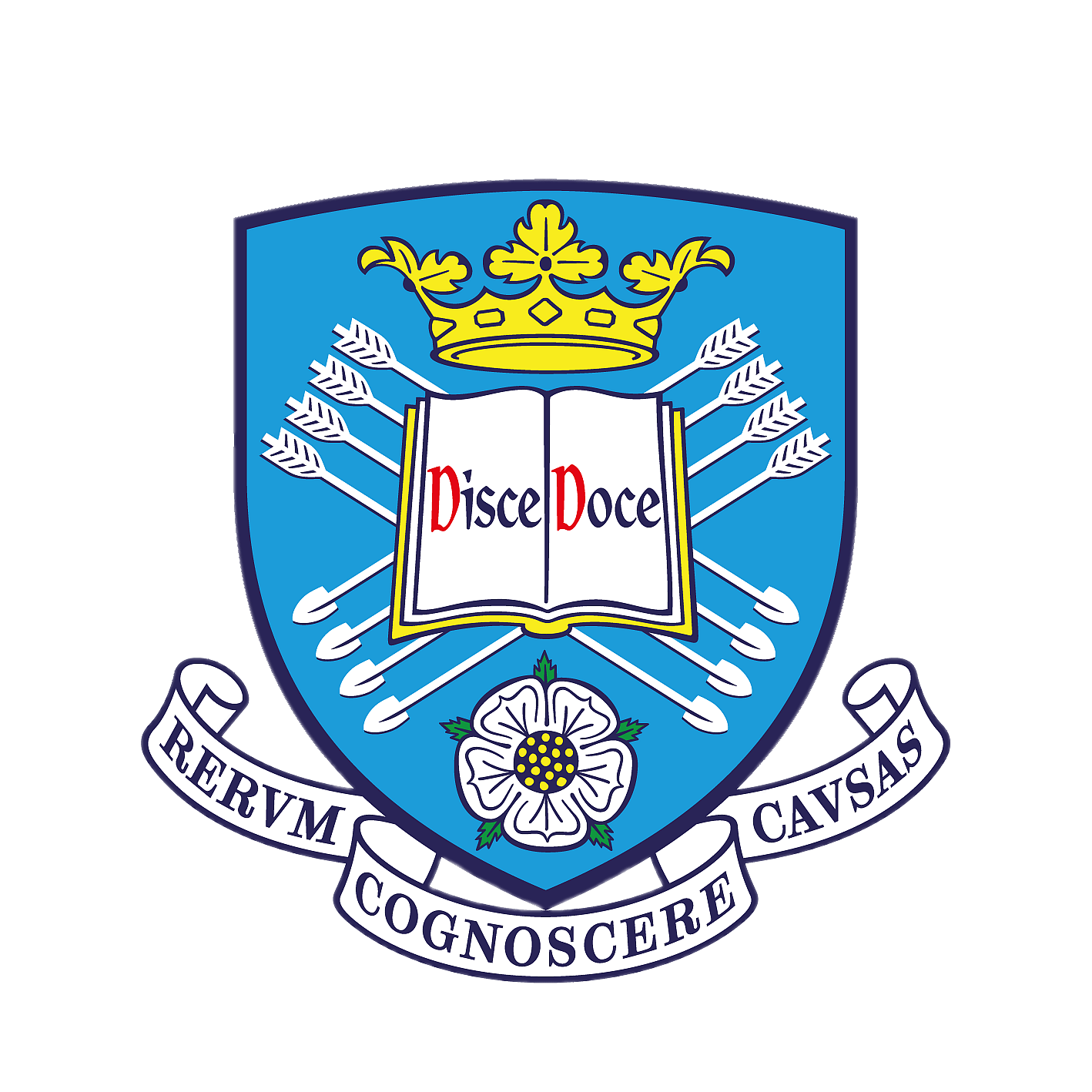
**Design of an Approximate 8-bit MAC Unit: A Step Towards RISC-V SoC Integration**

****

**Amaan Mujawar**

University of Sheffield

School of Electrical and Electronic Engineering

*Supervised by*

Mr. Niel Powell

*A report submitted in partial fulfilment of the*

*requirements for the degree of*

Masters of Engineering

7th May 2025

**Declaration**

All sentences or passages quoted in this report from other people’s work have been specifically acknowledged by clear cross-referencing to author, work and page(s). Any illustrations that are not the work of the author of this report have been used with the explicit permission of the originator and are specifically acknowledged. I understand that failure to do this amount to plagiarism and will be considered grounds for failure in this

project and the degree examination as a whole.

