A Mini Project Report

Entitled

VLSI Implementation of Approximate MAC Units

Submitted to the Department of Electronics Engineering
In Partial Fulfillment of the Requirements for the Degree of

MASTER OF TECHNOLOGY
(VLSI & EMBEDDED SYSTEMS)

SUBMITTED BY

Mr. RAJA BHARGAV

(Roll. No. P24VL005)

: GUIDE:

Prof. Dr. Zuber M.Patel
Associate Professor
ECED, SVNIT

DEPARTMENT OF ELECTRONICS ENGINEERING
SARDAR VALLABHBHAI NATIONAL INSTITUTE OF TECHNOLOGY, SURAT-395007
YEAR: 2024-25

Sardar Vallabhbhai National Institute of Technology

Surat - 395 007, Gujarat, India

DEPARTMENT OF ELECTRONICS ENGINEERING

CERTIFICATE

This is to certify that RAJA BHARGAV, P24VL005, Full-time M. Tech. student has presented and submitted a his Mini Project report on 'VLSI Implementation of Approximate MAC Units', in partial fulfillment of the requirement for the award of the degree **MASTER OF TECHNOLOGY** in ELECTRONICS ENGINEERING with specialization in **VLSI & EMBEDDED SYSTEMS** during the year 2024-2025.

Guide	Examiner1	Examiner2

P.G. Incharge (VLSI & ES)	Head of the Department

Seal of the Department

Jan-May 2025

Acknowledgement

I would like to express my heartfelt gratitude to my guide, Prof. Dr. Zuber M.Patel, for their invaluable guidance, constant encouragement, and insightful expertise throughout the preparation of this Mini Project report on "VLSI Implementation of Approximate MAC units." Their consistent support and constructive feedback were pivotal in shaping this work.

I also extend my deep appreciation to Prof. Dr. Jignesh N. Sarvaiya, Head of the Department, Dept. Of ECE for their encouragement and for providing the necessary infrastructure and resources to complete this report successfully.

Raja Bhargav
P24VL005
S.V. National Institute of Technology, Surat

Date:
Place:

Abstract

Multiply-Accumulate (MAC) units form the computational backbone of most machine learning and deep learning architectures, particularly in convolution and fully connected layers. However, the high computational demand of these applications leads to increased energy consumption and silicon area, making them less suitable for edge devices with limited resources. In this project, we propose an Approximate MAC unit that integrates a Wallace Tree-based multiplier and Carry Skip Adders (CSAs) to achieve efficient computation with reduced power, delay, and area. Approximation techniques are strategically applied within the multiplier and adder blocks to minimize complexity while tolerating small, application-agnostic errors. The design targets energy-constrained scenarios, such as neural network inference on embedded systems, where exact precision is not critical. Simulation results demonstrate that the proposed architecture offers a favorable trade-off between computational accuracy and hardware efficiency, making it a promising candidate for next-generation approximate hardware accelerators in deep learning applications.

Declaration

I declare that this written submission represents my ideas in my own words and where others' ideas or words have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

Raja Bhargav
P24VL005

Date:
Place:

:: CONTENT ::

***

List of Figures

***

List of Tables

***

List of Acronyms

Acronym	Full Form
VLSI	Very Large Scale Integration
MAC	Multiply-Accumulate
ML	Machine Learning
DL	Deep Learning
DSP	Digital Signal Processing
CSA	Carry Save Adder
CLA	Carry Look Ahead Adder
FFT	Fast Fourier Transform
FMA	Fused Multiply-Add
IoT	Internet of Things
CFA	Circuit Functional Approximation
VOS	Voltage Over-Scaling
OC	Over-Clocking
AWTM	Approximate Wallace Tree Multiplier
IMPACT	Inexact Multiplier-based Power and Area-Constrained Technology (Approx. Adder)
AXA	Approximate XOR/XNOR-based Adder
RCPA	Reverse Carry Propagation Adder
OLOCA	One-Level-Only Configurable Approximate Adder

Acronym	**Full Form**
ACA	Accuracy-Configurable Adder
GDA	Graceful Degradation Adder
GeAr	Generic Accuracy-Reconfigurable Adder
RAP-CLA	Reconfigurable Approximate Parallel CLA
SARA	Simple Approximate Ripple-carry Adder
BSCA	Block Speculative Carry Adder
DRUM	Dynamic Range Unbiased Multiplier
RoBa	Rounding-Based Approximate Multiplier
PR	Partial Product Reduction (context-specific)
AFMU	Approximate Floating-point Multiplier Unit
TOSAM	Truncated Operand Symmetric Approximate Multiplier
REALM	Range Enhanced Approximate Logarithmic Multiplier
ILM	Improved Logarithmic Multiplier
TLM	Two-Level Logarithmic Multiplier
CNN	Convolutional Neural Network
PPA	Power, Performance, Area
K-Map	Karnaugh Map

***

Chapter: 1
Introduction

1.1 Introduction

Machine learning (ML) and deep learning (DL) have witnessed rapid growth in recent years, with applications ranging from computer vision and natural language processing to autonomous systems and healthcare. These applications typically rely on highly parallel and compute-intensive operations, most notably the Multiply-Accumulate (MAC) operation. In neural networks, especially during inference, a large number of MAC operations are performed across layers to compute activations, making MAC units a key determinant of system performance, power consumption, and silicon area.

While precise arithmetic is traditionally preferred, studies have shown that many ML/DL algorithms are inherently error-tolerant, especially during inference. This opens up the opportunity to apply approximate computing techniques to reduce power and area without significantly degrading model accuracy. Approximate arithmetic units, particularly approximate MACs, are gaining attention for their potential to optimize hardware for energy-constrained edge devices such as smartphones, IoT nodes, and wearable AI systems.

In this project, we present an Approximate MAC architecture optimized for deep learning inference tasks. The design employs a Wallace Tree-based multiplier, known for its fast partial product reduction, and a Carry Skip Adder (CSA) structure, which accelerates addition by bypassing carry chains. Approximation is introduced at strategic stages to reduce logic complexity, thus improving energy efficiency and speed.

The key objectives of this work are:

a. To design an approximate MAC unit with low power, reduced delay, and compact area.
b. To evaluate the trade-offs between accuracy and hardware efficiency.

c. To demonstrate the applicability of the proposed MAC in ML/DL workloads, especially on edge devices.

1.2 Details of MAC

A Multiply-Accumulate (MAC) unit is a fundamental building block in digital signal processing (DSP) and various computational tasks. It performs a series of mathematical operations crucial for applications like convolution, filtering, and machine learning.

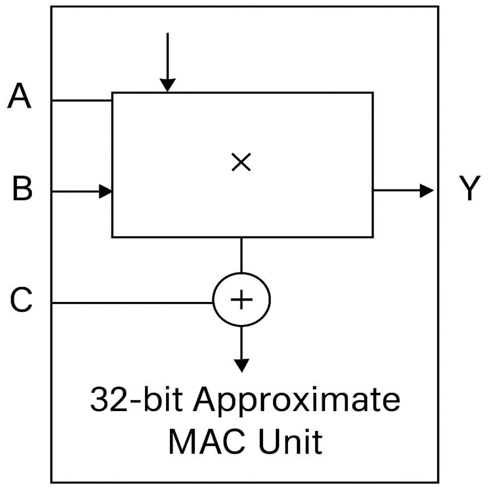


Fig 1.1 Block diagram of MAC

1.2.1 Purpose of a MAC Unit:

The MAC unit is designed to perform the following operation efficiently:

$$\text{MAC Output} = \sum(A_i \times B_i) + C$$

Where , Ai and Bi are the input operands (typically integers or floating-point numbers).

C is an accumulation value (previous result or bias).

The product of A i and Bi is computed, and the result is added to C.

1.2.2 Applications of MAC Units:

Digital Signal Processing (DSP): Fast Fourier Transform (FFT), filtering, and convolution.

Machine Learning and AI: Dot product calculations in neural network layers.

Graphics Processing: Image processing, computer vision, and 3D transformations.

Embedded Systems: Real-time audio and video processing.

1.2.3 Basic Operation of a MAC Unit:

Multiplication: Takes two input values (A and B) and computes their product (P=A×B).

Accumulation: Adds the product to an accumulated sum (S=S+P).

Result Storage: The result is stored in an accumulator register.

1.2.4 MAC Unit Architecture:

The architecture of a typical MAC unit includes the following components:

Multiplier: A hardware module (like a Wallace tree multiplier or Booth multiplier) to multiply the inputs.

Adder: Adds the product to the accumulated value.

Accumulator Register: Holds the ongoing sum to be updated with every clock cycle.

Control Logic: Manages the sequencing of operations, especially in pipelined designs.

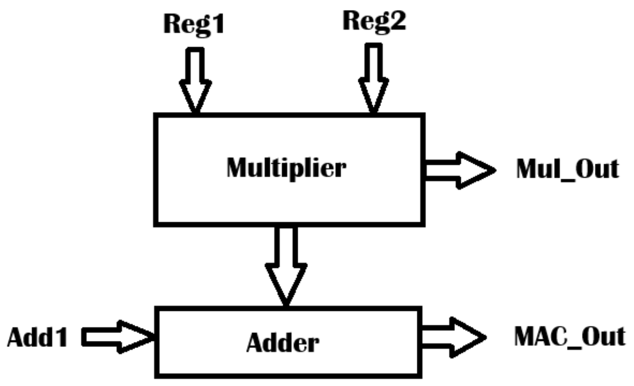


Fig 1.2 MAC Architecture

1.2.5 Key Characteristics:

Throughput: The number of MAC operations performed per second. Measured in MACs per second (MAC/s).

Latency: The time taken to complete one MAC operation. Pipelining can reduce the effective latency.

Precision: Supports fixed-point or floating-point arithmetic, depending on the application. Precision trade-offs impact the accuracy of results.

1.2.6 Variants of MAC Units:

Approximate MAC Units: Used in scenarios where a bit of error is tolerable (e.g., image processing) to save power and area.

Example: Accuracy-configurable MAC units for energy-efficient applications.

Fused Multiply-Add (FMA) Units: Perform multiplication and addition in a single step to reduce rounding errors.

Common in floating-point computations (e.g., IEEE 754).

1.3 Organization of Report

This report is organized into several key sections:

1. **Introduction**: Provides an overview of the growing demand for energy-efficient and high-performance arithmetic circuits, particularly in applications such as machine learning and signal processing. It introduces the concept of approximate computing and its relevance to MAC units.
2. **Background**: Reviews the conventional MAC unit architectures and their limitations in terms of power consumption and area. It covers related work and the significance of Wallace Tree multipliers and carry-save addition.
3. **Proposed Architecture**: Describes the approximate MAC unit designed using a Wallace Tree multiplier for partial product reduction and a 32-bit Carry Save Adder

with embedded 5-bit Carry Look Ahead Adders for efficient summation. Design considerations and architectural benefits are detailed.

4. **Implementation and Results**: Outlines the simulation environment, implementation methodology, and performance metrics. The results demonstrate improvements in terms of speed and area efficiency, with acceptable trade-offs in computational accuracy.
5. **Conclusion and Future Work**: Summarizes the outcomes and suggests possible directions for further enhancement.

***

Chapter: 2
Background

2.1 Introduction to Multiply-Accumulate (MAC) Units

The Multiply-Accumulate (MAC) operation is a fundamental computational primitive widely used in digital signal processing (DSP), machine learning (ML), and scientific computing. It involves multiplying two numbers and accumulating the result into a register:

$$MAC=\sum(Ai \times Bi)$$

Due to its extensive usage in neural networks, convolution operations, and matrix multiplications, the performance of MAC units significantly impacts the overall system efficiency.

2.2 Motivation for Approximate Computing

The rise of data-intensive applications leads to an increasing demand for high-performance and energy-efficient hardware. In many of these applications (e.g., image processing, machine learning), perfect accuracy is not always necessary. This gives rise to Approximate Computing, a design paradigm that exploits the inherent error tolerance of such applications to trade off computational accuracy for improvements in power, area, and speed.

2.3 Hardware Approximate Computation Techniques

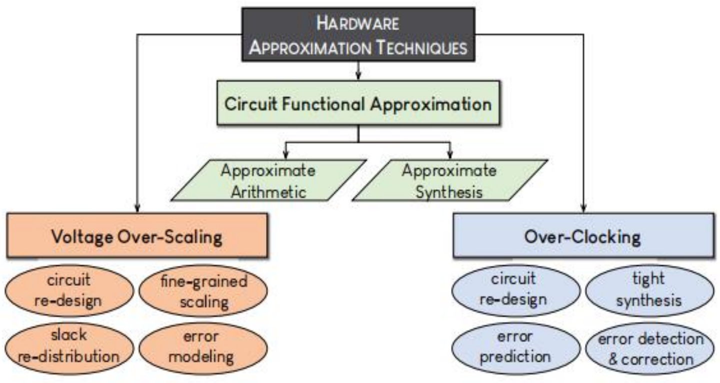


Fig 2.1 Classification of HW approximation techniques in three classes: Circuit Functional Approximation,Voltage Over-Scaling, and Over-Clocking[1].

An overview of hardware-level approximation techniques that operate at the lower tiers of the digital design abstraction hierarchy. These methods are primarily aimed at enhancing key performance metrics such as area efficiency, power reduction, and computational speed—especially in the context of accelerators, processors, and hardware computing platforms. Here are different hardware approximation classes listed below table.

HW Approximation Class	**Technique / Approach**
Adder Approximation	Use of Approximate Full Adder Cells
	Segmentation and Carry Prediction
Multiplier Approximation	Truncation and Rounding
	Approximate Radix Encodings
	Use of Approximate Compressors
	Logarithmic Approximation
Divider Approximation	Bit-width Scaling
	Use of Approximate Adder/Subtractor Cells
	Simplification of Computations
Approximate Synthesis	Structural Netlist Transformation
	Boolean Rewriting
	High-Level Approximate Description
	Evolutionary Synthesis
Voltage Over-Scaling	Slack Re-distribution
	Circuit Re-design and Architecture Modification
	Fine-Grained Scaling
	Error Modeling
Over-Clocking	Tight Synthesis
	Circuit Re-design and Architecture Modification
	Error Detection & Correction
	Error Prediction

Table 2.1: Classification of Circuit functional Approximation Techniques[1]

Hardware approximation techniques are generally divided into three main categories:
(i) Circuit Functional Approximation (CFA),
(ii) Voltage Over-Scaling (VOS), and
(iii) Over-Clocking (OC).

Each category introduces trade-offs between accuracy and efficiency. Functional approximation (CFA) deliberately modifies the logic or architecture of a circuit to simplify its operation, resulting in functional errors. In contrast, VOS and OC techniques primarily target energy and speed improvements by relaxing timing constraints, which can lead to timing errors due to insufficient setup or hold times. A classification diagram (see Figure 2.1) further breaks down these techniques into their respective subcategories, providing a detailed taxonomy of approximation strategies used in hardware design. In this project we worked on circuit functional Approximation, where mostly on Adders and multipliers for MAC operation.

2.4 Circuit Functional Approximation

Circuit Functional Approximation (CFA) involves simplifying an accurate circuit design at the logic level to reduce its complexity. This simplification helps in saving power and improving performance, especially in power-hungry applications. Common CFA techniques include:

1. Modifying the original truth table of the circuit,
2. Replacing the exact hardware algorithm with an approximate version,
3. Using small inexact components as basic building blocks, and
4. Applying approximate synthesis methods during circuit generation.

CFA is primarily applied to arithmetic circuits, since they are central to processors and hardware accelerators. Because of their heavy use in computations, optimizing these circuits through approximation can significantly enhance overall system efficiency.

2.5 Approximate MAC unit

Approximate MAC units aim to reduce the energy and area cost of conventional MAC architectures by introducing controlled inaccuracies in either the multiplier, the adder, or both. These units are particularly effective in applications where: Minor inaccuracies do not significantly affect output quality (e.g., object recognition). Power and area savings are prioritized over exact numerical correctness.

