Bar-Ilan University

Faculty of Engineering

Approximate Computation

David Darvish
207816711

Eliran Faridi
318869047

Academic Advisors:
Dr. Itamar Levi
Mr. Oren Ganon

October 2024

Contents

Abstract

This project investigates the application of approximate computing in implementing the Discrete Fourier Transform (DFT) to reduce computational complexity through controlled accuracy loss. We specifically focused on Fast Fourier Transform (FFT) methods, namely Decimation in Time (DIT) and Decimation in Frequency (DIF), which are widely recognized for their efficiency in computing the DFT and are crucial for various signal processing applications.

Using the TIE language, we designed an approximate multiplier as a core component of the FFT. This multiplier is essential for performing the arithmetic operations required in FFT calculations. By implementing it as an approximate component, we evaluated its impact on overall computational performance.

While we successfully developed the approximate multiplier, integrating this component into a complete FFT implementation remains a direction for future work. Such integration is vital for fully realizing the advantages of approximation in FFT computations, potentially leading to enhanced efficiency in digital signal processing systems.

Introduction

Approximate computing is an innovative approach in digital signal processing that enhances computational efficiency by tolerating a degree of inaccuracy, thus conserving resources such as power, hardware space, and processing time.

In this project, we aimed to implement an approximate multiplier that performs calculations that are close to, but not exact. We achieved this by incorporating approximate adders and compressors as fundamental components of the multiplier, leading to savings in hardware space and processing resources.

The approximate multiplier plays a crucial role in the implementation of the Fast Fourier Transform (FFT) algorithm, a fundamental tool for transforming signals from the time domain to the frequency domain. Given that the FFT requires numerous multiplications, utilizing the approximate multiplier significantly improves efficiency, particularly in applications where absolute precision is less critical, such as image processing, video analysis, and machine learning.

Theoretical Background

Approximate computing (AC) represents a transformative approach in the design of circuits and systems, focusing on energy efficiency by allowing for reduced accuracy in computational results where permissible. This paradigm is particularly relevant given the ongoing miniaturization of electronic devices, which demands designs that are both fault-tolerant and resilient to variations in manufacturing processes at the nanoscale.

Traditional fault-tolerance methods, which rely heavily on redundancy in hardware, time, or information, often lead to increased energy consumption—a trade-off that has become a significant challenge in maintaining reliability while enhancing energy efficiency. AC adopts a perspective where errors are not just inevitable but integral to the computational process, especially in systems where some level of error can be tolerated without critical impact on the final outcome.

This philosophy is particularly applicable in fields such as multimedia processing, data mining, and machine learning, where a slight loss in precision can be exchanged for substantial gains in energy efficiency and processing speed.

AC encompasses a broad spectrum of research areas, from circuit design at the transistor and logic level to programming languages. It includes innovations in arithmetic circuit design, memory systems like SRAM, DRAM, and non-volatile storage, and various processor architectures, including neural networks, general-purpose processors, and reconfigurable systems such as GPUs and FPGAs.

The practical applications of AC are extensive, touching on image and signal processing, classification, recognition, and machine learning, illustrating its broad potential impact. Despite these advancements, AC faces significant challenges and prospects, particularly in how it integrates with brain-inspired computing, which has gained traction as a method for achieving energy-efficient computing through device parallelism and circuit adaptability.

The central questions involve determining the extent to which AC can be effectively implemented within larger systems to meet the diverse quality requirements of error-tolerant applications and to ascertain whether the performance and energy efficiencies observed at the component level can be sustained when scaled up to full systems.

These questions highlight the ongoing need to balance error tolerance with quality assurance to make AC a practical and efficient approach for future computing needs.

1 Discrete Fourier Transform (DFT)

Discrete Fourier Transform (DFT) Overview

The Discrete Fourier Transform (DFT) is a fundamental tool in the field of digital signal processing, allowing the conversion of signals from the time domain to the frequency domain. This transformation is crucial because it reveals the frequency components of a time-domain signal, which is essential for many applications in signal analysis, compression, and filtering.

Applications of DFT

DFT is extensively used in:

- Audio engineering, for effects such as equalization and pitch correction.
- Image processing, for filtering and image compression.
- Telecommunications, where it forms the basis for techniques like OFDM (Orthogonal Frequency-Division Multiplexing) used in modern communication systems like WiFi and LTE.

DFT's Relation to Approximate Computing

Approximate computing is a computational technique that seeks to improve the efficiency of computing systems by allowing for controlled errors in calculations, which can significantly reduce the power consumption and increase the speed of computations. This approach is particularly useful in applications where an exact solution is not necessary, such as multimedia processing, machine learning, and large-scale data analysis.

Connection with DFT

- **Reducing Computational Complexity:** DFT, specifically through its fast computing algorithm FFT (Fast Fourier Transform), is computationally demanding. Approximate computing can be used to reduce the precision of the computation, thereby decreasing the computational load and power consumption at the expense of a controlled degradation in signal quality.
- **Error-Tolerant Applications:** Many applications that use DFT can tolerate some level of inaccuracies. For example, in image and audio

processing, minor losses in high-frequency components (which often represent less perceptible details) might not significantly affect the user experience but can simplify the computation.

- **Selective Accuracy:** In approximate computing, the accuracy of the computation can be tailored to different parts of the signal. For example, more significant frequency components of a signal can be computed with higher accuracy compared to less significant ones.

- **Energy-Efficient Implementations:** For battery-operated devices where power consumption is a critical concern, applying approximate computing techniques to DFT can lead to more energy-efficient implementations. This is especially relevant in mobile devices and embedded systems where extended battery life is crucial.

Research and Development

Recent research in the field of approximate computing with relation to DFT includes developing algorithms that selectively reduce the precision of mathematical operations or simplify the algorithms themselves, achieving a trade-off between accuracy and computational resources. These approaches aim to optimize the performance of systems where slight inaccuracies are acceptable for gaining substantial improvements in efficiency.

Conclusion

The interplay between DFT and approximate computing represents a significant opportunity for optimizing signal processing tasks. By allowing for controlled inaccuracies, approximate computing techniques can enhance the efficiency of DFT computations, particularly in applications where exact precision is not paramount, thereby contributing to the advancement of both signal processing and computing technology.

2 Fast Fourier Transform (FFT) algorithms

Definition of the Fourier Transform

A fast Fourier transform (FFT) is any fast algorithm for computing the DFT. The development of FFT algorithms had a tremendous impact on computational aspects of signal processing and applied science. The DFT of an N-point signal

$$\{x[n], 0 \leq n \leq N-1\}$$

is defined as:

$$X[k] = \sum_{n=0}^{N-1} x[n] W_N^{-nk}, \quad 0 \leq k \leq N-1$$

where

$$W_N = e^{j\frac{2\pi}{N}} = \cos\left(\frac{2\pi}{N}\right) + j\sin\left(\frac{2\pi}{N}\right)$$

is the principal N-th root of unity.

Direct DFT Computation

Direct computation of $X[k]$ for $0 \leq k \leq N-1$ requires:

$(N-1)^2$ complex multiplications and $N(N-1)$ complex additions.

Radix-2 FFT

The radix-2 FFT algorithms are used for data vectors of lengths $N = 2^k$. They proceed by dividing the DFT into two DFTs of length $N/2$ each, and iterating. There are several types of radix-2 FFT algorithms, the most common being the decimation-in-time (DIT) and the decimation-in-frequency (DIF). This terminology will become clear in the next sections.

The development of the FFT will call on two properties of W_N.