Amaan Mujawar

**Abstract**

The rapid growth in data-intensive applications, such as signal processing, image processing, and multimedia tasks, has led to a demand for more efficient computing solutions. Many signal processing, image processing, and multimedia tasks are inherently error-tolerant and can produce results that appear indistinguishable to the human eye, even without the need for exact computations. By leveraging this error tolerance approximate computing provides an approach, offering reduced power consumption and increased processing speeds by tolerating minor inaccuracies in calculations.

The primary goal of this project is to design and implement a MAC unit that seamlessly integrates with the RISC-V processor architecture while achieving performance improvements, such as lower power consumption, faster processing speeds, and tunable accuracy levels. The project utilizes the Xilinx Vivado Design Suite for design, simulation, and synthesis, and the implementation is carried out on the Artix Nexus 7 FPGA platform. Detailed performance metrics, including power consumption, speed benchmarks, and accuracy, are measured and evaluated to ensure the validity of the design.

The design has been prepared with future integration in mind and the capability of a final integration into a RISC-V System on Chip (SoC) offering a scalable and flexible hardware solution for approximate computing in embedded systems. The results demonstrate significant power and speed improvements with acceptable accuracy losses, highlighting the potential of approximate computing techniques for future high-performance computing applications.

**Note: Check Vivado performance metrics page to validate claim**

**Individual Contribution**

This dissertation focuses on the design and implementation of a 8-bit Multiply-Accumulate (MAC) unit utilizing approximate computing techniques for power and performance optimization. While the project is primarily based on existing architecture design, I have tried an alternative approach at integrating and implementing these designs in my work. The architecture draws from existing research, hardware, and software resources. As the sole contributor to this project, I was responsible for all stages of the design process, including conceptualization, implementation, and testing. The design is built upon a solid foundation of existing knowledge in approximate computing, particularly in the areas of multipliers, adders, and hardware accelerators. Throughout the project, I leveraged several external resources, including research papers, existing open-source platforms, and simulation tools, which helped inform and guide the development process. These resources were critical to refining the approach and ensuring the robustness of the design.

The initial phase of the project involved a comprehensive literature review of approximate computing techniques, particularly focusing on approximate multipliers, adders, and their use in hardware acceleration. Building on existing research, I selected an approximate Dadda multiplier and a carry-lookahead adder (CLA) for integration into the MAC unit. I used the Xilinx Vivado Design Suite for the development and simulation of the hardware, including creating the Verilog HDL code for both the 8-bit and 32-bit versions of the MAC unit.

The MAC unit was designed using a pipelined architecture to improve throughput and performance, which was key in handling the high-speed requirements of modern data-intensive applications. A significant portion of my work focused on the integration of fuzzy memoization into the MAC design, enhancing the efficiency of the unit, although calibration was required for full functionality. I also developed comprehensive test benches and used simulation tools in Vivado to validate the performance, speed, accuracy, and power consumption of the unit.

External resources included several research papers on FPGA-based approximate multipliers and adders, which were instrumental in refining the design. I also made use of various software tools such as Git for version control, Python for testing scripts, and Vivado for synthesis and simulation.

This dissertation represents the culmination of my individual efforts, drawing on existing work and resources to achieve a novel design for approximate computing in hardware acceleration.

**1**

**Introduction**

The advancement of digital technology has revolutionized the way data is processed and analyzed, enabling a wide range of applications from machine learning and computer vision to real-time multimedia and big data analytics. These data-intensive domains place enormous demands on computing systems, pushing the boundaries of traditional processor architectures in terms of speed, energy efficiency, and scalability. Historically, improvements in computing performance have been driven by Moore’s Law, which predicted the exponential scaling of transistor density in integrated circuits. However, as transistor sizes approach physical and economic limits, the benefits of traditional scaling have diminished, resulting in a pressing need for alternative methods to achieve continued improvements in computational efficiency.

In response to these limitations, approximate computing has emerged as a promising paradigm that challenges the conventional notion of precise computation. Instead of striving for exact outputs, approximate computing allows for controlled inaccuracies, trading off precision for significant gains in performance, power efficiency, and silicon area reduction. This trade-off is particularly acceptable in many real-world applications—such as image processing, signal filtering, and machine learning—where outputs are inherently resilient to minor errors and human perception masks small computational inaccuracies.