2.5.1 Approximate Adders:

Extensive research has been conducted on designing approximate adders to achieve reductions in area and power consumption, which are crucial for energy-efficient systems. Broadly, approximation in adders is achieved through two main techniques:

Using inexact full adder cells, and Applying segmentation and carry prediction methods. One of the well-known designs, the IMPACT adder, utilizes simplified full adder cells at the transistor level, enabling up to 45% area savings. Similarly, the AXA adder uses a 4-transistor XOR/XNOR structure that significantly reduces dynamic power consumption by 31%. The RCPA adder employs a novel approach using reverse carry propagation in its hybrid design to enhance performance. On a higher abstraction level, the OLOCA adder divides the addition operation into two segments: an accurate portion and an approximate portion, where simple logic like OR gates and fixed outputs (e.g., constant '1') are used to simplify computation. To further reduce delay due to carry propagation, researchers have proposed carry prediction-based techniques.

For instance, Kim et al. designed a predictor that uses the less significant bits to estimate carry values, achieving speeds 2.4× faster than conventional ripple-carry adders. Similarly, Hu and Qian developed a carry speculation mechanism with built-in error and sign correction, resulting in 4.3× speed improvement and 47% power reduction.

Given that application accuracy requirements may change dynamically, recent studies have focused on configurable approximate adders. For example: The ACA (Accuracy-Configurable Adder) includes multiple sub-adders and a runtime error detection-correction module, supporting both approximate and accurate modes.

The GDA (Graceful Degradation Adder) selects between carry sources to adjust accuracy on-the-fly.

The GeAr adder supports configurable approximation modes using uniform sub-adder blocks with a correction unit. The RAP-CLA adder splits the carry look-ahead logic into accurate and approximate sections, with power-gating applied to save energy during approximate operation. The SARA design avoids extra prediction circuitry by using a lightweight carry estimation approach in ripple-carry sub-adders. Lastly, the BSCA adder adopts a block-based speculative strategy, integrating sub-adders, prediction units, and correction logic to balance performance and error resilience.

2.5.2 Approximate Multipliers

Multipliers are another major target of approximation, given their central role in digital signal processing and machine learning workloads. Approximate multiplication techniques can be grouped into four main categories:

1. Truncation and Rounding:

These techniques simplify computation by reducing the operand bit-width or discarding insignificant partial products. For instance, the DRUM multiplier[12] adjusts bit-width dynamically based on leading '1's and achieves up to 60% power savings. The RoBa multiplier[13] uses exponent-of-two rounding and segmented shifting, while the PR multiplier skips selected partial products for energy efficiency. These methods are also used in floating-point units, such as the AFMU multiplier[14]. Vahdat et al. propose the TOSAM multiplier [15] that truncates the input operands according to their leading ‘1’ bit. To decrease the error, the truncated values are rounded to the nearest odd number.

2. Approximate Radix Encoding:

In these designs, approximation is introduced during partial product generation using modified encoding schemes. For example, radix-4 and radix-8 encoders are simplified to approximate least significant bits. Some designs, like in [16] and [17], use hybrid encodings that combine accurate and approximate schemes depending on the bit position. Higher-order encodings, such as radix-256, have also been proposed for improved efficiency.

3. Approximate Compressors:

Compressors are used in the partial product summation stage, where simplified versions help reduce logic complexity. Designs like those by Momeni et al. alter the truth table of standard 4:2 compressors, resulting in lightweight variants. Some compressors can switch between accurate and approximate modes dynamically. Others, such as those designed using FinFETs or stacking circuits, show significant power and area reductions while still maintaining acceptable accuracy levels.

4. Logarithmic Multipliers:

These approaches replace traditional multiplication with logarithmic operations. For instance, the ALM multiplier uses inexact logarithm converters and adders to simplify mantissa operations. The REALM design uses segmentation of input operand ranges with compensation factors, while the ILM multiplier simplifies operands by rounding them to the nearest power-of-two. The TLM design further reduces bit-widths in two stages—for inputs and mantissas—to achieve additional efficiency.

These various approaches to approximate addition and multiplication demonstrate the trade-offs between accuracy and efficiency. By selecting the appropriate approximation strategy based on application requirements, designers can optimize for power, speed, or area while tolerating small, controlled errors.

2.6 Challenges in Approximate MAC Design

While approximate MAC units offer benefits, they also pose challenges: Ensuring acceptable error metrics like Mean Error Distance (MED), Error Rate (ER), and Peak Signal-to-Noise Ratio (PSNR).

1. Balancing hardware savings and quality of results.
2. Application-specific tuning of approximation level.
3. Testing and verification complexities due to non-deterministic outputs.

***

Chapter: 3
Approximate MAC Architecture

.

3.1 Approximate Multiplier

This section begins by outlining the recursive multiplication method and subsequently presents a newly developed approximate pipelined multiplier architecture. Given the design objective of achieving high computational speed with minimal power consumption, a novel mechanism known as Carry-in Prediction Logic is also introduced and discussed.

3.1.1 Recursive Multiplication

In recursive multiplication, a large multiplication operation can be divided into several smaller units, each capable of being processed within a single clock cycle. Assume two input operands,

$$A = A_H A_L \text{ AND } X = X_H X_L$$

where A_H , A_L, X_H, and X_L are each 8-bit values. The complete multiplication $A \times X$ is thus broken into four smaller products: $A_H \times X_L$, $A_H \times X_H$, $A_L \times X_L$, and $A_L \times X_H$. Each of these is a 8×8 multiplication, which can be computed independently and then added together. This recursive division strategy is represented in Figure 3.1.

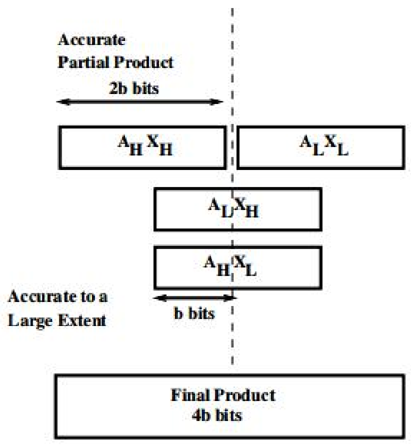


Fig 3.1 Recursive Multiplication[2]

In case we further divide multiplication into very small partial products (e.g. a 16 × 16 multiplication broken down into 2 × 2 multiplications), the size of recursive tree will be very large. As a result, the overhead required for addition stage will increase substantially. Further, this approach will not be helpful in reducing the critical path. In our approach, we consider a pipelined architecture of a 2b × 2b recursive multiplier broken down into four b × b multiplications as shown in Fig 3.2. This pipelined architecture is found to be more optimum because further dividing multiplication (b × b) into smaller partial products would not only increase the overhead of the addition stage but may require either more clock cycles to complete the addition or may need a longer clock period.

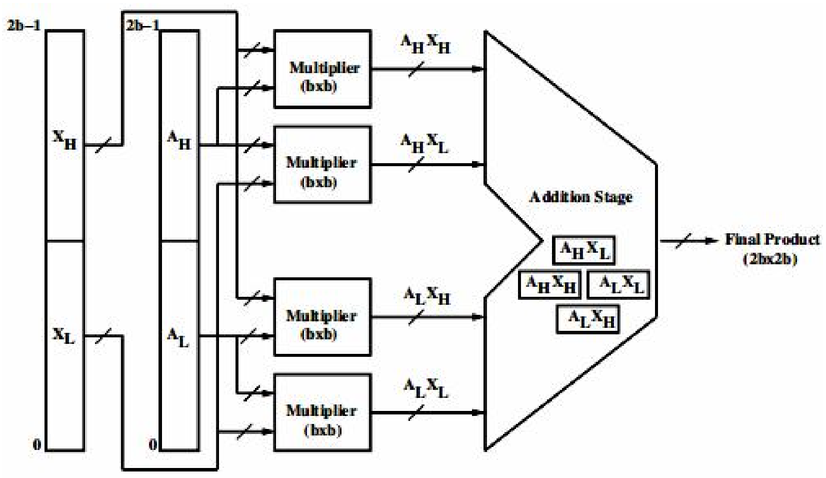


Fig 3.2 Pipelined Architecture for recursive Multiplier[2]

3.2 Proposed Multiplier Architecture

The proposed architecture prioritizes accuracy by aiming to reduce the error in the final product of the multiplication between operands A and X. To achieve this, a specific logic is designed to ensure higher precision in the most significant bits (MSBs) of the result. In particular, the upper 2b bits (from a total output width of 4b bits) are targeted to be mostly accurate. While these bits are not guaranteed to be entirely error-free, they are calculated with high precision to significantly reduce overall error without incurring excessive hardware cost.The most significant 2b bits are computed with high accuracy.

Of the remaining least significant 2b bits:

The lower b bits are made accurate.

The upper 3/2 b bits are intentionally left approximate.

This configuration strikes an effective trade-off between computational accuracy and hardware efficiency. By ensuring that the least significant b/2b/2b/2 bits are accurate (with minimal overhead) and that the top 2b bits are mostly accurate, the design achieves a good balance of speed, power consumption, and silicon area.

To implement this, the architecture utilizes approximate b×b multipliers for calculating the partial products AH×XL, AL×XH, and AL×XL as shown in Fig 3.3. For the critical term AH×XH, which contributes significantly to the MSBs, an accurate b×b multiplier is employed to ensure reliable computation.

Furthermore, to minimize delay in the critical path, each approximate partial product is split into two components: an accurate portion and an approximate portion. This partitioning supports faster execution while still maintaining acceptable accuracy in key sections of the result.

A central element of this approach is the introduction of a Carry-in Prediction Logic, which is applied exclusively to the approximate partial product computations—namely AH×XL, AL×XH, and AL×XL. This logic is not used for the fully accurate AH×XH AH ×XH path. The carry-in prediction helps reduce delay in the addition stages by estimating carry propagation in advance, thereby accelerating computation in the approximate paths.

3.2.1 Carry in Prediction Algorithm

In the proposed architecture, our aim is to make the most significant 2b (out of a total of 4*b* bits) as accurate to a large extent and we achieve this as illustrated in Fig 3.1 In this illustration, the most significant b bits of the b × *b* approximate partial products (AHXL*, ALXH*, and ALXL) must be accurate to a certain extent so that the sum of upper b bits of AHXL, upper b bits of ALXH and the 2b bits of AHXH achieves higher degree of accuracy.

In order to make the approximate partial product multipliers fast, it is necessary to break the carry chain which reduces the horizontal critical path. Therefore, we propose a novel Carry-in Prediction Logic. This logic reduces the horizontal critical path by half and makes the upper b bits of approximate partial product multipliers accurate to some extent. We divide the products AHXL, ALXH, and ALXL of 2b bits each into two parts: ”Accurate to a large extent” part and ”inaccurate” part.

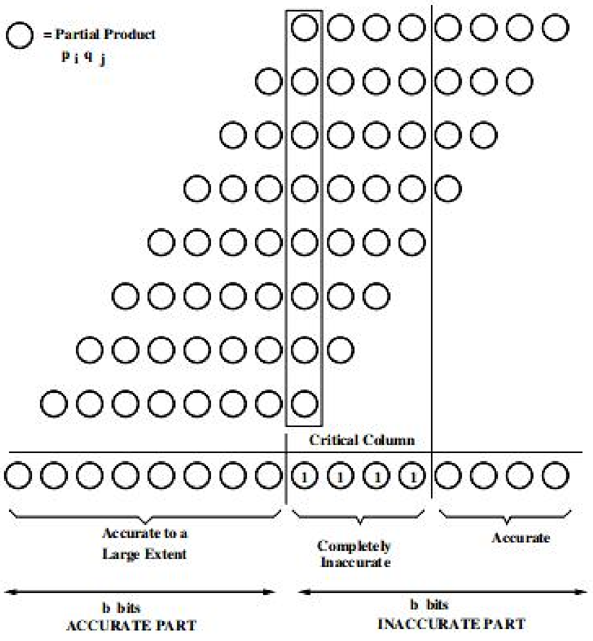


Fig 3.3 Carry in prediction of b=8 i.e 16 X 16 Multiplier[3].

Both these parts are of *b* bits each as shown in the Fig 3.4 for b= 8. The circles in figure represent the tree of partial products (p_iq_j) obtained from multiplication of multiplicand (say P = $p_{b-1}p_{b-2}...p_1p_0$) and multiplier (say *Q*= $q_{b-1}q_{b-2}...q_1q_0$). Please note, here P can take values of AH or AL and Q can take values of XH or XL as required for AHXL, *ALXH*, and ALXL evaluation. As evident from the figure, inaccurate part further consists of a completely inaccurate part and a completely accurate part of b/2 bits each. In rest of the paper, we denote the ”accurate to a large extent” part as Accurate part and the remaining lower *b* bits of approximate partial product as ”Inaccurate” part.

Carry-in Prediction necessitates the pre-computation of carry-in. Since we are dealing with error resilient systems, we can further simplify and approximate the evaluation of carryin so that reduction in latency is not achieved at the cost of power. We consider the cases of b= 4 and b = 6 in order to better explain carry-in pre-computation procedure.

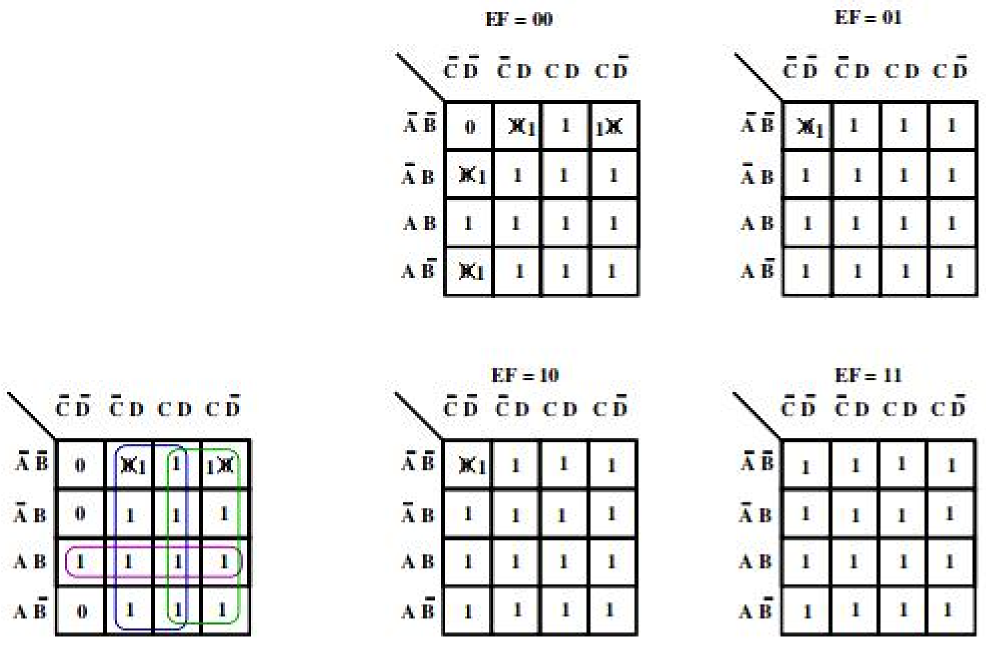


Fig 3.4 Pre-computation of Carry in prediction[3]

The pre-computation is made hardware efficient by making minor changes in the K-Maps of the carry-in expressions as shown in Fig 3.4. Here A, B, . . . , F are the elements in critical column. The original K-Maps are obtained from the statement of Carry-in Prediction Logic i.e. Cin= 1 if 2 or more elements of critical column are 1. Fig. 3.5 further derives for b= 4: By making changes in 2 cases out of 16, we can simplify the Carry-in expression to Cin = A+B +C + D.

Similar results can also be derived for b> 4 for b = 6: We make changes in 6 cases out of 64 and get Cin = A + B + C + D + E + F.

This is same as OR operation of all the elements present in critical column. Therefore, in general, we can state that for large b(greater than 4), one should take the OR of all the elements present in critical column to get Cin.