The first property is:

$$W_N^2 = W_{N/2}$$

which is derived as

$$W_N^2 = e^{-j\frac{2\pi}{N/2}} = W_{N/2}$$

More generally, we have:

$$W_N^{2nk} = W_{N/2}^{nk}$$

The second property is:

$$W_N^{k+N/2} = -W_N^k$$

Thus,

$$W_N^{k+N/2} = e^{-j\frac{2\pi(k+N/2)}{N}} = e^{-j\frac{2\pi k}{N}}e^{-j\pi} = -W_N^k$$

$$W_N^{k+N/2} = \cdots = -W_N^k$$

3 Decimation-In-Time (DIT) FFT

Definition

Consider an N-point signal $x[n]$ of even length.
The derivation of the DIT radix-2 FFT begins by splitting the sum into two parts — one part for the even-indexed values $x[2n]$ and one part for the odd-indexed values $x[2n+1]$.
Define two $N/2$-point signals $x_0[n]$ and $x_1[n]$ as:

$$x_0[n] = x[2n] \quad x_1[n] = x[2n+1], \quad 0 \leq n \leq \frac{N}{2} - 1$$

The DFT of the N-point signal $x[n]$ can be written as:

$$X[k] = \sum_{n=0,\text{ even}}^{N-1} x[n]W_N^{-nk} + \sum_{n=0,\text{ odd}}^{N-1} x[n]W_N^{-nk}$$

which can be written as

$$\begin{aligned}
X[k] &= \sum_{n=0}^{N/2-1} x[2n]W_N^{-2nk)} + \sum_{n=0}^{N/2-1} x[2n+1]W_N^{-(2n+1)k} \\
&= \sum_{n=0}^{N/2-1} x_0[n]W_N^{-2nk} + \sum_{n=0}^{N/2-1} x_1[n]W_N^{-(2n+1)k} \\
&= \sum_{n=0}^{N/2-1} x_0[n]W_N^{-2nk} + W_N^{-k} \sum_{n=0}^{N/2-1} x_1[n]W_N^{-2nk} \\
&= \sum_{n=0}^{N/2-1} x_0[n]W_{N/2}^{-nk} + W_N^{-k} \sum_{n=0}^{N/2-1} x_1[n]W_{N/2}^{-nk}
\end{aligned}$$

We can also write it as:

$$X[k] = X_0[k] + W_N^{-k} X_1[k]$$

Such that:

$$X_0[k] = DFT_{\frac{N}{2}}\{x_0[n]\} = \sum_{n=0}^{N/2-1} x_0[n]W_{\frac{N}{2}}^{-nk}$$

$$X_1[k] = DFT_{\frac{N}{2}}\{x_1[n]\} = \sum_{n=0}^{N/2-1} x_1[n]W_{\frac{N}{2}}^{-nk}$$

Thus:

$$X[k] = X_0[k] + W_N^{-k} X_1[k], \quad \text{for } 0 \le k \le \frac{N}{2} - 1$$

$$X\left[k + \frac{N}{2}\right] = X_0[k] - W_N^{-k} X_1[k], \quad \text{for } 0 \le k \le \frac{N}{2} - 1$$

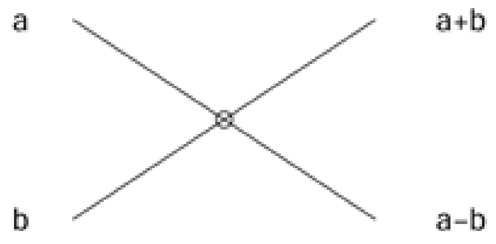
The multipliers W_N^k are known as twiddle factors.

Full Flowgraph

If $N/2$ can be further divided by 2, then this same procedure can be used to calculate the $N/2$-point DFTs. It is useful to illustrate the radix-2 FFT algorithm with a flowgraph, as developed here.

The flowgraph for the sum and difference operation is called the *butterfly*. This unit will be used as a shorthand notation for the sum and difference, to simplify the flowgraphs for the FFT.

Figure 1: Unit Butterfly Diagram



The decimation-in-time FFT for an 8-point DFT consists of:

1. 2 4-point DFT computations
2. the twiddle factors
3. butterflies

The decomposition of the N-point DFT into two $N/2$-point DFTs can be repeated (provided N is divisible by 4).

The 2-point DFT is simply a butterfly (sum/difference), so the final FFT has 3 stages.

Figure 2: 8-point DIT FFT algorithm diagram

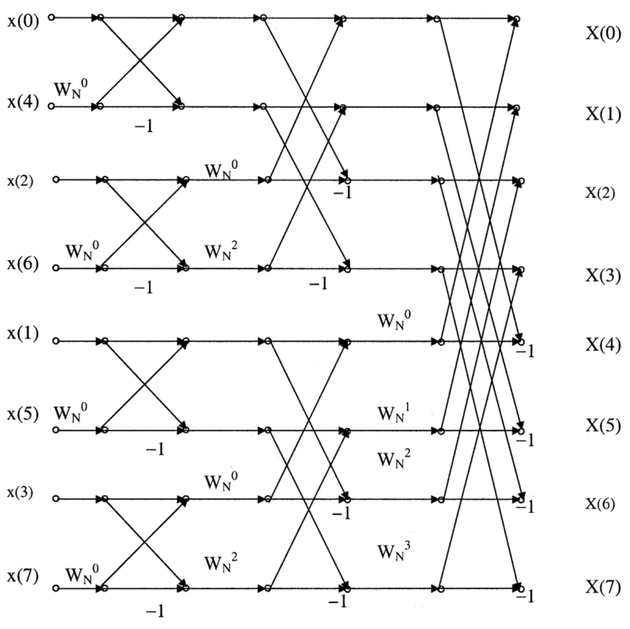
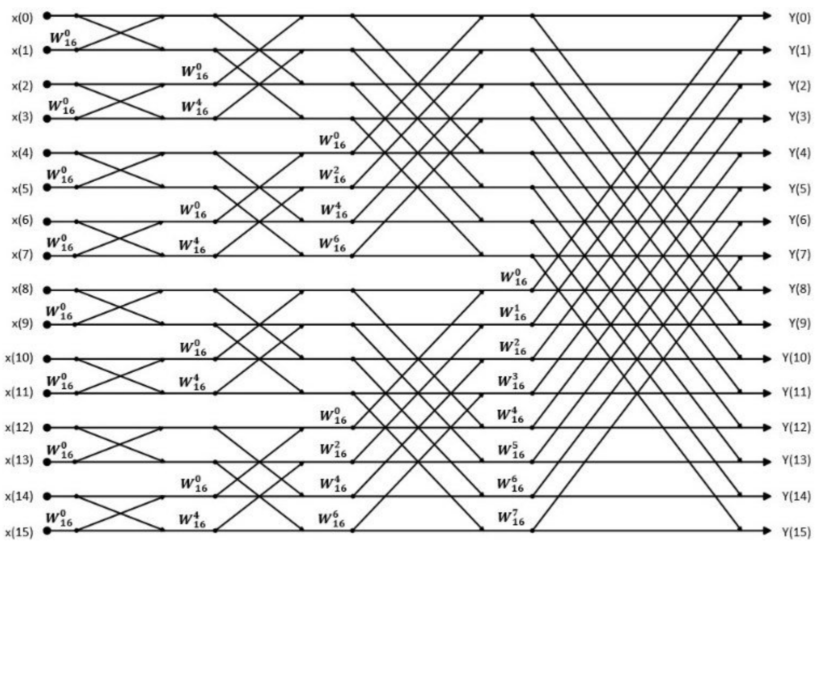


Figure 3: 16-point DIT FFT algorithm diagram



4 Comparison between DIT (Decimation in Time) and DIF (Decimation in Frequency)

Basic Structure

DIT:

In DIT, the input sequence undergoes a bit-reversed reordering process before any computation begins. The core operation starts by first applying multiplications (twiddle factors) and then performing additions. This sequence of operations can be useful for approximation techniques, such as trimming precision during the multiplication phase, allowing early approximations in the pipeline.

DIF:

In contrast, DIF operates with inputs in their natural order, and the operations start with additions followed by multiplications. The natural input order in DIF reduces the pre-processing burden, but approximations are best applied post-addition, where intermediate results can be selectively downscaled or approximated, especially in resource-constrained environments.

Data Flow and Impact on Approximate Techniques

DIT:

Since DIT requires input reordering (bit-reversal), it introduces an additional pre-processing step. This step can become a bottleneck in approximation strategies that rely on data locality and memory efficiency. However, because DIT first performs multiplications (which are often more computationally expensive), it provides opportunities for multiplier approximation techniques like truncated multipliers or precision scaling. These can reduce power consumption and hardware complexity.

DIF:

In DIF, the output is generated in bit-reversed order, which can be advantageous in approximation strategies where output fidelity is less critical. Since the algorithm starts with additions, it is easier to introduce approximations in the addition stage by using reduced-precision adders or eliminating low-order bits in intermediate sums. This makes DIF a good candidate for approximate computing where error-tolerance is permissible at the output stage.