Given the growing need for energy-efficient computing, especially in embedded and portable systems, the integration of approximate computing techniques into hardware accelerators offers an exciting opportunity. In particular, arithmetic units like the Multiply-Accumulate (MAC) unit, which are fundamental to numerous digital signal processing (DSP) and machine learning tasks, present an ideal candidate for approximation. The motivation for this dissertation stems from the need to explore such opportunities by designing a power- and performance-optimized MAC unit for a modern Reduced Instruction Set Computing (RISC-V) System on Chip (SoC) environment.

* 1. **Aims and Objectives**

The central aim of this project is to design and implement a 8-bit MAC unit that incorporates approximate computing methods, with a focus on achieving lower power consumption and faster execution time while maintaining acceptable accuracy levels for targeted applications. The MAC unit is intended for eventual integration into a RISC-V SoC, offering a scalable and efficient solution for energy-constrained computing systems.

The specific objectives of the project include:

1. **Literature Review**: To conduct a comprehensive review of existing approximate computing techniques, with an emphasis on approximate multipliers and adders suitable for integration into a MAC unit.
2. **Design**: To architect a pipelined 8-bit MAC unit incorporating selected approximation techniques such as fuzzy memoization and compressor-based multipliers, balancing performance with accuracy.
3. **Implementation**: To implement the proposed design using Verilog HDL, targeting the Xilinx Vivado Design Suite and the Artix Nexus 7 FPGA for simulation, synthesis, and hardware validation.
4. **Integration**: To prepare the MAC unit for integration into a RISC-V SoC, ensuring compatibility and modularity to support future expansion.
5. **Evaluation**: To evaluate the implemented design using metrics such as power consumption, execution speed, and computational accuracy, and to compare results against conventional designs or benchmarks.
   1. **Overview of the Dissertation**

This dissertation is structured to guide the reader through the full design and evaluation process of the approximate MAC unit. Following this introduction, Chapter 2 presents a detailed literature review of approximate computing techniques, focusing on hardware-level implementations and their application in arithmetic units. Chapter 3 describes the architectural design and methodology employed in building the MAC unit, including approximation strategies and pipelining structure. Chapter 4 discusses the implementation process in Verilog and the FPGA setup used for testing. Chapter 5 details the evaluation framework and presents the results of simulation and synthesis, analyzing the trade-offs in power, performance, and accuracy. Finally, Chapter 6 summarizes the findings, discusses limitations, and proposes directions for future work, particularly the full integration of the MAC unit into a RISC-V SoC.

**2**

**Literature Review**

The exponential growth in computational demands, driven by applications in machine learning,

multimedia processing, and big data analytics, has strained traditional digital design paradigms. Classical computing architectures prioritise precision and exactness, which come at the cost of increased power consumption, area usage, and latency. With the diminishing benefits of Moore’s Law and the rising need for energy-efficient hardware, approximate computing has emerged as a transformative approach to hardware design. Approximate computing operates on the principle that not all applications require perfect accuracy. Many domains, especially those involving human perception or probabilistic outcomes, can tolerate small errors without significant degradation in performance. By trade-off of accuracy, approximate computing reduces hardware complexity, resulting in substantial improvements in energy efficiency, and processing speed. At the heart of this model shift are arithmetic units like adders and multipliers which constitute a significant portion of computational workloads in digital systems. Optimising these units for approximate computing forms the core of this paper’s contributions.

**2.1 Approximate Adders**

Adders are a fundamental component in digital circuits, responsible for executing arithmetic operations that often dominate computational workload. Traditional adder designs prioritise accuracy, however, approximate adders introduce intentional inaccuracies to achieve resource savings. A proposed approximation approach using Lower-Part OR-based Approximate Adders [1] aligns with similar research, introducing the concept of approximate adders as a means to trade-off accuracy for reduced power consumptions and area in energy-efficient VLSI systems. Ramasamy et al. proposed a carry- based approximate full adder, demonstrating that bypassing the carry propagation chain in the least significant bits (LSB) can drastically improve speed and reduce area at the cost of negligible error [3].

