Efficient Approximate Adders with Minimal Error for FPGA Implementation

A Project report submitted in partial fulfillment of the requirements for the Award of the Degree of

BACHELOR OF TECHNOLOGY

In

ELECTRONICS & COMMUNICATION ENGINEERING

Submitted By

A. SHANMUKHA POORNA CHAND	**21KT5A0410**
K. KEERTHI	**21KT5A0419**
B. AVINASH	**20KT1A04D7**
MD. AHMED ASHRAF	**20KT1A04F7**

Under The Guidance of

K. RAGHAVENDRA RAO **M. Tech**

Assistant Professor

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

(NBA & NAAC ACCREDITED)

POTTI SRIRAMULU CHALAVADI MALLIKARJUNA RAO COLLEGE OF ENGINEERING & TECHNOLOGY, KOTHAPET, VIJAYAWADA 520001

(Affiliated to JNTU Kakinada, Approved by AICTE - New Delhi)

2023-2024

POTTI SRIRAMULU CHALAVADI MALLIKARJUNA RAO
COLLEGE OF ENGINEERING & TECHNOLOGY
SPONSORED BY SKPVV HINDU HIGH SCHOOLS COMMITTEE, Estd : 1906
APPROVED BY AICTE, NEW DELHI, AFFILIATED TO JNTU KAKINADA
ACCREDITED BY NAAC, NBA for B.TECH PROGRAMS in CSE, ECE & EEE, ISO 9001:2015 CERTIFIED
7-3-6/1, RAGHAVA REDDY STREET, ONE TOWN, VIJAYAWADA - 520001
CONTACT : +91 -866 - 2423442 / E-MAIL : principal@pscmr.ac.in / info@pscmr.ac.in
COLLEGE CODE : PSCV [EAMCET, ECET, ICET, POLYCET]

DEPARTMENT

OF

ELECTRONICS AND COMMUNICATION ENGINEERING

CERTIFICATE

This is to certify that the project report entitled **"Efficient Approximate Adders with Minimal Error for FPGA Implementation"** being submitted by **A. Shanmukha Poorna Chand (21KT5A0410), K. Keerthi (21KT5A0419), B. Avinash (20KT1A04D7), and MD. Ahemad Ashraf (20KT1A04F7)** in partial fulfillment of the requirements for the award of the degree of BACHELOR OF TECHNOLOGY to the Jawaharlal Nehru Technological University, Kakinada is a record of bonafide work carried out under my guidance and supervision. The results embodied in this project report have not been submitted to any other university or institution for the award of any degree or diploma.

Project Guide
K. RAGHAVENDRA RAO

Head of the department
Mrs. A.V. Kiranmai

External Examiner

DECLARATION

We certify that the work contained in this report is original and has been done by us under the guidance of my supervisor. The work has not been submitted to any other Institute for any degree or diploma. We have followed the guidelines provided by the Institute in preparing the report. We have conformed to the norms and guidelines given in the Ethical Code of Conduct of the Institute. Whenever we have used materials (data, theoretical analysis, figures, and text) from other sources, we have given due credit to them by citing them in the text of the report and giving their details in the references. Further, we have taken permission from the copyright owners of the sources, whenever necessary.

Names of Project Members

A. SHANMUKHA POORNA CHAND	**21KT5A0410**
K. KEERTHI	**21KT5A0419**
B. AVINASH	**20KT1A04D7**
MD. AHEMAD ASHRAF	**20KT1A04F7**

ACKNOWLEDGEMENT

We are tremendously obliged to our guide **Mr. K. RAGHAVENDRA RAO**, Assistant Professor, Department of Electronics and Communication Engineering, **POTTI SRIRAMULU CHALAVADI MALLIKARJUNA RAO COLLEGE OF ENGINEERING & TECHNOLOGY** for his patient guidance, invaluable advice and selfless help rendered all through the progress of our work.

We sincerely thank the Hon'ble Secretary & correspondent **Sri. P. LAKSHMANA SWAMY**, for providing us with the good facilities in our college for carrying out the project successfully, and to our Principal **Prof. Dr. J. LAKSHMI NARAYANA** for their kind encouragement and their genial cooperation.

We are extremely grateful to thank our Head of the Department **Mrs. A.V. Kiranmai**, for her kind encouragement and extreme help through our thesis and we like to thank **all our teaching and non-teaching staff** of the Department of E.C.E.

We are thankful to all editors and reviewers of our papers published for their constructive suggestions. We thank all office staff, friends, and all other technical staff of the department of E.C.E.

LIST OF CONTENTS

ABSTRACT I

LIST OF FIGURES II

LIST OF TABLES IV

LIST OF ABBREVIATIONS V

CHAPTER 1. INTRODUCTION **1-8**

1.1 Objective 7

CHAPTER 2. LITERATURE SURVEY **9-14**

CHAPTER 3. EXISTING METHOD **15-20**

3.1 Half Adder 15

3.2 Full Adder 16

3.3 Ripple Carry Adder 16

3.4 Parallel Prefix Adder 18

3.5 Lower-Part-OR Adder 19

3.6 Disadvantages 20

CHAPTER 4. PROPOSED METHOD **21-26**

4.1 Proposed Approximate Adder 22

4.2 Proposed LEADx 24

4.3 Proposed APEx 24

4.4 Advantages and Applications 26

4.4.1 Advantages 26

4.4.2 Applications 26

CHAPTER 5. SOFTWARE REQUIREMENTS **27-50**

5.1 History of Verilog 27

5.2 Introduction to HDL 28

5.2.1 Design Styles 28

5.2.2 Bottom-Up Design 28

5.2.3 Top-Down Design 29

5.2.4 Features of Verilog 29

5.3 VLSI Design Flow 29

5.3.1 System Specification 30

5.3.2 Architectural Design 31

5.3.3 Behavioural or Functional Design 31

5.3.4 Logic Design 31

5.3.5 Circuit Design 31

5.3.6 Physical Design 32

5.3.7 Layout Verification 32

5.3.8 Fabrication and Testing 32

5.4 Module 32

5.4.1 Instances 33

5.4.2 Ports 33

5.4.3 Identifiers 33

5.4.4 Keylines 34

5.5 Data Types 34

5.5.1 Register Data Types 34

5.6 Modelling Concepts 35

5.6.1 Abstraction Levels 35

5.6.1.1 Behavioural or algorithmic Level 35

5.6.1.2 Register-Transfer Level 36

5.6.1.3 Gate Level 36

5.7 Operators 36

5.7.1 Arithmetic Operators 37

5.7.2 Relational Operators 37

5.7.3 Bit-Wise Operators 38

5.7.4 Logical Operators 39

5.7.5 Reduction Operators 39

5.7.6 Shift Operators 40

5.7.7 Concatenation Operators 41

5.7.8 Operator Precedence 41

5.7.9 Switch Level 41

5.8 Xilinx Verilog HDL Tutorial 42

5.8.1 Getting Started 42

5.8.2 Introduction to Xilinx 42

5.8.3 Programmable Logic Device: FPGA 43

5.8.4 Creating a New Project 43

5.8.5 Opening Designs 45

5.8.6 Editing The Verilog Source File 47

5.8.7 Configuring Project Settings 47

5.8.8 Synthesis and Implementation of the Design 48

5.9 Xilinx Vivado Simulation Procedure 48

5.9.1 Using the Schematic Window 49

5.9.2 Using the Project Summary 50

CHAPTER 6. RESULTS **51-57**

6.1 Multiplexer 51

6.2 Lookup Table 51

6.3 AAD2 52

6.4 Carry chain 52

6.5 Output buffer 53

6.6 VCC 53

6.7 Ground 54

6.8 LEADx 54

6.9 APEX 57

CHAPTER 7. CONCLUSION AND REFERENCES 61

ABSTRACT

In this project, a methodology for designing low error efficient approximate adders has been proposed. The proposed methodology utilizes FPGA resources efficiently to reduce the error of approximate adders. We propose two approximate adders for FPGAs using our methodology: low error and area efficient approximate adder (LEADx), and area and power efficient approximate adder (APEx).

Both approximate adders are composed of an accurate and an approximate part. The approximate parts of these adders are designed in a systematic way to minimize the mean square error (MSE). LEADx has lower MSE than the approximate adders in the literature.

APEx has smaller area and lower power consumption than the other approximate adders than the existing adders. As a case study, the approximate adders are used in video encoding application. LEADx provided better quality than the other approximate adders for video encoding application. Therefore, our proposed approximate adders can be used for efficient FPGA implementations of error tolerant applications. The effectiveness of the proposed method is synthesized and simulated using Xilinx ISE 14.7.

Keywords: - Approximate computing, approximate adder, FPGA, low error, low power, LUT

LIST OF FIGURES

FIGURE NO	FIGURE NAMES	PAGE NO
1.1	Generalized approximate adder	6
3.1	Half adder	15
3.2	Half adder truth Table	15
3.3	Full adder	16
3.4	Ripple carry adder	17
3.5	Generalized PPA	18
3.6	LOA approximate adder	20
4.1	Proposed approximate adder	22
4.2	Approximate adder 1	23
4.3	Truth table of proposed AAd2	23
4.4	LEADx	24
4.5	APEX	25
5.1	VLSI Design Flow	30
5.2	New Project Wizard—Project Type Page	44
5.3	Implemented design	45
5.4	Settings Dialog Box—Project Settings General Category	47
5.5	Schematic Window	49
5.6	Project Summary	50
6.1	Multiplexer	51
6.2	Lookup table	51
6.3	AAD2	52
6.4	Carry chain	52
6.5	Output buffer	53
6.6	VCC	53

6.7	Ground	54
6.8	RTL Schematic of LEADx	55
6.9	Simulation Results of LEADx	56
6.10	Area consumed by LEADx	56
6.11	Delay Produced by LEADx	56
6.12	Power Consumption of LEADx	57
6.13	RTL Schematic of APEX	58
6.14	Simulation Results of APEX	59
6.15	Area consumed by APEX	59
6.16	Delay Produced by APEX	60
6.17	Power Consumption of APEX	60

LIST OF TABLES

TABLE NO.	TABLE TITLE	PAGE NO
5.1	Relational Operators	37
5.2	Bitwise Operators	38
5.3	Logical Operators	39
5.4	Reduction Operators	40
5.5	Shift Operators	40
5.6	Operator precedence	41
6.1	Evaluation Table	60

LIST OF ABBREVIATIONS

Abbreviations	Description
FPGA	Field Programmable Gate Array
LEADx	Low error and area efficient approximate adder
APEX	Area and power efficient approximate adder
DSP	Digital Signal Processing
ANT	Algorithmic Noise Tolerance
SDC	Significance Driven Computation
VOS	Voltage Over Scaling
CLB	Configurable Logic Block
LUT	Look Up Table
LSP	Least Significant Part
MSP	Most Significant Part
MSE	Mean Square Error
MCM	Multiple Constant Multiplication
HEVC	High Efficiency Video Encoding
LOA	Lower Part of OR Adder
CLA	Carry Look Ahead Adder
ISH	Internal Self-Healing
RTL	Register Transfer Level
OVI	Open Verilog International
MAS	Micro- Architectural Specification
HDL	Hardware Description Language
CISC	Complex Instruction Set Computer
RISC	Reduced Instruction Set Computer
CPLD	Complex Programmable Logic Device

CHAPTER-I

INTRODUCTION

Approximate computing is a new design technique that trades off accuracy for performance, area and/or power consumption for error-tolerant applications such as video coding. The video compression is error-tolerant in nature since the only requirement is to produce output that has sufficient quality to provide good user experience. Therefore, approximate computing has a huge potential to improve the performance, area and/or power consumption of hardware implementations.

FPGAs serve as an excellent platform for a wide range of applications from small-scale embedded devices to high-performance computing systems due to their short time-to-market, enhanced flexibility and run-time re-configurability. However, despite supporting specialized hardware accelerators and co-processors, FPGA-based systems typically consume more power and/or energy, compared to their ASIC counterparts. Therefore, besides employing traditional energy optimization techniques, there is a need for exploring new avenues in energy-efficient computing solely for FPGA-based systems. One such attractive trend is the Approximate Computing paradigm, which is re-emerging due to the breakdown of Moore's law and Dennard scaling, and the ever-increasing demand for high-performance and energy efficiency [2].

Approximate computing trades the accuracy and precision of intermediate or final computations to achieve significant gains in critical path delay, area, power and/or energy consumption. This trade-off becomes beneficial for applications exhibiting inherent application resilience, i.e., the ability to produce viable output despite some of its computations being inaccurate because of approximations. A wide range of applications like image and video processing, data mining, machine learning, etc., in the recognition, mining and synthesis domains exhibit this property.

Existing approximate computing techniques and principles can be applied to different stages of the computing stack, ranging from logic and architectures at the hardware layer all the way up to compiler and programming language at the software layer. There is an extensive amount of research related to approximations at both hardware and software layers. Voltage over-scaling and functional approximation are the two major approximate computing knobs employed at the hardware level [4,8,10].

Approximations at the software level can be classified into two major categories:

(i) Loop perforation, function approximation

(ii) Programming language support.

Approximations at the hardware level are focused on basic computation modules like adders and multipliers. Research works like focus on modeling the error probability of the existing state of-the-art ASIC-based adders and recursive multiplier architectures.

More recent works focus on architecture-level approximations targeting application-specific domains like video processing to achieve energy efficiency. Major industrial players like Intel, IBM and Microsoft have also explored the approximate computing paradigm, and have demonstrated case studies on the design of energy-efficient hardware and software systems using approximation techniques. There has been a lot of research in the field of approximate computing, focusing mostly on ASIC-based systems. However, due to the underlying architectural differences between ASICs and FPGAs, these approximate computing principles are not directly applicable to FPGAs for achieving similar gains.

DIGITAL SIGNAL processing (DSP) blocks form the backbone of various multimedia applications used in portable devices. Most of these DSP blocks implement image and video processing algorithms, where the ultimate output is either an image or a video for human consumption.

Human beings have limited perceptual abilities when interpreting an image or a video. This allows the outputs of these algorithms to be numerically approximate rather than accurate. This relaxation on numerical exactness provides some freedom to carry out imprecise or approximate computation [5].

We can use this freedom to come up with low-power designs at different levels of design abstraction, namely, logic, architecture, and algorithm. The paradigm of approximate computing is specific to select hardware implementations of DSP blocks. An embedded reduced instruction set computing processor consumes 70% of the energy in supplying data and instructions, and 6% of the energy while performing arithmetic only. Therefore, using approximate arithmetic in such a scenario will not provide much energy benefit when considering the complete processor.

Programmable processors are designed for general-purpose applications with no application-specific specialization. Therefore, there may not be many applications that will be able to tolerate errors due to approximate computing. This also makes general-purpose processors not suited for using approximate building blocks. Therefore, in this project, we consider application-specific integrated circuit implementations of error-resilient applications like image and video compression.