3.2.2 Bit-Width Aware Approximate Multiplication

In the approximate multiplication, we divide the $b \times b$ accurate multiplier AHXH into 4 smaller components, each being a $b/2 \times b/2$ multiplier. This is because, when accurate AHXH is performed in parallel with approximate AHXL, ALXH and ALXL, the critical path will still be determined by the accurate multiplier. Therefore, recursively reducing it to smaller multipliers will make approximate $b \times b$ multipliers as deciding factors of critical path as they are more critical than accurate $b/2 \times b/2$ multipliers.

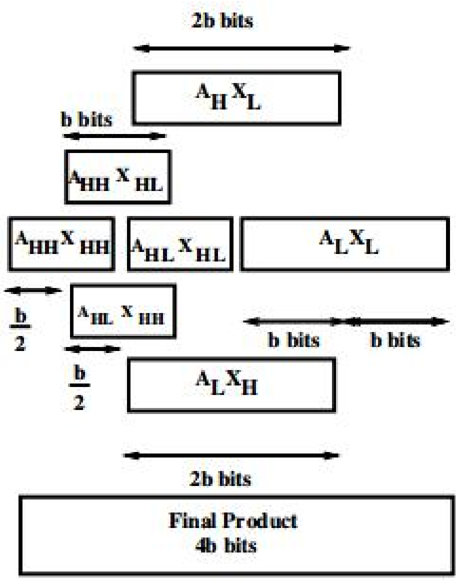


Fig 3.5 Latency driven Pipelined Approximate Multiplier[3]

In other words, the stage 1 of the pipelined approximate multiplier effectively consists of 7 multipliers. The designation of each of these multipliers and their respective arrangement for addition in second stage is depicted in Fig 3.5. Note that this kind of arrangement will not lead to any change in latency of second pipeline stage as we perform the addition of AHXL and ALXH in parallel with rest of the smaller multipliers. The latter additions generate a net sum of AHXH in almost the same time as the former takes to complete its addition. Now let us see how we can exploit this property in the proposed multiplication to configure its accuracy.

3.2.3 Accuracy Configuration Modes

Since now we have 7 multipliers in stage 1, we can vary the accuracy level of the proposed multiplier by varying the number of multipliers that are accurate. In any case, we keep the $A_{HH}X_{HH}$ as always accurate, so that the accuracy level does not fall below a certain level. Therefore, we obtain an accuracy configurable multiplier whose accuracy can be adjusted according to error tolerance of the application. The number of inaccurate multipliers used will directly determine the amount of power saved by the multiplier.

We propose 4 different modes of operations of our approximate multiplier based on accuracy levels. The proposed modes are given in Table I. Here ‘A’ stands for an accurate multiplier and ‘X’ stands for an inaccurate multiplier. We explain the bitwidth aware algorithm next.

Mode	$A_{HH}X_{HH}$	$A_{HH}X_{HL}$	$A_{HL}X_{HH}$	$A_{HL}X_{HL}$
1	A	I	I	I
2	A	A	I	I
3	A	A	A	I
4	A	A	A	A

Table 3.1: Modes of Operation of Accuracy Configurable Multiplier[3]

3.2.4 Approximate Wallace Tree Multiplier Design

In our accuracy-configurable design, we have reduced the horizontal critical path to just half. In order to reduce the vertical critical path as well (of AHXL, ALXH and ALXL), we use Wallace Tree Reduction [5]. The Wallace Trees are fast and hardware efficient for multiplication of more than 16 bits. Wallace tree height also grows as $\log_{3/2}(N/2)$. We call this Wallace tree based design as Approximate Wallace Tree Multiplier (AWTM). For an accurate 8 × 8 Wallace multiplication, it takes a total of 4 stages of reduction (each of which has a delay of 1 full adder) and then uses a 11-bit full adder to compute the final product. We use these 8 × 8 partial products to evaluate a 16 × 16 multiplication. Fig 3.6 shows that approximate partial product multipliers (AHXL, ALXH and ALXL), which are 8 × 8, take a total of 3 stages of reduction and further use a 6-bit full adder for final product evaluation. When compared in terms of critical paths (of stage 1 of pipelined 16 × 16 multiplier), an accurate 8 × 8 multiplier uses a delay of 15 full adders (4 stages and 11-bit full adder) and its approximate counterpart uses a delay of 9 full adders (3 stages and a 6-bit full adder). Theoretically, this leads to an improvement of 40% in latency of stage 1. Furthermore, for each of the AHXL, ALXH and ALXL, number of adders reduced is around 51.24% (48 full adders and 25 half adders are required for their accurate evaluation which is in contrast with 23 full adders and 13 half adders required for approximate evaluation) and hence for complete pipelined multiplier, total power reduction is expected to be around 38.42% (because *AHXH* is accurate and other 3 are inaccurate, therefore three-fourth of 51.24%). We validate these theoretical estimates by running actual simulations.

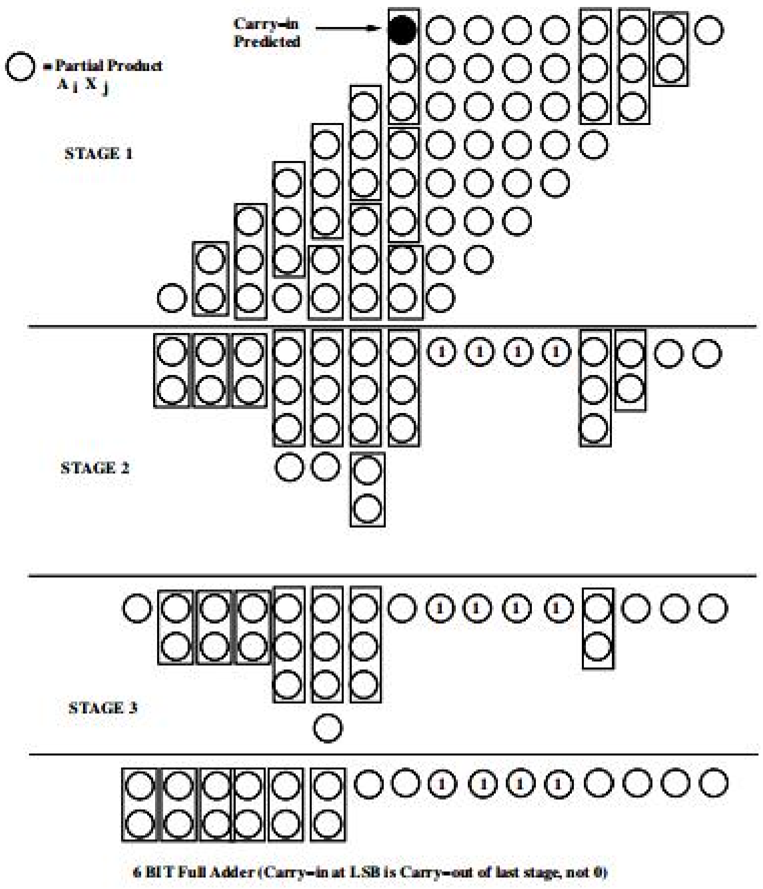

Carry-in Predicted
= Partial Product $A_i X_j$
STAGE 1
STAGE 2
STAGE 3
6 BIT Full Adder (Carry-in at LSB is Carry-out of last stage, not 0)

Fig 3.6 AWTM Implementation[3]

3.2 Proposed 32 bit Adder

Adders are one of the most widely implemented blocks of microprocessor chips and digital components in the digital integrated circuit design. They are the necessary part of Digital Signal Processing (DSP) applications. With the advances in technology, researchers have tried and are trying to design adders which offer either high speed, low power consumption, less area or the combination of them. Every adder generates a carry value that has to be propagated through the circuit within a series of adders. This contributes largely to the critical path delay of the circuit. By reducing the number of stages the carry has to be propagated, the delay in the circuit can be reduced. The required sum is selected using a multiplexer.

3.2.1 Proposed Full Adder Design

Mux based full adder circuit is obtained using two 2-to-1 multiplexers, 1 XOR gate and an NOT gate.

$$\text{Sum} = (b \odot cin)\, a + (b \oplus cin)(\sim a)$$

$$\text{Carry} = (b \odot cin)\, b + (b \oplus cin)\, a$$

Prof. S.Murugeswari and Dr. S.Kaja Mohideen have proposed this model. In this work, the detailed study about Wallace tree multiplier and truncated multipliers is done with both normal full adder and modified full adder-1 and it is shown that modified full adder-1 implementation occupies the area[6].

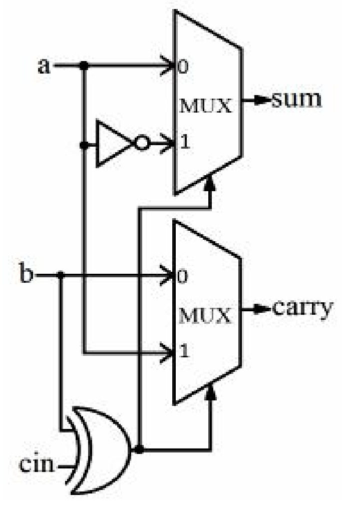


Fig 3.7 Modified Full Adder[4]

3.2.2 32 bit Carry skip Adder

A carry-skip adder (also known as a carry-bypass adder) is an adder implementation that improves on the delay of a ripple carry adder with little effort compared to other adders. The improvement of the worst-case delay is achieved by using several carry-skip adders to form a block-carry-skip adder. Unlike other fast adders, carry-skip adder performance is increased with only some of the combinations of input bits. This means, speed improvement is only probabilistic. Carry skip adder uses the principle of skip logic in carry propagation. It is used to speed addition operation by adding a propagation of carry bit around entire adder. It consists two logical gates AND gate is used for carry -in bit which compares with propagated signals.

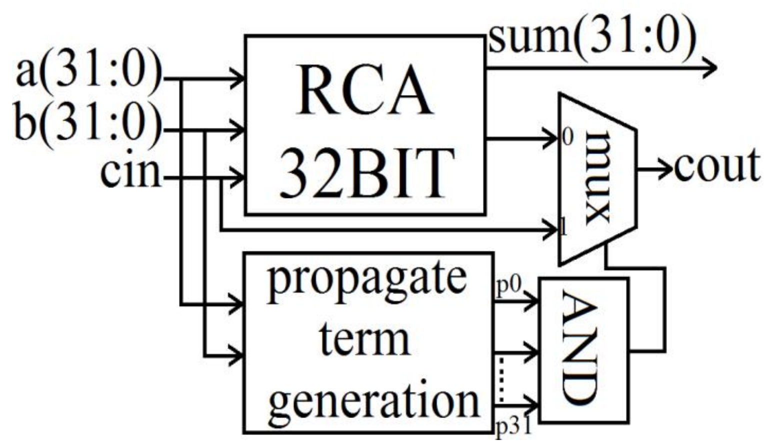


Fig 3.8 Carry skip Adder[6]

3.2.3 Proposed carry skip adder

In this CSA, I have used 6 5bit Carry look A head Adder, 6 2X1 Muxes, an 2 modified full adders. And implemented as shown in fig 3.9 below. Here every five bits we check the propagation of carry.

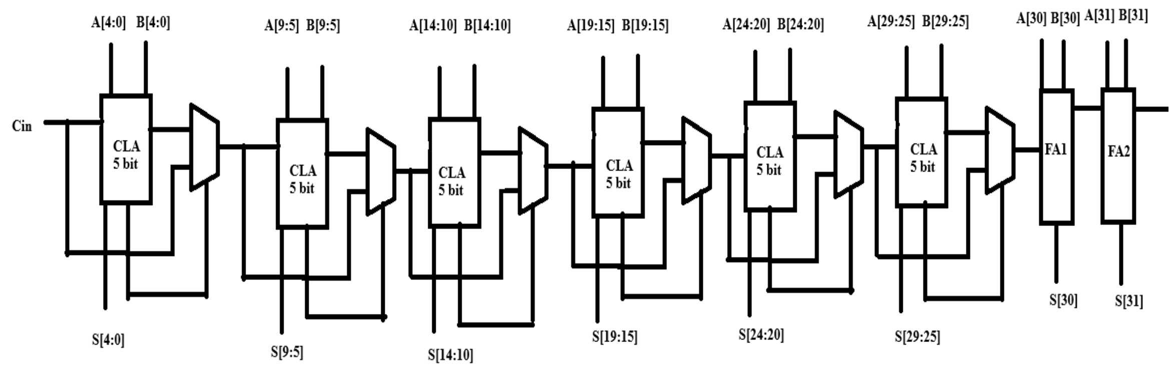


Fig 3.9 Proposed CSA

3.2.4 5bit Carry look ahead Adder

This gives us a way to determine the carry bit at each position without having to wait for carries to propagate up from less significant positions. We can then use the carry bit to derive the sum bits. An adder based on this formulation is called a carry-look-ahead adder. A 5-bit version of such an adder is illustrated in Figure 3.10. Each of the boxes at the top derives the generate and propagate signals for the corresponding bit position. The carry-look-ahead generator implements the equations shown above to derive the carry signals. These are combined with the propagate signals to derive the sum bits. The trade-off for getting the sum bits faster is the area and power consumed by the carry-look-ahead generator circuitry[7].

These can be derived from the below equations

For propagation,

$$p_i = x_i \oplus y_i$$

For Generation,

$$g_i = x_i \cdot y_i$$

For Sum and carry,

$$s_i = p_i \oplus c_i$$
$$c_{i+1} = g_i + p_i \cdot c_i$$

***

Chapter: 4
Approximate MAC Units Results

4.1 Simulation Results Using Vivado

To evaluate the performance of the proposed approximate MAC architecture, the design was implemented and simulated using Xilinx Vivado. Four different approximation modes were created to allow flexibility in the trade-off between accuracy and hardware efficiency:

a) **Mode 1**: Low Approximation
b) **Mode 2**: High Approximation
c) **Mode 3**: Very High Approximation
d) **Mode 4**: Accurate Mode (Baseline Reference)

Each mode alters the internal multiplier configuration, adjusting which sub-units operate in accurate or approximate modes.

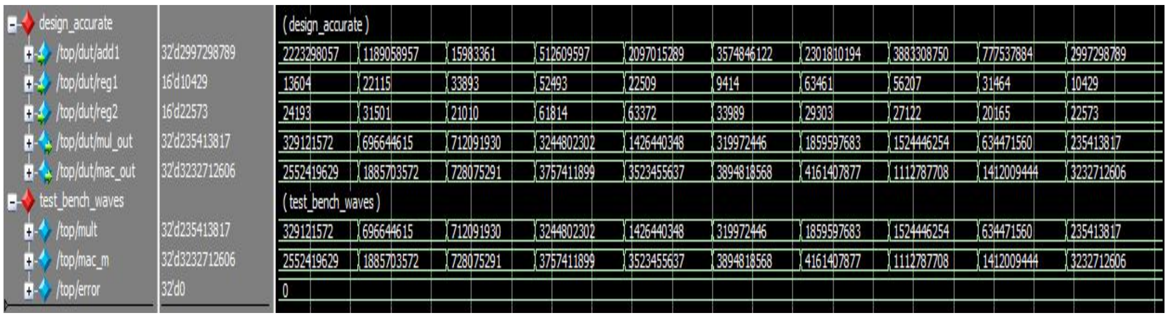
4.1.1 Accurate Mode Results:

This configuration uses fully accurate sub-modules in the MAC unit. It serves as a baseline to compare the performance and accuracy of other modes.

Fig 4.1 – 4.2: Show correct functional output from simulation and testbench.

