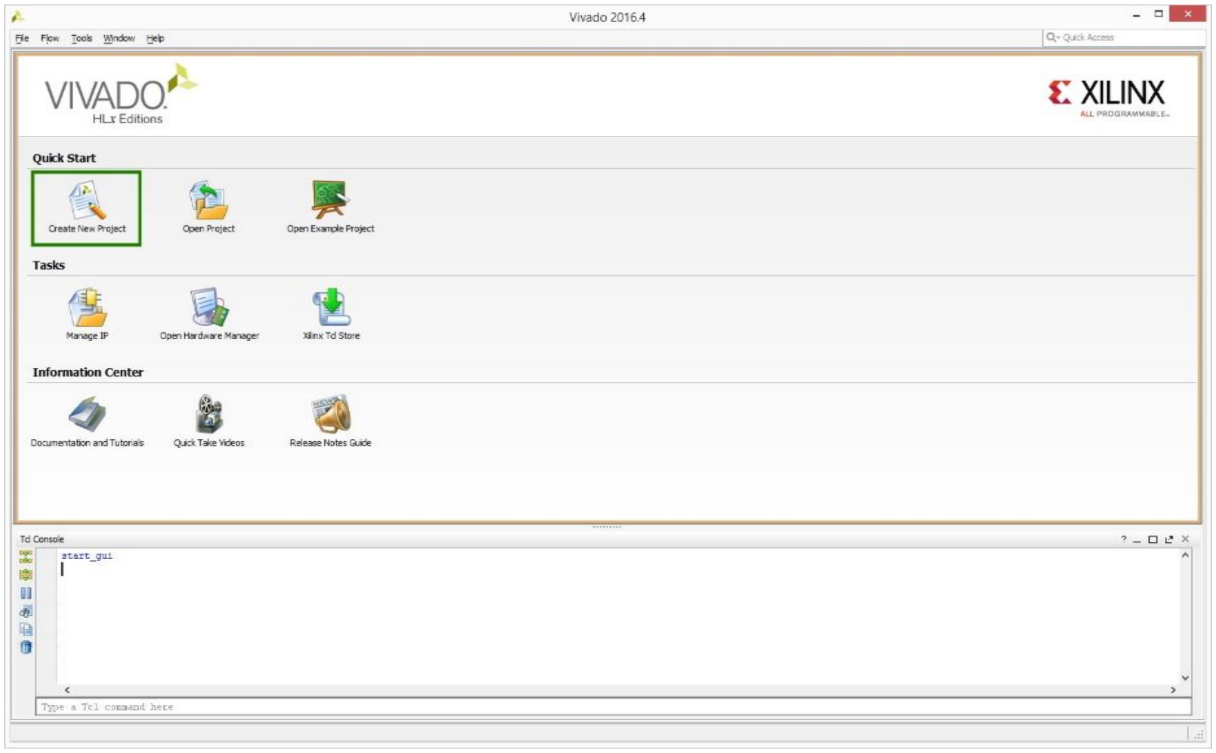
4.5 Overview of EDA (Electronic Design Automation) software

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-

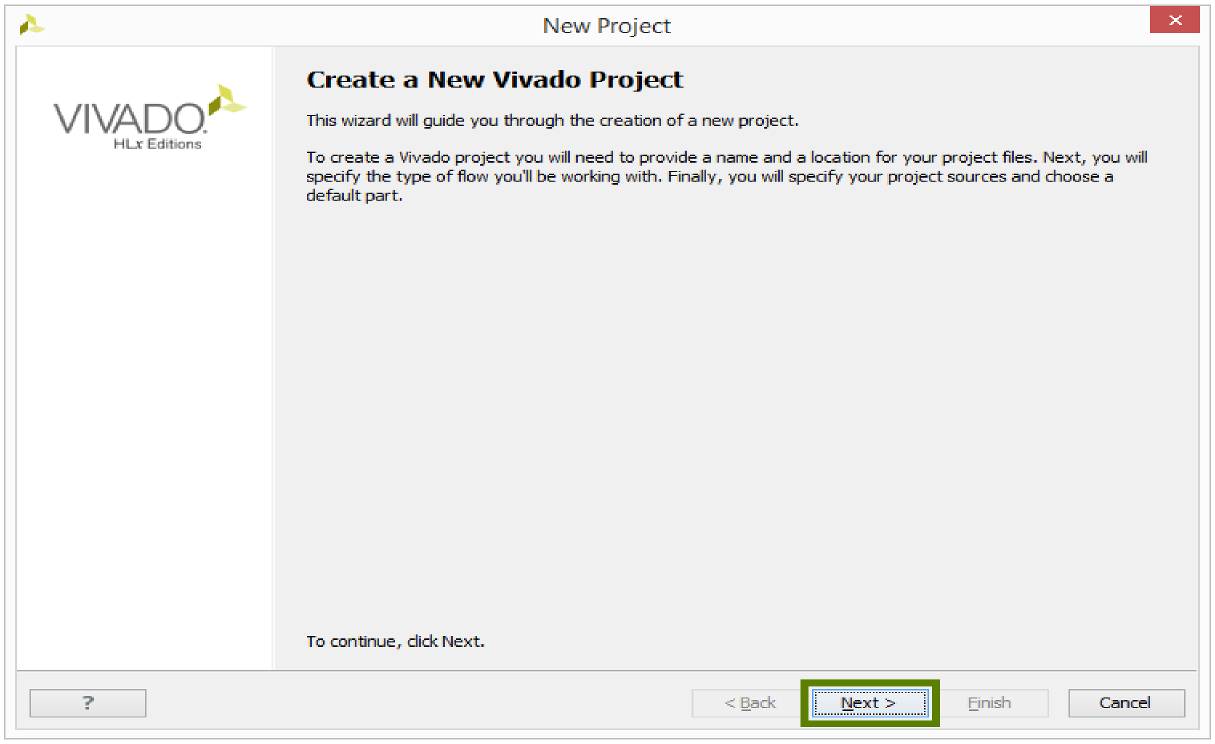
Creating a New Project

After launching Vivado, from the startup 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.



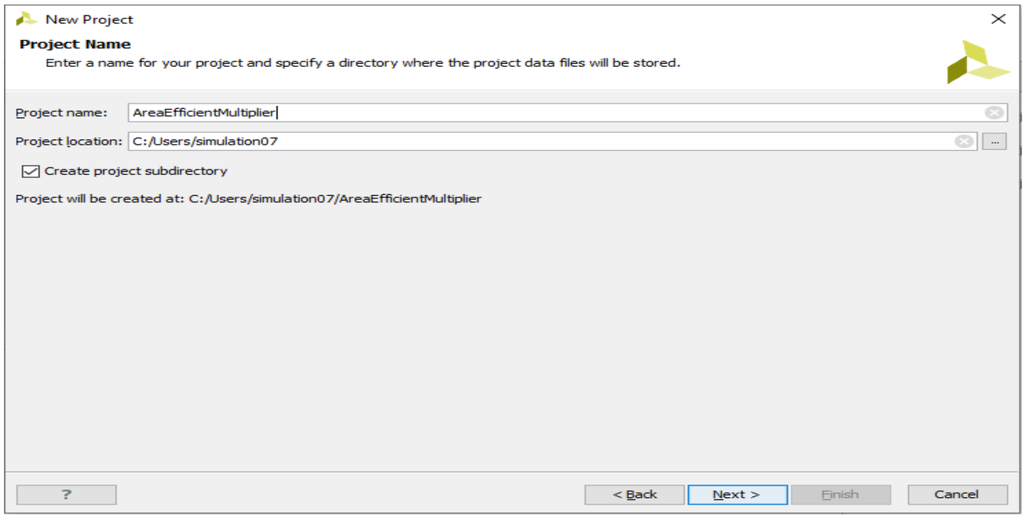
Create a New Vivado Project

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 necessary simply 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.

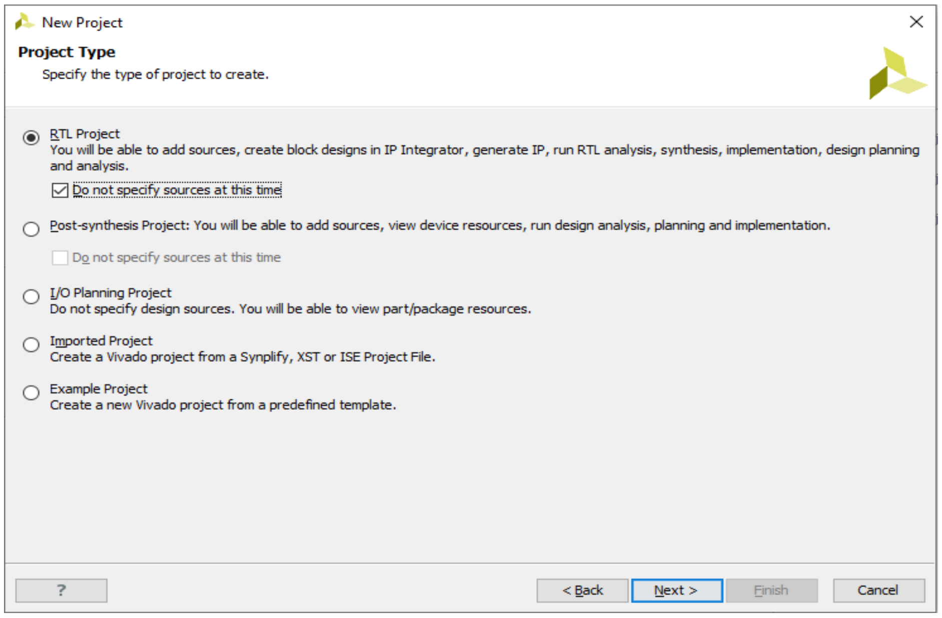
Project name: AreaEfficientMultiplier

Project location: C:/Users/simulation07

☑ Create project subdirectory

Project will be created at: C:/Users/simulation07/AreaEfficientMultiplier

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 and 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

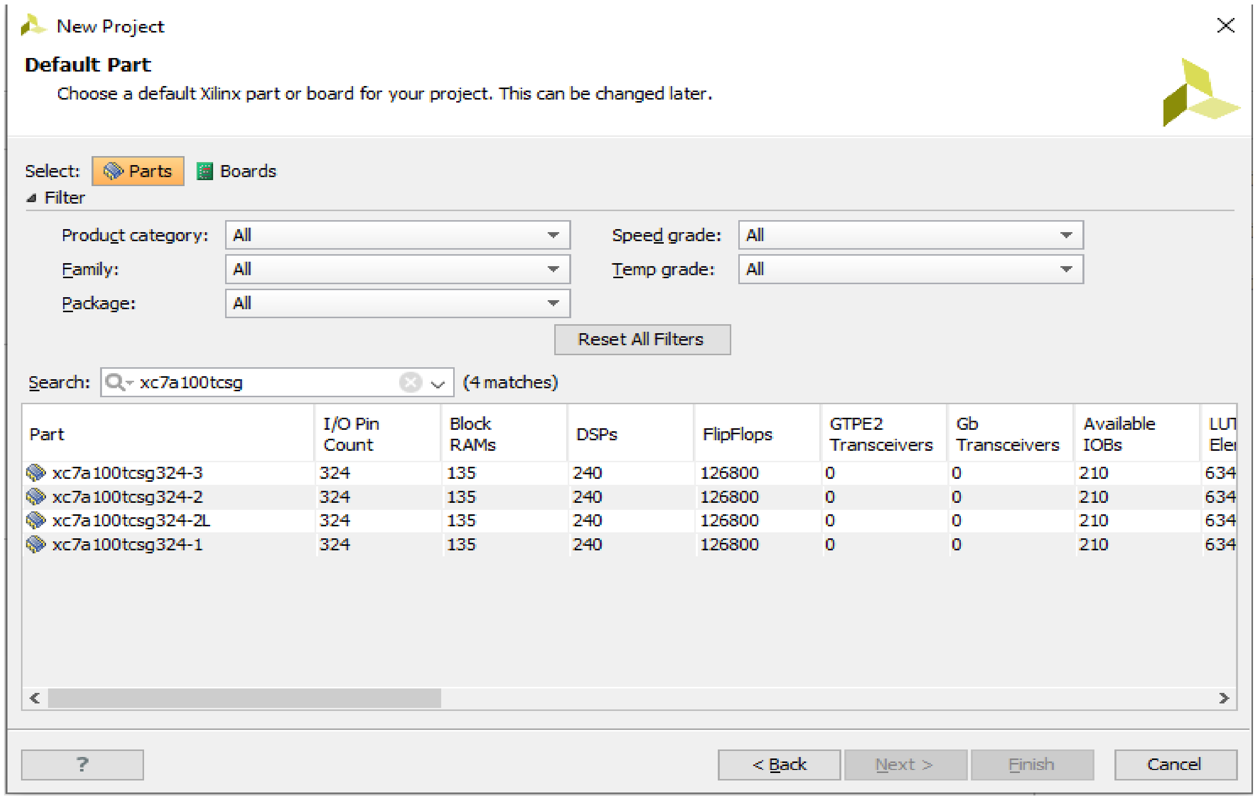
○ Post-synthesis Project: You will be able to add sources, view device resources, run design analysis, planning and implementation.
☐ Do not specify sources at this time

○ 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.