We target the most computationally intensive blocks in these applications and build them using approximate hardware to show substantial improvements in power consumption with little loss in output quality. Few works that focus on low-power design through approximate computing at the algorithm and architecture levels include algorithmic noise tolerance (ANT), significance driven computation (SDC), and non-uniform voltage over-scaling (VOS).

All these techniques are based on the central concept of VOS, coupled with additional circuitry for correcting or limiting the resulting errors. In a

fast but “inaccurate” adder is implemented in literature which is based on the idea that on average, the length of the longest sequence of propagate signals is approximately log n, where n is the bit-width of the two integers to be added. An error-tolerant adder that operates by splitting the input operands into accurate and inaccurate parts. However, neither of these techniques target logic complexity reduction.

A power-efficient multiplier architecture that uses a 2 × 2 inaccurate multiplier block resulting from Karnaugh map simplification. This paper considers logic complexity reduction using Karnaugh maps. Shin and Gupta and Phillips et al. also proposed logic complexity reduction by Karnaugh map simplification. Other works that focus on logic complexity reduction at the gate level. Other approaches use complexity reduction at the algorithm level to meet real-time energy constraints.

Proportional reductions in area, power and latency are not achieved as these approximations target transistor or gate-level truncations, leading to significant efficiency gains in ASICs but not in FPGAs. The most important factor in distinguishing ASICs and FPGAs is the way logic functions are realized. The basic building blocks for generating the required logic, in case of ASICs, are the logic gates, whereas in case of FPGAs, they are the lookup tables (LUTs) made of SRAM elements. Therefore, the approximations techniques for FPGAs should be amenable to the LUT structures instead of aiming at reducing the logic gates.

The basic building block of an FPGA are the Configurable logic blocks or CLBs. They are used to implement any kind of logic function using the switching/routing matrix. Each CLB consists of two slices FPGA family arranges all the CLBs in columns by deploying Advanced Silicon Modular Block (ASMBL) architecture. Each slice in this device consists of four 6-input LUTs, eight flip-flops and an efficient carry chain logic. The slices act

as the main function generators of any FPGA and in the Virtex-7 family they are categorized as SLICEL or logic slices, and SLICEM or memory slices.

The lookup tables present in these slices are 5x2 LUTs. This LUT6_2 is fabricated using two LUT5s and a multiplexer. These LUT5s are the basic SRAM elements which are used to realize the required logic function, by storing the output sequence of the truth-table in 1-bit memory locations, which are accessed using the address lines acting as inputs. These LUTs are made accessible using a wide range of lookup table primitives offered by the Xilinx UNISIM library, ranging from LUT1 to LUT7. These LUT primitives are instantiated with an INIT attribute, which is the truth table of the function required based on the input logic.

The LUT primitives are used to implement the required logic function which are then compacted and mapped onto the fabric resources available on the FPGA. Each of these LUT primitives take in an equivalent number of 1-bit inputs, and produce a unique 1-bit output. However, at the hardware level, each of these primitives are physically mapped to one of the two LUT5s present in the four LUT6_2 fabrics in a given slice of the CLB.

As per studies, the use of LUT primitives allows for Xilinx to efficiently optimize the combining and mapping of LUT primitives to reduce the area and latency of the synthesized designs. We use these LUT primitives to achieve significant area and performance gains for the approximate adder designs [2,3,5.

Approximate computing trades off accuracy to improve the area, power, and speed of digital hardware. Many computationally intensive applications such as video encoding, video processing, and artificial intelligence are error resilient by nature due to the limitations of human visual perception or nonexistence of a golden answer for the given problem. Therefore, approximate computing can be used to improve the area, power,

and speed of digital hardware implementations of these error tolerant applications.

A variety of approximate circuits, ranging from system level designs to basic arithmetic circuits, have been proposed in the literature. Adders are used in most digital hardware, not only for binary addition but also for other binary arithmetic operations such as subtraction, multiplication, and division. Therefore, many approximate adders have been proposed in the literature. All approximate adders exploit the fact that critical path in an adder is seldom used.

Approximate adders can be broadly classified into the following categories: segmented adders, which divide n-bit adder into several r-bit adders operating in parallel; speculative adders, which predict the carry using only the few previous bits; and approximate full-adder based adders, which approximate the accurate full-adder at transistor or gate level [1].

Segmented and speculative adders usually have higher speeds and larger areas than accurate adders. Approximate full-adder based approximate n-bit adders use m-bit approximate adder in the least significant part (LSP) and (n − m)-bit accurate adder in the most significant part (MSP), as shown in Figure 1.1.

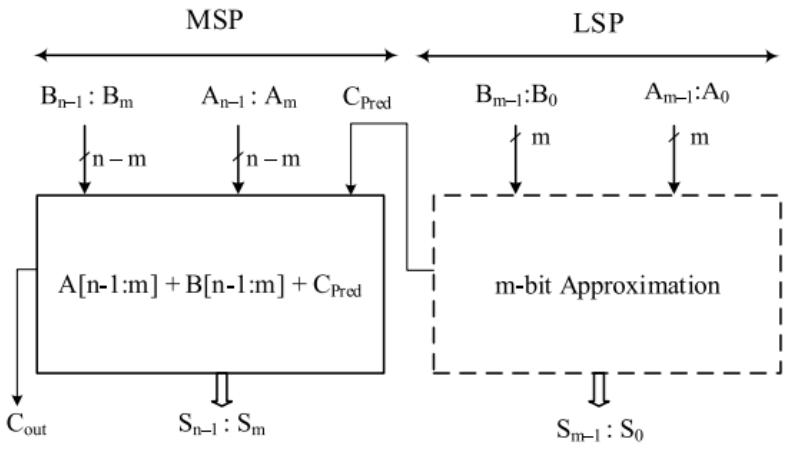

Fig 1.1: Generalized approximate adder

Most of the approximate adders in the literature have been designed for ASIC implementations. These approximate adders use gate or transistor level optimizations. Recent studies have shown that the approximate adders designed for ASIC implementations do not yield the same area, power, and speed improvements when implemented on FPGAs or fail to utilize FPGA resources efficiently to improve the output quality. This is mainly due to the difference in the way logic functions are implemented in ASICs and FPGAs.

The basic element of an ASIC implementation is a logic gate, whereas FPGAs use lookup tables (LUTs) to implement logic functions. Therefore, ASIC based optimization techniques cannot be directly mapped to FPGAs. FPGAs are widely used to implement error-tolerant applications using addition and multiplication operations. The efficiency of FPGA-based implementations of these applications can be improved through approximate computing. Only a few FPGA specific approximate adders have been proposed in the literature. These approximate adders focus on improving either the efficiency or accuracy. Therefore, the design of low error efficient approximate adders for FPGAs is an important research topic.

1.1 OBJECTIVE

In this project, we propose a methodology to reduce the error of approximate adders by efficiently utilizing FPGA resources, such as unused LUT inputs. We propose two approximate adders for FPGAs using our methodology based on the architecture shown in Figure 1.1. We propose a low error and area efficient approximate adder (LEADx) for FPGAs. It has lower mean square error (MSE) than the approximate adders in the literature. It achieves better quality than the other approximate adders for video encoding application.

We also propose an area and power efficient approximate adder (APEx) for FPGAs. Although its MSE is higher than that of LEADx, it is lower than that of the approximate adders in the literature. It has the same area, lower

MSE and less power consumption than the smallest and lowest power consuming approximate adder in the literature. It has smaller area and lower power consumption than the other approximate adders in the literature.

CHAPTER-2

LITERATURE SURVEY

G. A. Gillani et.al proposed a novel methodology for an internal-self-healing (ISH) that allows exploiting self-healing within a computing element internally without requiring a paired, parallel module, which extends the applicability to irregular/asymmetric datapaths while relieving the restriction of multiples of two for modules in a given datapath, as well as going beyond square functions. Approximate computing studies the quality-efficiency trade-off to attain a best-efficiency (e.g., area, latency, and power) design for a given quality constraint and vice versa. Recently, self-healing methodologies for approximate computing have emerged that showed an effective quality-efficiency tradeoff as compared to the conventional error-restricted approximate computing methodologies. However, the state-of-the-art self-healing methodologies are constrained to highly parallel implementations with similar modules (or parts of a datapath) in multiples of two and for square-accumulate functions through the pairing of mirror versions to achieve error cancellation. We employ our ISH methodology to design an approximate multiply-accumulate (xMAC), wherein the multiplier is regarded as an approximation stage and the accumulator as a healing stage. We propose to approximate a recursive multiplier in such a way that a near-to-zero average error is achieved for a given input distribution to cancel out the error at an accurate accumulation stage. To increase the efficacy of such a multiplier, we propose a novel 2 × 2 approximate multiplier design that alleviates the overflow problem within an n × n approximate recursive multiplier. The proposed ISH methodology shows a more effective quality-efficiency trade-off for an xMAC as compared with the conventional error-restricted methodologies for random inputs and for radio-astronomy calibration processing (up to 55% better quality output for equivalent-efficiency designs) [1].

E. Kalali and I. Hamzaoglu proposed approximation technique is also proposed. The proposed hardware implements angular prediction modes for all PU sizes (4x4 to 32x32). The proposed approximation technique significantly reduces area of the proposed hardware by enabling efficient use of one multiple constant multiplication (MCM) data-path to implement all constant multiplications using add and shift operations and by reducing amount of on-chip memory. It also reduces amount of computations and amount of on chip memory accessed. In this paper, an approximate High Efficiency Video Coding (HEVC) intra angular prediction technique is proposed for reducing area of HEVC intra prediction hardware. The proposed approximation technique uses closer neighboring pixels instead of distant neighboring pixels in intra angular prediction equations. It causes bit rate increase and 0.0028 dB PSNR loss on average. In this paper, an approximate HEVC intra angular prediction hardware implementing the proposed approximation technique is also proposed. FPGA and ASIC implementations of the proposed approximate hardware can process 24 and 40 quad full HD (3840x2160) video frames per second, respectively. The proposed approximate HEVC intra angular prediction hardware is the smallest and the second fastest HEVC intra prediction hardware in the literature. It is ten times smaller and 20% slower than the fastest HEVC intra prediction hardware in the literature. In this paper, first, data reuse technique is used to reduce amount of computations. Since some of the HEVC intra angular prediction equations use same Coeff and reference pixels, there are identical luminance angular prediction equations for each PU size. Since different PU sizes may use same neighboring pixels, there are also identical luminance angular prediction equations between different PU sizes. Data reuse technique calculates the common prediction equations for all luminance angular prediction modes only once and uses the result for corresponding modes. Since we use data reuse technique, instead of calculating intra prediction equations of different prediction modes and PUs

separately, we calculate all necessary intra prediction equations together and use the results for the corresponding prediction modes and Pus [2].

T. Ayhan and M. Altun aims to exploit approximate computing units in image processing systems and artificial neural networks. For this purpose, a general design methodology is introduced, and approximation-oriented architectures are developed for different applications. This paper proposes a method to compromise power/area efficiency of circuit-level design with accuracy supervision of system-level design. The proposed method selects approximate computational units that minimize the total computation cost, yet maintaining the ultimate performance. Unlike the previous case study for Sobel edge detector, each output of the approximate DCT unevenly contributes to the ultimate performance criteria, PSNR. The objective function is to minimize the total computational power consumption of the DCT module in a JPEG encoder. Moreover, the constraint b changes with the target compression ratio, or number of encoded frequency components, R. Therefore, computations to find the high frequency components are assigned a lower weight while writing the constraint equation. This is accomplished by formulating a linear programming problem, which can be solved by conventional linear programming solvers. Approximate computing units, such as multipliers, neurons, and convolution kernels, which are proposed by this paper, are suitable for power/area reduction through accuracy scaling. The formulation is demonstrated on applications in image processing, digital filters, and artificial neural networks. This way, the proposed technique and architectures are tested with different approximate computing units, as well as system-level requirement metrics, such as PSNR and classification performance. To successfully employ approximate circuits in systems of different scales, the tradeoffs between circuit costs and system's output quality are modeled as a linear program. Moreover, architectures that efficiently use approximate circuits are proposed for different applications. These applications include image processing and

artificial neural networks. The formulation is tested with systems used in edge detection, image compression, and ConvNets, showing the widespread applicability of the proposed technique [3].

W. Ahmad and I. Hamzaoglu, improved architecture for efficiently computing the sum of absolute differences (SAD) on FPGAs is proposed in this work. It is based on a configurable adder/subtractor implementation in which each adder input can be negated at runtime. The negation of both inputs at the same time is explicitly allowed and used to compute the sum of absolute values in a single adder stage. The architecture can be mapped to modern FPGAs from Xilinx and Altera. An analytic complexity model as well as synthesis experiments yield an average look-up table (LUT) reduction of 17.4% for an input word size of 8 bit compared to state-of-the-art. As the SAD computation is a resource demanding part in image processing applications, the proposed circuit can be used to replace the SAD core of many applications to enhance their efficiency. An improved SAD architecture is proposed which provides a significant resource reduction on current FPGAs. Its LUT complexity only grows with 2NB while the best state-of-the-art method grows with 2.5NB, leading to practical resource reductions of 17.4%. As the number of inputs in the multi-input addition is halved in the proposed method, it is expected that similar reductions are obtained when using advanced compressor tree optimization methods. The proposed circuit can be used to increase the efficiency of all the previously proposed applications from image processing that use the SAD metric in their core computation [4].

H. Jiang, C. Liu, L. Liu, F. Lombardi, and J. Han proposed the most important arithmetic modules in a processor, adders, multipliers, and dividers determine the performance and energy efficiency of many computing tasks. The demand of higher speed and power efficiency, as well as the feature of error resilience in many applications (e.g., multimedia, recognition, and data analytics), have driven the development of

approximate arithmetic design. In this article, a review and classification are presented for the current designs of approximate arithmetic circuits including adders, multipliers, and dividers. A comprehensive and comparative evaluation of their error and circuit characteristics is performed for understanding the features of various designs. By using approximate multipliers and adders, the circuit for an image processing application consumes as little as 47% of the power and 36% of the power-delay product of an accurate design while achieving similar image processing quality. Improvements in delay, power, and area are obtained for the detection of differences in images by using approximate dividers. Approximate Full Adders. In this type of design, approximate full adders are implemented in the LSBs of a multibit adder. It includes the simple use of OR gates (and one AND gate for carry propagation) in the so-called lower-part-OR adder (LOA), the approximate designs of the mirror adder (AMAs), and the approximate XOR/XNOR-based full adders (AXAs). Additionally, emerging technologies such as magnetic tunnel junctions have been considered for the design of approximate full adders for a shorter delay, a smaller area, and a lower power consumption. The critical path of this type of adders depends on its approximation scheme. For LOA, it is approximately O(log(n − l)), where l is the number of bits in the lower part of an adder. In the evaluation, LOA is selected as the reference design because the other designs require customized layouts at the transistor level; hence, they are not comparable with the other types of approximate adders that are approximated at the logic gate level. Finally, an adder with the LSBs truncated is referred to as a truncated adder that works with a lower precision. It is considered a baseline design [5].