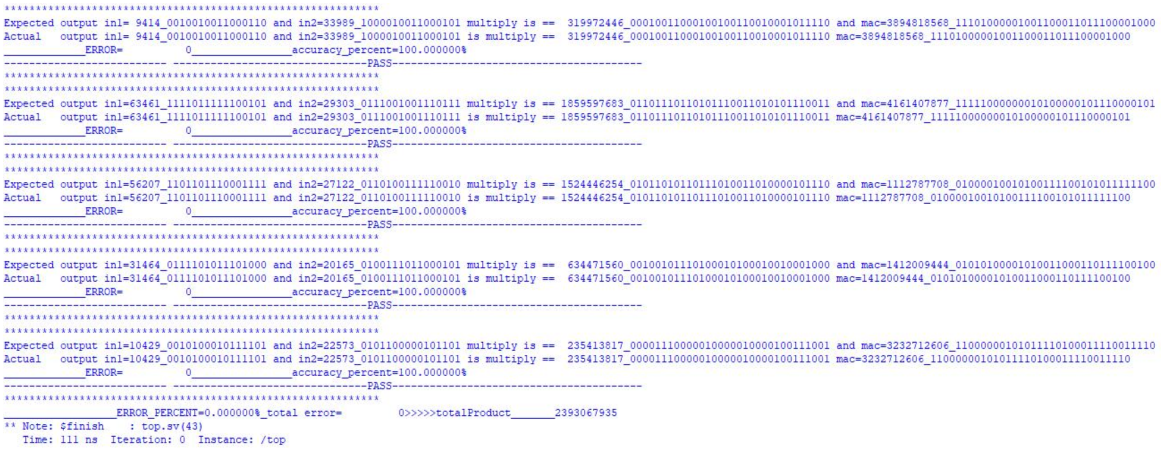
Fig 4.3: Presents timing analysis, indicating the highest latency due to no approximations.

Fig 4.4: Shows peak power usage (~76.66W).



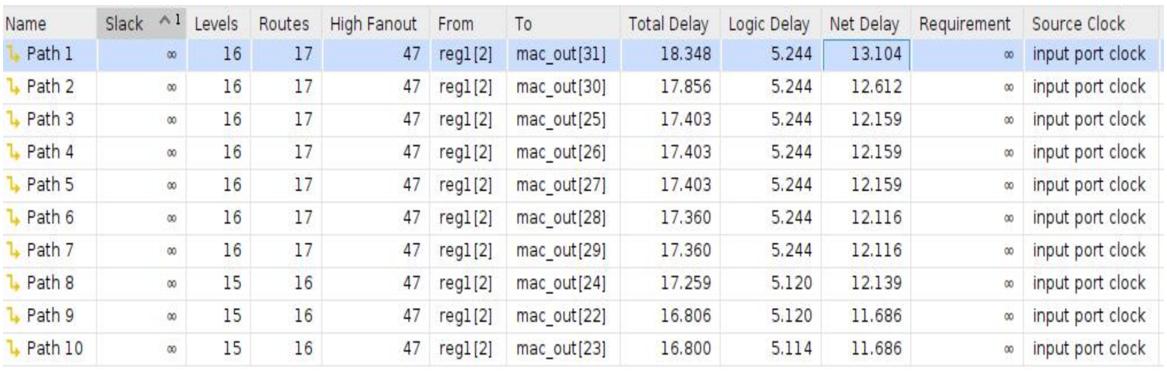
design_accurate		(design_accurate)									
/top/dut/add1	32'd2997298789	2223298057	1189058957	15983361	512609597	2097015289	3574846122	2301810194	3883308750	777537884	2997298789
/top/dut/reg1	16'd10429	13604	22115	33893	52493	22509	9414	63461	56207	31464	10429
/top/dut/reg2	16'd22573	24193	31501	21010	61814	63372	33989	29303	27122	20165	22573
/top/dut/mul_out	32'd235413817	329121572	696644615	712091930	3244802302	1426440348	319972446	1859597683	1524446254	634471560	235413817
/top/dut/mac_out	32'd3232712606	2552419629	1885703572	728075291	3757411899	3523455637	3894818568	4161407877	1112787708	1412009444	3232712606
test_bench_waves		(test_bench_waves)									
/top/mult	32'd235413817	329121572	696644615	712091930	3244802302	1426440348	319972446	1859597683	1524446254	634471560	235413817
/top/mac_m	32'd3232712606	2552419629	1885703572	728075291	3757411899	3523455637	3894818568	4161407877	1112787708	1412009444	3232712606
/top/error	32'd0	0									

Fig 4.1 Accurate Mode simulation results (MODE-4)



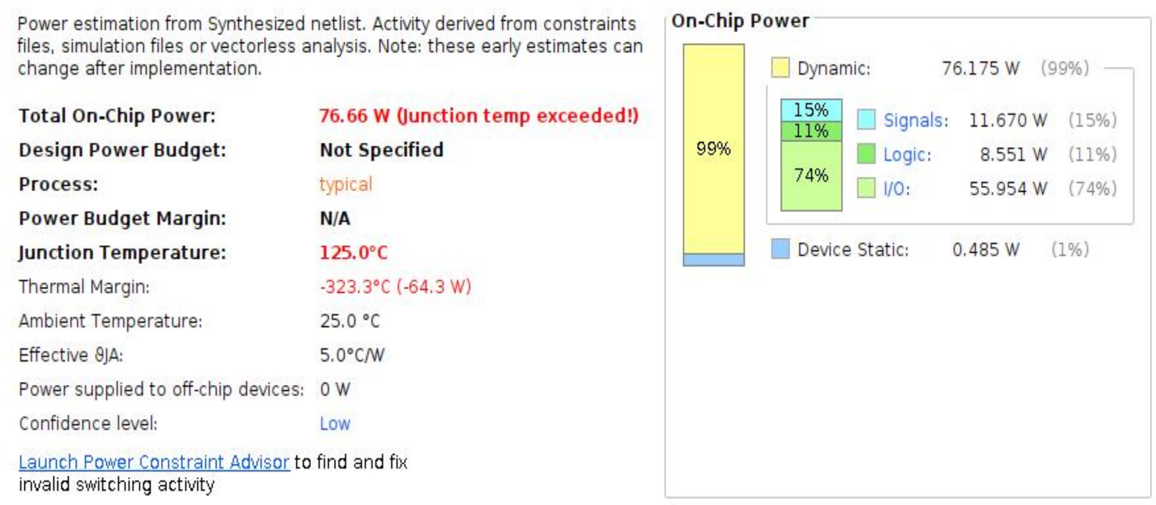
```
**********************************************************
Expected output in1= 9414_0010010011000110 and in2=33989_1000010011000101 multiply is ==  319972446_00010011000100100110010001011110 and mac=3894818568_11101000001001100011011100001000
Actual   output in1= 9414_0010010011000110 and in2=33989_1000010011000101 is multiply ==  319972446_00010011000100100110010001011110 mac=3894818568_11101000001001100011011100001000
_____________ERROR=          0________________accuracy_percent=100.000000%
-------------------------- ------------------------------PASS------------------------------------------
**********************************************************
**********************************************************
Expected output in1=63461_1111011111100101 and in2=29303_0111001001110111 multiply is == 1859597683_01101110110101110011010101110011 and mac=4161407877_11111000000010100000101110000101
Actual   output in1=63461_1111011111100101 and in2=29303_0111001001110111 is multiply == 1859597683_01101110110101110011010101110011 mac=4161407877_11111000000010100000101110000101
_____________ERROR=          0________________accuracy_percent=100.000000%
-------------------------- ------------------------------PASS------------------------------------------
**********************************************************
**********************************************************
Expected output in1=56207_1101101110001111 and in2=27122_0110100111110010 multiply is == 1524446254_01011010110111010011010000101110 and mac=1112787708_01000010010100111100101011111100
Actual   output in1=56207_1101101110001111 and in2=27122_0110100111110010 is multiply == 1524446254_01011010110111010011010000101110 mac=1112787708_01000010010100111100101011111100
_____________ERROR=          0________________accuracy_percent=100.000000%
-------------------------- ------------------------------PASS------------------------------------------
**********************************************************
**********************************************************
Expected output in1=31464_0111101011101000 and in2=20165_0100111011000101 multiply is ==  634471560_00100101110100010100010010001000 and mac=1412009444_01010100001010011000110111100100
Actual   output in1=31464_0111101011101000 and in2=20165_0100111011000101 is multiply ==  634471560_00100101110100010100010010001000 mac=1412009444_01010100001010011000110111100100
_____________ERROR=          0________________accuracy_percent=100.000000%
-------------------------- ------------------------------PASS------------------------------------------
**********************************************************
**********************************************************
Expected output in1=10429_0010100010111101 and in2=22573_0101100000101101 multiply is ==  235413817_00001110000010000010000100111001 and mac=3232712606_11000000101011110100011110011110
Actual   output in1=10429_0010100010111101 and in2=22573_0101100000101101 is multiply ==  235413817_00001110000010000010000100111001 mac=3232712606_11000000101011110100011110011110
_____________ERROR=          0________________accuracy_percent=100.000000%
-------------------------- ------------------------------PASS------------------------------------------
**********************************************************
_________________ERROR_PERCENT=0.000000%_total error=        0>>>>>totalProduct_______2393067935
** Note: $finish    : top.sv(43)
   Time: 111 ns  Iteration: 0  Instance: /top
```

Fig 4.2 Accurate Mode testbench results(MODE-4)



Name	Slack ^1	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock
Path 1	∞	16	17	47	reg1[2]	mac_out[31]	18.348	5.244	13.104	∞	input port clock
Path 2	∞	16	17	47	reg1[2]	mac_out[30]	17.856	5.244	12.612	∞	input port clock
Path 3	∞	16	17	47	reg1[2]	mac_out[25]	17.403	5.244	12.159	∞	input port clock
Path 4	∞	16	17	47	reg1[2]	mac_out[26]	17.403	5.244	12.159	∞	input port clock
Path 5	∞	16	17	47	reg1[2]	mac_out[27]	17.403	5.244	12.159	∞	input port clock
Path 6	∞	16	17	47	reg1[2]	mac_out[28]	17.360	5.244	12.116	∞	input port clock
Path 7	∞	16	17	47	reg1[2]	mac_out[29]	17.360	5.244	12.116	∞	input port clock
Path 8	∞	15	16	47	reg1[2]	mac_out[24]	17.259	5.120	12.139	∞	input port clock
Path 9	∞	15	16	47	reg1[2]	mac_out[22]	16.806	5.120	11.686	∞	input port clock
Path 10	∞	15	16	47	reg1[2]	mac_out[23]	16.800	5.114	11.686	∞	input port clock

Fig 4.3 Timing Analysis for MODE-4 Approximate MAC



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

Total On-Chip Power:	76.66 W (Junction temp exceeded!)
Design Power Budget:	Not Specified
Process:	typical
Power Budget Margin:	N/A
Junction Temperature:	125.0°C
Thermal Margin:	-323.3°C (-64.3 W)
Ambient Temperature:	25.0 °C
Effective ϑJA:	5.0°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power

🟨	Dynamic:	76.175 W	(99%)
🟦	Signals:	11.670 W	(15%)
🟩	Logic:	8.551 W	(11%)
🟩	I/O:	55.954 W	(74%)
🟦	Device Static:	0.485 W	(1%)

Fig 4.4 Power Analysis for MODE-4 Approximate MAC

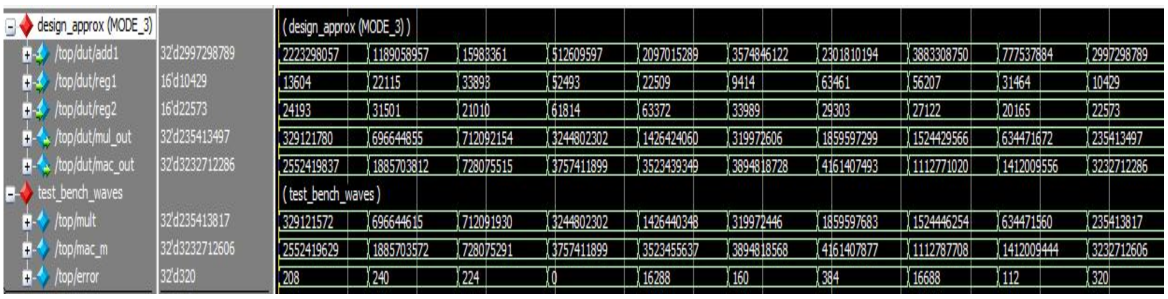
4.1.2 Very High Approximate Mode Results:

Only one multiplier is approximate, and three are accurate. Accuracy is nearly on par with Mode-4.

Fig 4.5 – 4.6: Output contains noticeable but tolerable errors.

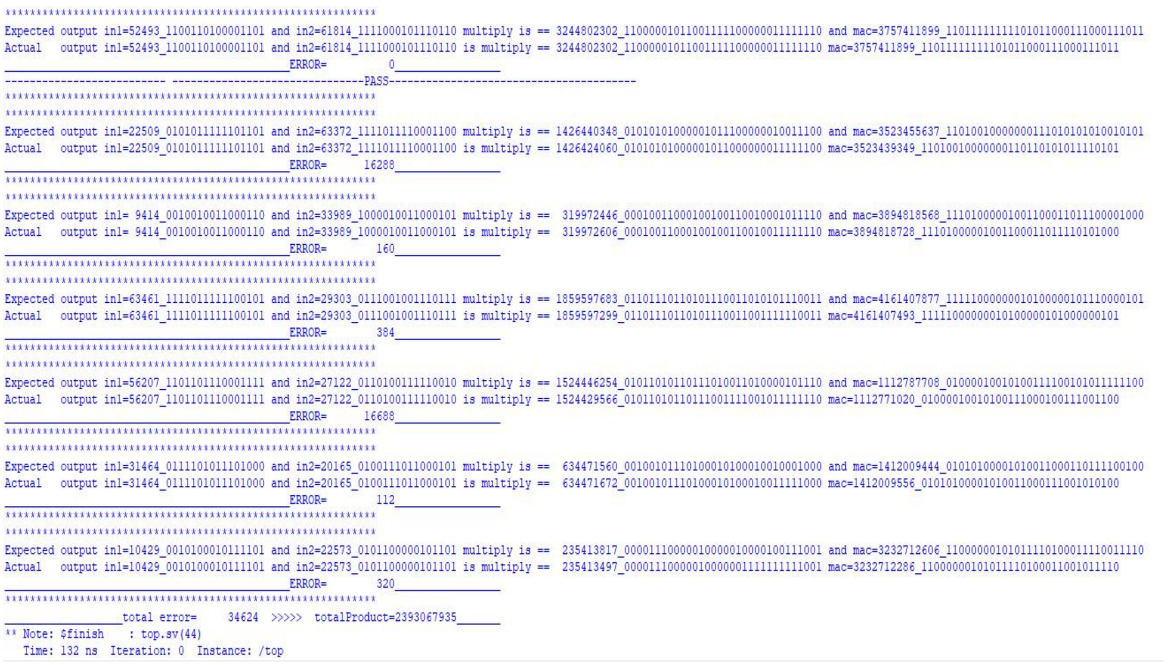
Fig 4.7: Shows significant reduction in delay compared to Mode-4.

Fig 4.8: Shows reduced power consumption (~69.34W).