Computational Complexity and Latency

Both DIT and DIF FFT algorithms share the same computational complexity of $O(N \log N)$, but their data flow and operation order provide different trade-offs for approximation.

DIT:

Multipliers are applied early, which can increase the algorithm's sensitivity to approximation errors in the early stages. However, the early application of multiplications means that the butterfly structure (core FFT operation) can benefit from fewer intermediate calculations, reducing the overall processing time if approximations are introduced effectively.

DIF:

Additions come first, which are less error-prone than multiplications when approximations are applied. Approximation techniques can focus on reducing precision after additions but before multiplications, where errors can be more easily absorbed without significant loss in output quality. This allows for latency reduction in applications where precision is traded for speed, especially in real-time processing systems.

Approximate Hardware Considerations

DIT:

Due to its reliance on multiplications upfront, DIT can benefit from hardware techniques like approximate multipliers that reduce power and area by eliminating partial products or using truncated bit widths. For instance, Baugh-Wooley multipliers can be adapted to introduce approximation schemes where less critical bits are ignored or approximated, making DIT well-suited for applications focused on power efficiency in high-throughput environments.

DIF:

With additions coming first, DIF is more naturally aligned with adder approximation techniques like Carry-Select Adders (CSAs) or Ripple Carry Adders (RCAs), which can be simplified by omitting low-order bits. This makes DIF more attractive in scenarios where fast, approximate addition is prioritized over precise multiplication. The trade-offs in hardware design can focus on reducing the overall footprint of adders while maintaining an acceptable level of accuracy in the final output.

Error Propagation and Control

DIT:

In DIT, since multiplications occur first, any approximation errors in the early stages can propagate through the subsequent additions, potentially compounding the error. Thus, error control mechanisms need to be more sophisticated, focusing on minimizing approximation in the early stages while tolerating more error as the algorithm progresses.

DIF:

In DIF, errors introduced in the addition phase are less likely to propagate as severely due to the later application of multiplications. This makes DIF a more error-resilient choice in applications where small errors in early additions are acceptable. Approximations applied to the output stage can be more aggressive without significantly impacting overall performance.

Application in Approximate Computing

DIT:

Best suited for applications where the primary concern is optimizing multiplication-heavy operations, especially in systems that can tolerate early-stage approximations but need precise addition in later stages.

DIF:

Suitable for scenarios where initial computations (additions) can be heavily approximated without significant degradation in output quality, making it ideal for error-tolerant systems like image or audio compression, where small inaccuracies are imperceptible.

Conclusion

For approximate computing applications, the choice between DIT and DIF depends on where and how much approximation can be introduced without significantly impacting system performance. DIT's structure lends itself to multiplier approximation, while DIF is better suited for adder approximations. Both can be used to reduce power consumption, hardware complexity, and latency in approximate computing systems, depending on the specific trade-offs required by the application.

5 Literature review

"Accelerating Convolutional Neural Networks With FFT on Embedded Hardware"

The article explores how approximate computing techniques can be applied to accelerate Convolutional Neural Networks (CNNs) on embedded hardware. The main focus is on using Fast Fourier Transform (FFT)-based methods to reduce the computational and memory overhead of convolution operations, which are the most resource-intensive part of CNNs.

Key Contributions to Approximate Computing:

1. FFT-Based Approximation in Convolution:

- Traditional convolution (Direct-Conv) is computationally expensive, especially for large CNNs. By switching to FFT-based convolution (FFT-Conv), the article demonstrates that convolution in the spatial domain can be approximated by element-wise multiplication in the frequency domain, which is computationally less expensive.

- FFT-based convolution reduces the order of computational complexity from $O(N^2K^2)$ in Direct-Conv to $O(N^2 \log N)$, where N is the image size and K is the filter size. This represents a significant reduction in hardware processing requirements.

2. FFT Overlap-and-Add (FFT-OVA-Conv) Method:

- One of the challenges with FFT-Conv is the mismatch in data and filter sizes, which results in high intermediate memory overhead. The FFT-OVA-Conv method solves this by breaking down data into smaller segments that can be processed independently, reducing memory demands.

- This method is especially important for embedded systems with limited on-chip cache and memory resources, making it highly efficient for hardware implementations like FPGA or many-core processors.

- Approximate computing in this context focuses on balancing the reduction in memory and computational complexity while accepting minor accuracy trade-offs, which is tolerable in many CNN applications like image processing.

3. **Hardware Implementations of Approximate FFT Convolutions:**
    - The study evaluates approximate techniques on platforms like PENC, ARM Cortex-A53, NVIDIA Jetson TX1, and SPARCNet (FPGA-based CNN accelerator).
    - On PENC, FFT-OVA-Conv offers a 2.9× speedup and 6.8× energy efficiency improvement over Direct-Conv, crucial for low-power embedded systems.
    - On ARM Cortex-A53, it delivers a 3.36× speedup and 2.72× higher throughput, highlighting its advantage for low-power CPUs.
    - SPARCNet on a Zynq FPGA achieves a 42 ms runtime and 137 mJ energy use, showing the benefits of hardware-accelerated FFT-based CNNs.

4. **Trade-Offs and Hardware Design Considerations:**
    - Approximate computing methods like FFT-based convolution introduce a trade-off between computational efficiency and accuracy. The paper shows that while FFT-OVA-Conv introduces some approximation errors, it maintains sufficient accuracy for tasks like image classification, with only minimal degradation in output quality (less than 1 dB PSNR drop).
    - For hardware engineers, this trade-off translates into significant reductions in power consumption, area, and execution time. Approximate computing allows for more efficient use of hardware resources, especially in FPGA designs or many-core architectures where minimizing data movement and power consumption is critical.

Conclusion:

The use of approximate FFT-based convolution techniques such as FFT-Conv and FFT-OVA-Conv enables significant performance improvements for CNNs on embedded hardware. By reducing computational complexity and memory overhead, these techniques offer a practical way to implement

CNNs in resource-constrained environments like FPGAs, low-power CPUs, and GPUs. For hardware engineers, the adoption of approximate computing techniques in CNN accelerators presents a path to achieving high efficiency in real-time, low-power applications, such as in autonomous vehicles or IoT devices.

"Accurate and Compact Approximate 4:2 Compressors with GDI Structure"

The 4:2 compressor is a digital logic component designed to compress four input bits into two output bits. In the context of approximate computing, it plays a crucial role in reducing the complexity and power consumption of multiplication operations, especially in error-tolerant applications like multimedia and image processing.

Key Features of the 4:2 Compressor:

1. **Input and Output:** The 4:2 compressor takes four input bits (A, B, C, and D) and produces two main outputs: the sum and carry. It also receives a carry-in (Cin) and generates a carry-out (Cout).
2. **Approximate Nature:** In this paper, the authors propose multiple approximate versions of the 4:2 compressor. These designs intentionally introduce small errors in less significant input combinations, reducing power consumption, area, and delay while maintaining high accuracy for most operations.
3. **Gate Diffusion Input (GDI) Structure:** The use of the GDI structure significantly reduces the number of transistors required for the compressor, leading to a more compact design with improved performance in terms of power and area efficiency.

Contribution to Approximate Multiplication:

The 4:2 compressors are utilized in the partial product reduction stage of a Dadda multiplier. By replacing exact compressors with approximate ones, the overall multiplier becomes faster and more energy-efficient, while still delivering results that are accurate enough for applications like image processing, where small errors are not perceptible to the human eye.

In approximate multiplication:

- **Reduction in Transistor Count:** This leads to smaller circuits and lower power consumption.
- **Increased Speed:** The approximate compressors reduce the number of computations needed, which decreases latency.

- **Accuracy Control:** The design includes options for balancing between accuracy and resource savings, allowing the user to choose different compressor configurations depending on the specific application.

In summary, the 4:2 approximate compressors are designed to optimize performance for multiplication by reducing complexity, power consumption, and delay, with a tradeoff of minimal accuracy loss that is acceptable in applications like image processing.

”Design of an Approximate FFT Processor Based on Approximate Complex Multipliers”

The article presents an innovative approach to improving the efficiency of Fast Fourier Transform (FFT) processors by employing approximate computing techniques.