A.C. Mert, H. Azgin, E. Kalali, and I. Hamzaoglu, Proposed the Approximate hardware designs have higher performance, smaller area or lower power consumption than exact hardware designs at the expense of lower accuracy. Absolute difference (AD) operation is heavily used in many applications such

as motion estimation (ME) for video compression, ME for frame rate conversion, stereo matching for depth estimation. Since most of the applications using AD operation are error tolerant by their nature, approximate hardware designs can be used in these applications. In this paper, novel approximate AD hardware designs are proposed. The proposed approximate AD hardware implementations have higher performance, smaller area and lower power consumption than exact AD hardware implementations at the expense of lower accuracy. They also have less error, smaller area and lower power consumption than the approximate AD hardware implementations which use approximate adders proposed in the literature. The three proposed approximate AD hardware, proposed_0 hardware consists of a subtractor and XOR gates. First, two 8-bit inputs A and B are subtracted with an exact subtractor hardware. Then, each bit of the subtraction result is XOR'ed with the sign bit of the subtraction result. If A >= B, the sign bit is 0. Therefore, each bit is XOR'ed with 0. In this case, proposed_0 hardware computes the correct absolute difference. If A < B, the sign bit is 1. Therefore, each bit is XOR'ed with 1. In this case, the output of proposed_0 hardware is 1 less than the correct absolute difference. Therefore, the maximum error of proposed_0 hardware is 1. In proposed_1 hardware, the most significant 7 bits of subtraction result is XOR'ed with the sign bit. But, the least significant bit of the subtraction result is not XOR'ed with the sign bit. Therefore, proposed_1 hardware has 1 less XOR gate than proposed_0 hardware. However, its maximum error is 2 which is 1 more than the maximum error of proposed_0 hardware. In proposed_2 hardware, the most significant 6 bits of subtraction result is XOR'ed with the sign bit. But, the least significant 2 bits of the subtraction result is not XOR'ed with the sign bit. Therefore, proposed_2 hardware has 2 less XOR gates than proposed_0 hardware. However, its maximum error is 4 which is 3 more than the maximum error of proposed_0 hardware [6].

CHAPTER-3

EXISTING METHOD

In the existing method, an approximate adder with accurate part at the MSP side of the adder and approximate adder at the LSP part of the adder.

3.1 HALF ADDER

The Half-Adder is a basic building block of adding two numbers as two inputs and produce out two outputs. The adder is used to perform OR operation of two single bit binary numbers. The augent and addent bits are two input states, and 'carry' and 'sum' are two output states of the half adder.

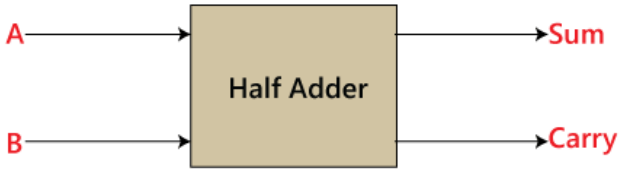

Fig 3.1: Half adder

Inputs		Outputs	
A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Fig 3.2: Half adder truth table

In the Figure,

1. 'A' and' B' are the input states, and 'sum' and 'carry' are the output states.
2. The carry output is 0 in case where both the inputs are not 1.
3. The least significant bit of the sum is defined by the 'sum' bit.

The SOP form of the sum and carry are as follows:

$$Sum=x'y+xy'$$

$$Carry = xy$$

3.2 FULL ADDER

A full adder is a logical circuit that performs an addition operation on three binary digits and just like the half adder, it also generates a carry out to the next addition column.

Here a Carry-in is a possible carry from a less significant digit, while a Carry-out represents a carry to a more significant digit.

In many ways, the full adder can be thought of as two half adders connected together, with the first half adder passing its carry to the second half adder as shown in figure 3.3.

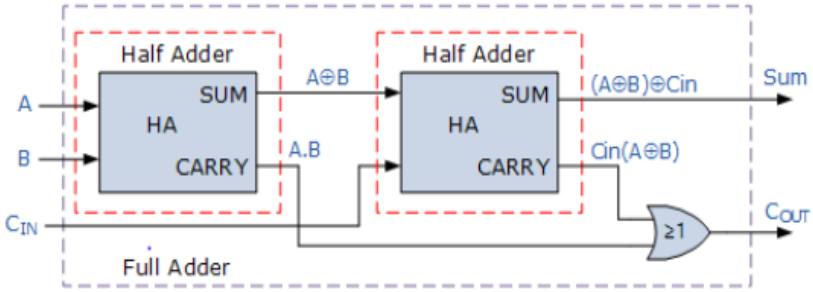

Fig 3.3: full adder

3.3 RIPPLE CARRY ADDER

A ripple carry adder is an important digital electronics concept, essential in designing digital circuits. Multiple full adder circuits can be cascaded in parallel to add an N-bit number. For an N- bit parallel adder, there must be N number of full adder circuits. A ripple carry adder is a logic circuit in which the carry-out of each full adder is the carry in of the succeeding next most significant full adder. It is called a ripple carry adder because each carry bit gets rippled into the next stage. In a ripple carry adder the sum and carry out bits of any half adder stage is not valid until the carry in of that stage occurs. Propagation delays inside the logic circuitry is the reason behind this. Propagation delay is time elapsed between the

application of an input and occurrence of the corresponding output. Consider a NOT gate, When the input is “0” the output will be “1” and vice versa. The time taken for the NOT gate’s output to become “0” after the application of logic “1” to the NOT gate’s input is the propagation delay here. Similarly the carry propagation delay is the time elapsed between the application of the carry in signal and the occurrence of the carry out (Cout) signal. Circuit diagram of a 4-bit ripple carry adder is shown in figure 3.4.

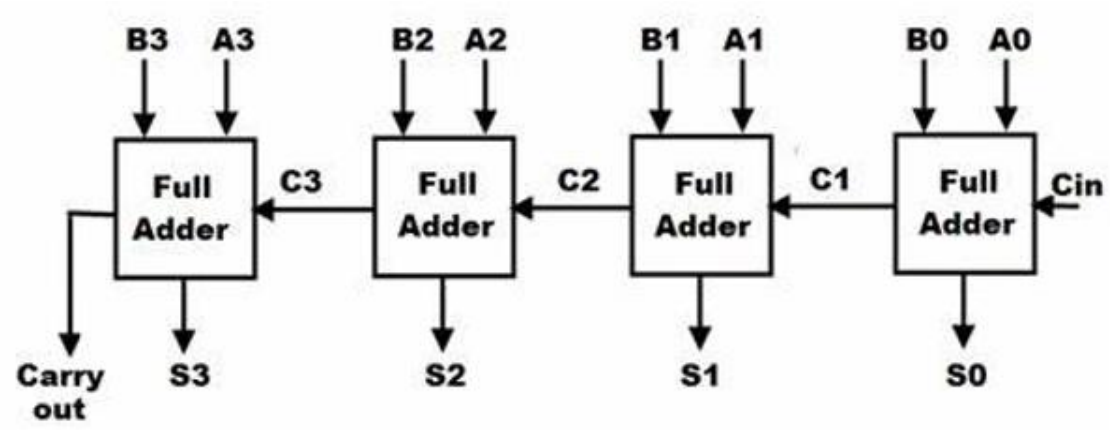

Fig 3.4: ripple carry adder

Binary addition is fundamental operation in most of the digital circuits. There are so many adders in the digital design. The selection of adder depends on its performance parameters. Adders are important elements in microprocessors, digital signal processors. ALU and in floating point arithmetic units. and memory addressing, in booth multipliers. They are also used in real time signal processing like signal processing, image processing etc. for human beings’ arithmetic calculations are easy to calculate when they are decimals i.e., base ten. But they became pragmatic if binary numbers are given. Therefore, binary addition is essential any improvement in binary addition can improve the performance of system. The fast and accuracy of system depends mainly on adder performance.

3.4 PARALLEL PREFIX ADDER

In this project designing and implementation of various parallel prefix adders on FPGA are described. Parallel Prefix Adders are also known as Carry Tree Adders. Parallel prefix adders are designed from carry look ahead adder as a base. Parallel prefix adders consist of three stages similar to CLA. Figure 3.5 shows the PPA structure.

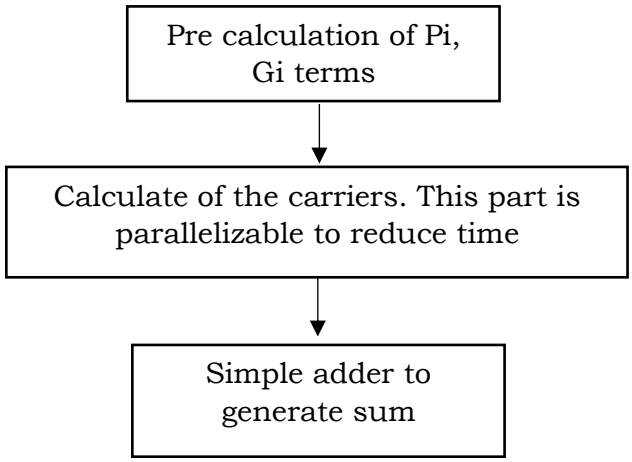

Fig 3.5: Generalized PPA

The parallel prefix adder employs three stages in preprocessing stage the generation of Propagate and Generate signals is carried out. The calculation of Generate (Gi) and Propagate (Pi) are calculated when the inputs A, B are given. As follows

$$Gi = Ai \text{ AND } Bi$$

$$Pi = Ai \text{ XOR } Bi$$

Gi indicates whether the Carry is generated from that bit. Pi indicates whether Carry is propagated from that bit. In carry generation stage of PPA, prefix graphs can be used to describe the tree structure. Here the tree structure consists of grey cells, black cells, and buffers. In carry generation

stage when two pairs of generate and propagate signals (Gm, Pm), (Gn, Pn) are given as inputs to the carry generation stage. It computes a pair of group generates and group propagate signals (Gm: n, Pm: n) which are calculated as follows Gm: n=Gm+ (Pm.Gn) Pm: n=Pm. Pn The black cell computes both generate and propagate signals as output. It uses two and gates and or gate. The grey cell computes the generate signal only. It uses only and gate, or gate. In post processing stage simple adder to generate the sum, Sum and carry out are calculated in post processing stage as follows Si=Pi XOR Ci-1 Cout=Gn-1 XOR (Pn-1 AND Gn-2) If Cout is not required it can be neglected.

Parallel prefix adders also known as carry tree adders. They pre-compute propagate and generate signals. These signals are combined using fundamental carry operator (fco). (g1, p1) o (g2, p2) = (g1+g2.p1, p1.p2) Due to associative law of the fundamental carry operator these operators can be combined in different ways to form various adder structures. For example 4 bit carry look ahead generator is given by C4= (g4, p4) o [(g3, p3) o [(g2, p2) o (g1, p1)]] Now in parallel prefix adders allow parallel operation resulting in more efficient tree structure for this 4 bit example. C4= [(g4, p4) o (g3, p3)] o [(g2, p2) o (g1, p1)] It is a key advantage of tree structured adders is that the critical path due to carry delay is of order log2N for N bit wide adder.

3.5 LOWER-PART-OR ADDER

“Lower-Part-OR Adder” Structure Description: Addition is a fundamental operation that can significantly influence the overall achievable performances. The Lower-part-OR Adder (LOA) divides a -bit addition into two -bit and -bit smaller parts. As shown in Figure 3.6, a -bit LOA exploits a regular smaller precise adder (is called sub-adder) that computes the precise values of the ‘ ’ most significant bits of the result (upper-part) along with some OR gates that approximate the ‘ ’ least significant result bits (lower-part) by applying bitwise OR to the respective

input bits. An extra AND gate is used to generate a carry-in for the upper part sub-adder when the most significant bits of both lower-part inputs are one. This helps to take into account the trivial carries from lower to upper parts of the LOA adder to decrease its imprecision. The sub-adder might have any desired precise structure such as carry look-ahead or ripple-carry adder, while using more efficient precise adders as the sub-adder directly leads to more efficient LOA. For a LOA with a constant Word-Length (WL or ‘ ’), higher ‘ ’ or Lower-Part Length (LPL) values decrease the area, delay, and power consumption of the LOA while at the same time increase its imprecision

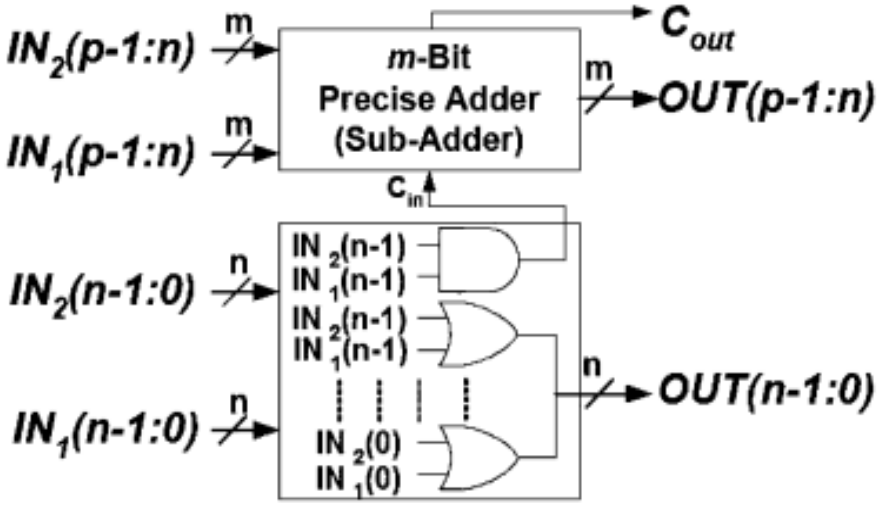

Fig 3.6: LOA approximate adder

3.6 DISADVANTAGES

- Accuracy is very low.
- Energy consumption is high
- Delay increases as the computation time of output increases.

CHAPTER-4

PROPOSED METHOD

In the proposed method, approximate full adders based on FPGA LUT's has been proposed. Two designs are proposed one establishes a tradeoff between power and accuracy and the other on area and accuracy. The proposed design methodology uses the approximate full adder based n-bit adder architecture shown in Fig. Generalized approximate adder, n-bit addition is divided into n-bit approximate adder in the LSP and (n−m)-bit accurate adder in the MSP.

Breaking the carry chain at bit-position m generally introduces an error of 2^m in the final sum. The error rate and error magnitude can be reduced by predicting the carry-in to the MSP (CMSP) more accurately and by modifying the logic function of LSP to compensate for the error. The carry to the accurate part can be predicted using any k-bit input pairs from the approximate part such that $k \leq m$. Most of the existing approximate adders use $k = 1$.