Once you select the RTL project, Click on next button.



New Project

Default Part

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

Select: Parts Boards

Filter

Product category: All

Family: All

Package: All

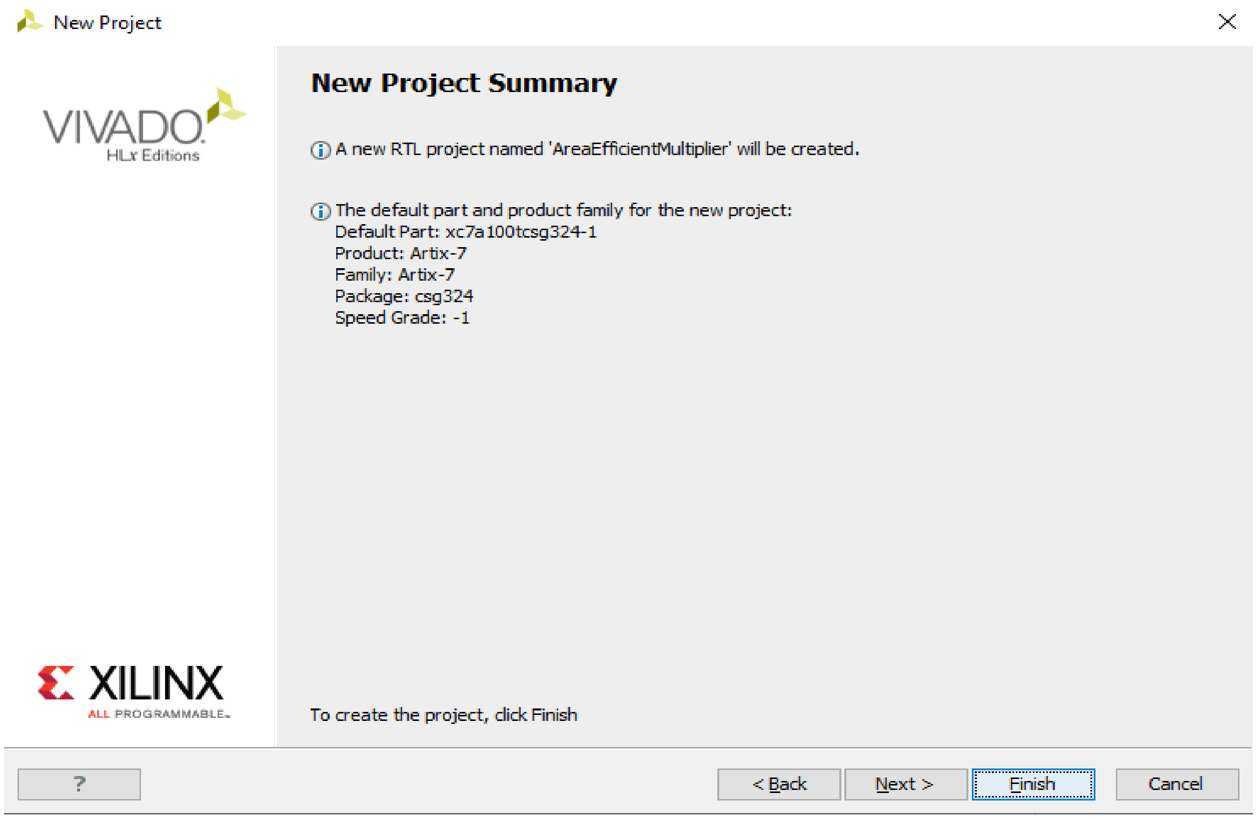
Speed grade: All

Temp grade: All

Reset All Filters

Search: xc7a100tcsg (4 matches)

Part	I/O Pin Count	Block RAMs	DSPs	FlipFlops	GTPE2 Transceivers	Gb Transceivers	Available IOBs	LUT Ele
xc7a100tcsg324-3	324	135	240	126800	0	0	210	634
xc7a100tcsg324-2	324	135	240	126800	0	0	210	634
xc7a100tcsg324-2L	324	135	240	126800	0	0	210	634
xc7a100tcsg324-1	324	135	240	126800	0	0	210	634



New Project

New Project Summary

A new RTL project named 'AreaEfficientMultiplier' 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:

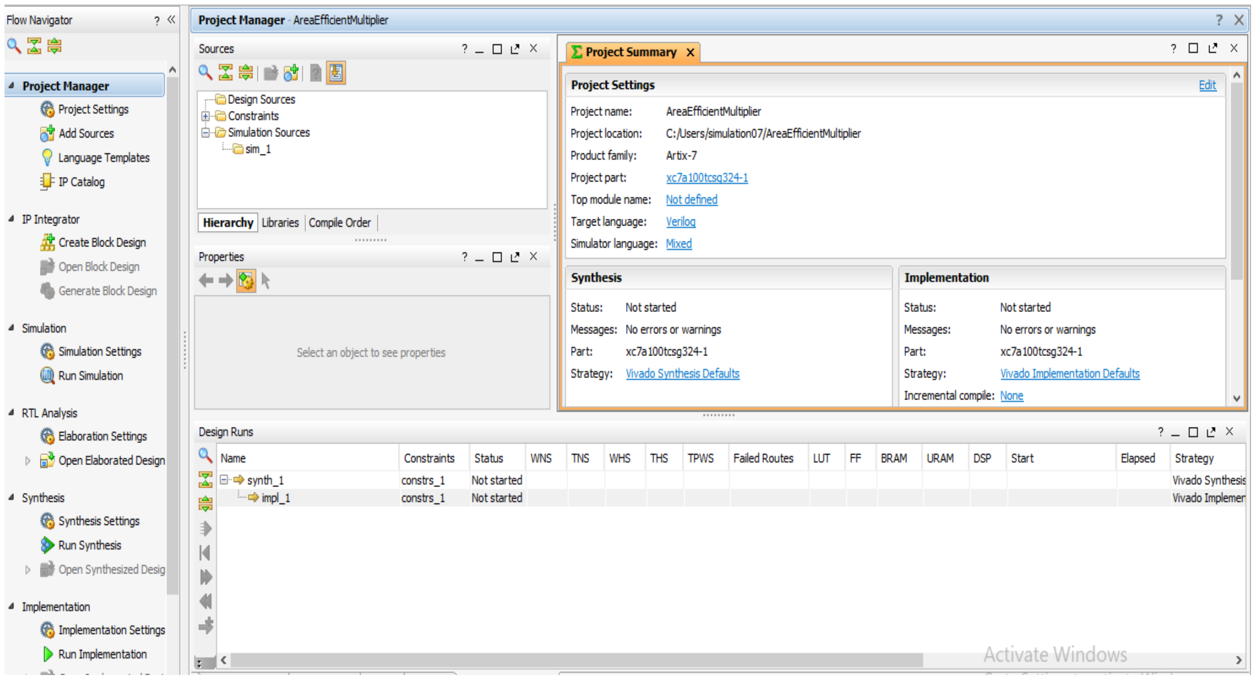
Working through the basic project flow

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 organized 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.



Project Settings

Project name:	AreaEfficientMultiplier
Project location:	C:/Users/simulation07/AreaEfficientMultiplier
Product family:	Artix-7
Project part:	xc7a100tcsg324-1
Top module name:	Not defined
Target language:	Verilog
Simulator language:	Mixed

Synthesis

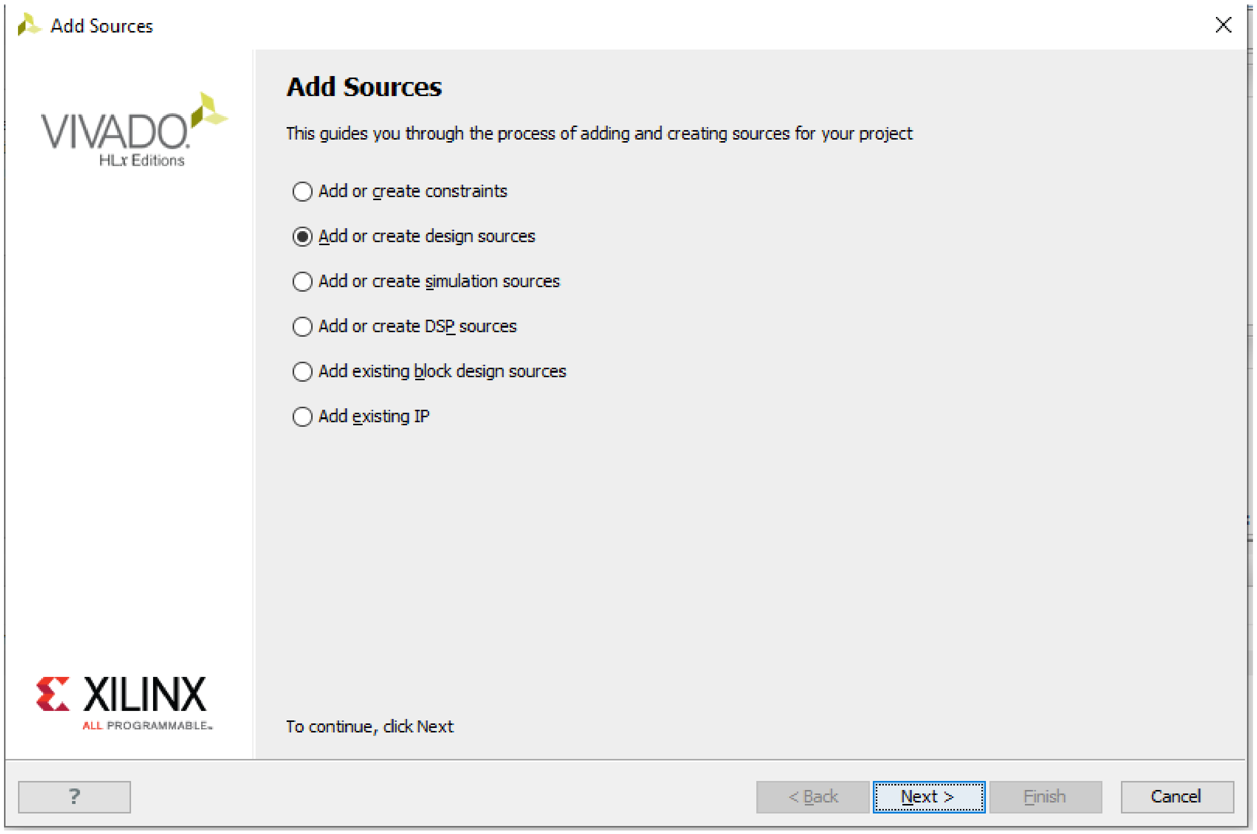
Status:	Not started
Messages:	No errors or warnings
Part:	xc7a100tcsg324-1
Strategy:	Vivado Synthesis Defaults

Implementation

Status:	Not started
Messages:	No errors or warnings
Part:	xc7a100tcsg324-1
Strategy:	Vivado Implementation Defaults
Incremental compile:	None

Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Failed Routes	LUT	FF	BRAM	URAM	DSP	Start	Elapsed	Strategy
synth_1	constrs_1	Not started														Vivado Synthesis
impl_1	constrs_1	Not started														Vivado Implemen

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


Add Sources

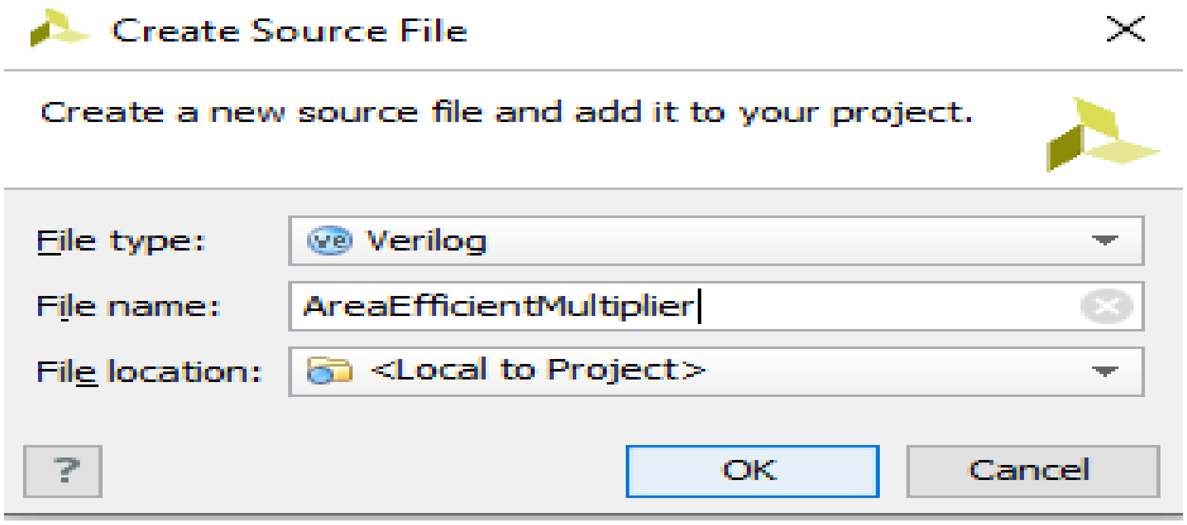
Add Sources

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.