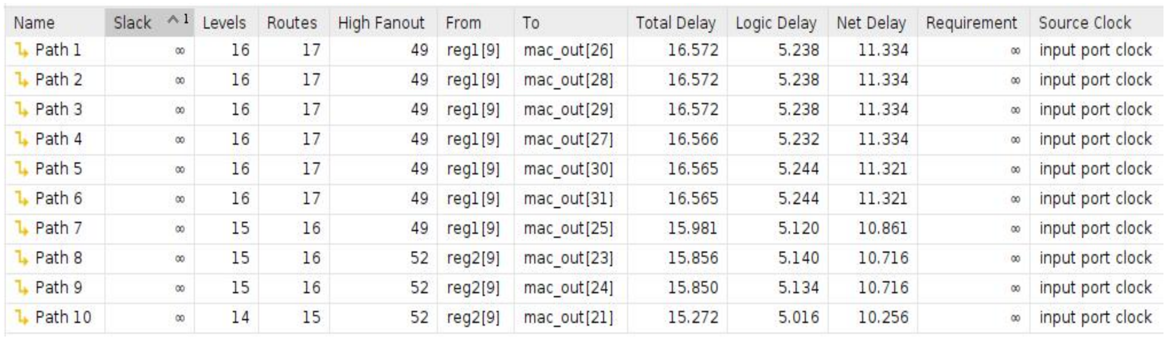
```
design_approx (MODE_3)                   (design_approx (MODE_3))
  /top/dut/add1     32'd2997298789   2223298057  1189058957  15983361    512609597   2097015289  3574846122  2301810194  3883308750  777537884   2997298789
  /top/dut/reg1     16'd10429        13604       22115       33893       52493       22509       9414        63461       56207       31464       10429
  /top/dut/reg2     16'd22573        24193       31501       21010       61814       63372       33989       29303       27122       20165       22573
  /top/dut/mul_out  32'd235413497    329121780   696644855   712092154   3244802302  1426424060  319972606   1859597299  1524429566  634471672   235413497
  /top/dut/mac_out  32'd3232712286   2552419837  1885703812  728075515   3757411899  3523439349  3894818728  4161407493  1112771020  1412009556  3232712286
test_bench_waves                         (test_bench_waves)
  /top/mult         32'd235413817    329121572   696644615   712091930   3244802302  1426440348  319972446   1859597683  1524446254  634471560   235413817
  /top/mac_m        32'd3232712606   2552419629  1885703572  728075291   3757411899  3523455637  3894818568  4161407877  1112787708  1412009444  3232712606
  /top/error        32'd320          208         240         224         0           16288       160         384         16688       112         320
```

Fig 4.5 Very High Approximate simulation results(MODE-3)



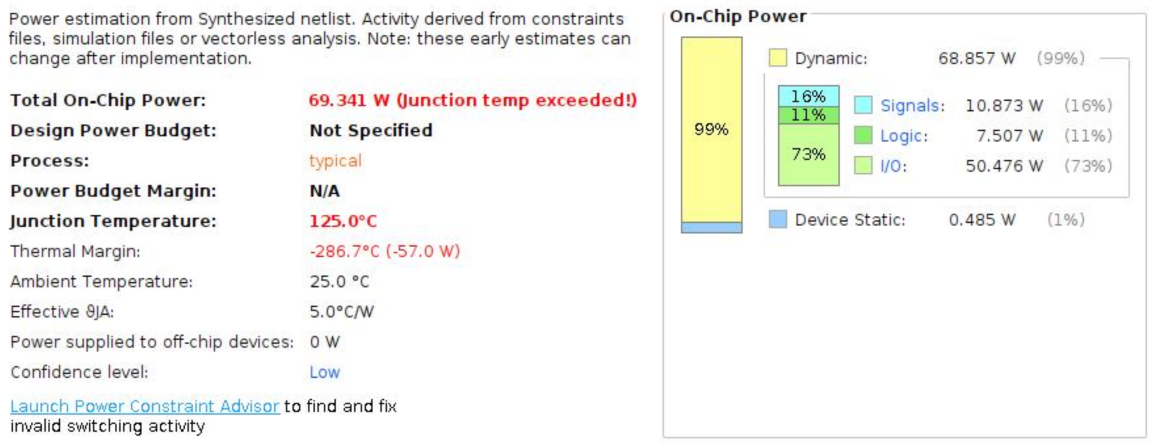
```
************************************************************
Expected output in1=52493_1100110100001101 and in2=61814_1111000101110110 multiply is == 3244802302_11000001011001111100000011111110 and mac=3757411899_11011111111101011000111000111011
Actual   output in1=52493_1100110100001101 and in2=61814_1111000101110110 is multiply == 3244802302_11000001011001111100000011111110 mac=3757411899_11011111111101011000111000111011
_______________________________________________ERROR=        0_________________
-------------------------- -------------------------------PASS----------------------------------------
************************************************************
************************************************************
Expected output in1=22509_0101011111101101 and in2=63372_1111011110001100 multiply is == 1426440348_01010101000001011100000010011100 and mac=3523455637_11010010000000111010101010010101
Actual   output in1=22509_0101011111101101 and in2=63372_1111011110001100 is multiply == 1426424060_01010101000001011000000011111100 mac=3523439349_11010010000000110110101011110101
_______________________________________________ERROR=    16288_________________
************************************************************
************************************************************
Expected output in1= 9414_0010010011000110 and in2=33989_1000010011000101 multiply is ==  319972446_00010011000100100110010001011110 and mac=3894818568_11101000001001100011011100001000
Actual   output in1= 9414_0010010011000110 and in2=33989_1000010011000101 is multiply ==  319972606_00010011000100100110010011111110 mac=3894818728_11101000001001100011011110101000
_______________________________________________ERROR=      160_________________
************************************************************
************************************************************
Expected output in1=63461_1111011111100101 and in2=29303_0111001001110111 multiply is == 1859597683_01101110110101110011010101110011 and mac=4161407877_11111000000010100000101110000101
Actual   output in1=63461_1111011111100101 and in2=29303_0111001001110111 is multiply == 1859597299_01101110110101110011001111110011 mac=4161407493_11111000000010100000101000000101
_______________________________________________ERROR=      384_________________
************************************************************
************************************************************
Expected output in1=56207_1101101110001111 and in2=27122_0110100111110010 multiply is == 1524446254_01011010110111010011010000101110 and mac=1112787708_01000010010100111100101011111100
Actual   output in1=56207_1101101110001111 and in2=27122_0110100111110010 is multiply == 1524429566_01011010110111001111001011111110 mac=1112771020_01000010010100111000100111001100
_______________________________________________ERROR=    16688_________________
************************************************************
************************************************************
Expected output in1=31464_0111101011101000 and in2=20165_0100111011000101 multiply is ==  634471560_00100101110100010100010010001000 and mac=1412009444_01010100001010011000110111100100
Actual   output in1=31464_0111101011101000 and in2=20165_0100111011000101 is multiply ==  634471672_00100101110100010100010011111000 mac=1412009556_01010100001010011000111001010100
_______________________________________________ERROR=      112_________________
************************************************************
************************************************************
Expected output in1=10429_0010100010111101 and in2=22573_0101100000101101 multiply is ==  235413817_00001110000010000010000100111001 and mac=3232712606_11000000101011110100011110011110
Actual   output in1=10429_0010100010111101 and in2=22573_0101100000101101 is multiply ==  235413497_00001110000010000001111111111001 mac=3232712286_11000000101011110100011001011110
_______________________________________________ERROR=      320_________________
************************************************************
________________total error=     34624  >>>>>  totalProduct=2393067935______
** Note: $finish    : top.sv(44)
   Time: 132 ns  Iteration: 0  Instance: /top
```

Fig 4.6 Very High Approximate testbench output(MODE-3)



Name	Slack ^1	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock
Path 1	∞	16	17	49	reg1[9]	mac_out[26]	16.572	5.238	11.334	∞	input port clock
Path 2	∞	16	17	49	reg1[9]	mac_out[28]	16.572	5.238	11.334	∞	input port clock
Path 3	∞	16	17	49	reg1[9]	mac_out[29]	16.572	5.238	11.334	∞	input port clock
Path 4	∞	16	17	49	reg1[9]	mac_out[27]	16.566	5.232	11.334	∞	input port clock
Path 5	∞	16	17	49	reg1[9]	mac_out[30]	16.565	5.244	11.321	∞	input port clock
Path 6	∞	16	17	49	reg1[9]	mac_out[31]	16.565	5.244	11.321	∞	input port clock
Path 7	∞	15	16	49	reg1[9]	mac_out[25]	15.981	5.120	10.861	∞	input port clock
Path 8	∞	15	16	52	reg2[9]	mac_out[23]	15.856	5.140	10.716	∞	input port clock
Path 9	∞	15	16	52	reg2[9]	mac_out[24]	15.850	5.134	10.716	∞	input port clock
Path 10	∞	14	15	52	reg2[9]	mac_out[21]	15.272	5.016	10.256	∞	input port clock

Fig 4.7 Timing Analysis for MODE-3 Approximate MAC



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

Total On-Chip Power:	69.341 W (Junction temp exceeded!)
Design Power Budget:	Not Specified
Process:	typical
Power Budget Margin:	N/A
Junction Temperature:	125.0°C
Thermal Margin:	-286.7°C (-57.0 W)
Ambient Temperature:	25.0 °C
Effective ϑJA:	5.0°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power

🟨	Dynamic:	68.857 W	(99%)
	Signals:	10.873 W	(16%)
	Logic:	7.507 W	(11%)
	I/O:	50.476 W	(73%)
🟦	Device Static:	0.485 W	(1%)

Fig 4.8 Power Analysis for MODE-3 Approximate MAC

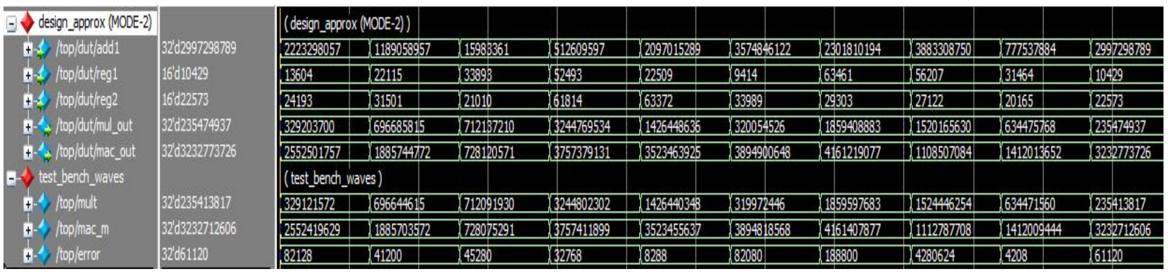
4.1.3 High Approximate Mode Results:

This configuration uses two accurate and two approximate multipliers, offering a balanced trade-off.

Fig 4.9 – 4.10: Output has minor errors.

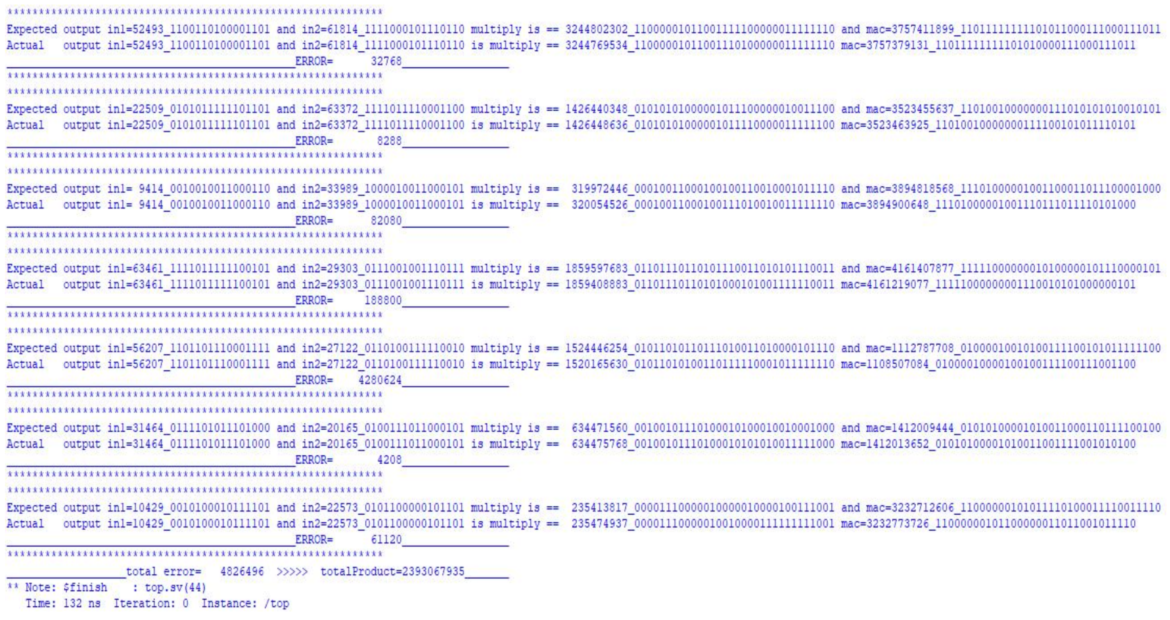
Fig 4.11: Moderate delay and acceptable timing closure.

Fig 4.12: Power drops to ~67.67W.



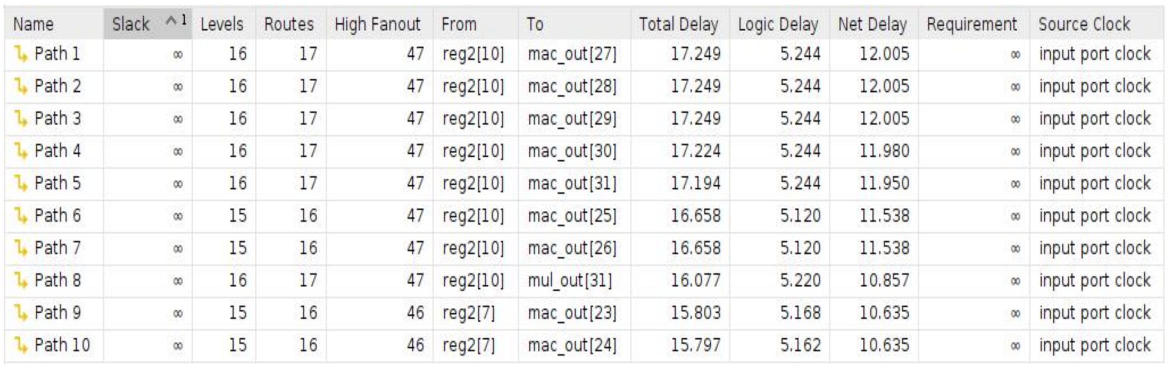
```
design_approx (MODE-2)                      ( design_approx (MODE-2) )
  /top/dut/add1     32'd2997298789   2223298057 1189058957 15983361 512609597  2097015289 3574846122 2301810194 3883308750 777537884 2997298789
  /top/dut/reg1     16'd10429        13604      22115      33893    52493      22509      9414       63461      56207      31464     10429
  /top/dut/reg2     16'd22573        24193      31501      21010    61814      63372      33989      29303      27122      20165     22573
  /top/dut/mul_out  32'd235474937    329203700  696685815  712137210 3244769534 1426448636 320054526 1859408883 1520165630 634475768 235474937
  /top/dut/mac_out  32'd3232773726   2552501757 1885744772 728120571 3757379131 3523463925 3894900648 4161219077 1108507084 1412013652 3232773726
test_bench_waves                            ( test_bench_waves )
  /top/mult         32'd235413817    329121572  696644615  712091930 3244802302 1426440348 319972446 1859597683 1524446254 634471560 235413817
  /top/mac_m        32'd3232712606   2552419629 1885703572 728075291 3757411899 3523455637 3894818568 4161407877 1112787708 1412009444 3232712606
  /top/error        32'd61120        82128      41200      45280    32768      8288       82080      188800     4280624    4208      61120
```

Fig 4.9 High Approximate Simulation Results(MODE-2)