FPGA implementation of accurate adder uses only 2 inputs and 1 output of each 6-input LUT. We propose to utilize the remaining 4, available but unused, inputs of the first LUT of the MSP to predict CMSP. Therefore, we propose to share the most significant 2 bits of both inputs of the LSP with the MSP for carry prediction. Sharing more bits of LSP with MSP will increase the probability of correctly predicting CMSP which will in turn reduce error rate. However, this will also increase the area and delay of the approximate adder.

To analyze the tradeoff between the accuracy and performance of an FPGA-based approximate adder with different values of k. For $k > 2$, the error rate reduces slightly at the cost of increased area and delay. On the other hand, for $k < 2$, the delay improves marginally at the cost of significant

increase in the error rate. Therefore, we propose using k = 2, as it provides good balance between accuracy and performance of approximate adders for FPGAs.

4.1 PROPOSED APPROXIMATE ADDER

In the proposed approximate adders, a carry is passed to the MSP if it is generated at bit position m − 1, or generated at bit position m − 2 and propagated at bit position m − 1. The CMSP can be described by (4) where G_i and P_i are generated and propagate signals of the i^{th} bit position, respectively. Architecture of the proposed approximate adders is shown in Figure 4.1.

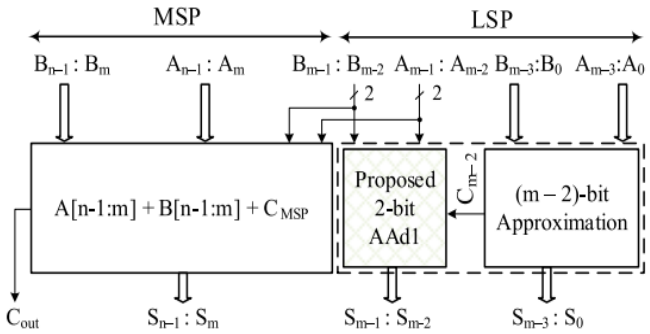

Fig 4.1: proposed approximate adder

It uses 2 MSBs of LSP to predict the CMSP, whereas their respective sum bits are computed using AAd1. AAd1 is only suitable when the Cout of 2-bit inputs is predicted accurately. Accurate prediction of Cout requires additional resources or unused LUT inputs. Therefore, to design area efficient approximate adders for FPGAs, AAd1 is not used in the least-significant m − 2 bits of the LSP. In this paper, we propose two n-bit approximate adders using the architecture in Figure 4.1.

The two proposed n-bit approximate adders use different approximate functions for the first m − 2 bits of the LSP. State-of-the-art FPGAs use 6-input LUTs. These LUTs can be used to implement two 5-input functions. The complexity of the implemented logic function does not affect

performance of LUT based implementation. A 2-bit adder has 5 inputs and two outputs. Therefore, a LUT can be used to implement a 2-bit approximate adder.

For an area efficient FPGA implementation, we propose to split the first m − 2 bits of LSP into d(m − 2)/2e groups of 2-bit inputs such that each group is mapped to a single LUT. Each group adds two 2-bit inputs with carry-in using an approximate 2-bit adder (AAd2). To eliminate the carry chain in LSP, we propose to equate Cout of ith group to one of the inputs of that group (Ai+1). This results in error in 8 out of 32 possible cases with an absolute error magnitude of 4 in each erroneous case. To reduce the error magnitude, we propose to compute the Si and Si+1 output bits as follows:

- If the Cout is predicted correctly, the sum outputs are also calculated accurately using standard 2-bit addition.
- If the Cout is predicted incorrectly and the predicted value of Cout is 0, both sum outputs are set to 1.
- If the Cout is predicted incorrectly and the predicted value of Cout is 1, both sum outputs are set to 0.

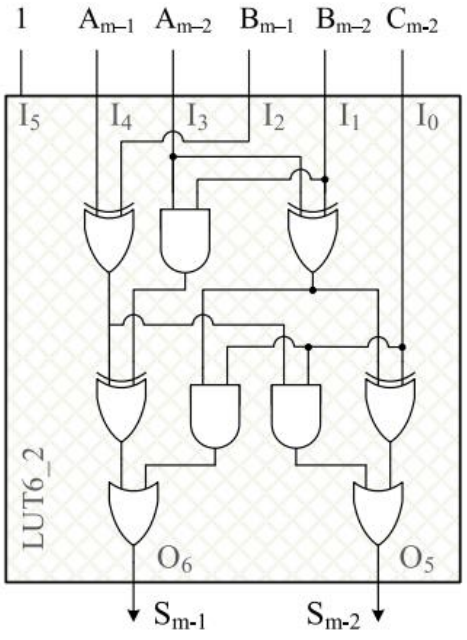

Fig 4.2: AAd1

A_{i+1}	A_i	B_{i+1}	B_i	C_{in}	C_{i+2}	S_{i+1}	S_i
0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	1
0	0	0	1	0	0	0	1
0	0	0	1	1	0	1	0
0	0	1	0	0	0	1	0
0	0	1	0	1	0	1	1
0	0	1	1	0	0	1	1
0	0	1	1	1	0	1	1
0	1	0	0	0	0	0	1
0	1	0	0	1	0	1	0
0	1	0	1	0	0	1	0
0	1	0	1	1	0	1	1
0	1	1	0	0	0	1	1
0	1	1	0	1	0	1	1
0	1	1	1	0	0	1	1
0	1	1	1	1	0	1	1
1	0	0	0	0	1	0	0
1	0	0	0	1	1	0	0
1	0	0	1	0	1	0	0
1	0	0	1	1	1	0	0
1	0	1	0	0	1	0	0
1	0	1	0	1	1	0	1
1	0	1	1	0	1	0	1
1	0	1	1	1	1	1	0
1	1	0	0	0	1	0	0
1	1	0	0	1	1	0	0
1	1	0	1	0	1	0	0
1	1	0	1	1	1	0	1
1	1	1	0	0	1	0	1
1	1	1	0	1	1	1	0
1	1	1	1	0	1	1	0
1	1	1	1	1	1	1	1

Figure 4.3: Truth table of proposed AAd2

This modification reduces the absolute error magnitude to 2 in two cases, and to 1 in the other six cases. The resulting truth table of AAd2 is given in Table 4.1. The error cases are shown in red. Since AAd2 produces an erroneous result in 8 out of 32 cases, the error probability of AAd2 is 0.25.

4.2 PROPOSED LEADX

The proposed LEADx approximate adder is shown in Figure 4.3. An n-bit LEADx uses d(m − 2)/2e copies of AAd2 adder in the least significant m − 2 bits of the approximate adder architecture shown in Fig. proposed approximate adder. In

LEADx, Cm−2 = Am−3. AAd2 implements a 5-to-2 logic function that is mapped to a single LUT. Similarly, AAd1 is also mapped to a single LUT. Therefore, dm/2e LUTs are used for the LSP. These LUTs work in parallel. Therefore, the delay of LSP is equal to the delay of a single LUT (tLUT). The critical path of LEADx is from the input Am−2 to the output Sn−1.

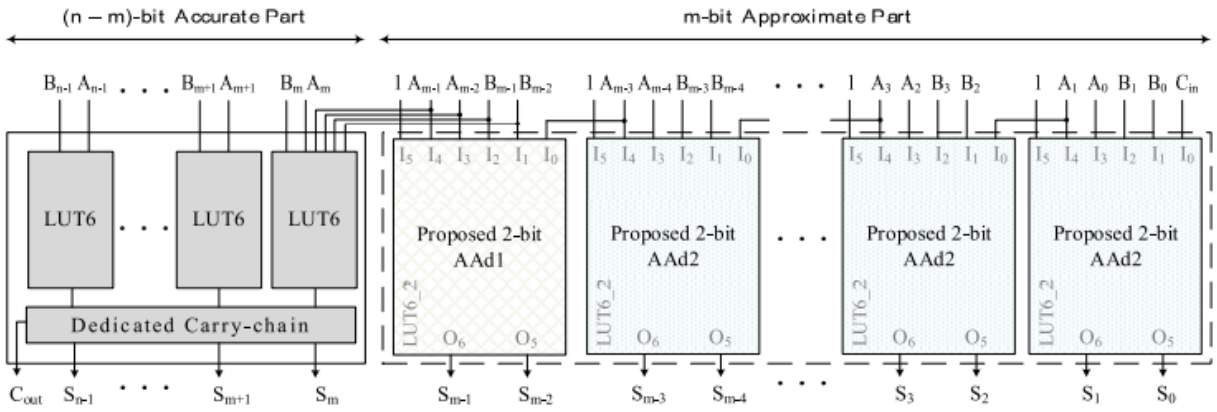

Fig 4.4: LEADx

4.3 PROPOSED APEX

APEx is also based on the approximate adder architecture shown in Figue 4.4. For the least significant m − 2 bits of the LSP, the aim is to find an approximate function with no data dependency. Carry should neither be

generated nor used for sum computation. A 1-bit input pair at any bit position i ≤ (m − 2) should produce a 1-bit sum output only. In general, any logic function with 1-bit output can be used as an approximate function to compute the approximate sum of 1-bit inputs at ith bit position. A constant 0 or constant 1 at the output are also valid approximate functions. Fixing the output to 0 or 1 will reduce the area and power consumption of the approximate adder because no hardware will be required for sum computation. If the least significant m − 2 bits are fixed to 1, the ME occurs when the inputs A0 to Am−3 and B0 to Bm−3 are all 0. With accurate addition, S0 to Sm−3 output bits are all 0 and carry is not propagated to m − 2 bit position. Fixing S0 to Sm−3 to 1 and carry-in for m − 2 bit position to 0 results in ME of 2m−2 − 1. The ME of constant 1 is less than the ME of constant 0.

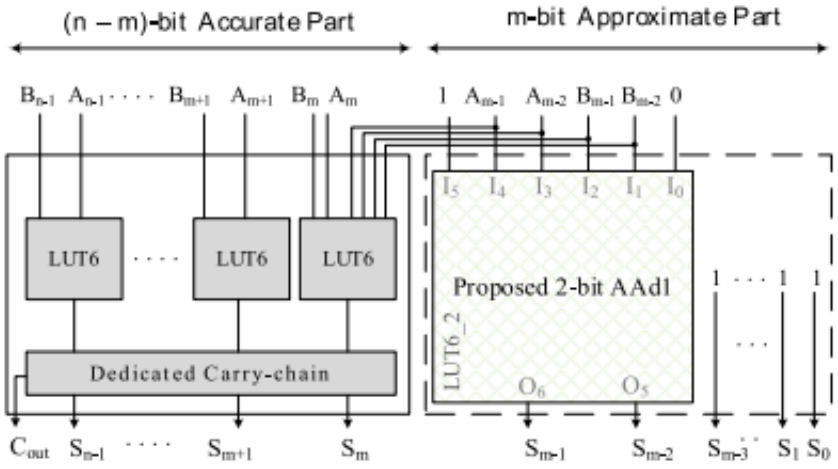

Fig 4.5: APEx

In the proposed APEx, the S0 to Sm−3 outputs are fixed to 1 and the Cm−2 is 0. This provides significant area and power consumption reduction at the expense of slight quality loss. It is important to note that this is different from bit truncation technique which fixes both the sum and carry outputs to 0. The ME of truncate adder is 2m+1 − 2 which is much higher than ME of APEx (2m−2 − 1). The proposed APEx approximate adder is

shown in Figure 4.4. Same as LEADx, the critical path of APEx is from the input Am−2 to the output Sn−1.

4.4 ADVANTAGES & APPLICATIONS

4.4.1 Advantages:

- Computational delay is reduced
- Improvement in accuracy of output is achieved
- Area and Energy consumption are optimized.

4.4.2 Applications:

- Digital signal processors
- Digital image processors
- Video processor applications
- MAC & Arithmetic circuits

CHAPTER 5

SOFTWARE REQUIREMENTS

5.1 HISTORY OF VERILOG

Verilog was started initially as a proprietary hardware modeling language by Gateway Design Automation Inc. around 1984. It is rumored that the original language was designed by taking features from the most popular HDL language of the time, called HiLo, as well as from traditional computer languages such as C. At that time, Verilog was not standardized and the language modified itself in almost all the revisions that came out within 1984 to 1990.

Verilog simulator was first used beginning in 1985 and was extended substantially through 1987. The implementation was the Verilog simulator sold by Gateway. The first major extension was Verilog-XL, which added a few features and implemented the infamous "XL algorithm" which was a very efficient method for doing gate-level simulation.

The time was late 1990. Cadence Design System, whose primary product at that time included thin film process simulator, decided to acquire Gateway Automation System. Along with other Gateway products, Cadence now became the owner of the Verilog language, and continued to market Verilog as both a language and a simulator.

At the same time, Synopsys was marketing the top-down design methodology, using Verilog. This was a powerful combination. In 1990, Cadence recognized that if Verilog remained a closed language, the pressures of standardization would eventually cause the industry to shift to VHDL. Consequently, Cadence organized the Open Verilog International (OVI), and in 1991 gave it the documentation for the Verilog Hardware Description Language. This was the event which "opened" the language.

5.2 INTRODUCTION TO HDL

- HDL is an abbreviation of Hardware Description Language. Any digital system can be represented in a REGISTER TRANSFER LEVEL (RTL) and HDLs are used to describe this RTL.
- Verilog is one such HDL and it is a general-purpose language –easy to learn and use. Its syntax is similar to C.
- The idea is to specify how the data flows between registers and how the design processes the data.
- To define RTL, hierarchical design concepts play a very significant role. Hierarchical design methodology facilitates the digital design flow with several levels of abstraction.
- Verilog HDL can utilize these levels of abstraction to produce a simplified and efficient representation of the RTL description of any digital design.
- For example, an HDL might describe the layout of the wires, resistors and transistors on an Integrated Circuit (IC) chip, i.e., the switch level or, it may describe the design at a more micro level in terms of logical gates and flip flops in a digital system, i.e., the gate level. Verilog supports all of these levels.

5.2.1 Design Styles:

Any hardware description language like Verilog can be design in two ways one is bottom-up design and other one is top-down design.

5.2.2 Bottom-Up Design:

The traditional method of electronic design is bottom-up (designing from transistors and moving to a higher level of gates and, finally, the system). But with the increase in design complexity traditional bottom-up designs have to give way to new structural, hierarchical design methods.

5.2.3 Top-Down Design:

For HDL representation it is convenient and efficient to adapt this design-style. A real top-down design allows early testing, fabrication technology independence, a structured system design and offers many other advantages. But it is very difficult to follow a pure top-down design. Due to this fact most designs are mix of both the methods, implementing some key elements of both design styles.

5.2.4 Features of Verilog HDL

- Verilog is case sensitive.
- Ability to mix different levels of abstract freely.
- One language for all aspects of design, testing, and verification.
- In Verilog, Keywords are defined in lower case.
- In Verilog, Most of the syntax is adopted from "C" language.
- Verilog can be used to model a digital circuit at Algorithm, RTL, Gate and Switch level.
- There is no concept of package in Verilog.
- It also supports advanced simulation features like TEXTIO, PLI, and UDPs.