FFT is a critical operation in digital signal processing (DSP), communications, and related fields, but it is resource-intensive, particularly in its use of complex multiplications. To address this, the authors propose the use of approximate multipliers to reduce power consumption, area, and delay, with only minor accuracy trade-offs.

Main Contributions:

1. **Approximate Radix-4 Booth Multipliers:**
    - Four different levels of approximation are applied to Radix-4 Booth multipliers, targeting the low-significant bits of the operands. By approximating these bits, the hardware complexity and power consumption are reduced, while the high-significant bits remain accurate to preserve overall performance.
    - These approximate multipliers are incorporated into the FFT’s complex multiplication stages, a key operation in FFT algorithms.
2. **Pipeline and Parallel FFT Architectures:**
    - The proposed approximate multipliers are implemented in both pipeline and parallel FFT architectures. These two structures differ in their approach to processing input data, but both benefit from the approximate multiplier designs in terms of reduced resource requirements and power consumption.
    - In the pipeline FFT architecture, power consumption is reduced by up to 69.9%, and the delay is reduced by 45.7%. In the parallel FFT, the reduction in Look-Up Tables (LUTs) reaches 29.1%.
3. **Error Tolerance:**
    - Approximate computing inherently introduces a level of error, but the authors show that the accuracy loss is minimal, with the Peak Signal-to-Noise Ratio (PSNR) dropping by less than 1 dB.

- This level of error is acceptable for many error-tolerant applications, such as multimedia, communication systems, and machine learning.

4. **Hardware Efficiency:**

   - The approximate FFT designs result in significant savings in hardware resources.
   - For example, in the 256-point FFT design, the use of approximate multipliers reduces the number of LUTs by up to 20.3% compared to exact FFT designs.
   - This makes the approximate FFT processor highly suitable for low-power and high-performance applications, such as in portable devices or systems where energy efficiency is critical.

Conclusion:

This article demonstrates how approximate computing can effectively reduce the resource demands of FFT processors, making them more efficient for modern applications. By using approximate Radix-4 Booth multipliers, the design achieves substantial improvements in power consumption, area, and performance, with only a slight accuracy loss. This work highlights the potential of approximate computing in enabling energy-efficient and high-performance solutions for complex mathematical operations like FFT, which are vital in many signal processing and communication applications.

"Rounding Technique Analysis for Power-Area & Energy Efficient Approximate Multiplier Design"

The article delves into the hardware implementation of Approximate Computing for multiplier designs, specifically using a rounding technique that enhances power, area, and energy efficiency. Approximate computing is critical in hardware design, especially for error-resilient applications like image processing, machine learning, and signal processing, where slight inaccuracies in computation can be tolerated for gains in efficiency.

Approximate Multiplier Design:

In traditional multipliers, each multiplication involves generating and reducing $N \times N$ partial products, where N is the bit width of the inputs. This can be resource-intensive in terms of area, power, and delay. Approximate computing tackles this problem by trading off accuracy for gains in these areas, which is beneficial for embedded and low-power hardware systems.

Rounding-Based Approach:

The core innovation in the paper is the rounding technique, where input data is pre-processed by rounding lower-significance bits before multiplication. This approach reduces the number of active partial products, cutting down on the complexity of the reduction process. By not considering the inactive partial products (which are zeros), the design can compress and compute only the relevant rows, resulting in significant savings in hardware resources.

1. **Partial Product Reduction:** The method leverages fewer compressor stages to reduce the partial products. Conventional designs use full adders and half adders to compress the products, while the proposed method uses a combination of approximate and accurate compressors, including OR gates instead of full adders in some stages. This substitution significantly reduces area and energy consumption.

2. **Impact on Hardware Metrics:**

    - **Power Consumption:** By reducing the number of active partial products and simplifying the reduction stages, the design achieves power savings of up to 65% compared to traditional multipliers like the DRUM multiplier.

- **Area Efficiency:** Area reduction is a major advantage, especially for larger bit-widths (e.g., 32-bit), where the proposed design cuts down area usage by up to 85% by eliminating unnecessary computations.
- **Delay:** The design improves delay by reducing the time-consuming process of generating and reducing partial products. The study shows up to 60% delay reduction compared to traditional methods for both 16-bit and 32-bit inputs.

3. **Trade-offs and Configurability:** Approximate computing in hardware requires careful balancing between the accuracy loss and the efficiency gains. The rounding technique allows for configurable accuracy by adjusting how aggressively lower-significance bits are rounded. This flexibility enables the design to meet different application needs, from high-speed, low-power embedded systems to more accuracy-demanding environments.

Practical Applications:

This rounding-based approximate multiplier is particularly useful for hardware engineers working on:

- **Low-power embedded systems:** Saves power and area, perfect for battery-operated devices.
- **FPGA-based systems:** The reduction in logic complexity translates to fewer resources being utilized on an FPGA, allowing for more efficient implementations of high-throughput systems such as CNN accelerators.
- **ASIC designs:** Custom silicon designs that prioritize power efficiency and die area can benefit greatly from the hardware simplifications introduced by this method, making it suitable for industrial applications like real-time image or signal processing.

Conclusion:

In conclusion, the use of approximate computing with a rounding technique in multipliers presents significant improvements in power, area, and delay, which are critical for hardware engineers designing low-power, high-efficiency systems. The flexibility in adjusting the accuracy based on application demands makes this technique highly adaptable for various computing environments.

6 Approximate Multiplier

In this project, we implement an approximate multiplier that multiplies two 16-bit numbers. The primary objective of this project is to explore approximate computing techniques to achieve significant reductions in power consumption, area and processing speed.

Initially, we will construct an approximate multiplier designed for multiplying two 8-bit numbers. This foundational design will allow us to explore the principles and techniques of approximate multiplication on a smaller scale. Once we have successfully implemented the 8-bit multiplier, we will proceed to expand our design to create a 16-bit by 16-bit approximate multiplier.

The expansion will be achieved by integrating four 8×8 multiplier components, effectively combining them to handle larger bit-width inputs. This modular approach not only facilitates scalability but also allows us to leverage the performance characteristics of the 8-bit multipliers. We will provide a detailed explanation of this expansion process in the following sections.

6.1 Implementation using the TIE language

We reviewed extensive literature and researched various approaches to approximate multipliers. This exploration led us to the intriguing concept of the approximate compressor, which is discussed in several academic papers we consulted.

In this project, we employed the TIE hardware description language to construct components specifically designed to support approximate computation. Our approach involved a combination of precise and approximate components to meet the computational requirements while optimizing for performance and efficiency.

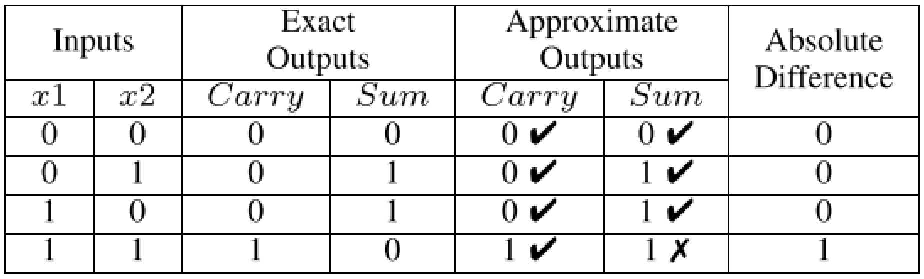
We drew upon the foundational knowledge gained from previous courses to design precise components such as the half adder, full adder, and the 4:2 compressor. These components ensured the accuracy needed in certain computations.

On the other hand, to achieve approximation where allowed, we incorporated components that operate with reduced accuracy. This was done by implementing approximate versions of the half adder, full adder and 4:2 compressor, guided by truth tables obtained from academic papers we reviewed. These sources provided valuable insights into how approximation could be efficiently applied without significantly compromising overall performance.