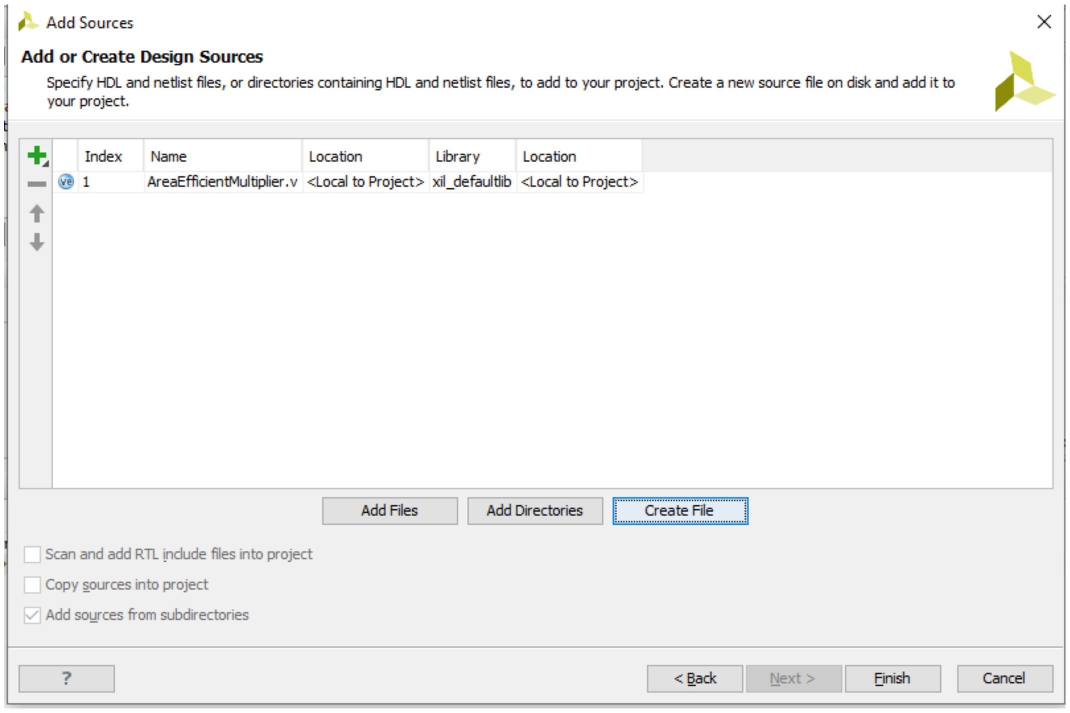
File type: Verilog

File name: AreaEfficientMultiplier

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	Location	Library	Location
1	AreaEfficientMultiplier.v	<Local to Project>	xil_defaultlib	<Local to Project>

- [ ] Scan and add RTL include files into project
- [ ] Copy sources into project
- [x] Add sources from subdirectories

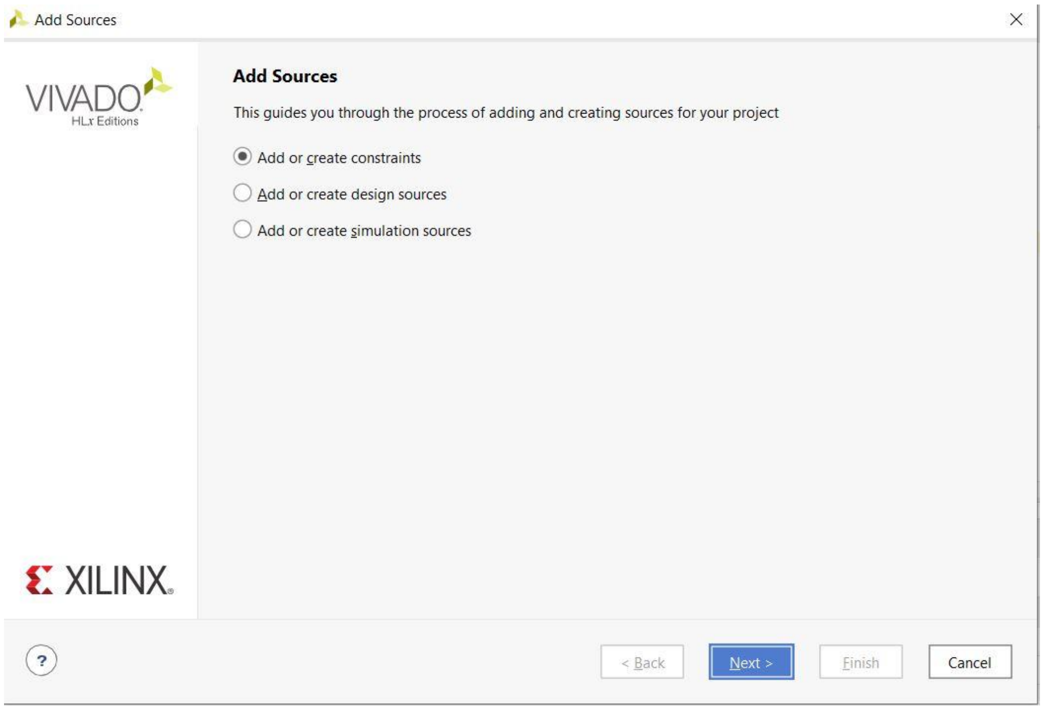
You can use the “define module” window to automatically write some of the Verilog code. Additional “I/O definitions” can be added by either clicking the green “+”symbol in the upper left or by simply clicking on the next empty line.

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.

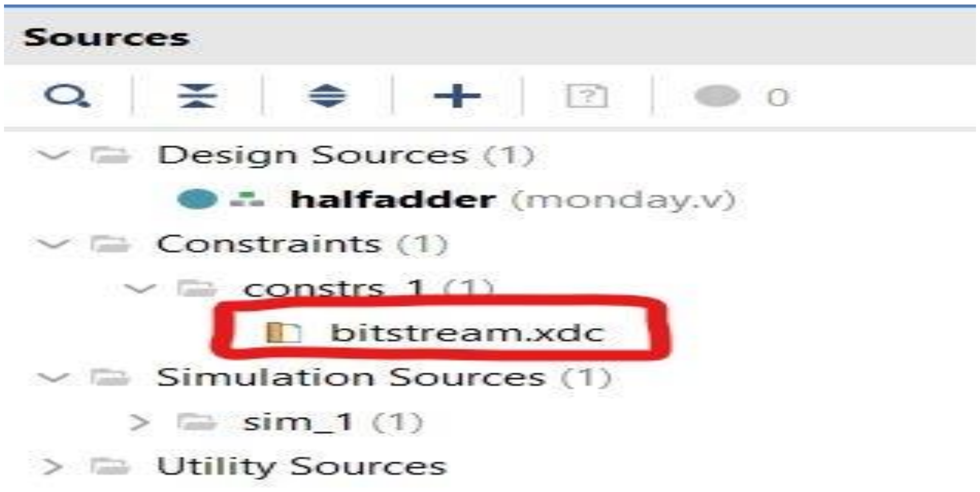
STEPS FOR CONSTRAINTS FILE CREATION

Click on “Add sources” under the project manager phase of the Flow navigator.

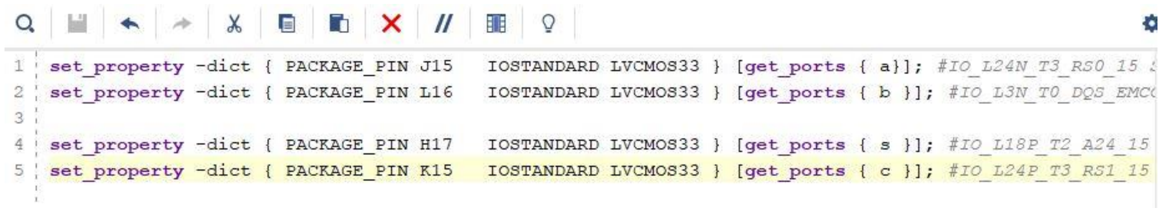
Select the “Add or create constraints” radialand then click the “Next>”button


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
< Back
Next >
Finish
Cancel

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".



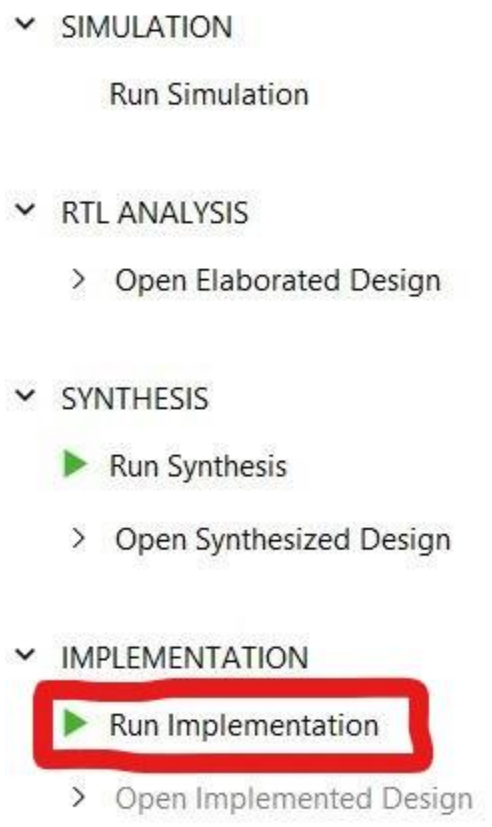
Write the bit file according to the requirement. The required memory locations are mapped to the program variables.