5.3 VLSI DESIGN FLOW

The VLSI design cycle starts with a formal specification of a VLSI chip, follows a series of steps, and eventually produces a packaged chip.

It provides a structured framework that helps designers navigate through different stages of the design process, from concept to production. The design flow encompasses several steps, including specification, design entry, synthesis, verification, layout, and fabrication.

One of the primary objectives of the design flow is to minimize the design cycle time while maximizing the quality and reliability of the final product. It allows designers to break down the complex VLSI design process

into smaller, manageable tasks, enabling them to focus on specific aspects of the design at each stage. By following a well-defined design flow, engineers can streamline the design process, reduce errors, and enhance the overall productivity of the VLSI designers.

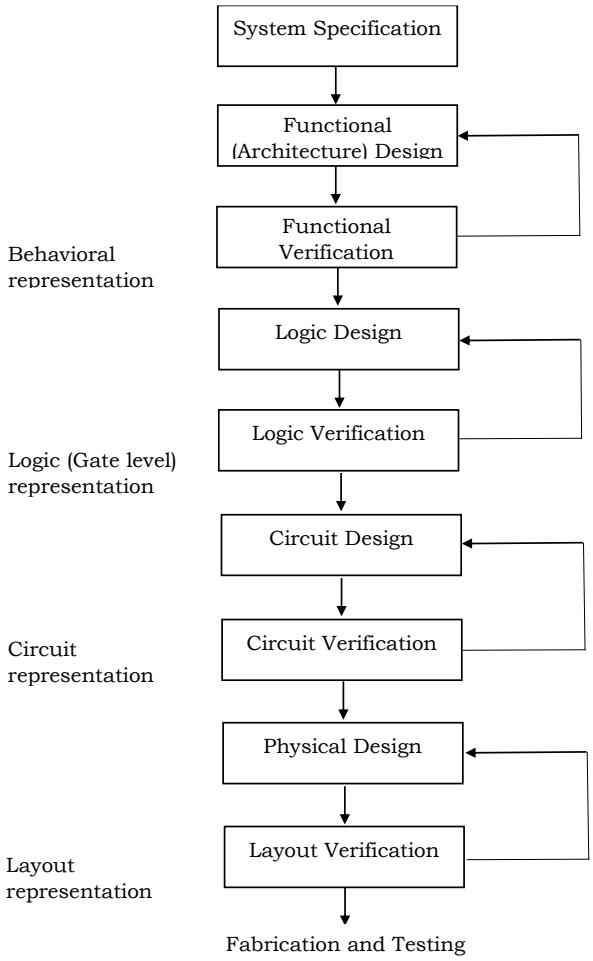

Fig 5.1: VLSI Design flow

5.3.1 System Specification:

The first step of any design process is to lay down the specifications of the system. System specification is a high-level representation of the system. The factors to be considered in this process include: performance, functionality, and physical dimensions like size of the chip.

The specification of a system is a compromise between market requirements, technology and economic viability. The end results are

specifications for the size, speed, power, and functionality of the VLSI system.

5.3.2 Architectural Design

The basic architecture of the system is designed in this step. This includes, such decisions as RISC (Reduced Instruction Set Computer) versus CISC (Complex Instruction Set Computer), number of ALUs, Floating Point units, number and structure of pipelines, and size of caches among others. The outcome of architectural design is a Micro-Architectural Specification (MAS).

5.3.3 Behavioral or Functional Design:

In this step, main functional units of the system are identified. This also identifies the interconnect requirements between the units. The area, power, and other parameters of each unit are estimated.

Modules. The key idea is to specify behavior, in terms of input, output and timing of each unit, without specifying its internal structure.

The outcome of functional design is usually a timing diagram or other relationships between units.

5.3.4 Logic Design:

In this step the control flow, word widths, register allocation, arithmetic operations, and logic operations of the design that represent the functional design are derived and tested.

This description is called Register Transfer Level (RTL) description. RTL is expressed in a Hardware Description Language (HDL), such as VHDL or Verilog. This description can be used in simulation and verification

5.3.5 Circuit Design:

The purpose of circuit design is to develop a circuit representation based on the logic design. The Boolean expressions are converted into a circuit representation by taking into consideration the speed and power

requirements of the original design. Circuit Simulation is used to verify the correctness and timing of each component

The circuit design is usually expressed in a detailed circuit diagram. This diagram shows the circuit elements (cells, macros, gates, transistors) and interconnection between these elements. This representation is also called a netlist. And each stage verification of logic is done.

5.3.6 Physical design:

In this step the circuit representation (or netlist) is converted into a geometric representation. As stated earlier, this geometric representation of a circuit is called a layout.

Layout is created by converting each logic component (cells, macros, gates, transistors) into a geometric representation (specific shapes in multiple layers), which perform the intended logic function of the corresponding component. Connections between different components are also expressed as geometric patterns typically lines in multiple layers.

5.3.7 Layout verification:

Physical design can be completely or partially automated and layout can be generated directly from netlist by Layout Synthesis tools. Layout synthesis tools, while fast, do have an area and performance penalty, which limit their use to some designs. These are verified.

5.3.8 Fabrication and Testing:

Silicon crystals are grown and sliced to produce wafers. The wafer is fabricated and diced into individual chips in a fabrication facility. Each chip is then packaged and tested to ensure that it meets all the design specifications and that it functions properly.

5.4 MODULE

A module is the basic building block in Verilog. It can be an element or a collection of low-level design blocks. Typically, elements are grouped into

modules to provide common functionality used in places of the design through its port interfaces, but hides the internal implementation.

Syntax:

```
module<module name> (<module_port_list>);

…..

<module internals> //contents of the module

….

Endmodule
```

5.4.1 Instances

A module provides a template from where one can create objects. When a module is invoked Verilog creates a unique object from the template, each having its own name, variables, parameters and I/O interfaces. These are known as *instances*.

5.4.2 Ports:

- Ports allow communication between a module and its environment.
- All but the top-level modules in a hierarchy have ports.
- Ports can be associated by order or by name.

You declare ports to be input, output or inout. The port declaration syntax is:

```
Input [range_val:range_var] list_of_identifiers;
output[range_val:range_var] list_of_identifiers;
inout[range_val:range_var] list_of_identifiers;
```

5.4.3 Identifiers

- Identifiers are user-defined words for variables, function names, module names, and instance names. Identifiers can be composed of letters, digits, and the underscore character.

- The first character of an identifier cannot be a number. Identifiers can be any length.
- Identifiers are case-sensitive, and all characters are significant.

An identifier that contains special characters, begins with numbers, or has the same name as a keyword can be specified as an escaped identifier. An escaped identifier starts with the backslash character(\) followed by a sequence of characters, followed by white space.

5.4.4 Keylines:

- Verilog uses keywords to interpret an input file.
- You cannot use these words as user variable names unless you use an escaped identifier.
- Keywords are reserved identifiers, which are used to define language constructs.
- Some of the keywords are always, case, assign, begin, case, end and end case etc.

5.5 DATA TYPES

Verilog Language has two primary data types:

- ***Nets*** - represents structural connections between components.
- ***Registers*** - represent variables used to store data.

Every signal has a data type associated with it. Data types are:

- ***Explicitly declared*** with a declaration in the Verilog code.
- ***Implicitly declared*** with no declaration but used to connect structural building blocks in the code. Implicit declarations are always net type "wire" and only one bit wide.

5.5.1 Register Data Types

- Registers store the last value assigned to them until another assignment statement changes their value.
- Registers represent data storage constructs.
- Register arrays are called memories.

- Register data types are used as variables in procedural blocks.
- A register data type is required if a signal is assigned a value within a procedural block
- Procedural blocks begin with keyword initial and always.

The data types that are used in register are register, integer, time and real.

5.6 MODELLING CONCEPTS

Modeling plays a significant role in the efficient simulation of VLSI circuits. By simplifying the models used to analyze these circuits, it is possible to perform transient analyses with reasonable accuracy at speeds of one or two orders of magnitude faster than in conventional circuit simulation programs.

5.6.1 Abstraction Levels:

- Behavioral level
- Register-Transfer Level
- Gate Level
- Switch level

5.6.1.1 Behavioral or algorithmic Level

- This level describes a system by concurrent algorithms (Behavioral).
- Each algorithm itself is sequential meaning that it consists of a set of instructions that are executed one after the other.
- The blocks used in this level are 'initial', 'always' ,'functions' and 'tasks' blocks
- The intricacies of the system are not elaborated at this stage and only the functional description of the individual blocks is prescribed.
- In this way the whole logic synthesis gets highly simplified and at the same time more efficient.

5.6.1.2 Register-Transfer Level:

- Designs using the Register-Transfer Level specify the characteristics of a circuit by operations and the transfer of data between the registers.
- An explicit clock is used. RTL design contains exact timing possibility, operations are scheduled to occur at certain times.
- Modern definition of a RTL code is "Any code that is synthesizable is called RTL code".

5.6.1.3 Gate Level:

- Within the logic level the characteristics of a system are described by logical links and their timing properties.
- All signals are discrete signals. They can only have definite logical values (`0', `1', `X', `Z`). The usable operations are predefined logic primitives (AND, OR, NOT etc gates).
- It must be indicated here that using the gate level modeling may not be a good idea in logic design.
- Gate level code is generated by tools like synthesis tools in the form of netlists which are used for gate level simulation and for backend.

5.7 OPERATORS

Verilog provided many different operators types. Operators can be,

- Arithmetic Operators
- Relational Operators
- Bit-wise Operators
- Logical Operators
- Reduction Operators
- Shift Operators
- Concatenation Operator

- Conditional Operator

5.7.1 Arithmetic Operators

- These perform arithmetic operations. The + and - can be used as either unary (-z) or binary (x-y) operators.
- Binary: +, -, *, /, % (the modulus operator)
- Unary: +, - (This is used to specify the sign)
- Integer division truncates any fractional part
- The result of a modulus operation takes the sign of the first operand
- If any operand bit value is the unknown value x, then the entire result value is x
- Register data types are used as unsigned values (Negative numbers are stored in two's complement form).

5.7.2 Relational Operators

Relational operators compare two operands and return a single bit 1or 0. These operators synthesize into comparators. Wire and reg variables are positive Thus (-3'b001) = = 3'b111 and (-3d001)>3d1 10, however for integers -1<>.

Table 5.1: Relational Operators

Operator	**Description**
a < b	a less than b
a > b	a greater than b
a <= b	a less than or equal to b
a >= b	a greater than or equal to b

The result is a scalar value

- 0 if the relation is false (a is bigger than b)
- 1 if the relation is true (a is smaller than b)
- x if any of the operands has unknown x bits (if a or b contains X)

Note: If any operand is x or z, then the result of that test is treated as false (0)

5.7.3 Bit-wise Operators

Bitwise operators perform a bit wise operation on two operands. This take each bit in one operand and perform the operation with the corresponding bit in the other operand. If one operand is shorter than the other, it will be extended on the left side with zeroes to match the length of the longer operand.

Table 5.2: Bitwise Operators

Operator	**Description**
~	Negation
&	AND
|	Inclusive OR
^	Exclusive OR
^~ or ~^	Exclusive NOR (Equivalent)

Computations include unknown bits, in the following way:

```
-> ~x = x

-> 0&x = 0

-> 1&x = x&x = x

-> 1|x = 1

-> 0|x = x|x = x

-> 0^x = 1^x = x^x = x

-> 0^~x = 1^~x = x^~x = x
```

When operands are of unequal bit length, the shorter operand is zero-filled in the most significant bit positions.

5.7.4 Logical Operators

Logical operators return a single bit 1 or 0. They are the same as bit-wise operators only for single bit operands. They can work on expressions, integers or groups of bits, and treat all values that are nonzero as "1". Logical operators are typically used in conditional (if … else) statements since they work with expressions.

Table 5.3: Logical Operators

Operator	**Description**
!	Logic negation
&&	Logical AND
||	Logical OR

Expressions connected by && and || are evaluated from left to right

Evaluation stops as soon as the result is known

The result is a scalar value:

- 0 if the relation is false
- 1 if the relation is true
- x if any of the operands has x (unknown) bits

5.7.5 Reduction Operators

Reduction operators operate on all the bits of an operand vector and return a single-bit value. These are the unary (one argument) form of the bit-wise operators.

Table 5.4: Reduction Operators

Operator	Description
&	AND
~&	NAND
|	OR
~|	NOR
^	XOR
^~ or ~^	XNOR

- Reduction operators are unary.
- They perform a bit-wise operation on a single operand to produce a single bit result.
- Reduction unary NAND and NOR operators operate as AND and OR respectively, but with their outputs negated.

5.7.6 Shift Operators

Shift operators shift the first operand by the number of bits specified by the second operand. Vacated positions are filled with zeros for both left and right shifts (There is no sign extension).

Table 5.5: Shift Operators

Operator	Description
<<	Left shift
>>	Right Shift

- The left operand is shifted by the number of bit positions given by the right operand.
- The vacated bit positions are filled with zeroes

5.7.7 Concatenation Operator

- The concatenation operator combines two or more operands to form a larger vector.
- Concatenations are expressed using the brace characters { and }, with commas separating the expressions within.
- ->Example: + {a, b[3:0], c, 4'b1001} // if a and c are 8-bit numbers, the results has 24 bits
- Un-sized constant numbers are not allowed in concatenations

5.7.8 Operator Precedence

Table 5.6: Operator Precedence

Operator	**Symbols**
Unary, Multiply, Divide, Modulus	!,~,*,/,%
Add, Subtract, Shift	+,-,<<,>>
Relation, Equality	<,>,<=,>=,==,!=,===,!==
Reduction	&,!&,^,^~,|,~|
Logic	&&, ||
Conditional	?:

5.7.9 Switch Level:

This is the lowest level of abstraction. A module can be implemented in terms of switches, storage nodes and interconnection between them. However, as has been mentioned earlier, one can mix and match all the levels of abstraction in a design. RTL is frequently used for Verilog description that is a combination of behavioral and dataflow while being acceptable for synthesis.

5.8 XININX VERILOG HDL TUTORIAL

5.8.1 Getting started

Frist we need to download and install Xilinx and ModelSim. These tools both have free student versions. Please accomplish Appendix B, C, and D in that order before continuing with this tutorial. Additionally if you wish to purchase your own Spartan3 board, you can do so at Digilent's Website. Digilent offers academic pricing. Please note that you must download and install Digilent Adept software. The software contains the drivers for the board that you need and also provides the interface to program the board.