**2.2 Approximate Multipliers**

Multiplication is a computationally intensive operation, making approximate multipliers a critical focus for energy-efficient design. Approximate multipliers reduce the complexity of partial product summation, which directly impacts delay and power consumption. Novel hardware design of approximate multipliers is provided, Lower-Part OR-based Approximate Multiplier [1], integrating the concept of Wallace Tree multipliers for accurate MSBs and OR- based logic for approximate LSBs. The combination of these techniques results in a novel multiplier design that balances accuracy, speed and resource utilisation, suitable for FPGA based implementations.

**2.3 Approximate Matrix Multiplication**

Matrix multiplication is a fundamental operation in numerous computational tasks, including AI, scientific computing, and graphics processing. Despite its importance, research into approximate matrix multipliers is limited. The proposed matrix multiplier design is a significant step forward, as it combines approximate multipliers and adders in a single hardware implementation [1], by targeting an FPGA platform and demonstrating scalability across different matrix sizes and bit-widths.

**2.4 Compressor-Based Approximate Multiplier**

Traditional Multiplier Architectures typically involve, partial product generation, partial product

accumulation, and final addition. Partial product generation involves producing intermediate results by multiplying bits of input operands, followed by partial product accumulation, summing the intermediate results using adders or compressors, and lastly final addition produces the output from accumulated partial products. A compressor is a combinational logic circuit used to sum multiple binary inputs and produce a small number of outputs, usually two, a sum and a carry. The most commonly used compressors are 3:2, 4:2 and 5:2 [2]. In traditional designs, compressors play a critical role in the accumulation phase. However, conventional exact compressors are power-intensive and complex, especially in FPGA-based implementations due to limited logic resources and/or cascading delays and increased power consumption from logic circuits.

**2.4.1 Conventional 3:2 Compressors (Full Adder)**

A 3:2 compressor is equivalent to a full adder. It takes three input bits and outputs two bits, a sum, the least significant bit of the result, and a carry, the most significant bit of the result.

𝑆 = 𝐴 ⊕ 𝐵 ⊕ 𝐶𝑖𝑛

𝐶𝑜𝑢𝑡 = (𝐴 ⋅ 𝐵) + (𝐵 ⋅ 𝐶𝑖𝑛) + (𝐴 ⋅ 𝐶𝑖𝑛)

This is the simplest compressor and serves as the building block for higher-order compressors.

**2.4.2 Conventional 4:2 Compressors**

A 4:2 compressor takes 4 inputs and produces two output bits and an additional carry-in and carry-out.

𝑆 = 𝐴 ⊕ 𝐵 ⊕ 𝐶 ⊕ 𝐷 ⊕ 𝐶𝑖𝑛

𝐶 = (𝐴 ⋅ 𝐵) + (𝐶 ⋅ 𝐷) + (𝐶𝑖𝑛 ⋅ (𝐴 ⊕ 𝐵 ⊕ 𝐶 ⊕ 𝐷))

This style of compressor is advantageous as it reduces four rows of partial products to two, with a carry propagated to next stage and minimises delay compared to a series of 3:2 compressors. Most

commonly used in high-performance multipliers to speed up partial product reduction especially in Dadda multipliers.

**2.4.3 Conventional 5:2 Compressors**

The 5:2 compressor takes five input bits and produces two output bits, along with two carry bits (one from previous stage and one for the next stage).

𝑆 = 𝐴 ⊕ 𝐵 ⊕ 𝐶 ⊕ 𝐷 ⊕ 𝐸 ⊕ 𝐶𝑖𝑛1 ⊕ 𝐶𝑖𝑛2

𝐶 = 𝑀𝑎𝑗𝑜𝑟𝑖𝑡𝑦 𝑓𝑢𝑛𝑐𝑡𝑖𝑜𝑛 𝑜𝑓 𝑖𝑛𝑝𝑢𝑡𝑠

A 5:2 compressor is particularly efficient for reducing a large number of partial product rows in multipliers, for instance 16x16 or 32x32.

**2.4.4 Approximate m:2 Compressor**

Traditional compressors focus on exact computations, which are not always necessary for error-tolerant applications such as image processing or machine learning. The Approximate m:2 Compressor is designed to aggregate multiple elements in two equal-weight outputs while minimising hardware complexity and power consumption.

The Approximate m:2 Compressor is designed into two output bits, Sum (S) and Carry (C) outputs

represent the cumulative result of m elements with reduced precision. Probability analysis is used to determine the most partial product values are concentrated between 0 and 2, making it feasible to represent them with two outputs.