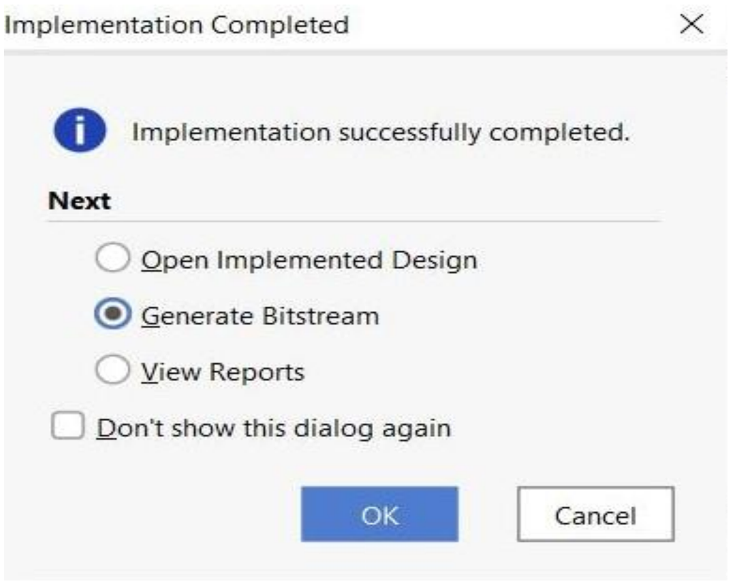
```
set_property -dict { PACKAGE_PIN J15   IOSTANDARD LVCMOS33 } [get_ports { a}]; #IO_L24N_T3_RS0_15
set_property -dict { PACKAGE_PIN L16   IOSTANDARD LVCMOS33 } [get_ports { b }]; #IO_L3N_T0_DQS_EMC

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 half adder. 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.

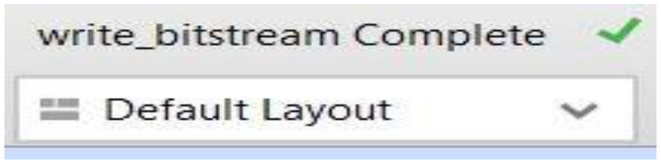
Next

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

☐ Don't show this dialog again

OK Cancel

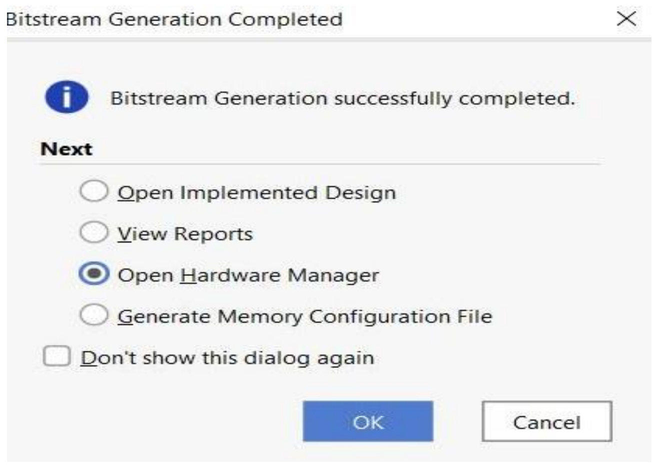
Click on "Generate Bit stream" and then "OK".


write_bitstream Complete ✓

Default Layout

This is an indication for completion of the generation of the Bit stream 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.

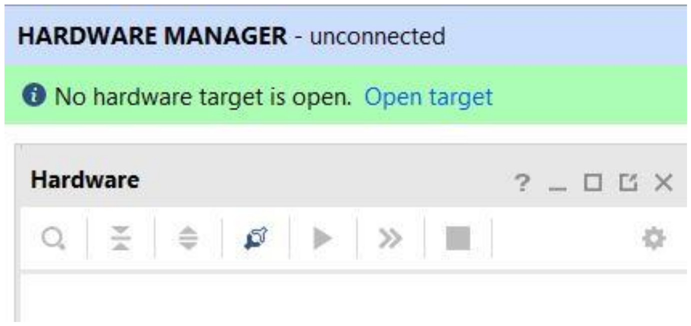
Next

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

☐ Don't show this dialog again

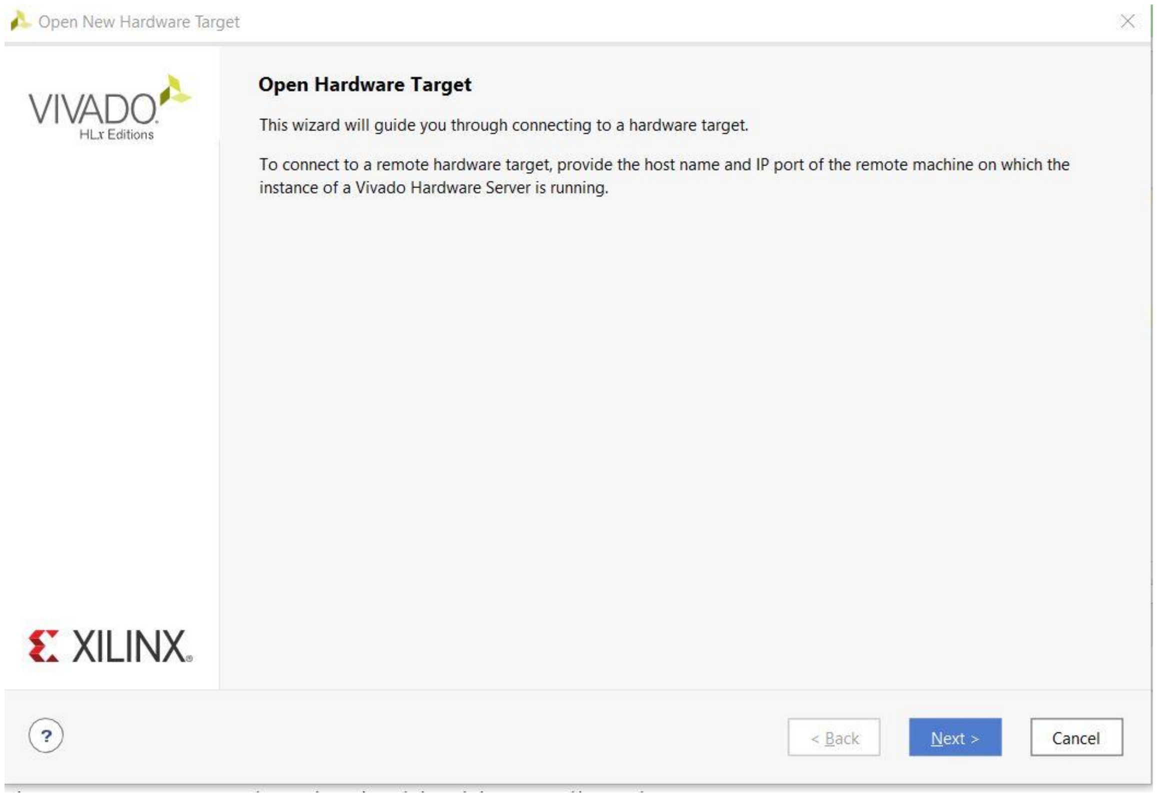
OK Cancel

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.

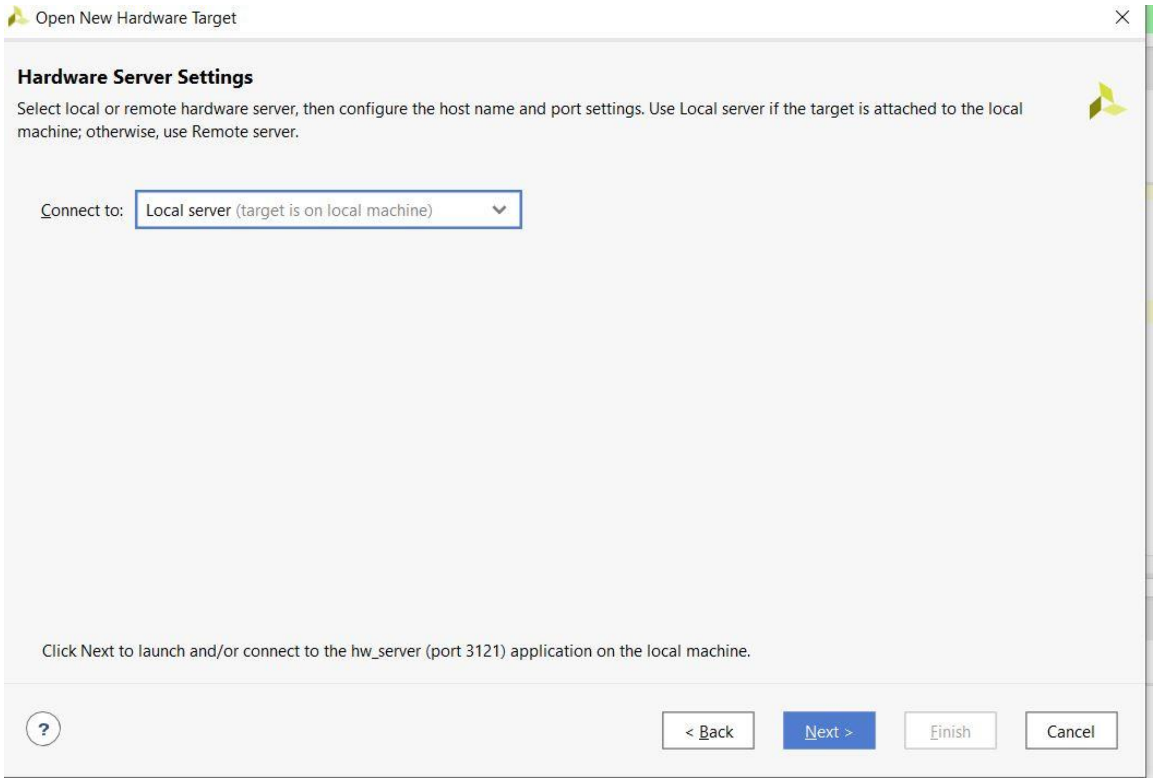


Open Hardware Target

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.

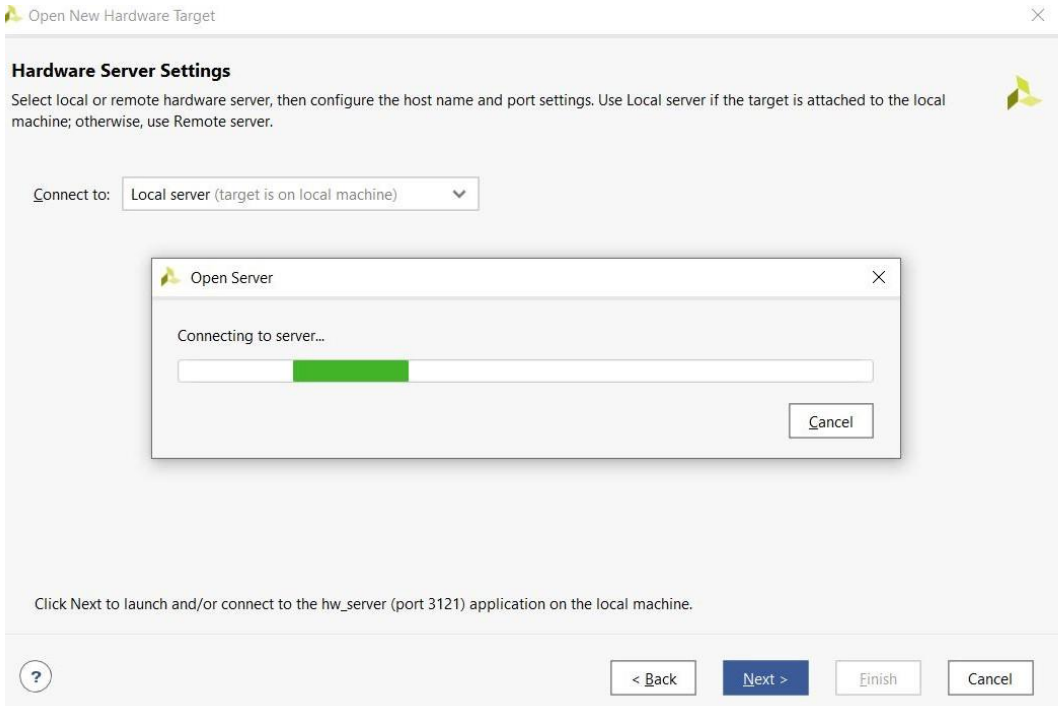

Open New Hardware Target

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.

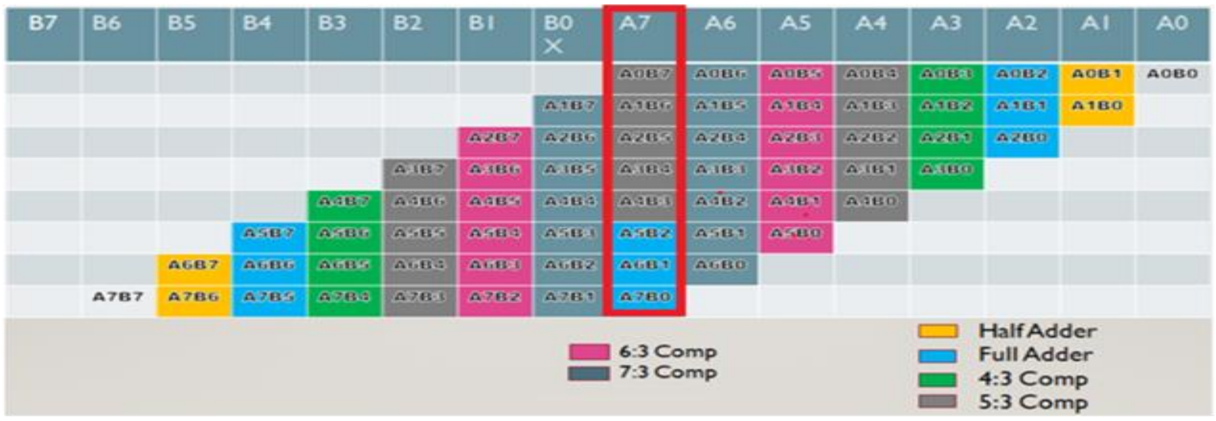
Select your specific Hardware device. Click the "Finish" button and Vivado will attempt to connect to your specified hardware. Now click "Program device" under the program and debug phase of the Flow Navigator and then your specific device from the menu that appears.

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

CHAPTER 5

METHODOLOGY

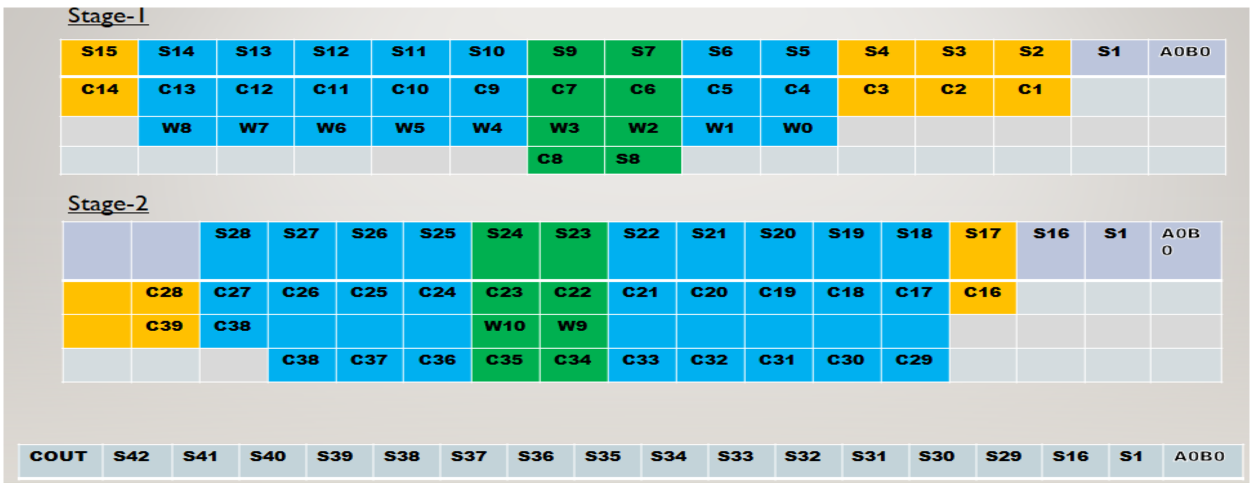
Method 1: Grouping 8 bits using 5:3 compressor and a full adder



B7 B6 B5 B4 B3 B2 B1 B0 X A7 A6 A5 A4 A3 A2 A1 A0
A0B7 A0B6 A0B5 A0B4 A0B3 A0B2 A0B1 A0B0
A1B7 A1B6 A1B5 A1B4 A1B3 A1B2 A1B1 A1B0
A2B7 A2B6 A2B5 A2B4 A2B3 A2B2 A2B1 A2B0
A3B7 A3B6 A3B5 A3B4 A3B3 A3B2 A3B1 A3B0
A4B7 A4B6 A4B5 A4B4 A4B3 A4B2 A4B1 A4B0
A5B7 A5B6 A5B5 A5B4 A5B3 A5B2 A5B1 A5B0
A6B7 A6B6 A6B5 A6B4 A6B3 A6B2 A6B1 A6B0
A7B7 A7B6 A7B5 A7B4 A7B3 A7B2 A7B1 A7B0
6:3 Comp
7:3 Comp
Half Adder
Full Adder
4:3 Comp
5:3 Comp

Fig5(a) Grouping 8 bits using 5:3 compressor and full adder

Here in this method, we use 4:3, 5:3, 6:3, 7:3 compressors. For Example, 5:3 compressor is used to add 5 bits which is generally done by using a full adder and a half adder. Similar case with other compressors also. Wallace tree method is used to find the result of multiplication of binary numbers. In this method, binary bits are added and the result is transferred to next stage as follows.



Stage-I
S15 S14 S13 S12 S11 S10 S9 S7 S6 S5 S4 S3 S2 S1 A0B0
C14 C13 C12 C11 C10 C9 C7 C6 C5 C4 C3 C2 C1
W8 W7 W6 W5 W4 W3 W2 W1 W0
C8 S8
Stage-2
S28 S27 S26 S25 S24 S23 S22 S21 S20 S19 S18 S17 S16 S1 A0B0
C28 C27 C26 C25 C24 C23 C22 C21 C20 C19 C18 C17 C16
C39 C38 W10 W9
C38 C37 C36 C35 C34 C33 C32 C31 C30 C29
COUT S42 S41 S40 S39 S38 S37 S36 S35 S34 S33 S32 S31 S30 S29 S16 S1 A0B0

Fig5(b) Stages involving in binary bits addition of 8-bit multiplication using 5:3 compressor