5.8.2 Introduction to Xilinx

Xilinx Tools is a suite of software tools used for the design of digital circuits implemented using Xilinx Field Programmable Gate Array (FPGA) or Complex Programmable Logic Device (CPLD). The design procedure consists of (a) design entry, (b) synthesis and implementation of the design, (c) functional simulation and (d) testing and verification. Digital designs can be entered in various ways using the above CAD tools: using a schematic entry tool, using a hardware description language (HDL) – Verilog or VHDL or a combination of both. In this lab we will only use the design flow that involves the use of Verilog HDL.

The CAD tools enable you to design combinational and sequential circuits starting with Verilog HDL design specifications. The steps of this design procedure are listed below:

1. Create Verilog design input file(s) using template driven editor.
2. Compile and implement the Verilog design file(s).
3. Create the test-vectors and simulate the design (functional simulation) without using a PLD (FPGA or CPLD).
4. Assign input/output pins to implement the design on a target device.
5. Download bitstream to an FPGA or CPLD device.

7. Test design on FPGA/CPLD device

A Verilog input file in the Xilinx software environment consists of the following segments:

Header: module name, list of input and output ports.

Declarations: input and output ports, registers and wires.

Logic Descriptions: equations, state machines and logic functions.

End: endmodule

All your designs for this lab must be specified in the above Verilog input format. Note that the *state diagram* segment does not exist for combinational logic designs.

5.8.3 Programmable Logic Device: FPGA

In this lab digital designs will be implemented in the Basys2 board which has a Xilinx Spartan3E –XC3S250E FPGA with CP132 package. This FPGA part belongs to the Spartan family of FPGAs. These devices come in a variety of packages. We will be using devices that are packaged in 132 pin package with the following part number: XC3S250E-CP132.

5.8.4 Creating a New Project

Creating Projects, you can use the New Project wizard to easily create different types of projects in the Vivado IDE. To open the New Project wizard, select File > New Project. This wizard enables you to specify a project location and name and create the types of projects shown in figure 5.2.

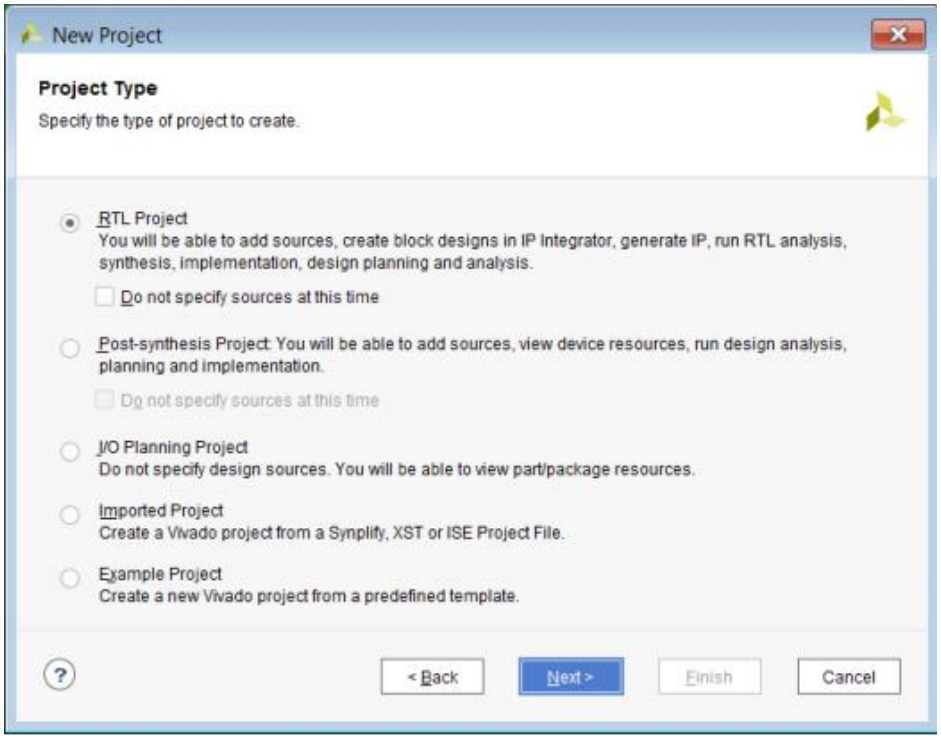

Fig 5.2: New Project Wizard—Project Type Page

Project Name: Write the name of your new project which is user defined.

Project Location: The directory where you want to store the new project in the specified location in one of your drive. In above window they are stored in location c drive which is not correct, the location of software and code should not be same location and Clicking on NEXT.

For each of the properties given below, click on the '**value**' area and select from the list of values that appear.

- **Device Family**: Family of the FPGA/CPLD used. In this laboratory we will be using the Spartan3E FPGA's.
- **Device**: The number of the actual device. For this lab you may enter **XC3S250E** (this can be found on the attached prototyping board)
- **Package**: The type of package with the number of pins. The Spartan FPGA used in this lab is packaged in CP132 package.
- **Speed Grade**: The Speed grade is "-4".

- **Synthesis Tool**: **XST** [VHDL/Verilog]
- **Simulator:** The tool used to simulate and verify the functionality of the design. Then click on **NEXT** to save the entries.

5.8.5 Opening Designs:

Use the Flow Navigator or Flow menu to select the following commands:

- Open Elaborated Design
- Open Synthesized Design
- Open Implemented Design

The Flow > Open Implemented Design command populates the Vivado IDE as shown in figure 5.3.

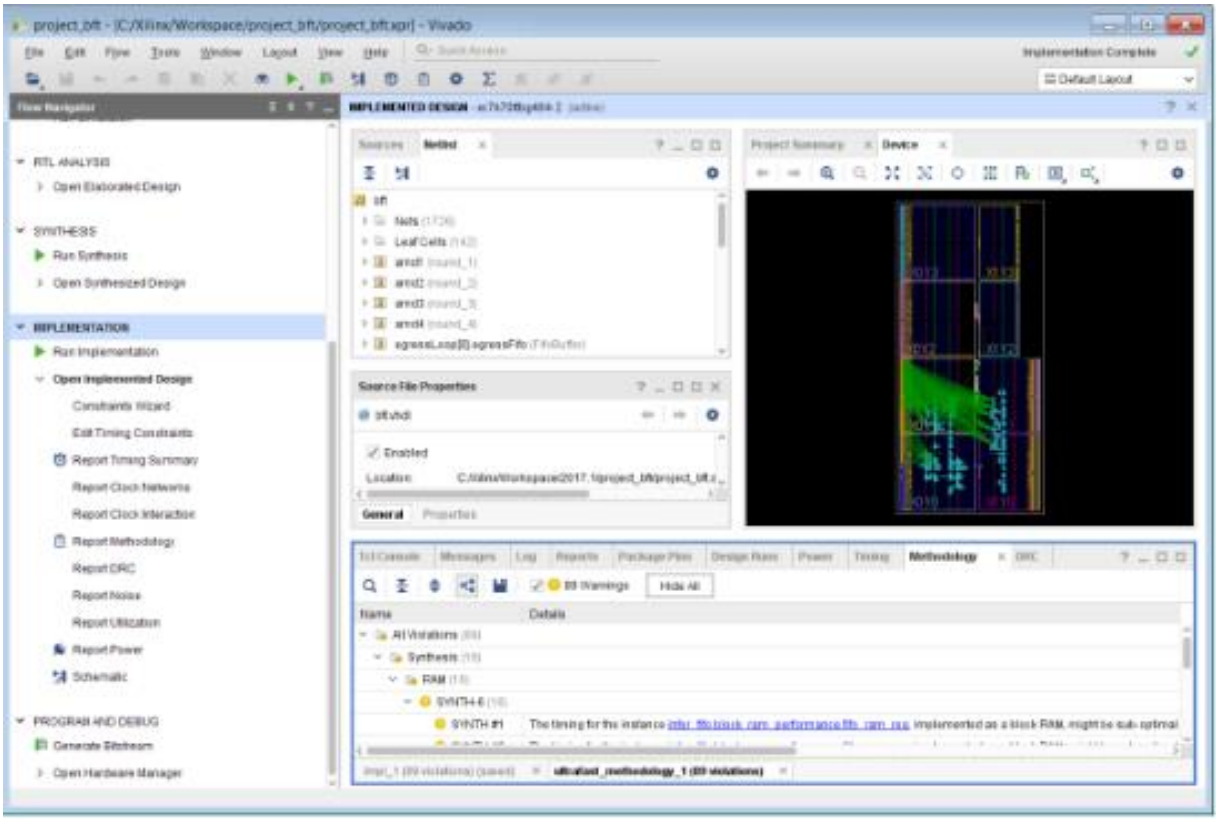

Fig 5.3: Implemented design

All project files such as schematics, netlists, Verilog files, VHDL files, etc., will be stored in a subdirectory with the project name.

In order to open an existing project in Xilinx Tools, select **File->Open Project** to show the list of projects on the machine. Choose the project you want and click **OK**.

If creating a new source file, click on the NEW SOURCE.

Creating a Verilog HDL input file for a combinational logic design:

In this lab we will enter a design using a structural or RTL description using the Verilog HDL. You can create a Verilog HDL input file (**.v** file) using the HDL Editor available in the Xilinx Vivado Tools (or any text editor).

In the previous window, click on the NEW SOURCE

(Note: "**Add to project**" option is selected by default. If you do not select it then you will have to add the new source file to the project manually.)

Select **Verilog Module** and in the "File Name:" area, enter the name of the Verilog source file you are going to create. Also make sure that the option **Add to project** is selected so that the source need not be added to the project again. Then click on **Next** to accept the entries.

In the **Port Name** column, enter the names of all input and output pins and specify the **Direction** accordingly. A Vector/Bus can be defined by entering appropriate bit numbers in the **MSB/LSB** columns. Then click on **Next>** to get a window showing all the new source information above window. If any changes are to be made, just click on **<Back** to go back and make changes. If everything is acceptable, click on **Finish > Next > Next > Finish** to continue.

Once you click on **Finish**, the source file will be displayed in the sources window in the **Project Navigator.** If a source has to be removed, just right click on the source file in the **Sources in Project** window in the **Project Navigator** and select **remove** in that. Then select **Project -> Delete Implementation Data** from the Project Navigator menu bar to remove any related files.

5.8.6 Editing the Verilog source file

The source file will now be displayed in the **Project Navigator** window (Figure 5.8). The source file window can be used as a text editor to make any necessary changes to the source file. All the input/output pins will be displayed. Save your Verilog program periodically by selecting the **File->Save** from the menu. You can also edit Verilog programs in any text editor and add them to the project directory using "Add Copy Source".

Here in the above window we will write the Verilog programming code for specified design and algorithm in the window.

After writing the programming code we will go for the synthesis report.

5.8.7 Configuring Project Settings

You can configure the Project Settings in the Settings dialog box to meet your design needs. These settings include general settings, related to the top module definition and language options, as well as simulation, elaboration, synthesis, implementation, bitstream, and IP settings.

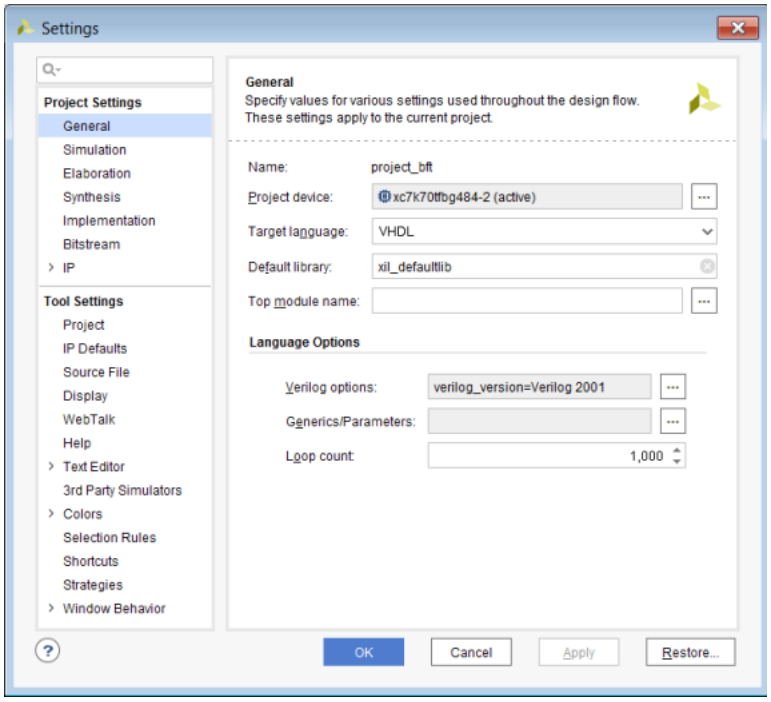

Fig 5.4: Settings Dialog Box—Project Settings General Category

To open the Settings dialog box, use any of the following methods:

• In the Flow Navigator Project Manager section, click Settings.

• Select Tools > Settings.

• In the main toolbar, click the Settings toolbar button.

• In the Project Summary, click the Edit link next to Settings.

5.8.8 Synthesis and Implementation of the Design*:*

The design has to be synthesized and implemented before it can be checked for correctness, by running functional simulation or downloaded onto the prototyping board. With the top-level Verilog file opened (can be done by double-clicking that file) in the HDL editor window in the right half of the Project Navigator, and the view of the project being in the **Module view** , the **implement design** option can be seen in the **process view**. **Design entry utilities** and **Generate Programming File** options can also be seen in the process view.

To synthesize the design, double click on the **Synthesize Design** option in the **Processes window**.

To implement the design, double click the **Implement design** option in the **Processes window**. It will go through steps like **Translate, Map and Place & Route**. If any of these steps could not be done or done with errors, it will place a **X** mark in front of that, otherwise a tick mark will be placed after each of them to indicate the successful completion

After synthesis right click on synthesis and click view text report in order to generate the report of our design.

5.9 XILINX VIVADO SIMULATION PROCEDURE:

After completion of synthesis we will go simulation in order to verify the functionality of the implemented design.

Click on **Run Simulation** and set the module that is need to Run

Next **double click on Run Behavioral Simulation** to check the errors. If no errors are found then double click on simulate behavioral model to get the output waveforms.

After clicking on **simulate behavioral model**, the simulation widow will appear pass the input values by making force constant and if it is clock by making force clock. Mention the simulation period and run for certain time and results will appear as shown in following window. Verify the results to the given input values.

5.9.1 Using the Schematic Window:

You can generate a Schematic window for any level of the logical or physical hierarchy. You can select a logic element in an open window, such as a primitive or net in the Netlist window, and use the Schematic command in the popup menu to create a Schematic window for the selected object.
An elaborated design always opens with a Schematic window of the top-level of the design, as shown in figure 5.5.