𝑆 = 𝐴 + 𝐵 + 𝐶 + ⋯ + 𝑚 =

𝐶 = (𝐴 ⋅ (𝐵 + 𝐶 + ⋯ + 𝑚)) + (𝐵 ⋅ (𝐶 + ⋯ + 𝑚)) + ⋯

The use of OR-based logic gates reduces the number of LUTs required, compared to traditional

compressors using XOR and AND gates furthermore reducing propagation delay. Fewer logic gates results in lower power consumption.

**2.5.1 Once-Through Multiplier Architecture**

The primary design objectives of CAM2 are to minimise power consumption, area utilisation and

delay by using simple logic operations, such as OR- gates, instead of more complex compressors in specific stages [2].

The CAM2 is implemented in three stages:

• **Stage 1:** Initial compression of partial products using carry-lagged compressors.

• **Stage 2:** Approximate compression of remaining partial products using OR operations.

• **Stage 3:** Final summation using carry chains to product the final product.

**2.5.2 Hardware Efficiency and Metrics**

CAM2 achieves a power reduction of 57.90% compared to exact multipliers. The use of OR operations significantly reduce the area, leading to a 33.80% reduction in LUT usage. By simplifying logic in Stage 2 and avoiding recursive carry propagation, CAM2 reduces delay by 24.78% [2], with a Mean Relative Error Distance (MRED) of 5.86% and an Error Rate (ER) of 84.50%. CAM2 sacrifices accuracy for greater efficiency, making it suitable for applications where minor inaccuracies are acceptable.

**2.6 Hardware Efficiency and Metrics**

Fuzzy memoization is an approximate computing technique based on instruction memoization here

approximate results are cached and reused instead of recomputing exact results. Both Instruction and Fuzzy memoization store input and output data for a process as an entry is the memo table, and reuse it to reduce execution time by skipping the original process [4].

This is particularly useful in applications where slight inaccuracies in the output are acceptable, such as image processing, machine learning, and signal processing.

**2.6.1 Cache Initialisation**

A cache (or lookup) table is created to store the results of previous computations. Each entry in the cache consists of inputs and outputs. The original input value of vectors used for computation and the output is the result of those inputs. For instance, in a MAC operation:

• **Inputs:** A and B

• **Outputs:** A \* B + previous result

**2.6.2 Fuzzy Matching**

Search the cache for a stored input that is “similar enough” to the new input, based on a predefined

similarity threshold. If a similar input is found (cache hit), reuse the cached result instead of performing the computation. If no similar input is found (cache miss), compute the result, store it in the cache, and use it for future queries [4].

**2.6.3 Similarity Metrics**

Several metrics can be used to determine the similarity between inputs, depending on the application. Hamming distance counts the number of bit positions in which the inputs differ, Euclidean distance measures the geometric distance between two vectors in a multi-dimensional space. Manhattan distance measures the sum of absolute differences between corresponding elements of two vectors and custom thresholding is a user-defined threshold that dictates the maximum allowable threshold difference between inputs.

**2.6.4 Updating the Cache**

When a cache miss occurs, the cache is updated with the new input-output pair. Depending on the cache size and replacement policy, older or less relevant entries may be removed. Common cache replacement policies include:

• **Least Recently Used (LRU):** Removes the least recently accessed entry.

• **First-In-First-Out (FIFO):** Removes the oldest entry.

• **Random Replacement:** Randomly selects and entry to replace.

**2.7 Proposed Novel Approximate Arithmetic Unit Design**

The proposed Approximate Arithmetic Unit Design (AAUD) is a hybrid architecture that combines multiple approximation techniques to achieve significant improvements in power consumption, area, delay, tailored for a MAC unit. The AAUD integrates OR-based approximate adders, approximate multipliers, compressor-based approximations, and fuzzy memoization. This

architecture is designed for error-tolerant applications like fuzzy-filter-based FIR filters used in image processing where small inaccuracies in computations are acceptable.

**2.7.1 Architecture**

The architecture of the AAUD is split into 4 stages. Approximate Partial Product Generation, Partial Product Reduction with Compressor-Based Approximations, Approximate Adders for

Accumulation and Fuzzy Memoization for reuse of results. The detailed AAUD-MAC flow is provided below:

**1. Input Fetch:** Fetch inputs A and B for the MAC operation.

**2. Fuzzy Memoization Lookup:** Search the cache for a similar input pair using Hamming distance.

**3. Cache Hit:** If a similar input is found, reuse the cached result. If not, proceed to the next

step.