The output of a full adder will be sum and carry. The sum is transferred to same level of next stage and carry is transferred to next level of next stage. Similarly, if we take 5:3 compressor, it will generate three outputs namely sum, carry, auxiliary carry. Now sum will be transferred to same level of next stage, carry is transferred to next level of next stage and auxiliary carry is transferred to second next level of the next stage. If after grouping any bit is left ungrouped, then it is transferred directly to next stage. For example, if we group 8 bits using 7:3 compressor, one bit is left ungrouped. The first three output bits of 7:3 compressor follow same procedure as discussed previously and the left bit is transferred directly to next stage. Similarly, we group bits of next stage using same procedure and the outputs are transferred to succeeding next stage until we get the final result.

Method 2. Grouping 8 bits using 8:4 compressor

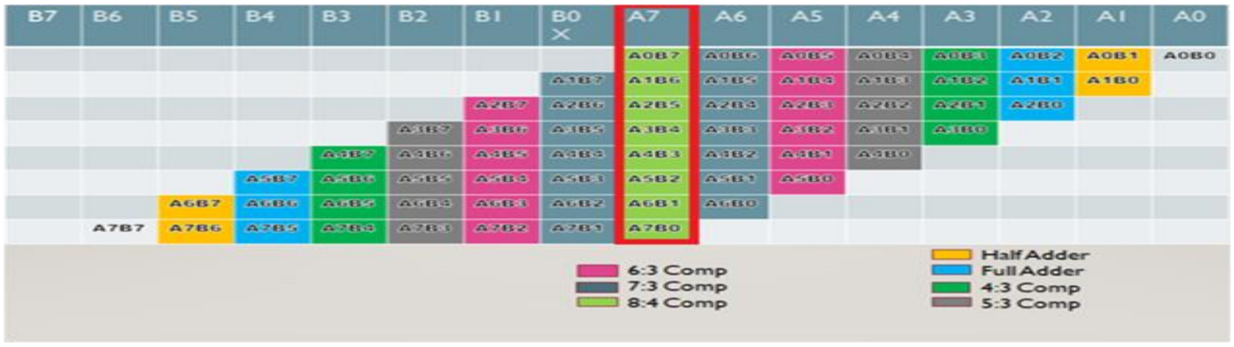
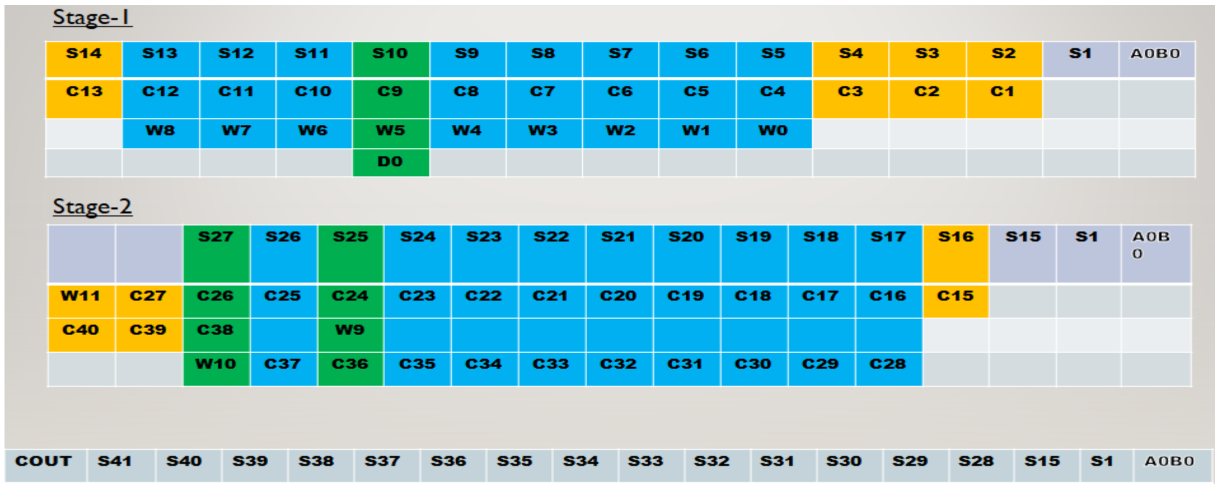


Fig5(c) Grouping 8 bits using 8:4 compressor



Stage-I
S14 S13 S12 S11 S10 S9 S8 S7 S6 S5 S4 S3 S2 S1 A0B0
C13 C12 C11 C10 C9 C8 C7 C6 C5 C4 C3 C2 C1
W8 W7 W6 W5 W4 W3 W2 W1 W0
D0
Stage-2
S27 S26 S25 S24 S23 S22 S21 S20 S19 S18 S17 S16 S15 S1 A0B0
W11 C27 C26 C25 C24 C23 C22 C21 C20 C19 C18 C17 C16 C15
C40 C39 C38 W9
W10 C37 C36 C35 C34 C33 C32 C31 C30 C29 C28
COUT S41 S40 S39 S38 S37 S36 S35 S34 S33 S32 S31 S30 S29 S28 S15 S1 A0B0

Fig5(d) Stages involving in binary bits addition of 8-bit multiplication using 8:4 compressor

The same methodology we used in case of method 1 is implemented here. But the only difference is 8 bits are grouped directly using 8:4 compressor.

CHAPTER 6

SIMULATION RESULTS AND CONCLUSIONS

6.1. Software: Xilinx Vivado

- We have written code for design of efficient multiplier using verilog language.
- Verilog is a Hardware Description Language which is used to design and describe digital circuits.
- The Verilog HDL is an IEEE standard - number 1364. Verilog is intended to be used for verification through simulation, for timing analysis, for test analysis and for logic synthesis.
- Verilog is an easier language as it’s syntax is based on C language. Verilog supports a design at many levels of abstraction. The major three are? Behavioral level, Register-transfer level and Gate level. We have simulated the code in vivado software and analyzed the results.

6.2. Simulation Results of Proposed Work

A) Method 1:

a) RTL Diagram

Register Transfer Level (RTL) is an abstraction for defining the digital portions of a design. The figure below represents the RTL diagram of 8 bit multiplier designed using method 1.

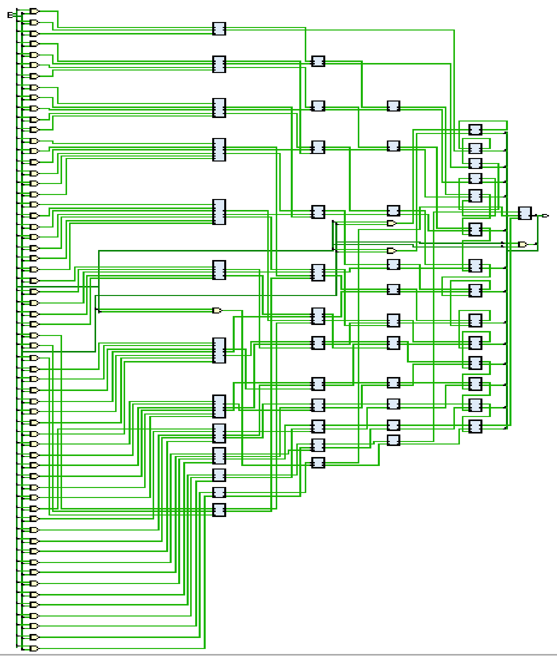


Fig 23. RTL diagram

b) Synthesized Schematic Diagram

The Synthesized Schematic diagram for the above RTL diagram is as shown. It represents how the actual components are connected on the artix-7 Fpga board.

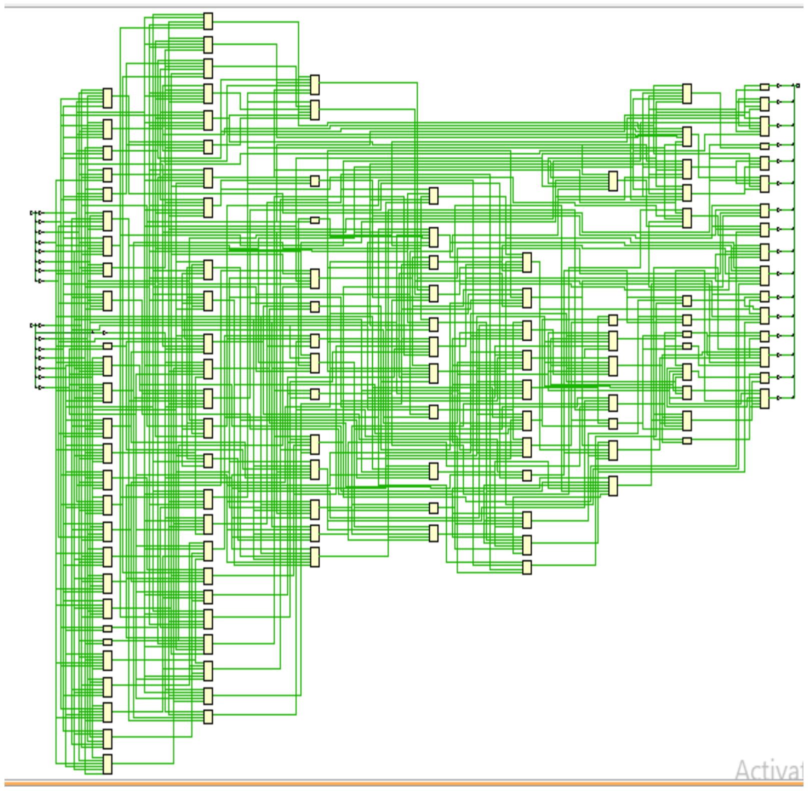
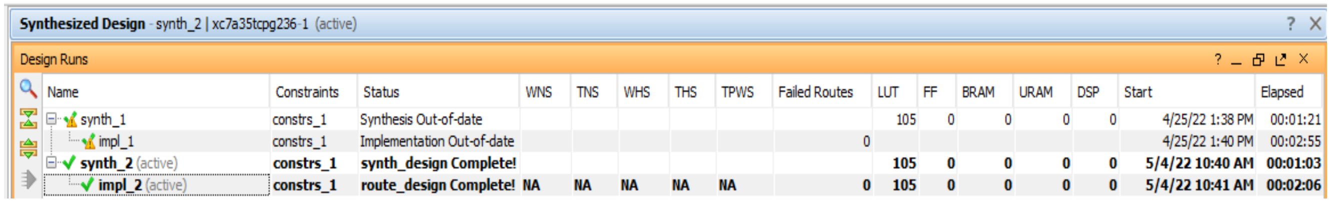


Fig 24. Synthesized schematic diagram

c) Area Utilization Report

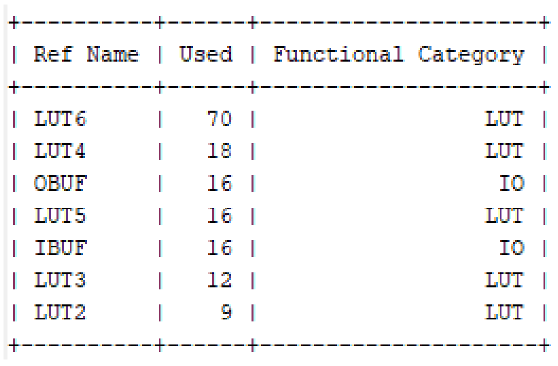


Synthesized Design - synth_2 | xc7a35tcpg236-1 (active)

Design Runs

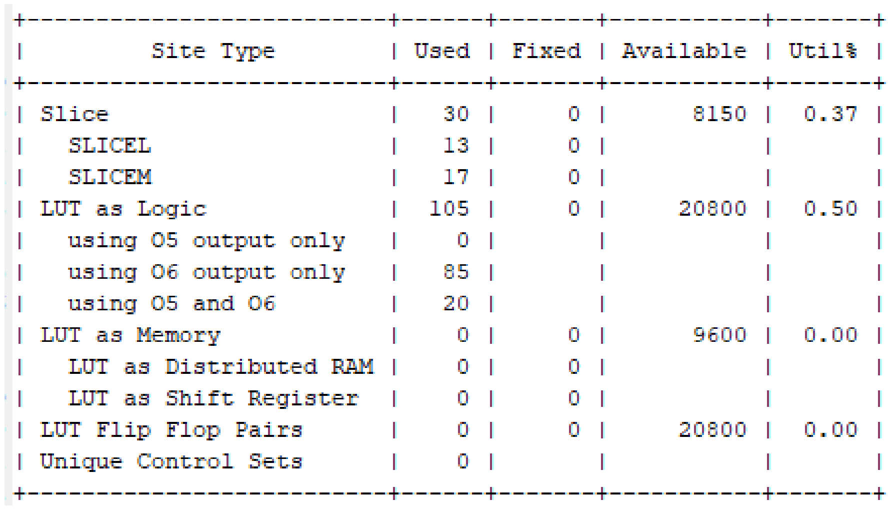
Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Failed Routes	LUT	FF	BRAM	URAM	DSP	Start	Elapsed
synth_1	constrs_1	Synthesis Out-of-date							105	0	0	0	0	4/25/22 1:38 PM	00:01:21
impl_1	constrs_1	Implementation Out-of-date						0						4/25/22 1:40 PM	00:02:55
synth_2 (active)	**constrs_1**	**synth_design Complete!**							**105**	**0**	**0**	**0**	**0**	**5/4/22 10:40 AM**	**00:01:03**
impl_2 (active)	**constrs_1**	**route_design Complete!**	**NA**	**NA**	**NA**	**NA**	**NA**	**0**	**105**	**0**	**0**	**0**	**0**	**5/4/22 10:41 AM**	**00:02:06**

***Fig 25.** Design synthesize status*