6.2 Exact and Approximate components

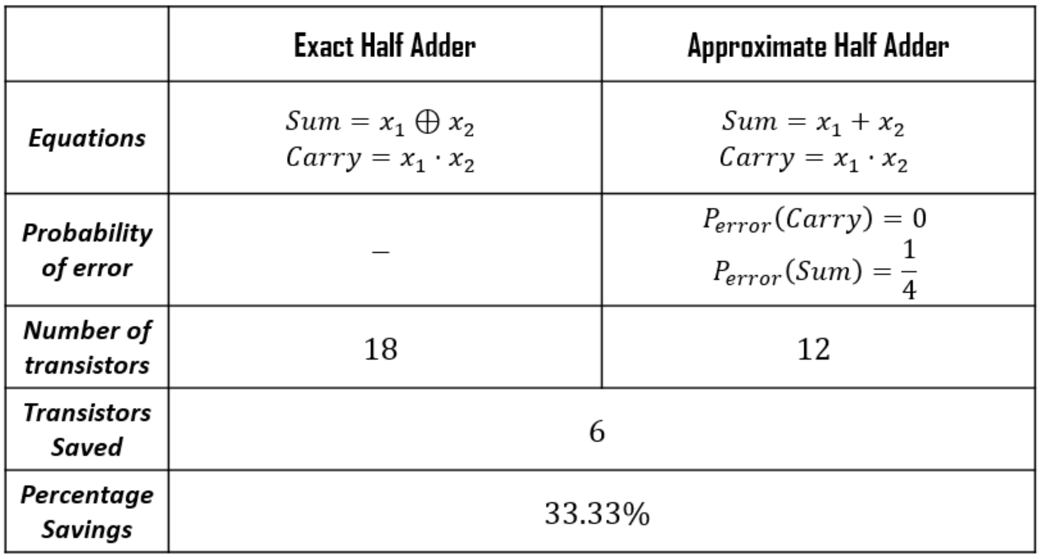
In this section, we will present the key components we developed and employed in the design of our approximate multiplier [2]. Each component plays a critical role in achieving the desired balance between performance and accuracy, while also optimizing resource utilization. We will provide a detailed overview of the design decisions, architectural choices, and techniques applied to create an efficient and effective approximate multiplication unit.

6.2.1 Exact and Approximate half adder



Inputs		Exact Outputs		Approximate Outputs		Absolute Difference
$x1$	$x2$	$Carry$	Sum	$Carry$	Sum	
0	0	0	0	0 ✔	0 ✔	0
0	1	0	1	0 ✔	1 ✔	0
1	0	0	1	0 ✔	1 ✔	0
1	1	1	0	1 ✔	1 ✗	1

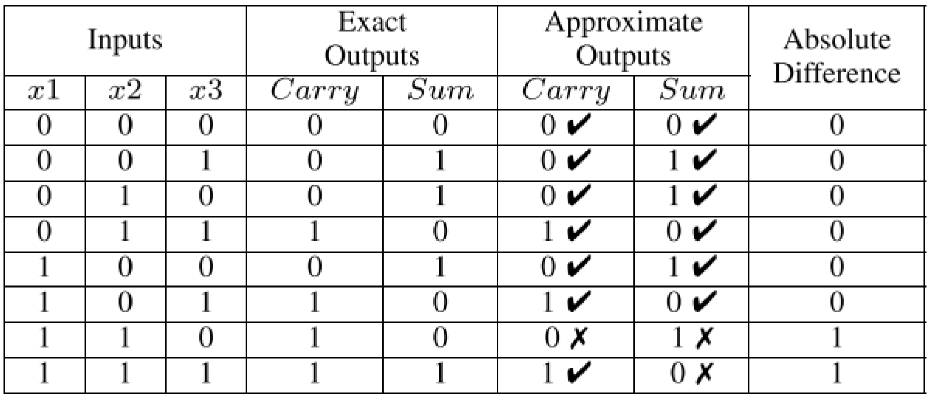
Figure 4: Truth table for Exact and Approximate half adder



	Exact Half Adder	**Approximate Half Adder**
Equations	$Sum = x_1 \oplus x_2$ $Carry = x_1 \cdot x_2$	$Sum = x_1 + x_2$ $Carry = x_1 \cdot x_2$
Probability of error	–	$P_{error}(Carry) = 0$ $P_{error}(Sum) = \frac{1}{4}$
Number of transistors	18	12
Transistors Saved	6	
Percentage Savings	33.33%	

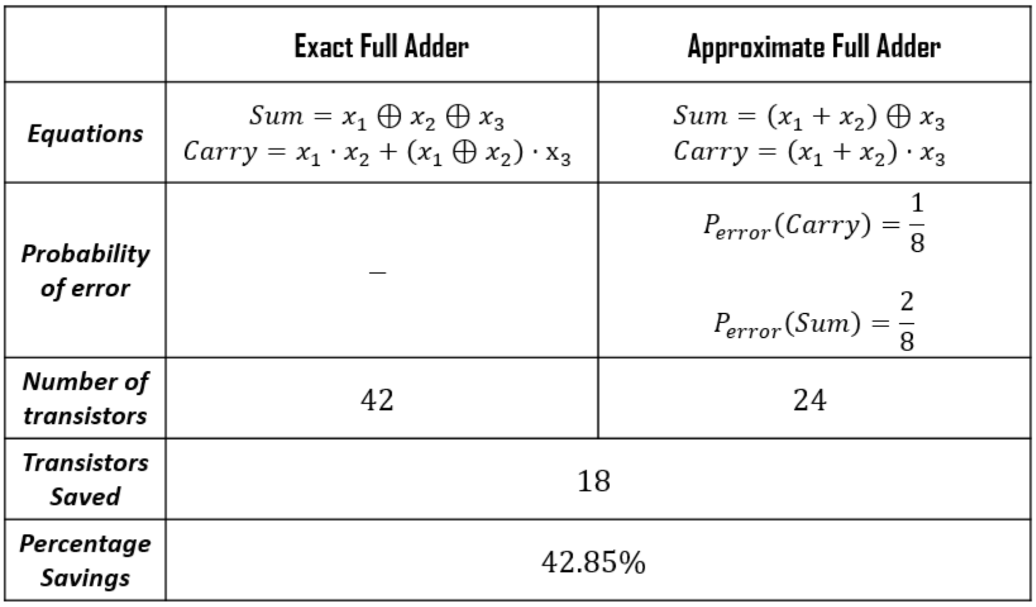
Figure 5: Comparison of Exact and Approximate Half Adders

6.2.2 Exact and Approximate full adder



Inputs			Exact Outputs		Approximate Outputs		Absolute Difference
$x1$	$x2$	$x3$	$Carry$	Sum	$Carry$	Sum	
0	0	0	0	0	0 ✓	0 ✓	0
0	0	1	0	1	0 ✓	1 ✓	0
0	1	0	0	1	0 ✓	1 ✓	0
0	1	1	1	0	1 ✓	0 ✓	0
1	0	0	0	1	0 ✓	1 ✓	0
1	0	1	1	0	1 ✓	0 ✓	0
1	1	0	1	0	0 ✗	1 ✗	1
1	1	1	1	1	1 ✓	0 ✗	1

Figure 6: Truth table for Exact and Approximate full adder

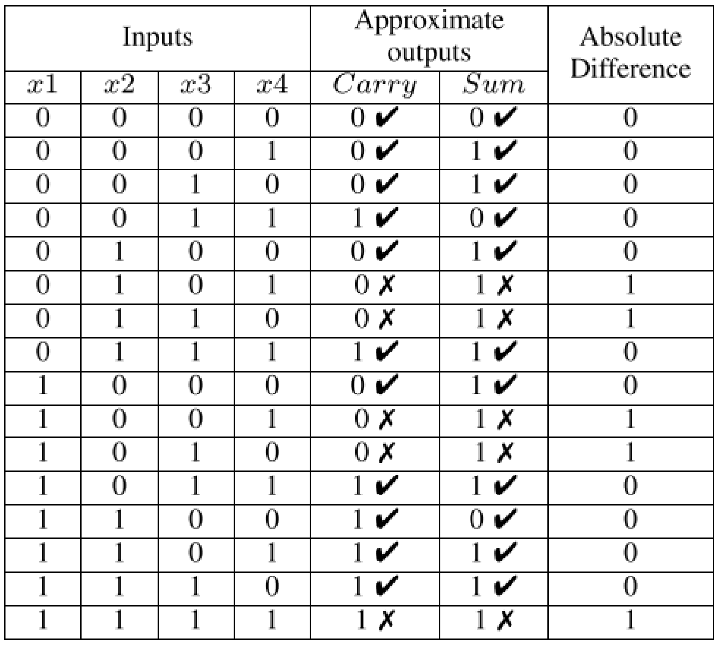


	Exact Full Adder	**Approximate Full Adder**
Equations	$Sum = x_1 \oplus x_2 \oplus x_3$ $Carry = x_1 \cdot x_2 + (x_1 \oplus x_2) \cdot \mathrm{x}_3$	$Sum = (x_1 + x_2) \oplus x_3$ $Carry = (x_1 + x_2) \cdot x_3$
Probability of error	–	$P_{error}(Carry) = \frac{1}{8}$ $P_{error}(Sum) = \frac{2}{8}$
Number of transistors	42	24
Transistors Saved	18	
Percentage Savings	42.85%	