**4. Approximate Multiplication:** Compute the product using the Lower-Part OR-based Approximate Multiplier

**5. Partial Product Reduction:** Use Approximate m:2 Compressors to reduce the partial products.

**6. Approximate Addition:** Accumulate the reduced partial products using the Lower-Part OR-based Approximate Adder

**7. Cache Update:** Store the result in the cache for future reuse.

The proposed AAUD-MAC is ideal for a Fuzzy Memoized FIR Filter used in image denoising noisy image pixels. Each pixel is passed through the FIR filter, where the MAC operations are accelerated using AAUD-MAC. Common pixel patterns are cached and reused, reducing redundant computations. The AAUD combines various approximation technique to deliver a highly efficient MAC unit suitable for energy-constrained application. Its hybrid design leverages

approximate arithmetic and fuzzy memoization, achieving significantly improvements in power, area, and delay making it a viable solution for real-time, error-tolerant applications such as image processing.

**3**

**Design Methodology**

**3.1 Overview**

This chapter presents the main technical developments of the project of the Approximate 8-bit Multiply-Accumulate (MAC) unit. It begins with a discussion of the overall architecture, followed by a detailed examination of the approximate components chosen, such as the Dadda multiplier and a Lower Approximate Carry Look-Ahead (CLA) adder. The aim is to balance computational efficiency and accuracy, particularly for applications in image processing where some degree of imprecision can be tolerated. The chapter also explains the reasoning behind using pipelining and fuzzy memoization to optimize performance and efficiency, laying the foundation for the implementation phase that follows.

**3.2 Dadda-Based Multiplier Design**

**3.2.1 Introduction**

Multiplication is a complex process and is the primary cause of time consumption during any operation. Utilizing approximate computing in multipliers can aid in large scale operations [Ankit Gupta]. The multiplier used is based on the Dadda tree reduction technique, chosen for its efficiency in reducing the number of reduction stages compared to Wallace trees. The Dadda multiplier is optimized for speed and area, making it an acceptable choice for hardware implementations where resource constraints are critical. The multiplier design can be split into three phases; partial product generation by bit-by-bit multiplication, partial product accumulation and final summation of partial products.

**3.2.2 Partial Product Generation**

In the proposed design, the first stage of the multiplier generates 64 partial products using AND gates. These products form an 8x8 matrix of bits which will be reduced in the subsequent stages with the use of reduction tree using compressors, discussed further on. This operation is repeated for all 64 combinations of the 8-bit inputs. The summation process has been performed by dividing the partial products into two blocks; MSB (accurate) block and a LSB (approximate) block. The (approximate) LSB block employs OR gates to generate the final product. This dividing methodology has been discussed in many research papers based on numerous factors such as power consumption, delay and accuracy [Ankit Gupta].

**3.2.3 Reduction Tree using Approximate Compressors**

In a multiplier, compressor is a circuit to accumulate and reduce multiple rows of partial products into two rows for the final adder. To accumulate the partial products efficiently, exact and inexact compressors are proposed. The reduction of partial products is performed in multiple levels using a combination of half adders (HA), full adders (FA), and 4:2 compressors. This approach to partial product reduction is a crucial step in the implementation of the Dadda-based multiplier, as it significantly affects the overall speed, area, and power consumption of the multiplier unit. In traditional Wallace multipliers, partial product bits are reduced level by level until only two rows remain for final addition. The Dadda multiplier improves upon this by minimizing the number of required adders at each stage, thus optimizing the number of operations and reducing logic depth. However, to achieve further gains in performance, this design incorporates compressor-based reduction, particularly 4:2 compressors, alongside conventional full and half adders.

Compressors, such as the 4:2 compressors, are well suited for dense accumulation of partial products because they can take four input bits and produce two output bits (sum and carry), thus reducing the vertical height of the partial product matrix more aggressively than full adders. This not only reduces the total number of reduction levels but also shortens the critical path delay, which is key in improving the multiplier's clock speed and throughput.

Additionally, by using compressors instead of a chain of full adders, the design achieves a more balanced load distribution across levels. This has positive implications for power consumption, which is particularly important in embedded systems.