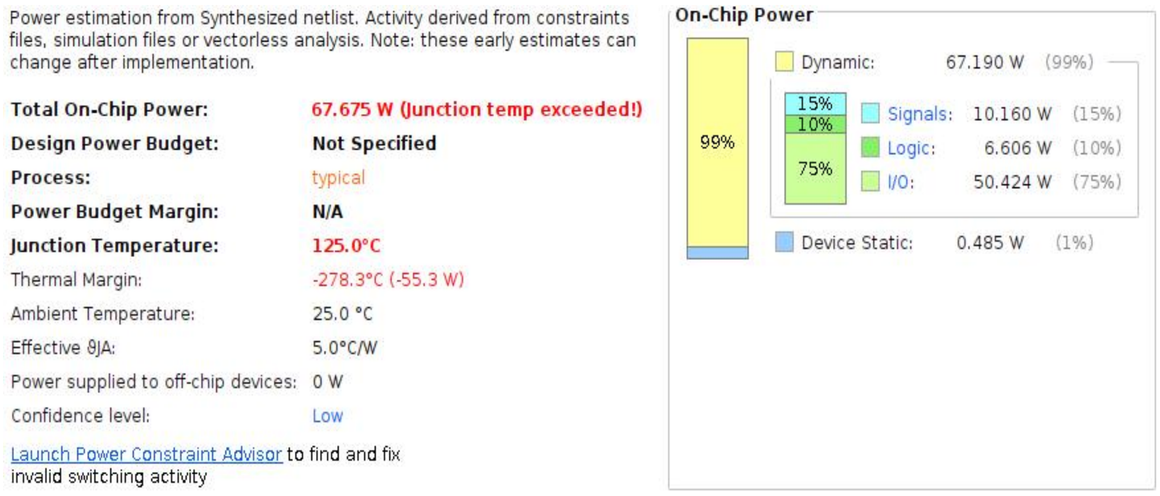
```
************************************************************
Expected output in1=52493_1100110100001101 and in2=61814_1111000101110110 multiply is == 3244802302_11000001011001111100000011111110 and mac=3757411899_11011111111101011000111000111011
Actual   output in1=52493_1100110100001101 and in2=61814_1111000101110110 is multiply == 3244769534_11000001011001110100000011111110 mac=3757379131_11011111111101010000111000111011
______________________________________________ERROR=      32768_______________
************************************************************
************************************************************
Expected output in1=22509_0101011111101101 and in2=63372_1111011110001100 multiply is == 1426440348_01010101000001011100000010011100 and mac=3523455637_11010010000000111010101010010101
Actual   output in1=22509_0101011111101101 and in2=63372_1111011110001100 is multiply == 1426448636_01010101000001011110000011111100 mac=3523463925_11010010000000111100101011110101
______________________________________________ERROR=       8288_______________
************************************************************
************************************************************
Expected output in1= 9414_0010010011000110 and in2=33989_1000010011000101 multiply is ==  319972446_00010011000100100110010001011110 and mac=3894818568_11101000001001100011011100001000
Actual   output in1= 9414_0010010011000110 and in2=33989_1000010011000101 is multiply ==  320054526_00010011000100111010010011111110 mac=3894900648_11101000001001110111011110101000
______________________________________________ERROR=      82080_______________
************************************************************
************************************************************
Expected output in1=63461_1111011111100101 and in2=29303_0111001001110111 multiply is == 1859597683_01101110110101110011010101110011 and mac=4161407877_11111000000010100000101110000101
Actual   output in1=63461_1111011111100101 and in2=29303_0111001001110111 is multiply == 1859408883_01101110110101000101001111110011 mac=4161219077_11111000000001110010101000000101
______________________________________________ERROR=     188800_______________
************************************************************
************************************************************
Expected output in1=56207_1101101110001111 and in2=27122_0110100111110010 multiply is == 1524446254_01011010110111010011010000101110 and mac=1112787708_01000010010100111100101011111100
Actual   output in1=56207_1101101110001111 and in2=27122_0110100111110010 is multiply == 1520165630_01011010100110111110001011111110 mac=1108507084_01000010000100100111100111001100
______________________________________________ERROR=    4280624_______________
************************************************************
************************************************************
Expected output in1=31464_0111101011101000 and in2=20165_0100111011000101 multiply is ==  634471560_00100101110100010100010010001000 and mac=1412009444_01010100001010011000110111100100
Actual   output in1=31464_0111101011101000 and in2=20165_0100111011000101 is multiply ==  634475768_00100101110100010101010011111000 mac=1412013652_01010100001010011001111001010100
______________________________________________ERROR=       4208_______________
************************************************************
************************************************************
Expected output in1=10429_0010100010111101 and in2=22573_0101100000101101 multiply is ==  235413817_00001110000010000010000100111001 and mac=3232712606_11000000101011110100011110011110
Actual   output in1=10429_0010100010111101 and in2=22573_0101100000101101 is multiply ==  235474937_00001110000010010000111111111001 mac=3232773726_11000000101100000011011001011110
______________________________________________ERROR=      61120_______________
************************************************************
________________total error=   4826496  >>>>>  totalProduct=2393067935_______
** Note: $finish    : top.sv(44)
   Time: 132 ns  Iteration: 0  Instance: /top
```

Fig 4.10 High Approximate testbench output(MODE-2)



Name	Slack ^1	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock
Path 1	∞	16	17	47	reg2[10]	mac_out[27]	17.249	5.244	12.005	∞	input port clock
Path 2	∞	16	17	47	reg2[10]	mac_out[28]	17.249	5.244	12.005	∞	input port clock
Path 3	∞	16	17	47	reg2[10]	mac_out[29]	17.249	5.244	12.005	∞	input port clock
Path 4	∞	16	17	47	reg2[10]	mac_out[30]	17.224	5.244	11.980	∞	input port clock
Path 5	∞	16	17	47	reg2[10]	mac_out[31]	17.194	5.244	11.950	∞	input port clock
Path 6	∞	15	16	47	reg2[10]	mac_out[25]	16.658	5.120	11.538	∞	input port clock
Path 7	∞	15	16	47	reg2[10]	mac_out[26]	16.658	5.120	11.538	∞	input port clock
Path 8	∞	16	17	47	reg2[10]	mul_out[31]	16.077	5.220	10.857	∞	input port clock
Path 9	∞	15	16	46	reg2[7]	mac_out[23]	15.803	5.168	10.635	∞	input port clock
Path 10	∞	15	16	46	reg2[7]	mac_out[24]	15.797	5.162	10.635	∞	input port clock

Fig 4.11 Timing Analysis for MODE-2 Approximate MAC



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

Total On-Chip Power:	**67.675 W (Junction temp exceeded!)**
Design Power Budget:	**Not Specified**
Process:	typical
Power Budget Margin:	**N/A**
Junction Temperature:	**125.0°C**
Thermal Margin:	-278.3°C (-55.3 W)
Ambient Temperature:	25.0 °C
Effective ϑJA:	5.0°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power

Dynamic:	67.190 W	(99%)	
Signals:	10.160 W	(15%)	
Logic:	6.606 W	(10%)	
I/O:	50.424 W	(75%)	
Device Static:	0.485 W	(1%)	

Fig 4.12 Power Analysis for MODE-2 Approximate MAC

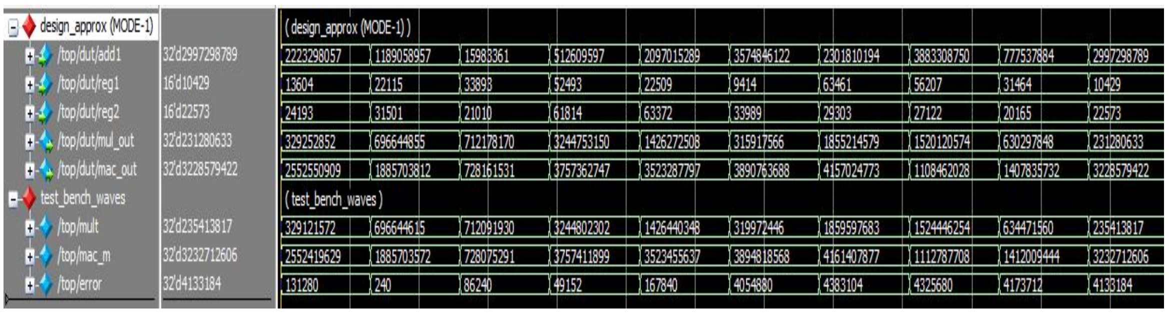
4.1.4 Low Approximate Mode Results:

Here, three out of four multipliers are approximate. This mode prioritizes power and area reduction over accuracy.

Fig 4.13 – 4.14: Output closely matches accurate output.

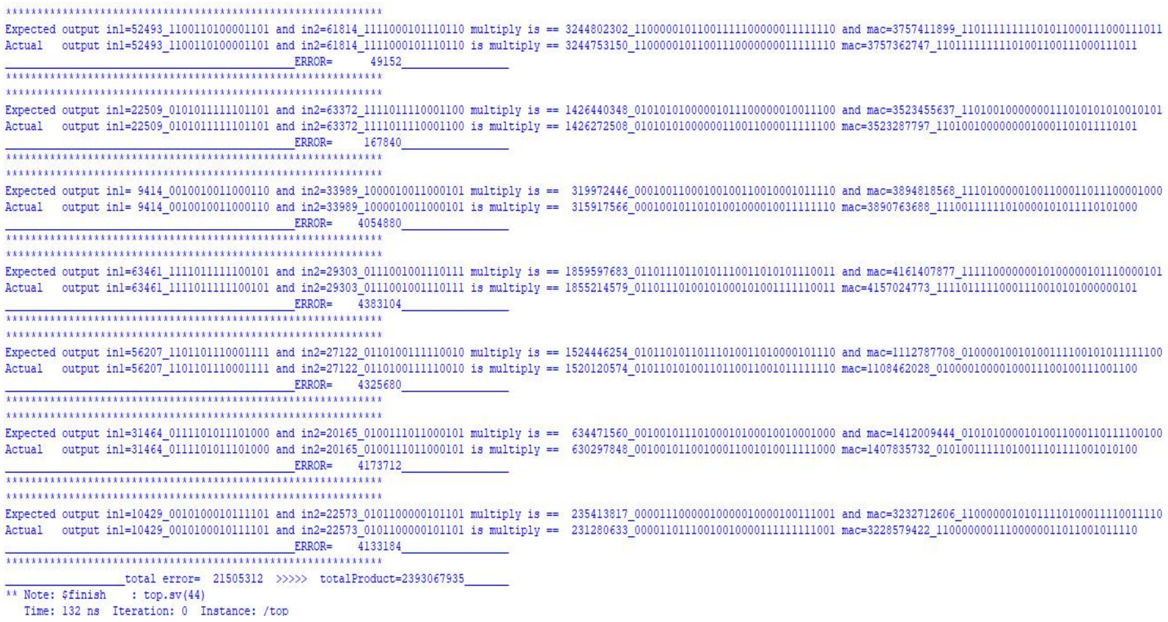
Fig 4.15: Latency significantly reduced compared to Mode-4.

Fig 4.16: Lowest power consumption (~65.33W).



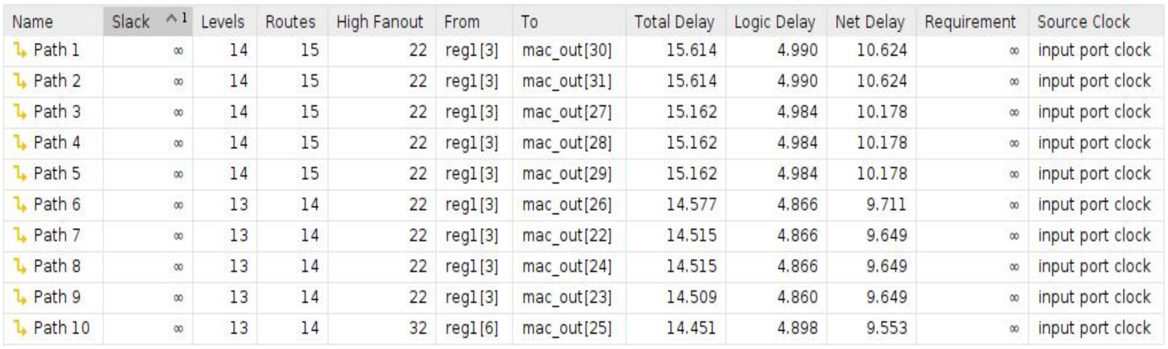
```
design_approx (MODE-1)                  ( design_approx (MODE-1) )
  /top/dut/add1     32'd2997298789      2223298057  1189058957  15983361    512609597   2097015289  3574846122  2301810194  3883308750  777537884   2997298789
  /top/dut/reg1     16'd10429           13604       22115       33893       52493       22509       9414        63461       56207       31464       10429
  /top/dut/reg2     16'd22573           24193       31501       21010       61814       63372       33989       29303       27122       20165       22573
  /top/dut/mul_out  32'd231280633       329252852   696644855   712178170   3244753150  1426272508  315917566   1855214579  1520120574  630297848   231280633
  /top/dut/mac_out  32'd3228579422      2552550909  1885703812  728161531   3757362747  3523287797  3890763688  4157024773  1108462028  1407835732  3228579422
test_bench_waves                        ( test_bench_waves )
  /top/mult         32'd235413817       329121572   696644615   712091930   3244802302  1426440348  319972446   1859597683  1524446254  634471560   235413817
  /top/mac_m        32'd3232712606      2552419629  1885703572  728075291   3757411899  3523455637  3894818568  4161407877  1112787708  1412009444  3232712606
  /top/error        32'd4133184         131280      240         86240       49152       167840      4054880     4383104     4325680     4173712     4133184
```

Fig 4.13 Low Approximate Simulation Results(MODE-1)



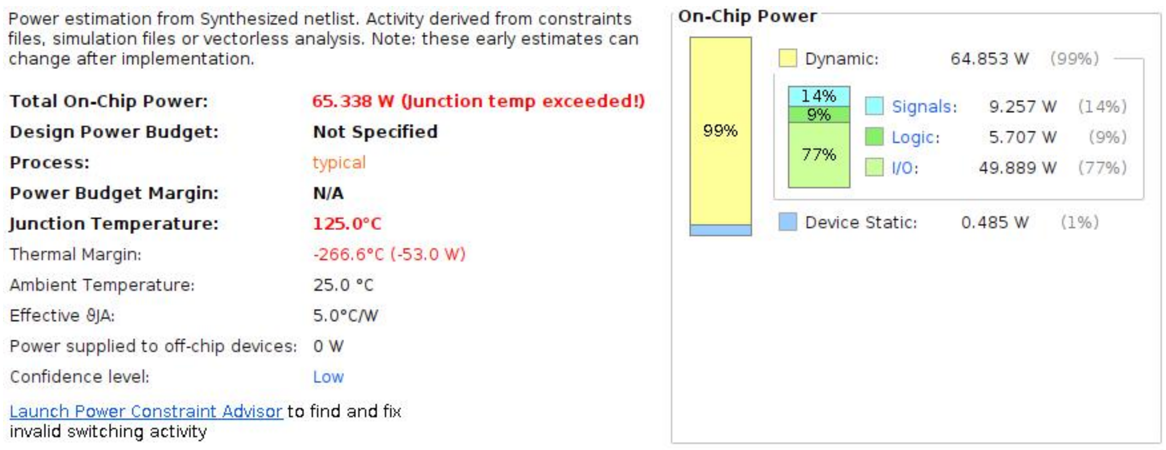
```
*********************************************************
Expected output in1=52493_1100110100001101 and in2=61814_1111000101110110 multiply is == 3244802302_11000001011001111100000011111110 and mac=3757411899_11011111111010110001110001110111
Actual   output in1=52493_1100110100001101 and in2=61814_1111000101110110 is multiply == 3244753150_11000001011001110000000011111110 mac=3757362747_11011111111010011001110001110111
_______________________________________ERROR=     49152_______________
*********************************************************
*********************************************************
Expected output in1=22509_0101011111101101 and in2=63372_1111011110001100 multiply is == 1426440348_01010101000001011100000010011100 and mac=3523455637_11010010000000111010101010010101
Actual   output in1=22509_0101011111101101 and in2=63372_1111011110001100 is multiply == 1426272508_01010101000000110011000011111100 mac=3523287797_11010010000000001000110101110101
_______________________________________ERROR=    167840_______________
*********************************************************
*********************************************************
Expected output in1= 9414_0010010011000110 and in2=33989_1000010011000101 multiply is ==  319972446_00010011000100100110010001011110 and mac=3894818568_11101000001001100011011100001000
Actual   output in1= 9414_0010010011000110 and in2=33989_1000010011000101 is multiply ==  315917566_00010010110101001000010011111110 mac=3890763688_11100111111010000101011110101000
_______________________________________ERROR=   4054880_______________
*********************************************************
*********************************************************
Expected output in1=63461_1111011111100101 and in2=29303_0111001001110111 multiply is == 1859597683_01101110110101110011010101110011 and mac=4161407877_11111000000010100000101110000101
Actual   output in1=63461_1111011111100101 and in2=29303_0111001001110111 is multiply == 1855214579_01101110100101000101001111110011 mac=4157024773_11110111110001110010101000000101
_______________________________________ERROR=   4383104_______________
*********************************************************
*********************************************************
Expected output in1=56207_1101101110001111 and in2=27122_0110100111110010 multiply is == 1524446254_01011010110111010011010000101110 and mac=1112787708_01000010010100111100101011111100
Actual   output in1=56207_1101101110001111 and in2=27122_0110100111110010 is multiply == 1520120574_01011010100110110011001011111110 mac=1108462028_01000010000100011100100111001100
_______________________________________ERROR=   4325680_______________
*********************************************************
*********************************************************
Expected output in1=31464_0111101011101000 and in2=20165_0100111011000101 multiply is ==  634471560_00100101110100010100010010001000 and mac=1412009444_01010100001010011000110111100100
Actual   output in1=31464_0111101011101000 and in2=20165_0100111011000101 is multiply ==  630297848_00100101100100011001010011111000 mac=1407835732_01010011111010011101111001010100
_______________________________________ERROR=   4173712_______________
*********************************************************
*********************************************************
Expected output in1=10429_0010100010111101 and in2=22573_0101100000101101 multiply is ==  235413817_00001110000010000010000100111001 and mac=3232712606_11000000101011110100011110011110
Actual   output in1=10429_0010100010111101 and in2=22573_0101100000101101 is multiply ==  231280633_00001101110010010000111111111001 mac=3228579422_11000000011100000011011001011110
_______________________________________ERROR=   4133184_______________
*********************************************************
________________total error=  21505312  >>>>>  totalProduct=2393067935_______
** Note: $finish    : top.sv(44)
   Time: 132 ns  Iteration: 0  Instance: /top
```