```
+----------+------+---------------------+
| Ref Name | Used | Functional Category |
+----------+------+---------------------+
| LUT6     |   70 |                 LUT |
| LUT4     |   18 |                 LUT |
| OBUF     |   16 |                  IO |
| LUT5     |   16 |                 LUT |
| IBUF     |   16 |                  IO |
| LUT3     |   12 |                 LUT |
| LUT2     |    9 |                 LUT |
+----------+------+---------------------+
```

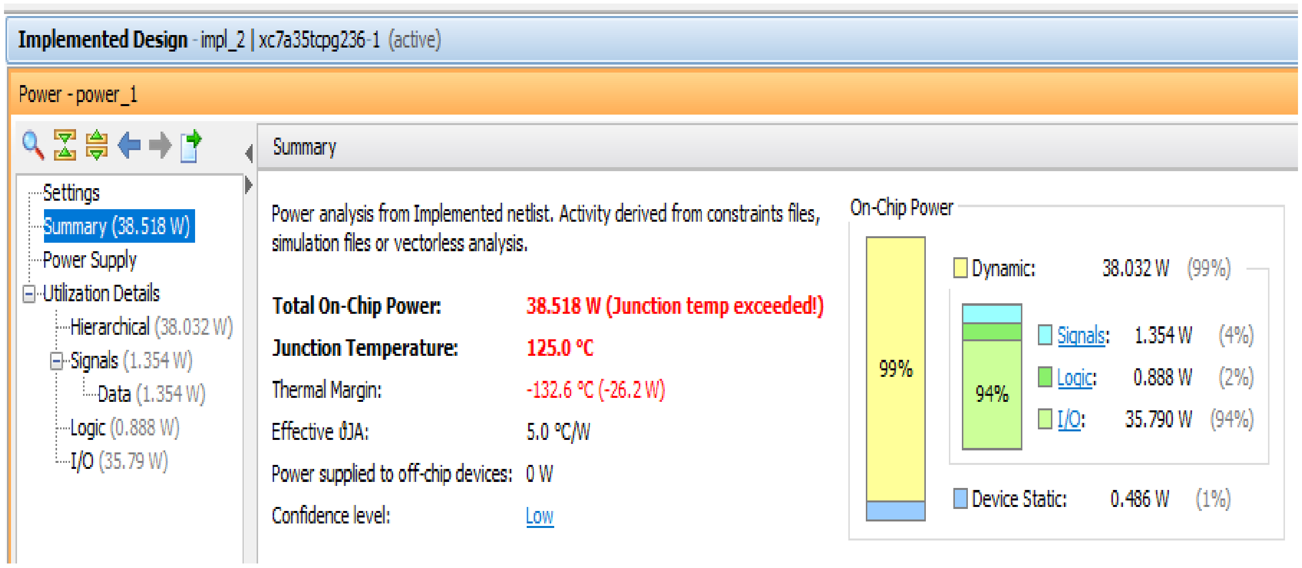
Fig 26. LUT table