Figure 7: Comparison of Exact and Approximate Full Adders

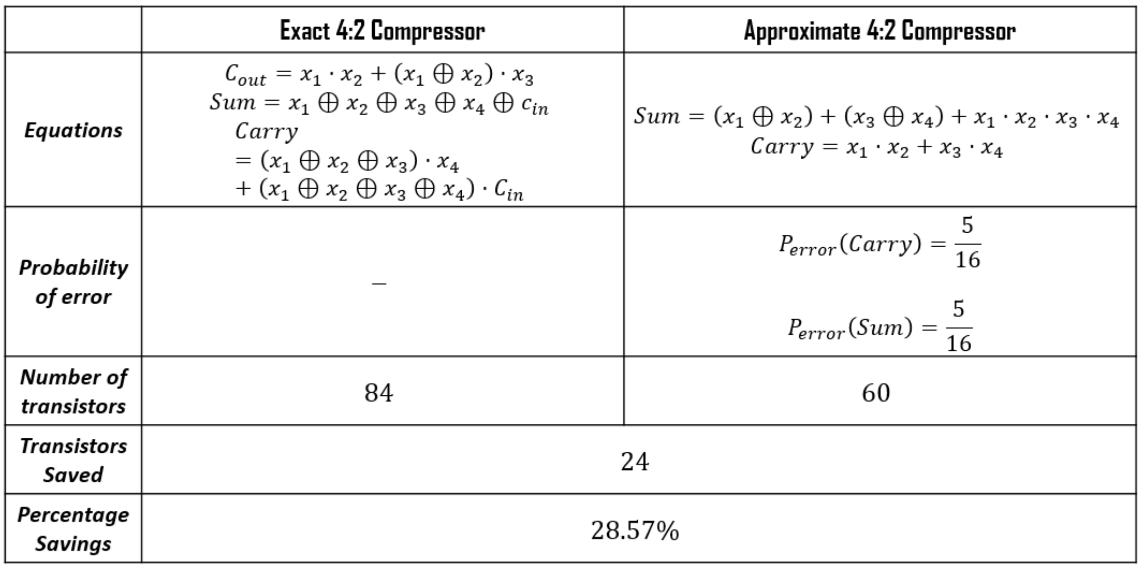
In the approximation of a full-adder, one of the two XOR gates is replaced with an OR gate in the Sum calculation. This results in an error in the last two cases out of eight cases. Carry is modified as in equation above, introducing one error. This approach provides more simplification while maintaining the difference between the original and approximate values as one.

6.2.3 Exact and Approximate 4:2 compressor



Inputs				Approximate outputs		Absolute Difference
$x1$	$x2$	$x3$	$x4$	$Carry$	Sum	
0	0	0	0	0 ✔	0 ✔	0
0	0	0	1	0 ✔	1 ✔	0
0	0	1	0	0 ✔	1 ✔	0
0	0	1	1	1 ✔	0 ✔	0
0	1	0	0	0 ✔	1 ✔	0
0	1	0	1	0 ✗	1 ✗	1
0	1	1	0	0 ✗	1 ✗	1
0	1	1	1	1 ✔	1 ✔	0
1	0	0	0	0 ✔	1 ✔	0
1	0	0	1	0 ✗	1 ✗	1
1	0	1	0	0 ✗	1 ✗	1
1	0	1	1	1 ✔	1 ✔	0
1	1	0	0	1 ✔	0 ✔	0
1	1	0	1	1 ✔	1 ✔	0
1	1	1	0	1 ✔	1 ✔	0
1	1	1	1	1 ✗	1 ✗	1

Figure 8: Truth table for 4:2 Aprroximate compressor



	Exact 4:2 Compressor	**Approximate 4:2 Compressor**
Equations	$C_{out} = x_1 \cdot x_2 + (x_1 \oplus x_2) \cdot x_3$ $Sum = x_1 \oplus x_2 \oplus x_3 \oplus x_4 \oplus c_{in}$ $Carry$ $= (x_1 \oplus x_2 \oplus x_3) \cdot x_4$ $+ (x_1 \oplus x_2 \oplus x_3 \oplus x_4) \cdot C_{in}$	$Sum = (x_1 \oplus x_2) + (x_3 \oplus x_4) + x_1 \cdot x_2 \cdot x_3 \cdot x_4$ $Carry = x_1 \cdot x_2 + x_3 \cdot x_4$
Probability of error	–	$P_{error}(Carry) = \frac{5}{16}$ $P_{error}(Sum) = \frac{5}{16}$
Number of transistors	84	60
Transistors Saved	24	
Percentage Savings	28.57%	

Figure 9: Comparison of Exact and Approximate 4:2 Compressor

Conclusion

- **Transistor Reduction:** All the approximate components show a reduction in the number of transistors used. Specifically, the approximate half adder achieves a 33.33% savings, the approximate full adder achieves a 42.85% savings, and the 4:2 approximate compressor achieves a 28.57% savings. These reductions are significant for optimizing hardware resources.

- **Error Probability:** While the approximate versions introduce a probability of error, this trade-off allows for substantial hardware savings. The error probabilities are relatively low, typically around $\frac{1}{8}$ to $\frac{5}{16}$, suggesting these approximate designs could be suitable for applications where occasional small errors are tolerable.

- **Implications and Applications:** This approach demonstrates the potential of approximate computing to achieve more energy efficient designs in digital systems. By carefully choosing where to apply approximation, hardware complexity and power consumption can be significantly reduced while maintaining acceptable accuracy for many applications. This technique could be particularly valuable in fields such as signal processing, machine learning, or multimedia, where small inaccuracies often do not significantly affect the final output quality.

6.2.4 The implementation of the components using the TIE language

We implemented each component as a separate module, allowing for modular reuse in the construction of the approximate multiplier. This modular approach facilitated easier integration and testing, ensuring flexibility and scalability in the design process.

```
module ha ( in [0:0] a, in [0:0] b, out [0:0] sum, out
    ↪ [0:0] carry)
{
    assign sum = a ^ b;
    assign carry = a & b;
}

module approximate_ha ( in [0:0] a, in [0:0] b, out
    ↪ [0:0] sum, out [0:0] carry)
{
    assign sum = a + b;
    assign carry = a & b;
}
```

Listing 1: Exact and Approximate half adder

```
module fa ( in [0:0] a, in [0:0] b, in [0:0] c, out
    ↪ [0:0] sum, out [0:0] carry)
{
    assign sum = a ^ b ^ c;
    assign carry = a & b | (a ^ b) & c;
}

module approximate_fa ( in [0:0] a, in [0:0] b, in
    ↪ [0:0] c, out [0:0] sum, out [0:0] carry)
{
    assign sum = (a | b) ^ c;
    assign carry = (a | b) & c;
}
```

Listing 2: Exact and Approximate full adder

```
module accurate_compressor_4_2 ( in [0:0] a, in [0:0]
    ↪ b, in [0:0] c, in [0:0] d, in [0:0] c_in, out
    ↪ [0:0] sum, out [0:0] carry, out [0:0] c_out)
{
    wire i_sum;
    fa full_adder_1 (a, b, c, i_sum, c_out);
    fa full_adder_2 (i_sum, d, c_in, sum, carry);
}

module approximate_compressor_4_2 ( in [0:0] a, in
    ↪ [0:0] b, in [0:0] c, in [0:0] d, out [0:0] sum,
    ↪ out [0:0] carry)
{
    assign sum = ((a==1 && b==0 && c==1 && d==0) ||
    ↪ (a==0 && b==1 && c==0 && d==1)) ? 1 : (a ^ b + c
    ↪ ^ d + a * b * c * d);
    assign carry = (a == 1 && b == 1 && c == 1 && d ==
    ↪ 1) ? 1 : (a * b + c * d);
}
```

Listing 3: Exact and Approximate 4:2 compressor

6.3 High Speed Multiplier Architectures

The high-speed arithmetic circuits are desirable for all computing applications, specifically in Digital Signal Processing (DSP) algorithms. Multiplication is one of the most extensively used operations in DSP algorithms; therefore, speed optimization of the multipliers is critical in order to speed up the DSP algorithms.