Fig 4.14 Low Approximate testbench output(MODE-1)



Name	Slack ^1	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock
Path 1	∞	14	15	22	reg1[3]	mac_out[30]	15.614	4.990	10.624	∞	input port clock
Path 2	∞	14	15	22	reg1[3]	mac_out[31]	15.614	4.990	10.624	∞	input port clock
Path 3	∞	14	15	22	reg1[3]	mac_out[27]	15.162	4.984	10.178	∞	input port clock
Path 4	∞	14	15	22	reg1[3]	mac_out[28]	15.162	4.984	10.178	∞	input port clock
Path 5	∞	14	15	22	reg1[3]	mac_out[29]	15.162	4.984	10.178	∞	input port clock
Path 6	∞	13	14	22	reg1[3]	mac_out[26]	14.577	4.866	9.711	∞	input port clock
Path 7	∞	13	14	22	reg1[3]	mac_out[22]	14.515	4.866	9.649	∞	input port clock
Path 8	∞	13	14	22	reg1[3]	mac_out[24]	14.515	4.866	9.649	∞	input port clock
Path 9	∞	13	14	22	reg1[3]	mac_out[23]	14.509	4.860	9.649	∞	input port clock
Path 10	∞	13	14	32	reg1[6]	mac_out[25]	14.451	4.898	9.553	∞	input port clock

Fig 4.15 Timing Analysis for MODE-1 Approximate MAC



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

Total On-Chip Power:	65.338 W (Junction temp exceeded!)
Design Power Budget:	Not Specified
Process:	typical
Power Budget Margin:	N/A
Junction Temperature:	125.0°C
Thermal Margin:	-266.6°C (-53.0 W)
Ambient Temperature:	25.0 °C
Effective ϑJA:	5.0°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power

Dynamic:	64.853 W	(99%)
Signals:	9.257 W	(14%)
Logic:	5.707 W	(9%)
I/O:	49.889 W	(77%)
Device Static:	0.485 W	(1%)

Fig 4.16 Power Analysis for MODE-1 Approximate MAC

4.2 Comparative Analysis

As seen, a trade-off between accuracy, delay, and power is evident across different modes. While Mode-4 offers 100% accuracy, Mode-1 and Mode-2 significantly improve delay and power at a minor cost to precision.

MODE	Error (%)	Accuracy(%)	Net Delay(ns)	Power(W)
1	0.89	99.10	10.624	65.338
2	0.201	99.798	12.005	67.675
3	0.00145	99.9985	11.334	69.341
4	0	100	13.104	76.66

Table 4.1 Overall Approximate Mac Comparison

The proposed Approximate MAC unit, especially in its high and low approximation configurations, demonstrates excellent potential for energy-efficient applications such as edge computing and deep learning inference. The results validate the effectiveness of configurable approximation, enabling adaptable designs that meet specific application constraints.

***

Conclusion and Future Work

In this work, we presented a high-efficiency Approximate MAC Unit design based on a Wallace Tree multiplier and a 32-bit Carry Save Adder enhanced with 5-bit Carry Look Ahead Adders. The architecture achieves a favorable balance between speed, area, and power efficiency while introducing minimal approximation error. Our simulations confirm that the proposed design significantly outperforms conventional accurate MAC units in terms of throughput and silicon footprint, making it suitable for low-power and error-resilient applications such as image processing and deep neural networks.

Future enhancements to this design could include:

Dynamic Approximation Control: Integrating a mechanism to adjust approximation levels based on real-time performance or energy constraints.

Extension to Floating-Point Operations: Expanding the approximate MAC concept to support floating-point arithmetic for broader applicability.

ASIC Implementation and Fabrication: Moving from simulation to physical implementation to evaluate real-world power, performance, and area (PPA) metrics.

Error-Resilience Testing: Applying the architecture to specific use cases such as convolution neural networks (CNNs) to assess tolerance to approximation-induced errors.

Optimization for Emerging Technologies: Adapting the design for FinFET or other sub-10nm technologies to leverage better scalability.

***.

References

[1] Approximate Computing Survey, Part I: Terminology and Software & Hardware Approximation Techniques.

[2] K. Bhardwaj and P. S. Mane, “Acma: Accuracy-configurable multiplier architecture for error-resilient system-on-chip,” in Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), 2013 8th International Workshop on, July 2013, pp. 1–6.

[3] Kartikeya Bhardwaj, Pravin S. Mane, Jorg Henkel, Power- and Area-Efficient Approximate Wallace Tree Multiplier for Error-Resilient Systems.

[4] K. Anirudh Kumar Maurya, K.Bala Sindhuri, Y.Rama Lakshmanna, N.Udaya Kumar

Design and Implementation of 32-Bit Adders Using Various Full Adders

[5] C. S. Wallace, “A suggestion for a fast multiplier,” in Electronic Computers, 1964 IEEE Transactions on, 1964, pp. 14–17.

[6] Prof. S.Murugeswari and Dr. S.Kaja Mohideen, “Design of area efficient and low power multiplier using multiplexer based full adder” in International Conference on Current Trends in Engineering and Technology, ICCET, Coimbatore, pp.388-392, 2014.

[7] peter j. ashenden, Digital Design An Embedded Systems Approach Using Verilog Chapter 3,page no.96

[8] Approximate Computing Survey, Part II: Terminology and Software & Hardware Approximation Techniques.

[9] E. J. Swartzlander, “Truncated multiplication with approximate rounding,” in Signals, Systems, and Computers, 1999. Conference Record of the Thirty-Third Asilomar Conference on, vol. 2, oct. 1999, pp. 1480– 1483 vol.2.

[10] N. Zhu, W. L. Goh, W. Zhang, K. S. Yeo, and Z. H. Kong, “Design of low-power high-speed truncation-error-tolerant adder and its application in digital signal processing,” Very Large Scale Integration (VLSI) Systems, IEEE Transactions on, vol. 18, no. 8, pp. 1225–1229, aug. 2010.

[11] Fabio Frustaci, Stefania Perri, Pasquale Corsonello, and Massimo Alioto. 2020. Approximate multipliers with dynamic truncation for energy reduction via graceful quality degradation. IEEE Trans. on Circuits and Systems II: Express Briefs 67, 12 (2020), 3427–3431.

[12] Soheil Hashemi, R. Iris Bahar, and Sherief Reda. 2015. DRUM: A dynamic range unbiased multiplier for approximate applications. In Int’l. Conference on Computer-Aided Design (ICCAD). 418–425.

[13] Reza Zendegani, Mehdi Kamal, Milad Bahadori, Ali Afzali-Kusha, and Massoud Pedram. 2017. RoBA multiplier: A rounding-based approximate multiplier for high-speed yet energy-efficient digital signal processing. IEEE Trans. on Very Large Scale Integration (VLSI) Systems 25, 2 (2017), 393–401.

[14] Vasileios Leon, Theodora Paparouni, Evangelos Petrongonas, Dimitrios Soudris, and Kiamal Pekmestzi. 2021. Improving power of DSP and CNN hardware accelerators using approximate floating-point multipliers. ACM Trans. on Embedded Computing Systems 20, 5 (2021), 1–21

[15] Vassilis Vassiliadis, Konstantinos Parasyris, Charalambos Chalios, Christos D. Antonopoulos, Spyros Lalis, Nikolaos Bellas, Hans Vandierendonck, and Dimitrios S. Nikolopoulos. 2015. A programming model and runtime system for significance-aware energy-efficient computing. In ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP). 275–276.

[16] Haroon Waris, Chenghua Wang, and Weiqiang Liu. 2020. Hybrid low radix encoding-based approximate booth multipliers. IEEE Trans. on Circuits and Systems II: Express Briefs 67, 12 (2020), 3367–3371.

[17] Haroon Waris, Chenghua Wang, Weiqiang Liu, and Fabrizio Lombardi. 2021. AxBMs: Approximate radix-8 booth multipliers for high-performance FPGA-based accelerators. IEEE Trans. on Circuits and Systems II: Express Briefs 68, 5 (2021), 1566–1570.