```
+-------------------------+------+-------+-----------+-------+
|        Site Type        | Used | Fixed | Available | Util% |
+-------------------------+------+-------+-----------+-------+
| Slice                   |   30 |     0 |      8150 |  0.37 |
|   SLICEL                |   13 |     0 |           |       |
|   SLICEM                |   17 |     0 |           |       |
| LUT as Logic            |  105 |     0 |     20800 |  0.50 |
|   using O5 output only  |    0 |       |           |       |
|   using O6 output only  |   85 |       |           |       |
|   using O5 and O6       |   20 |       |           |       |
| LUT as Memory           |    0 |     0 |      9600 |  0.00 |
|   LUT as Distributed RAM|    0 |     0 |           |       |
|   LUT as Shift Register |    0 |     0 |           |       |
| LUT Flip Flop Pairs     |    0 |     0 |     20800 |  0.00 |
| Unique Control Sets     |    0 |       |           |       |
+-------------------------+------+-------+-----------+-------+
```

Fig 27. Logic and memory table

d) Power Report



Implemented Design - impl_2 | xc7a35tcpg236-1 (active)

Power - power_1

Summary

- Settings
- Summary (38.518 W)
- Power Supply
- Utilization Details
  - Hierarchical (38.032 W)
  - Signals (1.354 W)
    - Data (1.354 W)
  - Logic (0.888 W)
  - I/O (35.79 W)

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

Total On-Chip Power:	**38.518 W (Junction temp exceeded!)**
Junction Temperature:	**125.0 °C**
Thermal Margin:	-132.6 °C (-26.2 W)
Effective ϑJA:	5.0 °C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

On-Chip Power

Dynamic:	38.032 W	(99%)	99%
Signals:	1.354 W	(4%)	
Logic:	0.888 W	(2%)	
I/O:	35.790 W	(94%)	94%
Device Static:	0.486 W	(1%)	

Fig 28. Power report

B) Method 2

a) RTL Diagram

Register Transfer Level (RTL) is an abstraction for defining the digital portions of a design. The figure below represents the RTL diagram of 8 bit multiplier designed using method 2.

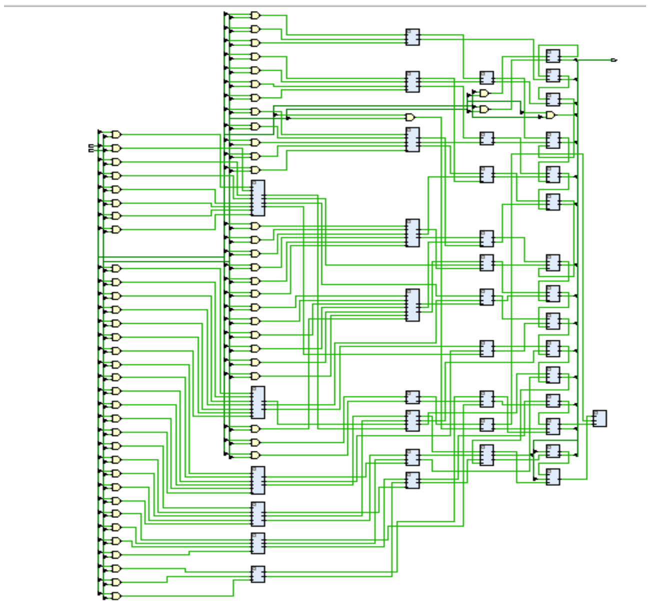


Fig 29. RTL diagram

b) Synthesized Schematic Diagram:

The Synthesized Schematic diagram for the above RTL diagram is as shown. It represents how the actual components are connected on the artix-7 Fpga board.

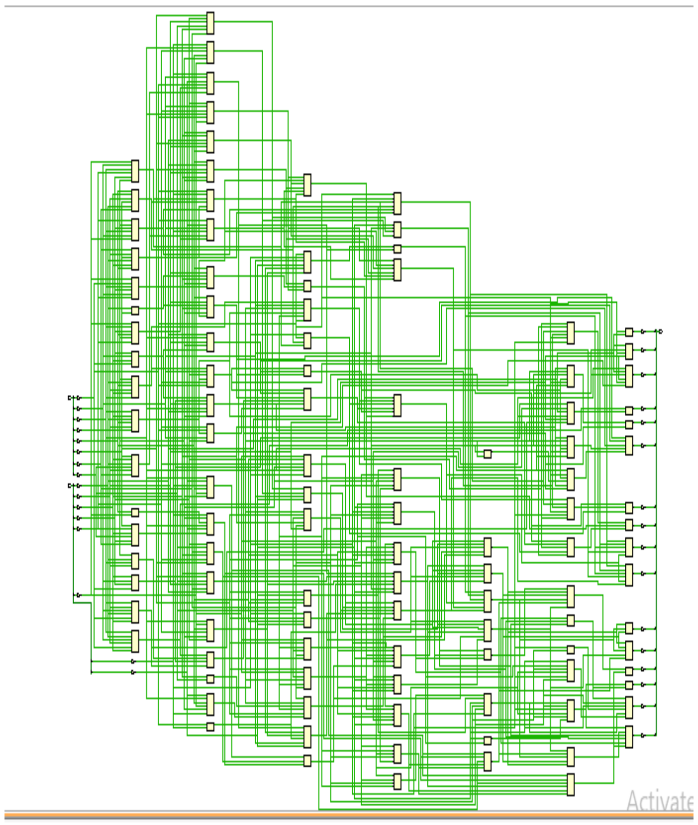
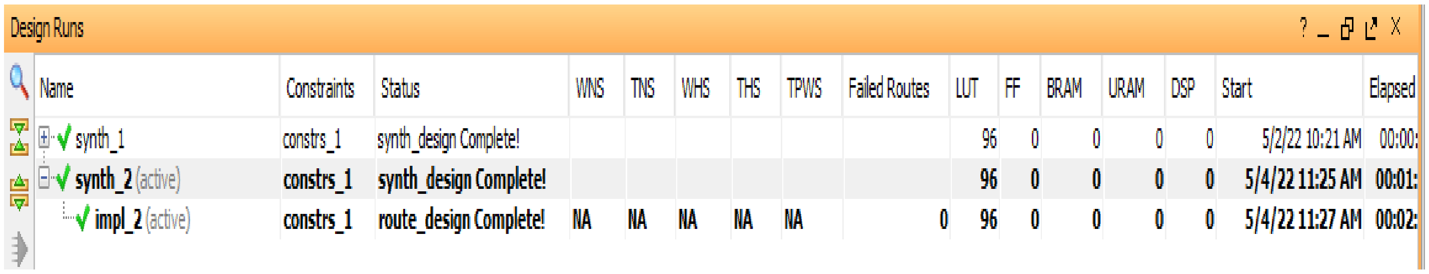


Fig 30. Synthesized schematic diagram

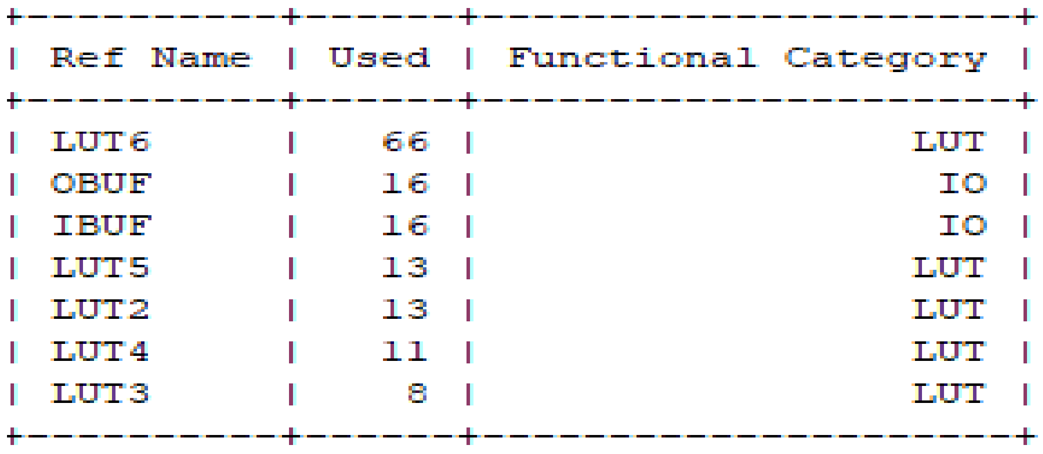
c) Area Utilization Report



Design Runs

Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Failed Routes	LUT	FF	BRAM	URAM	DSP	Start	Elapsed
synth_1	constrs_1	synth_design Complete!							96	0	0	0	0	5/2/22 10:21 AM	00:00:
synth_2 (active)	constrs_1	synth_design Complete!							96	0	0	0	0	5/4/22 11:25 AM	00:01:
impl_2 (active)	constrs_1	route_design Complete!	NA	NA	NA	NA	NA	0	96	0	0	0	0	5/4/22 11:27 AM	00:02:

Fig 31. Design synthesize status

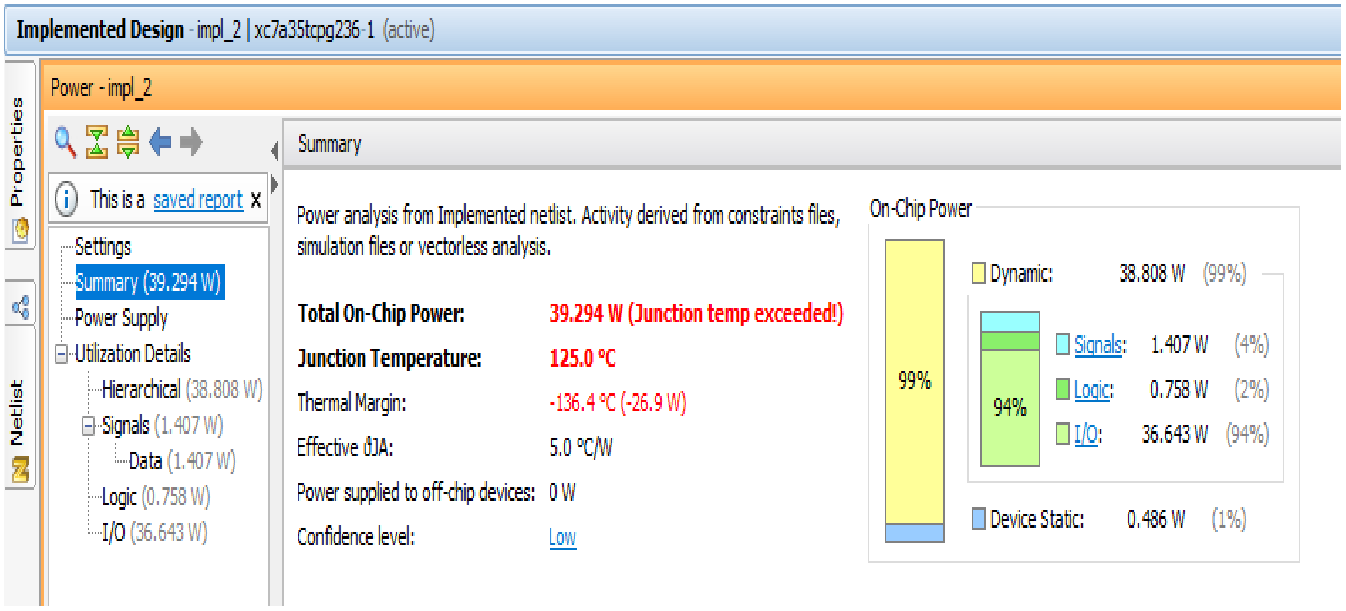