A large number of high-speed multiplier architectures are proposed in the literature, among which tree multipliers are considered to be one of the fastest. Figure 10 shows the general block diagram of the tree-based multipliers.

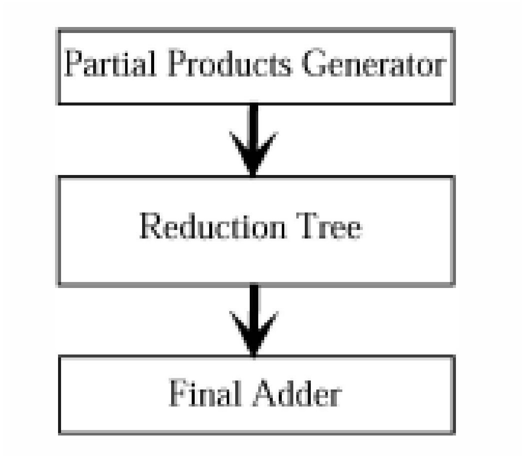


Figure 10: Block Diagram of Tree Based Multipliers

As can be seen the operation of the tree multipliers is divided into three steps.

1. **Partial product generation:** The simplest circuit for generating the partial product tree consists of AND gates. An N-bit multiplier requires N^2 AND gates to generate the partial product tree of N rows.

2. **Reduction Tree:** In this step, Full Adders (FAs), Half Adders (HAs) and 4:2 approximate Compressors are used to add all the columns of the partial product tree in a parallel fashion. The partial product tree is reduced stage by stage until only two rows are left.

3. **Final Adder:** In this step, the remaining two rows of the partial product tree are added by a ripple carry adder to produce the final product.

We will use the following structure in order to build the approximate 8×8 bit multiplier.

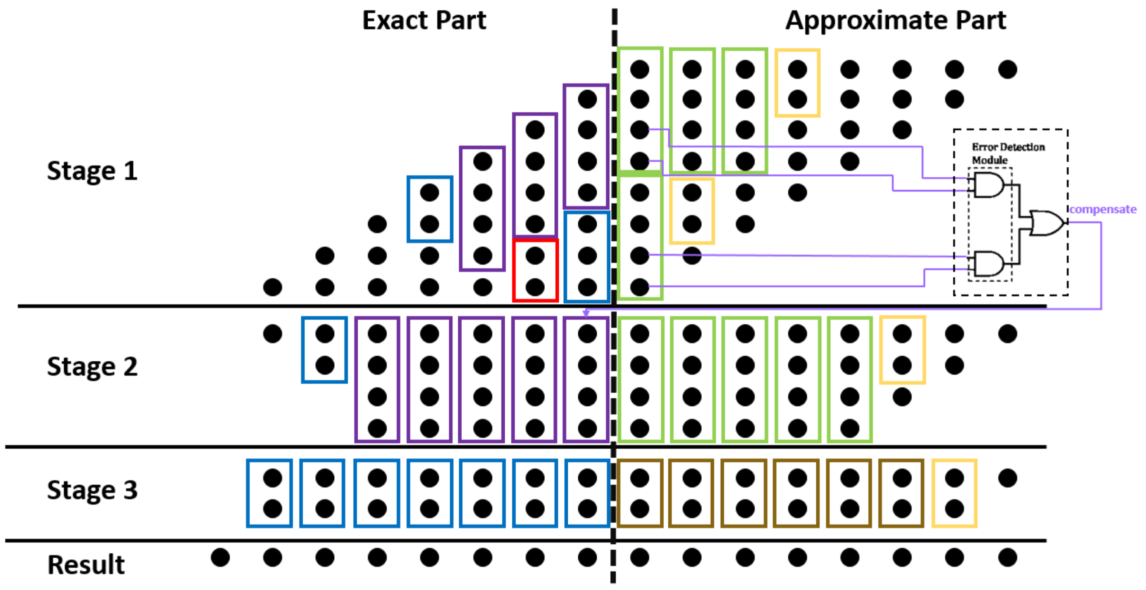


Figure 11: 8 x 8 Approximate Multiplier Architecture

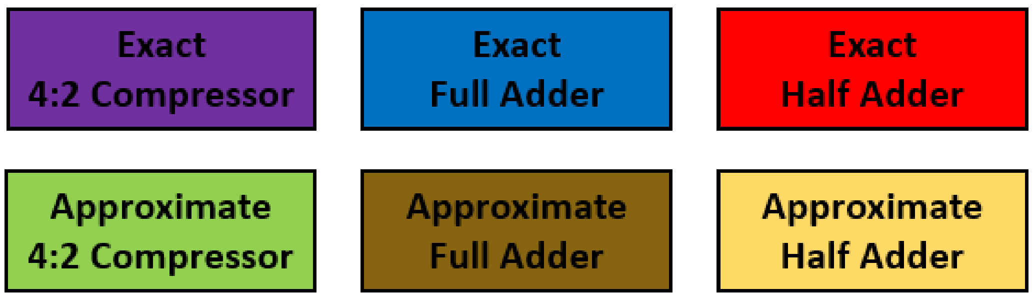


Figure 12: colored explain

Generate Partial Product:

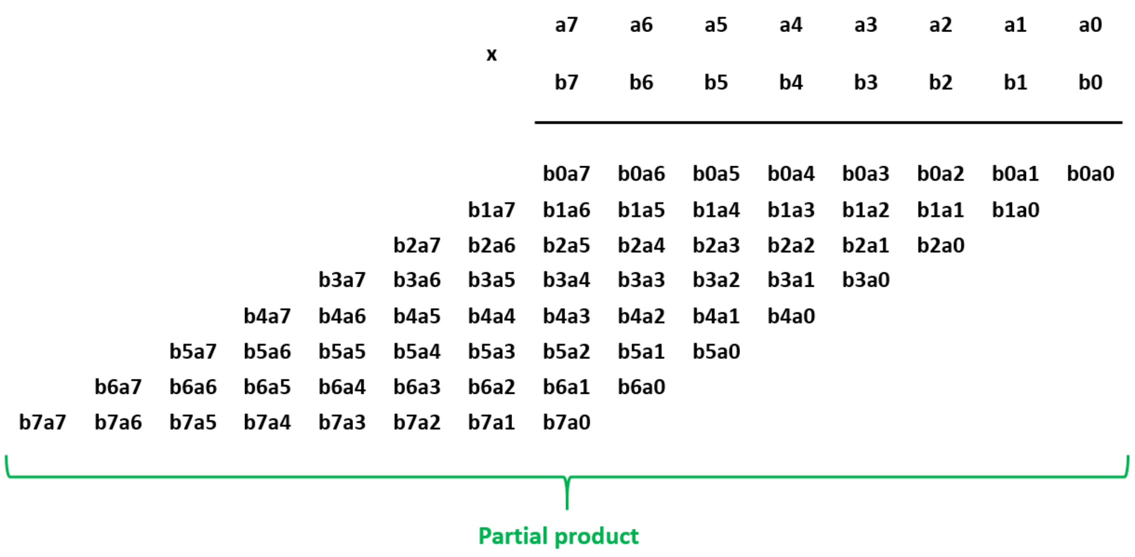

a7 a6 a5 a4 a3 a2 a1 a0
x
b7 b6 b5 b4 b3 b2 b1 b0
b0a7 b0a6 b0a5 b0a4 b0a3 b0a2 b0a1 b0a0
b1a7 b1a6 b1a5 b1a4 b1a3 b1a2 b1a1 b1a0
b2a7 b2a6 b2a5 b2a4 b2a3 b2a2 b2a1 b2a0
b3a7 b3a6 b3a5 b3a4 b3a3 b3a2 b3a1 b3a0
b4a7 b4a6 b4a5 b4a4 b4a3 b4a2 b4a1 b4a0
b5a7 b5a6 b5a5 b5a4 b5a3 b5a2 b5a1 b5a0
b6a7 b6a6 b6a5 b6a4 b6a3 b6a2 b6a1 b6a0
b7a7 b7a6 b7a5 b7a4 b7a3 b7a2 b7a1 b7a0
Partial product