***

A. Verilog design code for MAC Units

```
module ha(input a, b, output s0, c0);

  assign s0 = a ^ b;

  assign c0 = a & b;

endmodule

module fa(input a, b, cin, output sum, cout);

wire condition=b^cin;

assign sum = condition ? (~a) : a;

assign cout= condition ?  a : b;

endmodule

module adder4 (input [4:0]in1,in2,input cin,output [4:0]sum,output cout);

wire [4:0]P,G;

wire [5:0]c;

assign P=in1^in2;

assign G=in1 & in2;

assign c[0]=cin;

assign c[1]=G[0] | P[0] & cin;

assign c[2]=G[1] | P[1] & G[0] | P[1] & P[0] & cin;

assign c[3]=G[2] | P[2] & G[1] | P[2] & P[1] & G[0] | P[2]&P[1]&P[0]&cin;

assign c[4]=G[3] | P[3] & G[2] | P[3] & P[2] & G[1] | P[3]&P[2]&P[1]&G[0] |
P[3]&P[2]&P[1]&P[0]&cin;

assign c[5]=G[4] | P[4] & G[3] | P[4] & P[3] & G[2] | P[4]&P[3]&P[2]&G[1] |
P[4]&P[3]&P[2]&P[1]&G[0] | P[4]&P[3]&P[2]&P[1]&P[0]&cin;

assign cout=c[5];

assign sum = P ^ c;

endmodule

module adder4c (input [4:0]in1,in2,output [4:0]sum,output cout);

wire [4:0]P,G;
wire [5:0]c;

assign P=in1^in2;

assign G=in1 & in2;

assign c[0]=1'b0;

assign c[1]=G[0];

assign c[2]=G[1] | P[1] & G[0];

assign c[3]=G[2] | P[2] & G[1] | P[2] & P[1] & G[0];

assign c[4]=G[3] | P[3] & G[2] | P[3] & P[2] & G[1] | P[3]&P[2]&P[1]&G[0];

assign c[5]=G[4] | P[4] & G[3] | P[4] & P[3] & G[2] | P[4]&P[3]&P[2]&G[1] | P[4]&P[3]&P[2]&P[1]&G[0];

assign cout=c[5];

assign sum = P ^ c;

endmodule

module add21(input [20:0]add1,add2,output [20:0]sum,output cout);

   wire [3:0]c;

   adder4c a1(add1[4:0],add2[4:0],sum[4:0],c[0]);

   adder4 a2(add1[9:5],add2[9:5],c[0],sum[9:5],c[1]);

   adder4 a3(add1[14:10],add2[14:10],c[1],sum[14:10],c[2]);

   adder4 a4(add1[19:15],add2[19:15],c[2],sum[19:15],c[3]);

   assign sum[20]=add1[20]^add2[20];

   assign cout = (add1[20] & c[3]) | (add2[20] & c[3]) | (add1[20] & add2[20]);


endmodule

module CLA( input [7:0]A, B, input Cin, output [7:0] S,  output Cout );

  wire [7:0] Ci;

   assign Ci[0] = Cin;

  assign Ci[1] = (A[0] & B[0]) | ((A[0]^B[0]) & Ci[0]);

  assign Ci[2] = (A[1] & B[1]) | ((A[1]^B[1]) & ((A[0] & B[0]) | ((A[0]^B[0]) & Ci[0])));

  assign Ci[3] = (A[2] & B[2]) | ((A[2]^B[2]) & ((A[1] & B[1]) | ((A[1]^B[1]) & ((A[0] & B[0]) | ((A[0]^B[0]) & Ci[0])))));

  assign Ci[4] = (A[3] & B[3]) | ((A[3]^B[3]) & Ci[3]);

  assign Ci[5] = (A[4] & B[4]) | ((A[4]^B[4]) & Ci[4]);

  assign Ci[6] = (A[5] & B[5]) | ((A[5]^B[5]) & Ci[5]);
  assign Ci[7] = (A[6] & B[6]) | ((A[6]^B[6]) & Ci[6]);
  assign Cout = (A[7] & B[7]) | ((A[7]^B[7]) & Ci[7]);
  assign S = A^B^Ci;
endmodule
module multiplier_4b(input [3:0] A, B, output [7:0] P);
  wire [3:0] R0, R1, R2, R3;
  wire [4:0] X,Y;
  wire [5:-2] M;
  wire [4:-3] N;
  wire car;
  // generation of partial products;
  genvar g;
  generate
    for(g = 0; g<4; g=g+1) begin
      and (R0[g], A[g], B[0]);
      and (R1[g], A[g], B[1]);
      and (R2[g], A[g], B[2]);
      and (R3[g], A[g], B[3]);
    end
  endgenerate

  // generation of sums and carries
  ha add1(R0[1],R1[0],X[0],Y[0]);
  fa add2(R0[2],R1[1],R2[0],X[1],Y[1]);
  fa add3(R0[3],R1[2],R2[1],X[2],Y[2]);
  fa add4(R1[3],R2[2],R3[1],X[3],Y[3]);
  ha add5(R2[3],R3[2],X[4],Y[4]);
  ha add6(X[1],Y[0],M[0],N[0]);
  fa add7(X[2],Y[1],R3[0],M[1],N[1]);
  ha add8(X[3],Y[2],M[2],N[2]);
  ha add9(X[4],Y[3],M[3],N[3]);
  ha add10(R3[3],Y[4],M[4],N[4]);
```

```
 assign M[-1]=X[0];
 assign M[-2]=R0[0];
 assign M[5]=0;
 assign N[-1]=0;
 assign N[-2]=0;
 assign N[-3]=0;
 CLA add11(M,N,1'b0,P,car);
endmodule
module approx_mult(input [7:0]in1,in2,output [15:0]out_oppx);
    wire [7:0]reg_par[7:0];
    genvar i;
    wire cin;
    for(i=0;i<8;i=i+1)begin
            assign reg_par[i] = in1 & {8{in2[i]}};
    end
    assign out_oppx[4]=1'b1;
    assign out_oppx[5]=1'b1;
    assign out_oppx[6]=1'b1;
    assign out_oppx[7]=1'b1;
    assign cin=reg_par[0][7] | reg_par[1][6] | reg_par[2][5] | reg_par[3][4] | reg_par[4][3] |
reg_par[5][2] | reg_par[6][1] | reg_par[7][0];
    assign out_oppx[0]=reg_par[0][0];
    wire [13:1]s;
    wire [13:1]c;
    ha h1(reg_par[0][1],reg_par[1][0],s[1],c[1]);
    assign out_oppx[1]=s[1];
    wire [9:0]ss;
    wire [9:0]css;
    fa f1(reg_par[0][2],reg_par[1][1],reg_par[2][0],s[2],c[2]);
    ha h2(s[2],c[1],ss[0],css[0]);
    assign out_oppx[2]=ss[0];
    wire [6:0]sss;
```

```
      wire [6:0]csss;
      fa f2(reg_par[0][3],reg_par[1][2],reg_par[2][1],s[3],c[3]);
      fa f3(s[3],reg_par[3][0],c[2],ss[1],css[1]);
      ha h3(ss[1],css[0],sss[0],csss[0]);
      assign out_oppx[3]=sss[0];
      fa f4(reg_par[1][7],reg_par[2][6],cin,s[4],c[4]);
      fa f5(reg_par[3][5],reg_par[4][4],reg_par[5][3],s[5],c[5]);
      ha h4(reg_par[6][2],reg_par[7][1],s[6],c[6]);
      fa f6(s[4],s[5],s[6],ss[2],css[2]);
      assign out_oppx[8]=ss[2];
      fa f7(reg_par[2][7],reg_par[3][6],reg_par[4][5],s[7],c[7]);
      fa f8(reg_par[5][4],reg_par[6][3],reg_par[7][2],s[8],c[8]);
      fa f9(s[7],s[8],c[4],ss[3],css[3]);
      ha h5(c[5],c[6],ss[9],css[9]);
      fa f10(ss[3],ss[9],css[2],sss[1],csss[1]);
      assign out_oppx[9]=sss[1];
      fa f11(reg_par[3][7],reg_par[4][6],reg_par[5][5],s[9],c[9]);
      ha h6(reg_par[6][4],reg_par[7][3],s[10],c[10]);
      fa f12(s[9],s[10],c[7],ss[4],css[4]);
      fa f13(ss[4],c[8],css[3],sss[2],csss[2]);
      fa f14(reg_par[4][7],reg_par[5][6],reg_par[6][5],s[11],c[11]);
      fa f15(s[11],reg_par[7][4],c[9],ss[5],css[5]);
      fa f16(ss[5],c[10],css[4],sss[3],csss[3]);
   //12
      fa f17(reg_par[5][7],reg_par[6][6],reg_par[7][5],s[12],c[12]);
      ha h7(s[12],c[11],ss[6],css[6]);
      ha h8(ss[6],css[5],sss[4],csss[4]);
      //13
      ha h9(reg_par[6][7],reg_par[7][6],s[13],c[13]);
      ha h10(s[13],c[12],ss[7],css[7]);
      ha h11(ss[7],css[6],sss[5],csss[5]);
      //14
```

```
    ha h12(reg_par[7][7],c[13],ss[8],css[8]);
    ha h13(ss[8],css[7],sss[6],csss[6]);
    //15 stage 3 adder
    wire [5:0]ca;
    fa a1(sss[2],css[9],csss[1],out_oppx[10],ca[0]);
    fa a2(sss[3],csss[2],ca[0],out_oppx[11],ca[1]);
    fa a3(sss[4],csss[3],ca[1],out_oppx[12],ca[2]);
    fa a4(sss[5],csss[4],ca[2],out_oppx[13],ca[3]);
    fa a5(sss[6],csss[5],ca[4],out_oppx[14],ca[4]);
    fa a6(css[8],csss[6],ca[5],out_oppx[15],ca[5]);
endmodule
module accurates(input [7:0]in1,in2,output [7:0]b,a,c,d);
    multiplier_4b mult_AHHXHL(in1[7:4], in2[3:0], c);
    multiplier_4b mult_AHHXHH(in1[7:4], in2[7:4], b);
    multiplier_4b mult_AHLXHL(in1[3:0], in2[3:0], a);
    multiplier_4b mult_AHLXHH(in1[3:0], in2[7:4], d);
endmodule
module acc_mult(input [7:0]b,a,c,d,output [15:0]ap);
    wire [7:0]s,cc;
    fa f1(a[4],c[0],d[0],s[0],cc[0]);
    fa f2(a[5],c[1],d[1],s[1],cc[1]);
    fa f3(a[6],c[2],d[2],s[2],cc[2]);
    fa f4(a[7],c[3],d[3],s[3],cc[3]);
    fa f5(b[0],c[4],d[4],s[4],cc[4]);
    fa f6(b[1],c[5],d[5],s[5],cc[5]);
    fa f7(b[2],c[6],d[6],s[6],cc[6]);
    fa f8(b[3],c[7],d[7],s[7],cc[7]);
    assign ap[4:0]={s[0],a[3:0]};
    wire [6:0]ccc;
    adder4 add(s[5:1],cc[4:0],1'b0,ap[9:5],ccc[0]);
    fa ff1(s[6],cc[5],ccc[0],ap[10],ccc[1]);
    fa ff2(s[7],cc[6],ccc[1],ap[11],ccc[2]);
```

```
     fa ff3(b[4],cc[7],ccc[2],ap[12],ccc[3]);
     ha h1(b[5],ccc[3],ap[13],ccc[4]);
     ha h2(b[6],ccc[4],ap[14],ccc[5]);
     ha h3(b[7],ccc[5],ap[15],ccc[6]);
endmodule
module acc_mult15(input [7:0]in1,in2,output [15:0]ap);
     wire [7:0]b,a,c,d;
     accurates acc(in1,in2,b,a,c,d);
     acc_mult mul(b,a,c,d,ap);
endmodule
module pip_acc(input [15:0]a,b,c,d, output [31:0]prod);
assign          prod[7:0]=b[7:0];
wire [15:0]x= {a[7:0],b[15:8]};
wire [15:0]s,cs;
genvar i;
for(i=0;i<=15;i=i+1)begin
     fa f_i(x[i],c[i],d[i],s[i],cs[i]);
end
assign prod[8]=s[0];
wire [6:0]cout;
adder4c ad1(s[5:1],cs[4:0],prod[13:9],cout[0]);
adder4  ad2(s[10:6],cs[9:5],cout[0],prod[18:14],cout[1]);
adder4  ad3(s[15:11],cs[14:10],cout[1],prod[23:19],cout[2]);
wire [4:0]bb={4'b0,cs[15]};
adder4 ad4(a[12:8],bb,cout[2],prod[28:24],cout[3]);
ha hf1(a[13],cout[3],prod[29],cout[4]);
ha hf2(a[14],cout[4],prod[30],cout[5]);
ha hf3(a[15],cout[5],prod[31],cout[6]);
endmodule

module adderpg4 (input [4:0]in1,in2,input cin,output [4:0]sum,output cout,output [4:0]PP,GG);
wire [4:0]P,G;
wire [5:0]c;

assign P=in1^in2;

assign G=in1 & in2;

assign {PP,GG}={P,G};

assign c[0]=cin;

assign c[1]=G[0] | P[0] & cin;

assign c[2]=G[1] | P[1] & G[0] | P[1] & P[0] & cin;

assign c[3]=G[2] | P[2] & G[1] | P[2] & P[1] & G[0] | P[2]&P[1]&P[0]&cin;

assign c[4]=G[3] | P[3] & G[2] | P[3] & P[2] & G[1] | P[3]&P[2]&P[1]&G[0] | P[3]&P[2]&P[1]&P[0]&cin;

assign c[5]=G[4] | P[4] & G[3] | P[4] & P[3] & G[2] | P[4]&P[3]&P[2]&G[1] | P[4]&P[3]&P[2]&P[1]&G[0] | P[4]&P[3]&P[2]&P[1]&P[0]&cin;

assign cout=c[5];

assign sum = P ^ c;

endmodule

module adderpg4c (input [4:0]in1,in2,output [4:0]sum,output cout, output [4:0]PP,GG);

wire [4:0]P,G;

wire [5:0]c;

assign P=in1^in2;

assign G=in1 & in2;

assign {PP,GG}={P,G};

assign c[0]=1'b0;

assign c[1]=G[0];

assign c[2]=G[1] | P[1] & G[0];

assign c[3]=G[2] | P[2] & G[1] | P[2] & P[1] & G[0];

assign c[4]=G[3] | P[3] & G[2] | P[3] & P[2] & G[1] | P[3]&P[2]&P[1]&G[0];

assign c[5]=G[4] | P[4] & G[3] | P[4] & P[3] & G[2] | P[4]&P[3]&P[2]&G[1] | P[4]&P[3]&P[2]&P[1]&G[0];

assign cout=c[5];

assign sum = P ^ c;

endmodule

module mux(input in0,in1,sel, output out);

   assign out=sel?in1:in0;

endmodule
```

```
module adder32(input [31:0]in1,in2,output [31:0]out);
    wire [29:0]P,G;
    wire [5:0]sk;
    wire [7:0]c;
    adderpg4c ad1(in1[4:0],in2[4:0],out[4:0],c[0],P[4:0],G[4:0]);
    mux skip1(c[0],1'b0,&(P[4:0]),sk[0]);
    adderpg4   ad2(in1[9:5],in2[9:5],sk[0],out[9:5],c[1],P[9:5],G[9:5]);
    mux skip2(c[1],sk[0],&(P[9:5]),sk[1]);
    adderpg4   ad3(in1[14:10],in2[14:10],sk[1],out[14:10],c[2],P[14:10],G[14:10]);
    mux skip3(c[2],sk[1],&(P[14:10]),sk[2]);
    adderpg4   ad4(in1[19:15],in2[19:15],sk[2],out[19:15],c[3],P[19:15],G[19:15]);
    mux skip4(c[3],sk[2],&(P[19:15]),sk[3]);
    adderpg4   ad5(in1[24:20],in2[24:20],sk[3],out[24:20],c[4],P[24:20],G[24:20]);
    mux skip5(c[4],sk[3],&(P[24:20]),sk[4]);
    adderpg4   ad6(in1[29:25],in2[29:25],sk[4],out[29:25],c[5],P[29:25],G[29:25]);
    mux skip6(c[5],sk[4],&(P[29:25]),sk[5]);
    fa f1(in1[30],in2[30],sk[5],out[30],c[6]);
    fa f2(in1[31],in2[31],c[6],out[31],c[7]);
endmodule
module accurate_mode(input [15:0]in1,in2,output [31:0]acc_prod);
    wire [15:0]ALXL,AHXH,AHXL,ALXH;
    acc_mult15 mul1(in1[7:0],in2[7:0],ALXL);
    acc_mult15 mul2(in1[15:8],in2[15:8],AHXH);
    acc_mult15 mul3(in1[15:8],in2[7:0],AHXL);
    acc_mult15 mul4(in1[7:0],in2[15:8],ALXH);
    pip_acc main(AHXH,ALXL,AHXL,ALXH,acc_prod);
endmodule

// VERY APPROXIMATE MULTIPLIER
module vhigh_approx(input [15:0]in1,in2,output [31:0]acc_prod);
    wire [15:0]ALXL,AHXH,AHXL,ALXH;
    approx_mult mult_ALXL(in1[7:0], in2[7:0], ALXL);
    acc_mult15 mul2(in1[15:8],in2[15:8],AHXH);
    acc_mult15 mul3(in1[15:8],in2[7:0],AHXL);
    acc_mult15 mul4(in1[7:0],in2[15:8],ALXH);
    pip_acc main(AHXH,ALXL,AHXL,ALXH,acc_prod);

endmodule
// HIGH_APPROXIMATE MODE
module high_approx(input [15:0]in1,in2,output [31:0]acc_prod);
    wire [15:0]ALXL,AHXH,AHXL,ALXH;
    approx_mult mult_ALXL(in1[7:0], in2[7:0], ALXL);
    acc_mult15 mul2(in1[15:8],in2[15:8],AHXH);
    approx_mult mult_AHXL(in1[15:8], in2[7:0], AHXL);
    acc_mult15 mul4(in1[7:0],in2[15:8],ALXH);
    pip_acc main(AHXH,ALXL,AHXL,ALXH,acc_prod);
endmodule
// LOW_APPROXIMATE MODE
module low_approx(input [15:0]in1,in2,output [31:0]acc_prod);
    wire [15:0]ALXL,AHXH,AHXL,ALXH;
    approx_mult mult_ALXL(in1[7:0], in2[7:0], ALXL);
    approx_mult mult_AHXL(in1[15:8], in2[7:0], AHXL);
    approx_mult mult_ALXH(in1[7:0], in2[15:8], ALXH);
    acc_mult15 mul2(in1[15:8],in2[15:8],AHXH);
    pip_acc main(AHXH,ALXL,AHXL,ALXH,acc_prod);
endmodule
module mac32(input [31:0] add1,input [15:0]reg1,reg2,output [31:0]mul_out,mac_out);
    accurate_mode m1(reg1,reg2,mul_out);
    adder32 ad1(add1,mul_out,mac_out);
Endmodule
```

B. Verilog Testbench for MAC design

```
`include"my_mac.sv"

module top;

reg [15:0] in1,in2;

reg [31:0] product,add1,mac_out;

reg seed=1231;

reg [31:0] mult,mac_m;

integer error1,error2,error;

integer count=20;

reg [31:0] total_error=0,total_product=0;

real error_percent;

real error_per;

mac32 dut(.add1(add1),.reg1(in1),.reg2(in2),.mul_out(product),.mac_out(mac_out));

initial begin

        repeat(count) begin

                in1=$random;

                in2=$random;

                add1=$random;

                #100;

                mult=in1*in2;

                mac_m=mult+add1;

        $display("************************************************************");

  $display("Expected output in1=%d_%b and in2=%d_%b multiply is == %d_%b and
  mac=%d_%b",in1,in1,in2,in2,mult,mult,mac_m,mac_m);

  $display("Actual   output in1=%d_%b and in2=%d_%b is multiply == %d_%b
    mac=%d_%b",in1,in1,in2,in2,product,product,mac_out,mac_out);

                error1=product-mult;

                error2=mult-product;

                error=error1>error2?error1:error2;
                total_product=total_product+mult;

                total_error=total_error+error;

                error_per=(error/mult)*100;

        $display("_____________ERROR=%d_%b__________e1=%d
e2=%d______error_percent=%d%%", error1>error2?error1:error2,
error1>error2?error1:error2,error1,error2,error_per);

        if(mult==product && mac_m==mac_out)

$display("------------------------- ------------------------------PASS---------");

else

$display("-----------FAIL----------");

$display("**********************************************************");

        end

        error_percent= (total_error/total_product) *100;

        $display("__________________ERROR_PERCENT=%d%%_total
error=%d>>>>>totalProduct_______",error_percent,total_error,total_product);

        #100 $finish;

end

endmodule
```

C. Key Terms and Definitions

Approximate Computing

A design paradigm that intentionally introduces minor inaccuracies in computation to achieve significant improvements in power, area, and speed. Suitable for applications where perfect accuracy is not critical, such as image processing and machine learning.

MAC Unit (Multiply-Accumulate Unit)

A fundamental arithmetic block that performs multiplication followed by addition:

$$MAC(a,b,c)=a\times b+c$$

Widely used in digital signal processing and neural networks.

Wallace Tree Multiplier

A hardware-efficient method for multiplying two binary numbers. It reduces the number of partial products using layers of carry-save adders, significantly increasing speed compared to array multipliers.

Carry Save Adder (CSA)

An adder that adds three or more binary numbers without propagating the carry immediately. It outputs a sum and carry vector, which are later added using a conventional adder. Useful in multi-operand addition scenarios like multipliers.

Carry Look Ahead Adder (CLA)

A type of fast adder that computes carry signals in advance based on input bits, reducing propagation delay compared to ripple carry adders. In this project, 5-bit CLAs are used to finalize the addition efficiently.

Partial Product

Intermediate results of bitwise multiplication of one operand with each bit of the other operand. These are summed to produce the final multiplication result.

Adder Tree

A hierarchical structure of adders used to sum multiple operands efficiently. Wallace Tree is a form of adder tree.

Latency

The total time taken to complete a single operation, from input to output. Reducing latency is crucial in real-time applications.

Throughput

The rate at which operations are completed. Higher throughput means more operations are processed in less time, essential for high-performance computing.

Error-Tolerant Design

Circuit design that allows controlled errors to reduce complexity, area, or power consumption without significantly degrading overall application-level performance.

***