Each stage in the reduction tree is constructed to meet the target column heights prescribed by the Dadda algorithm. Compressors are deployed in locations with higher column heights to maximize their impact, while full and half adders are used in sparser regions of the matrix to avoid unnecessary logic.

**3.2.4 Final Addition**

In the final stage of the Dadda multiplier, once the partial products have been reduced to two rows through successive levels of 4:2 compressors and (3,2) counters (full and half adders), a final addition is necessary to produce the complete 16-bit product. This is a critical stage in any multiplier architecture, as it determines the correctness and timing of the final result.

The two remaining rows represent intermediate results that must be added together using binary addition with full carry propagation. In the implemented design, this final addition is performed using a ripple-carry adder constructed from basic logic gates or structural Verilog. This method ensures exact summation of the bits and complete propagation of carry signals from the least significant to the most significant bits.

This stage is fully accurate, and no approximation is applied. The use of an exact binary adder guarantees that no error is introduced during the final summation, which is essential for applications requiring high numerical precision. While ripple-carry adders are not the fastest option in terms of delay (as the carry must propagate through every bit), they are hardware-efficient and easy to implement, making them a practical choice for moderate-speed designs such as this.

No attempts are made in this design to truncate, estimate, or bypass carry propagation, which are typical features of approximate adders. Instead, the focus remains on maintaining full accuracy of the product, particularly since all preceding reduction logic (e.g., compressors) is also exact. This ensures that the final 16-bit output accurately represents the multiplication of two 8-bit operands, without approximation-induced distortion.

**4**

**Implementation**

**4.1 Overview**

This chapter describes the practical realization of the design methodology presented in Chapter 3. It outlines the hardware description, synthesis flow, and testing procedures used to implement the 8-bit Dadda-based approximate MAC unit. The design was written in Verilog HDL and simulated using Xilinx Simulation Tool, with synthesis carried out using Xilinx Vivado.

**4.2 Module Integration Strategy**

The integration of the Dadda-based multiplier and the approximate LowerApproxCLA adder was carried out via a systematic, bottom-up approach. This methodology ensured that individual modules were fully tested and verified before being integrated into higher-level structures. The primary goal of the integration strategy was to balance accuracy, area, and performance while maintaining modularity for ease of testing, bug fixing and future design changes.

**4.2.1 Hierarchal Design**

The overall design was structured hierarchically, allowing individual functional blocks to be developed, tested and validated in isolation. The core of the arithmetic unit such as the approximate multiplier and adder were treated as independent elements, each consisting of further submodules such as partial product generators, 4:2 compressors, and carry-lookahead logic.

Individual modules were implemented using synthesizable Verilog and verified by dedicated testbenches. This modularity allowed the reuse of components especially in the multiplier, where repetitive structures such as compressors were extensively used.

**4.2.2 Pipelining Considerations**

In order to improve performance, the design incorporated pipelining between key computational stages. Pipelining was introduced after partial product generation and again following the addition stage to improve throughput to allow for higher clock frequencies. The pipeline registers were utilized to ensure correct timing without introducing data hazards or synchronization issues.

The utilization of pipeline stages was balanced with the requirement to keep latency low, which is particularly important for real-time processing such as in image filtering applications. Simulation results confirmed that the added stages did not significantly degrade overall latency, while offering a considerable increase in operating frequency.

**4.2.3 Integration Strategy**

The integration strategy was also modelled differently by the deliberate use of approximation within the datapath. During the multiplier stage, approximate 4:2 compressors were utilized in the reduction tree to speed up computation and reduce area. While traditional compressors aim for precise carry propagation, the approximate versions used selectively simplified the carry logic in less significant bit positions.

This approach was completed by the use of an approximate adder, which replaces the conventional carry-lookahead logic in the LSB region with a simplified logic block that reduces gate count and delay. The MSB region, preserves exact logic to avoid significant deviation from true results.

The architecture was designed in such a way that the flow of approximate data through the system could be isolated and analyzed. This helped in quantifying the impact of approximation on both error metrics and system-level outputs.

**5**

**Testing and Evaluation**

**5.1 Introduction**

This chapter presents the testing methodology, evaluation metrics, and implementation challenges encountered during the development of the proposed 8-bit Approximate MAC unit. Testing played a vital role in verifying the functionality, efficiency, and accuracy of the modules designed using approximate arithmetic principles, such as the use of approximate compressors and fuzzy memoization.