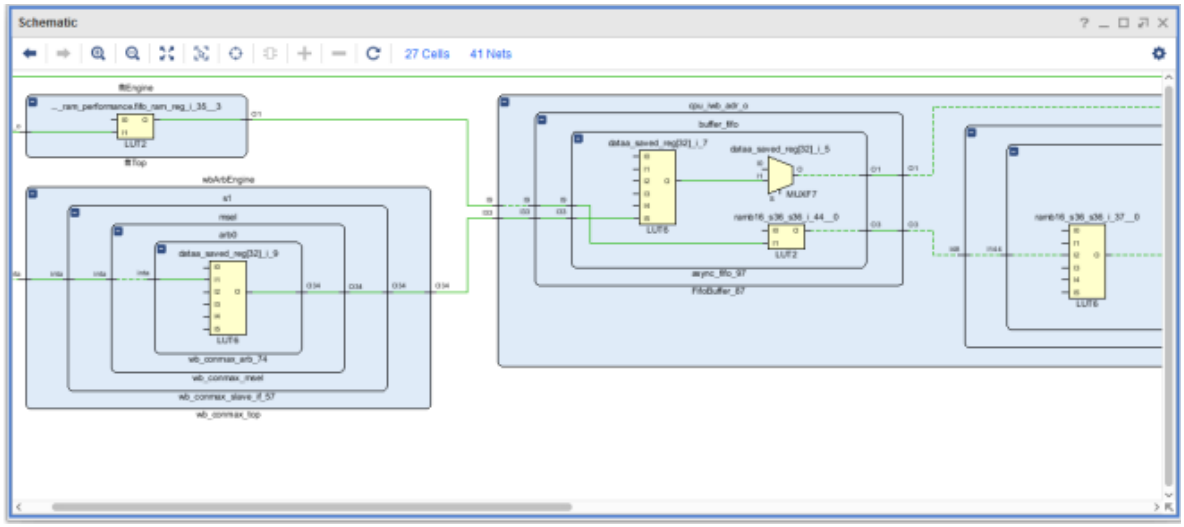

Fig 5.5: Schematic window

5.9.2 Using the Project Summary

The Vivado IDE includes an interactive Project Summary (Figure 3-11) that updates dynamically as design commands are run and the design progresses through the design flow. It provides project and design

information, such as the project part, board, and state of synthesis and implementation.

It also provides links to detailed information, such as links to the Messages and Reports windows as well as the Settings dialog box.

As synthesis and implementation complete, DRC violations, timing values, utilization percentages, and power estimates are also populated. To open the Project Summary, do either of the following:

- Select Window > Project Summary.
- Click the Project Summary toolbar button.

Fig 5.6: Project Summary

CHAPTER 6

RESULTS

6.1 MULTIPLEXER

A multiplexer (or mux; spelled sometimes as multiplexor), also known as a data selector, is a device that selects between several analog or digital input signals and forwards the selected input to a single output line. The selection is directed by a separate set of digital inputs known as select lines.

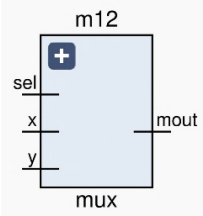

Figure 6.1: Multiplexer

6.2 LOOKUP TABLE

A lookup table is an array of data that maps input values to output values, thereby approximating a mathematical function. Given a set of input values, a lookup operation retrieves the corresponding output values from the table. The Schematic of LUT will be given as Figure 6.2.

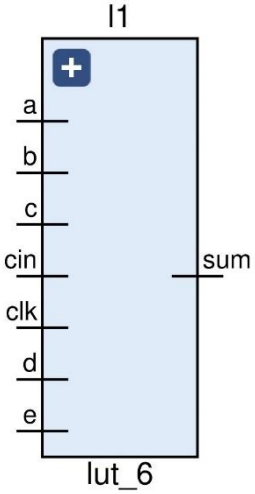

Figure 6.2: Look up table

6.3 AAD2

An approximate 2-bit adder (AAD2) is proposed suited for building an adder tree to accumulate partial product (PP) because it is less complicated than traditional adders. Compared to a one-bit-full adder, the critical path delay (CPD) is reduced significantly in the proposed methods. The Schematic of AAD2 is shown in Figure 6.3.

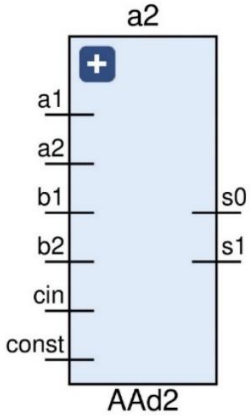

Figure 6.3: AAD2

6.4 CARRY CHAIN

The carry chain provides a fast carry function between the dedicated adders in the arithmetic mode.

The Schematic of the Carry chain is given at figure 6.4.

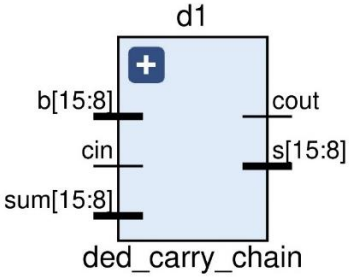

Figure 6.4: Carry chain

6.5 OUTPUT BUFFER

This design element is a simple output buffer used to drive output signals to the FPGA device pins that do not need to be 3-stated (constantly driven). Either an OBUF, OBUFT, OBUFDS, or OBUFTDS must be connected to every output port in the design. The schematic of OBUF will be given at Figure 6.5.

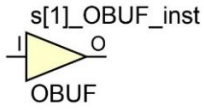

Figure 6.5: Output Buffer

6.6 VCC

VCC is the most common power supply pin, usually used to provide the forward voltage required in digital circuits. The voltage of VCC is usually 3.3V or 5V, but there are microcontrollers with other voltage levels. The schematic of VCC unit will be taken as shown in Figure 6.6.

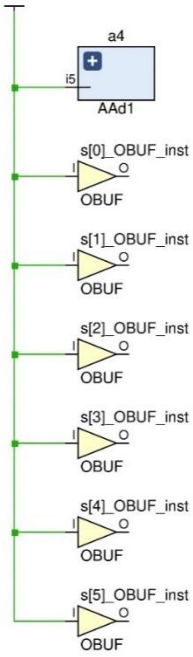

Figure 6.6: VCC

6.7 GROUND

In electronics, "ground" typically refers to a common reference point in a circuit against which other voltages are measured. Ground serves as a return path for electric current in a circuit and is often connected to the earth.

In practical terms, the ground connection helps stabilize voltages and provides a stable reference point for the entire circuit. It's crucial for safety reasons as well, as it can help prevent electric shocks and ensure proper functioning of electrical devices.

The Schematic of the Ground Circuit used in this Project will be given at Figure 6.7.

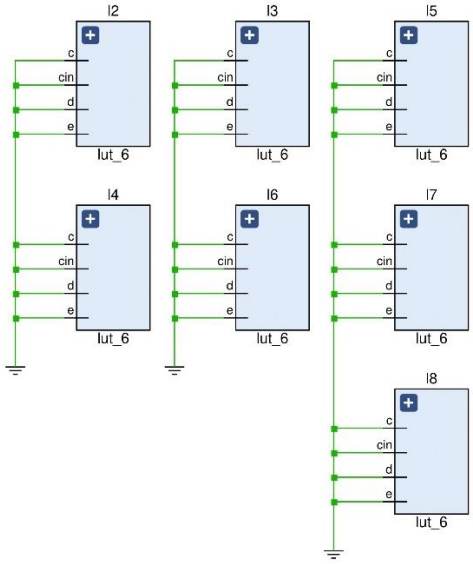

Figure 6.7: Ground

6.8 LEADX

These findings demonstrate that compared to the FPGA specific adders in the literature, our suggested LEADx has a smaller size, lower power, and superior quality. The findings demonstrate that, among FPGA-specific

approximate adders in the literature, DeMAS is the most effective. The RTL Schematic of the LEADx will be presented at Figure 6.8

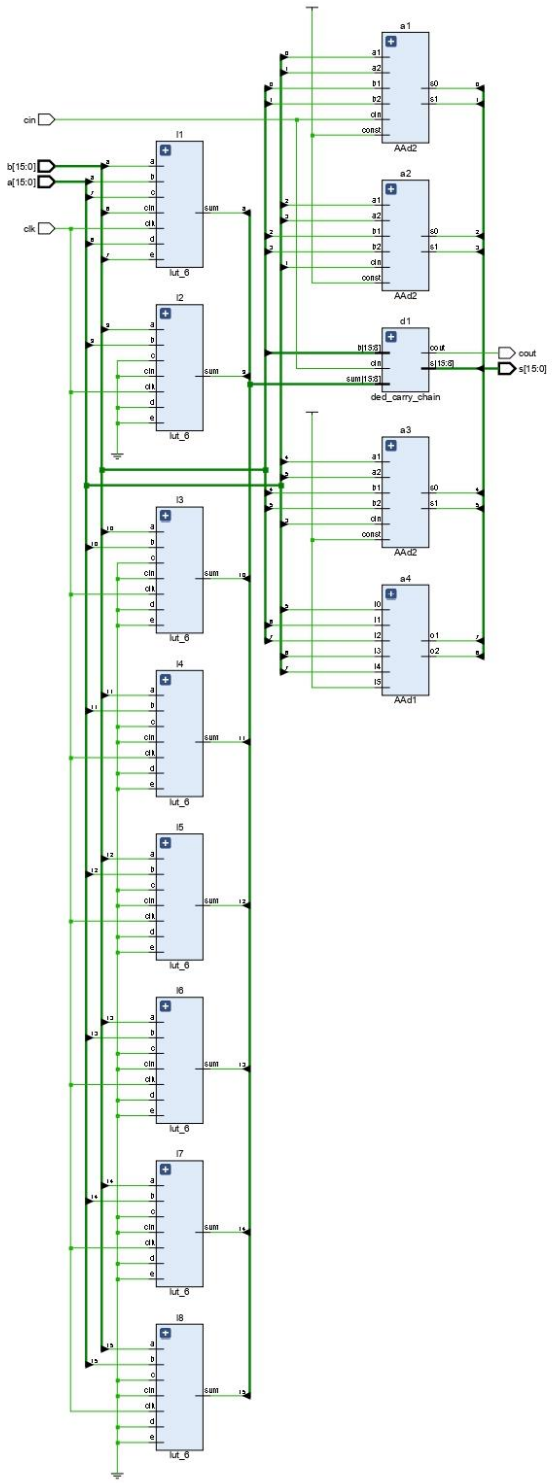

Fig 6.8: RTL Schematic of LEADx

When implemented on an FPGA, APEX, however, uses less power and produces better results than HOAANED at the same cost. APEX possesses over 60% lower MSE than HOAANED at the same cost. The Behavioural simulation of the LEADx will be illustrated at Figure 6.9.

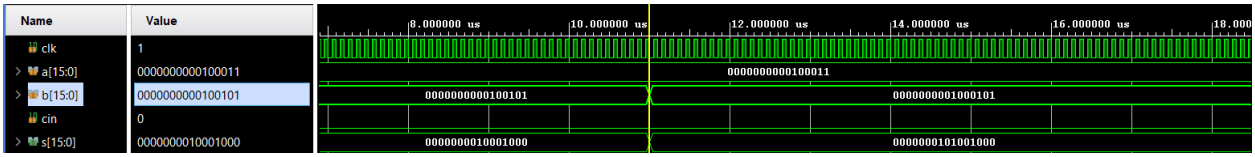

Fig 6.9: Simulation results of LEADx

Compared to DeMAS, LEADx has an 86% lower MSE and a 7% smaller area using an 8-bit approximation. Area consumed by the LEADx will be me ntioned at Figure 6.10. According to the research, LOA is among the most effective ASIC-based approximation adders. When implemented on an FPGA, LEADx outperforms LOA in terms of quality for the same cost.

Name	Slice LUTs (134600)	Slice Registers (269200)	Bonded IOB (400)	BUFGCTRL (32)
N LEADx	18	1	51	1
I1 (lut_6)	14	1	0	0

Fig 6.10: Area consumed by LEADx

While most approximate adders exhibit a linear decrease in LUTs with increasing approximation levels, the corresponding power reductions do not consistently follow this trend. However, APEX notably achieves significant power reduction compared to the accurate adder, albeit with a slight compromise in accuracy. The Delay of APEX will be mentioned at Figure 6.11.

Name	Slack	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock	Destination Clock	Exceptio
Path 1	∞	7	8	3	a[8]	cout	6.511	3.767	2.744	∞	input port clock		

Fig 6.11: Delay Produced by LEADx

By covering approximation levels from 4-bit to 20-bit within a 32-bit adder. Notably, the accurate 32-bit adder utilizes 32 LUTs and consumes 10.75 mW of power. While most approximate adders exhibit a linear decrease in LUTs with increasing approximation levels, the corresponding power reductions do not consistently follow this trend. The power consumption of LEADx will be given at Figure 6.12.

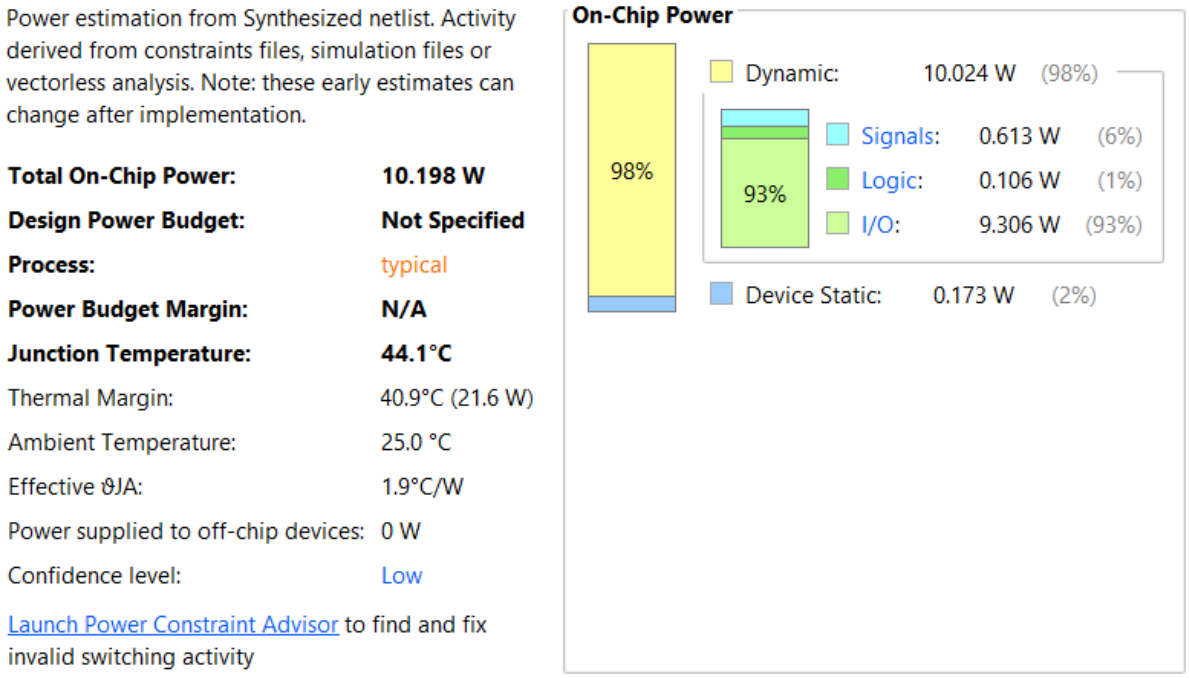

Fig 6.12: Power Consumption of LEADx

6.9 APEX

APEX and HOAANED exhibit the most efficient utilization of LUTs, with their usage being the lowest among all adders. This reduction in LUTs is primarily attributed to the integration of constant functions in their LSPs. Conversely, other approximate adders achieve LUT reduction through approximation techniques, enabling synthesis tools to consolidate two sum outputs into a single LUT. The RTL and Technological Schematics of APEX will be mentioned at Figure 6.13.