```
+-----------+-------+----------------------+
| Ref Name  | Used  | Functional Category  |
+-----------+-------+----------------------+
| LUT6      |    66 |                  LUT |
| OBUF      |    16 |                   IO |
| IBUF      |    16 |                   IO |
| LUT5      |    13 |                  LUT |
| LUT2      |    13 |                  LUT |
| LUT4      |    11 |                  LUT |
| LUT3      |     8 |                  LUT |
+-----------+-------+----------------------+
```

Fig 32. LUT table

d) Power Report

The following figure represents the Power analysis from Implemented netlist.



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

Total On-Chip Power:	39.294 W (Junction temp exceeded!)
Junction Temperature:	125.0 °C
Thermal Margin:	-136.4 °C (-26.9 W)
Effective ϑJA:	5.0 °C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

On-Chip Power

Dynamic:	38.808 W	(99%)
Signals:	1.407 W	(4%)
Logic:	0.758 W	(2%)
I/O:	36.643 W	(94%)
Device Static:	0.486 W	(1%)

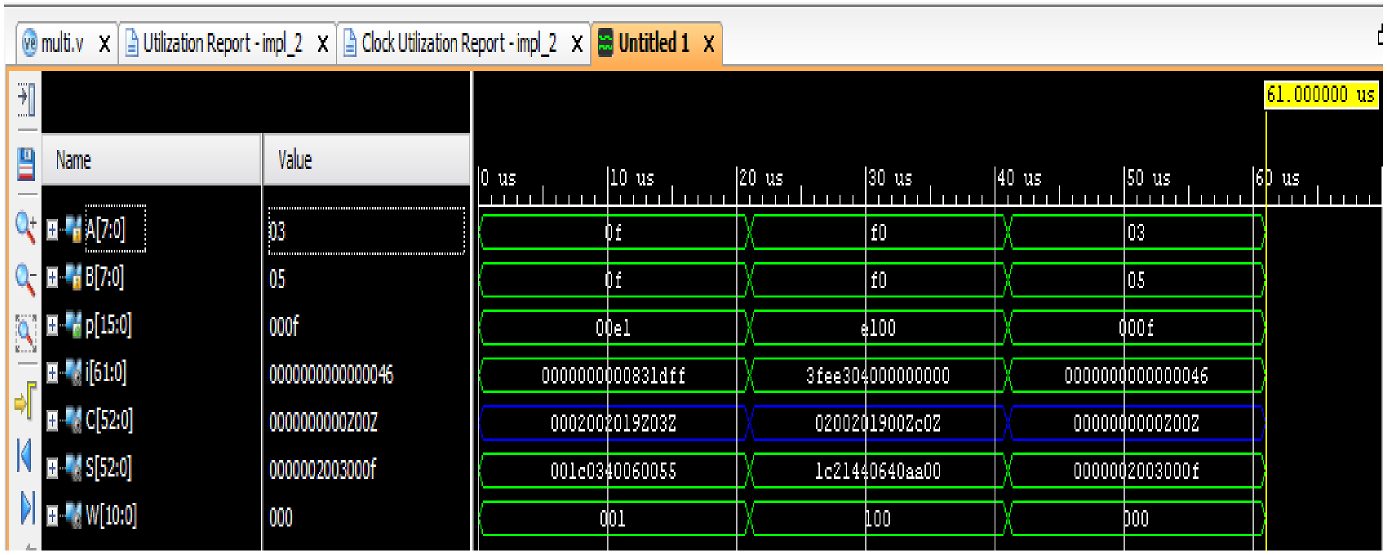
Fig 33. Power report

6.3 Theoretical calculation: from fig 21 we have taken three different binary values for multiplication

A = 00000011; B = 00000101

A*B = 00001111

6.4 Simulation Outputs

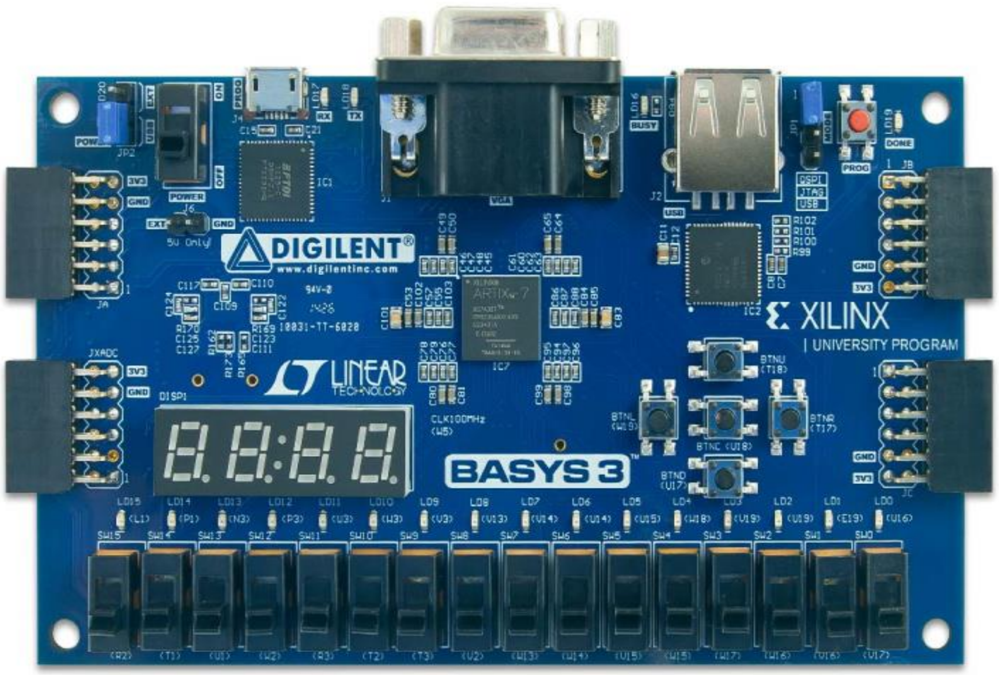


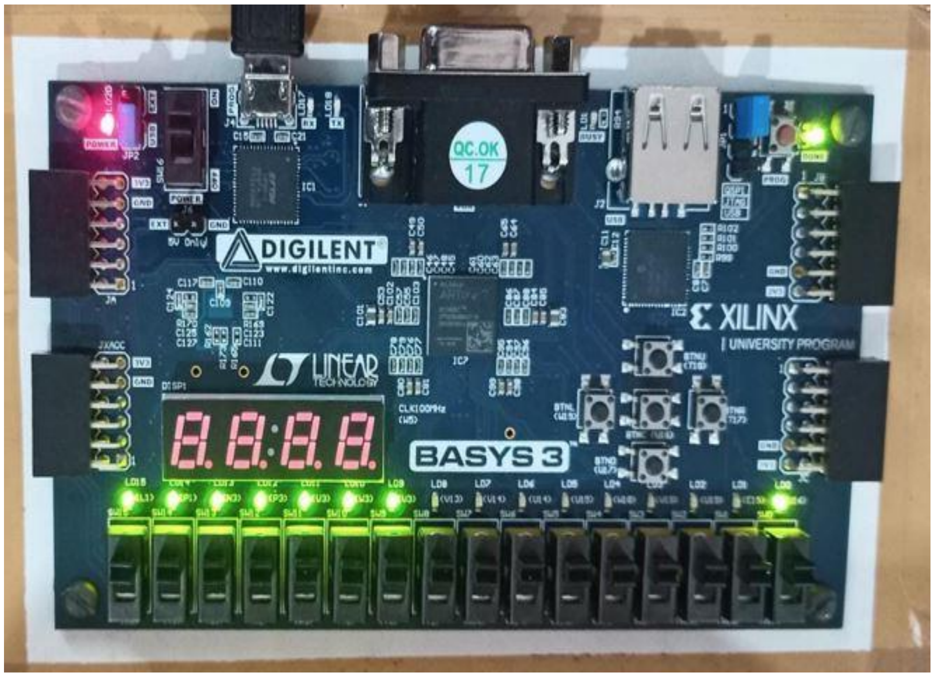
```
Name      Value              0 us ...             20 us ...            40 us ... 61 us
A[7:0]    03                 0f                   f0                   03
B[7:0]    05                 0f                   f0                   05
p[15:0]   000f               00e1                 e100                 000f
i[61:0]   0000000000000046   0000000000831dff     3fee304000000000     0000000000000046
C[52:0]   00000000002002     00020020192032       02002019002c02       00000000002002
S[52:0]   0000002003000f     001c0340060055       1c21440640aa00       0000002003000f
W[10:0]   000                001                  100                  000
```

Fig 34. Simulation output results

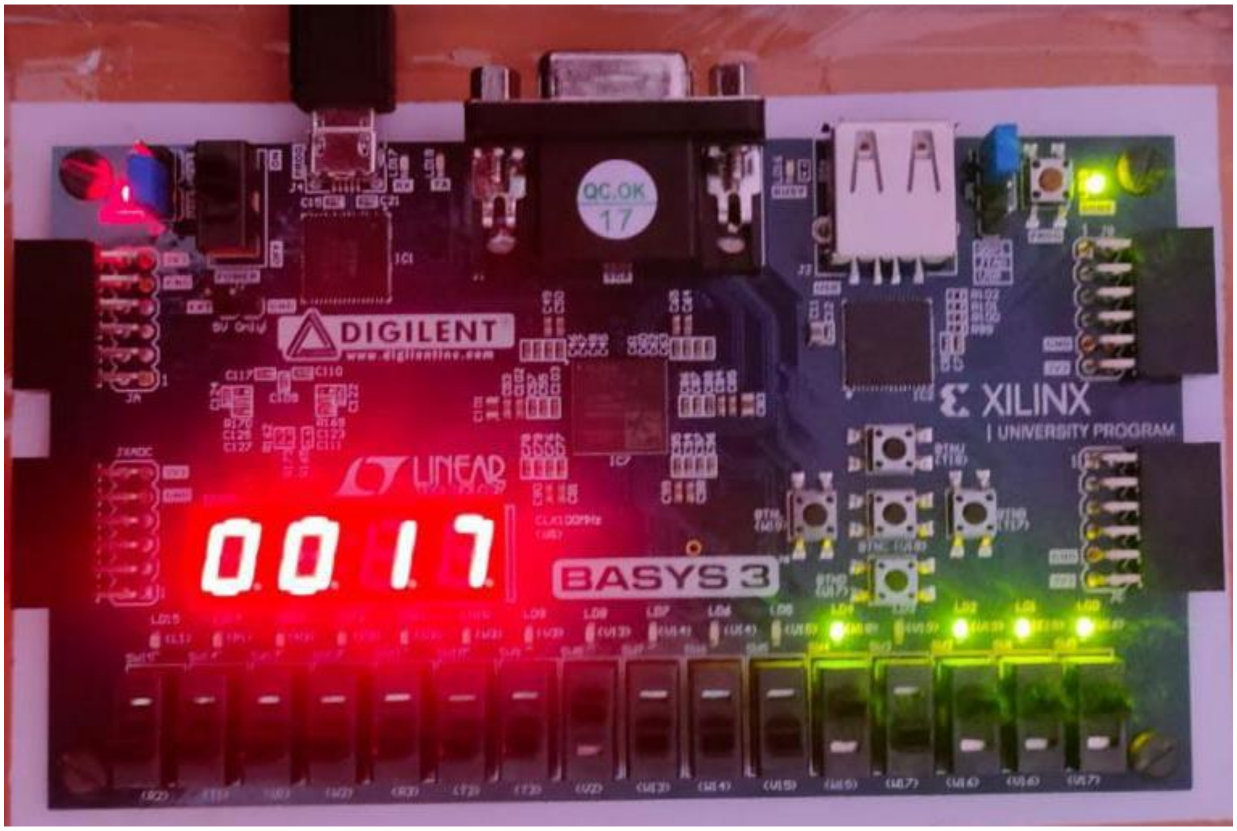
6.4 Hardware Implementation

In this work, all proposed models are implemented on Basys3 Artix7 FPGA with the help of Xilinx Vivado. For implementation first we have to map original input and output ports with board I/Os by generating constraint file. Next, the program can be dumped to FPGA kit to verify the simulation results practically.





INTERFERENCE 7 SEGMENT DISPLAY



Here multiplication between 1 and 23 is done and the output is 23 which is represented as 00010111 in binary format which can be observed using LEDs as shown in the figure. Additionally the resultant output is displayed on seven segment display which represents the hexadecimal equivalent of 23 which is nothing but 17.

CONCLUSION

In digital signal processing , multiplication is a key operation which determines the performance of the multiplier. Speed and area requirement of multiplier plays major part in the performance of multiplier. In this paper, We designed binary 8 bit multiplier using higher order compressors and adders using two methods and compared both methods such that area occupied in terms of LUTs is reduced.

REFERENCES

[1] D. Liu Embedded DSP Processor Design, 1st ed.Morgan KaufmannPublishing, 2008.

[2] K. K Parhi VLSI Digital Signal Processing Systems:Design and Implementation., 1st ed.John Wiley and Sons, 1999.

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

[4] A. Pishvaie, G. Jaberipur, and A. Jahanian, ”Improved CMOS (4; 2) compressor designs for parallel multipliers,” Computers and Electrical Engineering, vol. 38, no. 6, pp. 17031716, Nov. 2012.

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

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

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

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

[9] O. Kwan, K. Nawka, and E. Swartzlander Jr, ” A 16 bit by 16 bit MAC Design Using Fast 5:3 Compressor Cells,” Journal of VLSI Signal Processing, vol. 31, no. 2, pp. 77-89, July 2002.

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

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

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

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

[14] N.Zhu, W L Goh, and Kiat Seng Yeo, ”An enhanced low power high- speed adder for error tolerant application,” in proc. of 12 th International Symposium on Integrated Circuits (ISIC), Nov. 2009, pp. 69 - 72.

[15] Z. Yang, A. Jain, J. Liang, J. Han, and F. Lombardi, ”Approximate XOR/XNOR-based adders for inexact computing,” in proc. of IEEE International Conference on Nanotechnology (IEEE - NANO), Aug. 2013,pp. 690 - 693.

[16] H. Jiang, J.Han, and F. Lombardi, ”A comparative review and evaluation of approximate adders,” in proc. of ACM Great Lakes Symposium on VLSI(GLSVLSI), May. 2015, pp. 343 - 348.

[17] V. Gupta, D. Mohapatra, S. P. Park, A. Raghunathan, and K. Roy,”IMPACT: IMPrecise adders for low-power approximate computing,” in proc. of International Symposium on Low Power Electronics and Design (ISLPED), Aug. 2011, pp. 409 - 414.

[18] C. Liu, J. Han, and F. Lombardi, ”A Low-Power, High-Performance Approximate Multiplier with Configurable Partial Error Recovery,” in proc. of International Conference on Design Automation and Test in Europe (TEST), Mar. 2014.

[19] P. Kulkarni, P. Gupta, and M. D. Ercegovac, ”Trading accuracy for power in a multiplier architecture,” in Journal of Low Power Electronics, vol. 7, no. 4, pp. 490501, Dec. 2011.

[20] S. Balamurugan,and P.S Mallick, ”Fixed-width multiplier circuits using column bypassing and decompositon logic techniques,” in International journal on Electrical Engineering and Informatics, vol. 7, no. 4, pp. 655664, Dec. 2015.

[21] S. Balamurugan,S. Ghosh,Atul, S. Balakumaran, R. Marimuthu,and P.S Mallick, ”Design of low power fixed-width multiplier with row bypassing,” in IEICE Electronics Express, vol. 9, no. 20, pp. 15681575,Oct. 2012.

[22] H.R. Mahdiani, A. Ahmadi, S.M. Fakhraie, and C. Lucas, ”Bio-Inspired imprecise computational blocks for efficient VLSI implementation of soft-computing applications,” in IEEE Transactions on Circuits and Systems,vol. 57, no. 4, pp. 655664, Apr. 2010.

[23] K.Y. Kyaw, W.L. Goh, and K.S. Yeo, ”Low-power high-speed multiplier for error-tolerant application,” in proc. of IEEE International Conference of Electron Devices and Solid-State Circuits (EDSSC), Dec. 2010, pp. 1- 4.

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

[25] M.S.K Lau, K.V. Ling, and Y.C. Chu, ”Energy-aware probabilistic multiplier: design and analysis,” in proc. of international conference on Compilers, architecture, and synthesis for embedded systems, Oct. 2009, pp. 281- 290.

[26] R. Venkatesan, A. Agarwal, K. Roy, and A. Raghunathan, ”MACACO:Modeling and analysis of circuits for approximate computing,” in proc.of international conference on Compilers, architecture, and synthesis for embedded systems (ICCAD), Nov. 2011, pp. 667 - 673.

[27] F. Farshchi, M.S.Abrishami, and S.M. Fakhraie, ”New approximate multiplier for low power digital signal processing,” in proc. of 17th international Symposium on Computer Architecture and Digital Systems (CADS), Oct. 2013, pp. 25 - 30.

[28] H. Jiang, C. Liu, N. Maheshwari, F. Lombardi, and J. Han, ”A Comparative Evaluation of Approximate Multipliers,” in proc. of In International Symposium on Nanoscale Architectures (NANOARCH), Jul. 2016, pp. 191- 196.

[29] C.-H. Lin, and I.-C. Lin, ”High accuracy approximate multiplier with error correction,” in proc. of IEEE 31st International Conference on Computer Design (ICCD), Oct. 2013, pp. 33 - 38.

[30] Y. Bansal, and C. Madhu, ”A novel high-speed approach for 16x16 vedic multiplication with compressor adders,” Computers and Electrical Engineering, vol. 49, pp. 39-49, Jan. 2016.