Throughout the implementation lifecycle, rigorous simulation-based testing was employed alongside hardware synthesis trials on FPGA. These efforts ensured the developed modules met performance and area efficiency requirements. To track the iterative development process and capture real-time debugging efforts, version control was employed using Git. A detailed commit history serves as a chronological log of progress, refinements and issues encountered, forming a valuable resource to retrospectively analyze implementation challenges and the effectiveness of testing strategies.

**5.2 Testing Methodology**

The testing strategy adopted throughout the development of the approximate 8-bit MAC units was multi-tiered, combining simulation-based verification with automated Python functional validation. This approach ensured both correctness and performance consistency across evolving stages of approximation and hardware integration.

**5.2.1 Simulation Testing in Verilog**

Initial testing was conducted through custom testbenches written in Verilog. These testbenches were tailored to the specific module under development, including the Lower-Part Approximate OR-based Carry Lookahead Adder, the Parallel Prefix Approximate Multiplier (PPAM), and the complete pipelined MAC architecture. These simulations served as the primary means to confirm correct logic behavior, timing and synthesis compatibility.

**5.2.2 Python-Based Functional Testing**

To complement hardware-level simulations, a Python script was written to conduct automated validation. This script compared outputs from the Verilog simulation with expected values from an accurate software model. This form of golden-model testing provided a robust mechanism for checking the error introduced by approximation, as well as ensuring the correctness after implementation changes.

The test vectors generated and processed through the Python script were extensive, including random and edge-case data. This helped identify failure modes involving a case of partial integration of the PPAM, causing the results to not align with expectations. Later, a validation implementation involved more rigorous check for the CLA adder using this python tool, verifying both functionality and error characteristics in a controlled environment.

**5.2.3 Integration Testing with Pipelined MAC**

Once both the approximate multiplier and adder components were verified, they were integrated into the pipeline MAC architecture. This integration was not without challenges, as documented, a testbench-level modification was required to accommodate larger input vector lengths, pushing the design towards a more realistic operational scale.

**5.3 Implementation Challenges and Resolutions**

The development of the approximate MAC unit involved a wide range of challenges across design correctness, timing behavior, synthesis issues, and architectural calibration. These challenges were identified and documented through Git version control, enabling traceable resolution paths. Below is a comprehensive breakdown of key issues encountered during the design and implementation phases, and how they were addressed.

**5.3.1 Approximate Module Accuracy and Behavioral Bugs**

One of the earlier challenges was ensuring the approximate modules – particularly the Lower-Part Approximate Or-based Carry Lookahead Adder and the Parallel Prefix Approximate Multiplier functioned correctly within acceptable error margins. A correction within the Carry Lookahead Adder was documented, where behavioral inconsistencies were identified during simulation testing. This required a reassessment of the internal logic structure and a modification of the OR-based carry approximation path to ensure accurate summation in the lower bits.

The PPAM, implemented as part of the approximate Dadda multiplier, also posed a unique challenge in balancing hardware reduction steps with accuracy. During initial testing, it was noted that the multiplier failed to match expected outputs, leading to an enhancement in the Python validation script that enabled clear statistical analysis of the mismatch rate.

**5.2.2 Functional Integration and Testbench Constraints**

During module integration, the testbench was updated to accommodate longer simulation cycles and larger input datasets. Previously, the testbench was limited to test lengths that had rare failure conditions, which only emerged with large-scale testing. This change enabled more robust debugging of edge cases and better evaluation of accumulated approximation error over continuous operations.

Furthermore, the challenges in combining approximate modules into a pipelined structure required new signal alignment and reset logic. Getting the approximate multiplier and adder to synchronize effectively within the pipeline’s latency was particularly demanding. The solution involved fine-tuning intermediate register stages and verifying alignment across clock domains.

**5.2.3** **Fuzzy Memoization Calibration**

The fuzzy memoization unit presented a unique challenge. Although initial integration was successful, it was evident that memoization required calibration to ensure that repeated input patterns would consistently trigger valid cached outputs without corrupting precision. This was particularly important as memoization errors could propagate through the pipeline. The calibration effort required experimentation. The design was left modular and parametric to allow further refinement post-evaluation.

**5.4 Evaluation of Accuracy and Performance**