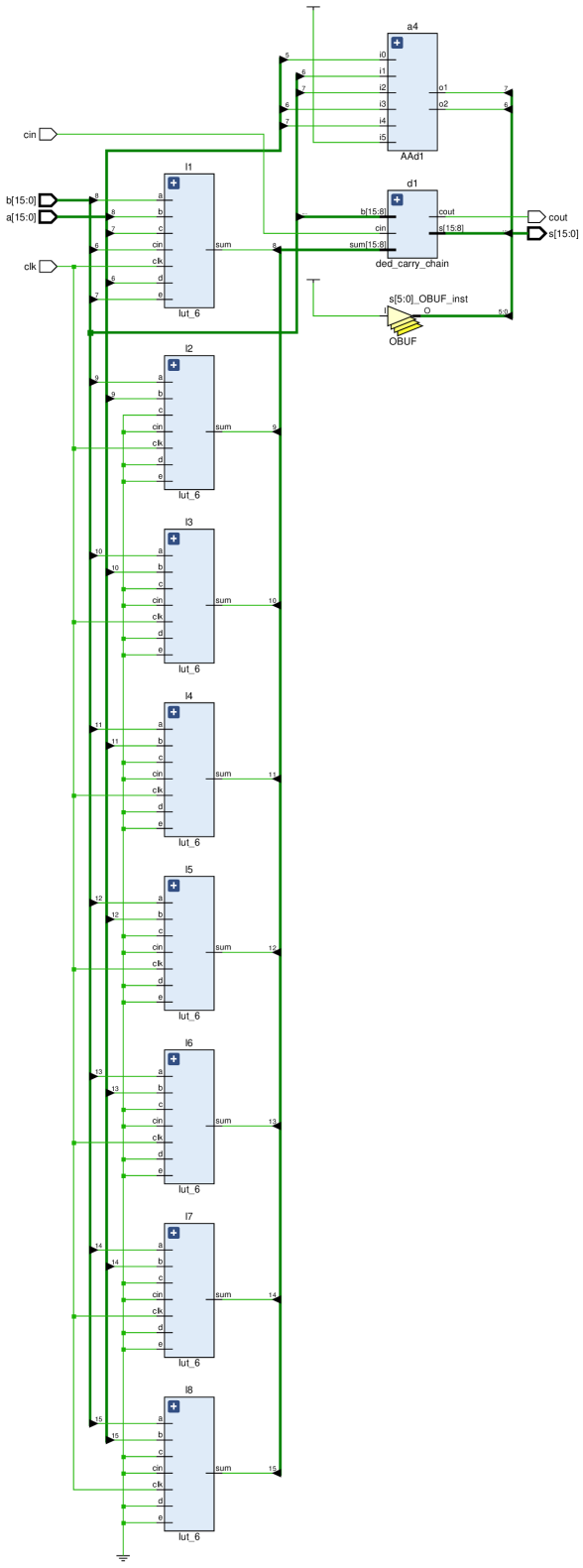

Fig 6.13: RTL Schematic of APEX

However, APEX notably achieves significant power reduction compared to the accurate adder, albeit with a slight compromise in accuracy. The reductions in LUTs, power consumption, and delay achieved by 64-bit approximate adders with 16-bit approximation compared to the accurate 64-bit adder. LEADx reduces LUTs by 12.5% compared to the accurate adder, whereas APEX surpasses with a 23.4% reduction in LUTs and a 21% decrease in power consumption compared to the accurate adder. The simulation results of APEX will be given at Figure 6.14.

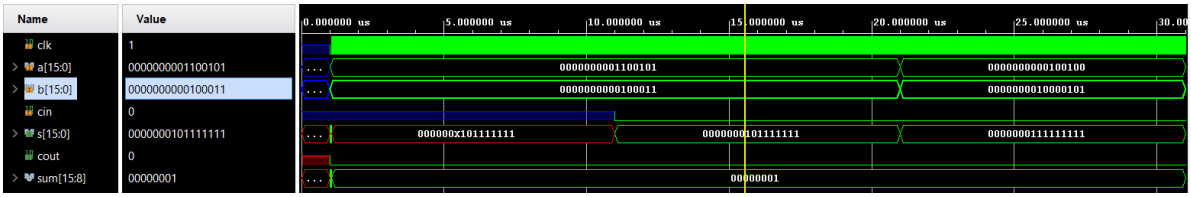

Fig 6.14: Simulation results of APEX

LEADx demonstrates a marginally lower power consumption than the accurate adder, while APEX stands out as the most power-efficient among all approximate adders. Specifically, in a 16-bit adder with 8-bit approximation, APEX consumes 29% less power than the accurate adder and 4.5% less than the secondlowest power-consuming adder, HOAANED. The Area of APEX will be shown at Figure 6.15.

Name	Slice LUTs (134600)	Slice Registers (269200)	Bonded IOB (400)	BUFGCTRL (32)
N APEx	15	1	40	1
I1 (lut_6)	14	1	0	0

Fig 6.15: Area consumed by APEX

While most approximate adders exhibit a linear decrease in LUTs with increasing approximation levels, the corresponding power reductions do not consistently follow this trend. However, APEX notably achieves significant power reduction compared to the accurate adder, albeit with a slight

compromise in accuracy. The Delay of APEX will be mentioned at Figure 6.16.

Name	Slack	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock	Destination Clock	Exceptio
Path 1	∞	7	8	3	a[8]	cout	6.511	3.767	2.744	∞	input port clock		

Fig 6.16: Delay Produced by APEX

By covering approximation levels from 4-bit to 20-bit within a 32-bit adder. Notably, the accurate 32-bit adder utilizes 32 LUTs and consumes 10.75 mW of power. While most approximate adders exhibit a linear decrease in LUTs with increasing approximation levels, the corresponding power reductions do not consistently follow this trend. The power consumption of LEADx will be given at Figure 6.17

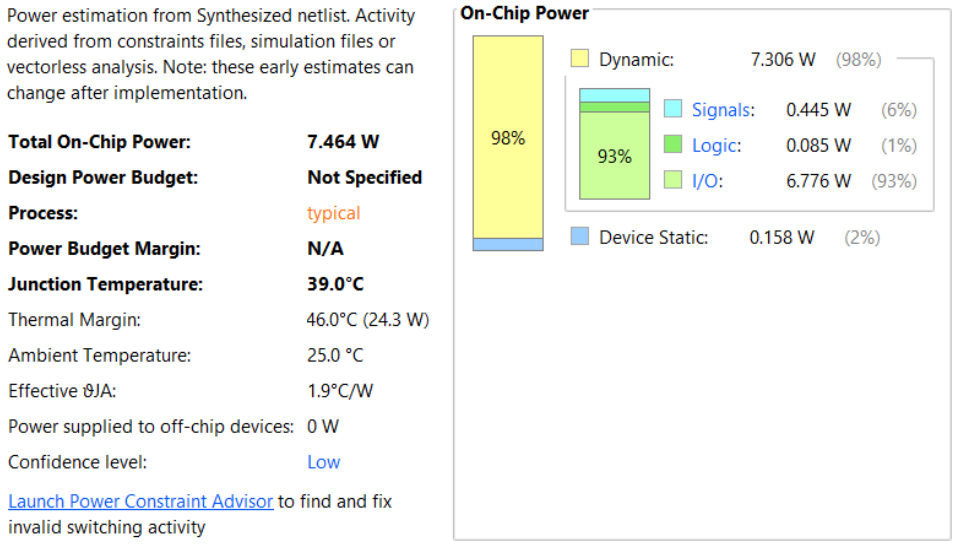

Fig 6.17: Power Consumption of APEX

Table 6.1: Evaluation table

	Area (LUT's)	**POWER**	**Delay (ns)**
LEADx	18	10.198	7.511
APEx	15	7.464	7.511

CHAPTER 7

CONCLUSION AND FUTURE SCOPE

In this paper, two low error efficient approximate adders for FPGAs are proposed, where approximation is done only in the LSP and the MSP is kept accurate. The first approximate adder, LEADx, achieves better area and delay compared to exact adder implementations. The second approximate adder, APEx, has reduced area, and less power consumption than the existing adders. It has smaller area and lower power consumption than the other approximate adders in the literature. Therefore, the proposed approximate adders can be used for FPGA implementations of error tolerant applications.

7.1 FUTURE SCOPE:

In the future the proposed adders can be designed efficiently by establishing a trade-off between error and parameters like area, delay and power by making efficient ways in carry chain or accurate adders etc. and as an application it can also be implemented in multiplier architectures at the partial product reduction process thereby improving the efficiency of the design.

REFERENCES:

[1] G. A. Gillani, M. A. Hanif, B. Verstoep, S. H. Gerez, M. Shafique, and A. B. J. Kokkeler, "MACISH: Designing approximate MAC accelerators with internal-self-healing," IEEE Access, vol. 7, pp. 77142–77160, 2019.

[2] E. Kalali and I. Hamzaoglu, "An approximate HEVC intra angular prediction hardware," IEEE Access, vol. 8, pp. 2599–2607, 2020.

[3] T. Ayhan and M. Altun, "Circuit aware approximate system design with case studies in image processing and neural networks," IEEE Access, vol. 7, pp. 4726–4734, 2019.

[4] W. Ahmad and I. Hamzaoglu, "An efficient approximate sum of absolute differences hardware for FPGAs," in Proc. IEEE Int. Conf. Consum. Electron. (ICCE), Las Vegas, NV, USA, Jan. 2021, pp. 1–5.

[5] H. Jiang, C. Liu, L. Liu, F. Lombardi, and J. Han, "A review, classification, and comparative evaluation of approximate arithmetic circuits," ACM J. Emerg. Technol. Comput. Syst., vol. 13, no. 4, pp. 1–34, Aug. 2017.

[6] A. C. Mert, H. Azgin, E. Kalali, and I. Hamzaoglu, "Novel approximate absolute difference hardware," in Proc. 22nd Euromicro Conf. Digit. Syst. Design (DSD), Kallithea, Greece, Aug. 2019, pp. 190–193.

[7] N. Van Toan and J.-G. Lee, "FPGA-based multi-level approximate multipliers for high-performance error-resilient applications," IEEE Access, vol. 8, pp. 25481–25497, 2020.

[8] L. Chen, J. Han, W. Liu, P. Montuschi, and F. Lombardi, "Design, evaluation and application of approximate high-radix dividers," IEEE Trans. Multi-Scale Comput. Syst., vol. 4, no. 3, pp. 299–312, Jul. 2018.

[9] A. K. Verma, P. Brisk, and P. Ienne, "Variable latency speculative addition: A new paradigm for arithmetic circuit design," in Proc. Design, Automat. Test Eur. (DATE), Munich, Germany, Mar. 2008, pp. 1250–1255.

[10] N. Zhu, W. L. Goh, G. Wang, and K. S. Yeo, "Enhanced low-power highspeed adder for error-tolerant application," in Proc. Int. SoC Design

Conf., Incheon, South Korea, Nov. 2010, pp. 323–327.

[11] B. Kahng and S. Kang, "Accuracy-configurable adder for approximate arithmetic designs," in Proc. 49th Annu. Design Automat. Conf. (DAC), New York, NY, USA, 2012, pp. 820–825.

[12] V. Gupta, D. Mohapatra, A. Raghunathan, and K. Roy, "Low-power digital signal processing using approximate adders," IEEE Trans. Comput.-Aided Design Integr. Circuits Syst., vol. 32, no. 1, pp. 124–137, Jan. 2013.

[13] W. Ahmad, B. Ayrancioglu, and I. Hamzaoglu, "Comparison of approximate circuits for H.264 and HEVC motion estimation," in Proc. 23rd Euromicro Conf. Digit. Syst. Design (DSD), Kranj, Slovenia, Aug. 2020, pp. 167–173.

[14] H. R. Mahdiani, A. Ahmadi, S. M. Fakhraie, and C. Lucas, "Bio-inspired imprecise computational blocks for efficient VLSI implementation of soft computing applications," IEEE Trans. Circuits Syst. I, Reg. Papers, vol. 57, no. 4, pp. 850–862, Apr. 2010.

[15] A. Dalloo, A. Najafi, and A. Garcia-Ortiz, "Systematic design of an approximate adder: The optimized lower part constant-OR adder," IEEE Trans. Very Large Scale Integr. (VLSI) Syst., vol. 26, no. 8, pp. 1595–1599, Aug. 2018.

[16] P. Balasubramanian, R. Nayar, D. L. Maskell, and N. E. Mastorakis, "An approximate adder with a near-normal error distribution: Design, error analysis and practical application," IEEE Access, vol. 9, pp. 4518–4530, 2021.

[17] S. Dutt, S. Nandi, and G. Trivedi, "Analysis and design of adders for approximate computing," ACM Trans. Embedded Comput. Syst., vol. 17, p. 40, Dec. 2017.

[18] D. Celia, V. Vasudevan, and N. Chandrachoodan, "Optimizing poweraccuracy trade-off in approximate adders," in Proc. Design, Automat. Test Eur. Conf. Exhib. (DATE), Dresden, Germany, Mar. 2018, pp. 1488–1491.

[19] A. Becher, J. Echavarria, D. Ziener, S. Wildermann, and J. Teich, “A LUT based approximate adder,” in Proc. IEEE 24th Annu. Int. Symp. Field Program. Custom Comput. Mach. (FCCM), Washington, DC, USA, May 2016, p. 27.

[20] B. S. Prabakaran, S. Rehman, M. A. Hanif, S. Ullah, G. Mazaheri, A. Kumar, and M. Shafique, “DeMAS: An efficient design methodology for building approximate adders for FPGA-based systems,” in Proc. Design, Automat. Test Eur. Conf. Exhib. (DATE), Dresden, Germany, Mar. 2018, pp. 917–920.

[21] S. Boroumand, H. P. Afshar, and P. Brisk, “Approximate quaternary addition with the fast carry chains of FPGAs,” in Proc. Design, Automat. Test Eur. Conf. Exhib. (DATE), Dresden, Germany, Mar. 2018, pp. 577–580.