Figure 13: Generate Partial Product

Stage 1 - Reduction Tree:

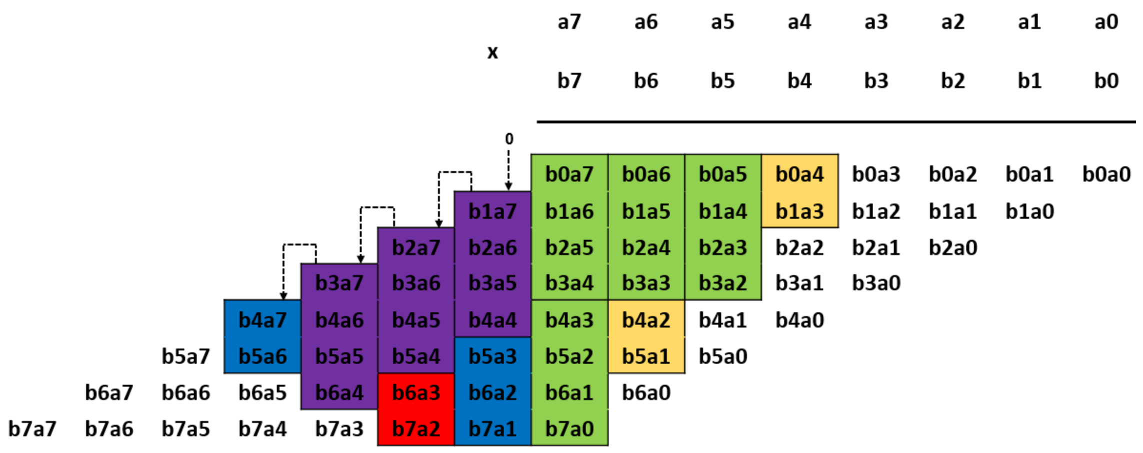

a7 a6 a5 a4 a3 a2 a1 a0
x
b7 b6 b5 b4 b3 b2 b1 b0
0
b0a7 b0a6 b0a5 b0a4 b0a3 b0a2 b0a1 b0a0
b1a7 b1a6 b1a5 b1a4 b1a3 b1a2 b1a1 b1a0
b2a7 b2a6 b2a5 b2a4 b2a3 b2a2 b2a1 b2a0
b3a7 b3a6 b3a5 b3a4 b3a3 b3a2 b3a1 b3a0
b4a7 b4a6 b4a5 b4a4 b4a3 b4a2 b4a1 b4a0
b5a7 b5a6 b5a5 b5a4 b5a3 b5a2 b5a1 b5a0
b6a7 b6a6 b6a5 b6a4 b6a3 b6a2 b6a1 b6a0
b7a7 b7a6 b7a5 b7a4 b7a3 b7a2 b7a1 b7a0

Figure 14: Stage 1

Stage 2 - Reduction Tree:

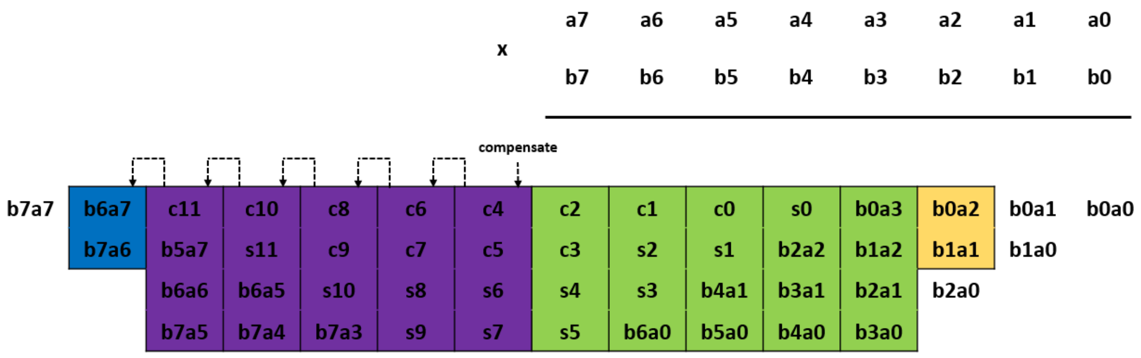

a7 a6 a5 a4 a3 a2 a1 a0
x
b7 b6 b5 b4 b3 b2 b1 b0
compensate
b7a7 b6a7 c11 c10 c8 c6 c4 c2 c1 c0 s0 b0a3 b0a2 b0a1 b0a0
b7a6 b5a7 s11 c9 c7 c5 c3 s2 s1 b2a2 b1a2 b1a1 b1a0
b6a6 b6a5 s10 s8 s6 s4 s3 b4a1 b3a1 b2a1 b2a0
b7a5 b7a4 b7a3 s9 s7 s5 b6a0 b5a0 b4a0 b3a0

Figure 15: Stage 2

Stage 3 - Final Adder:

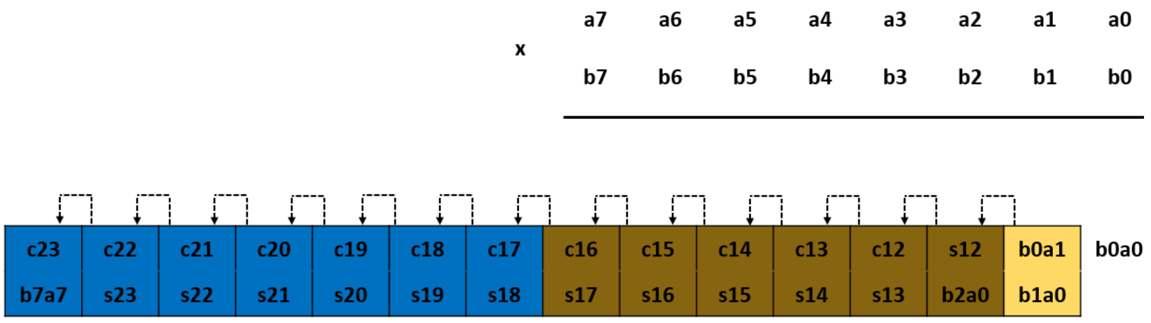

a7 a6 a5 a4 a3 a2 a1 a0
x
b7 b6 b5 b4 b3 b2 b1 b0
c23 c22 c21 c20 c19 c18 c17 c16 c15 c14 c13 c12 s12 b0a1 b0a0
b7a7 s23 s22 s21 s20 s19 s18 s17 s16 s15 s14 s13 b2a0 b1a0

Figure 16: Stage 3

Result:

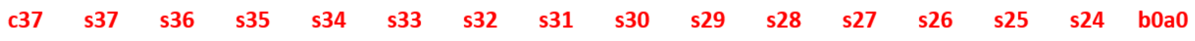

c37 s37 s36 s35 s34 s33 s32 s31 s30 s29 s28 s27 s26 s25 s24 b0a0

Figure 17: Result

6.3.1 TIE language implementation:

We designed dedicated hardware using the TIE language to implement the structure outlined above. Below, the code that delineates each step of the process.

```
 wire s0, s1, s2, s3, s4, s5, s6, s7, s8, s9, s10, s11;
 wire c0, c1, c2, c3, c4, c5, c6, c7, c8, c9, c10, c11;
 wire k1, k2, k3, k4, k5, k6, k7, k8;

 approximate_ha ha1 (a[4:4]&b[0:0], a[3:3]&b[1:1], s0, c0);
 approximate_compressor ac1 (a[5:5]&b[0:0], a[4:4]&b[1:1], a[3:3]&b[2:2], a[2:2]&b[3:3], s1, c1);
 approximate_compressor ac2 (a[6:6]&b[0:0], a[5:5]&b[1:1], a[4:4]&b[2:2], a[3:3]&b[3:3], s2, c2);
 approximate_ha ha2 (a[2:2]&b[4:4], a[1:1]&b[5:5], s3, c3);
 approximate_compressor ac3 (a[7:7]&b[0:0], a[6:6]&b[1:1], a[5:5]&b[2:2], a[4:4]&b[3:3], s4, c4);
 approximate_compressor ac4 (a[3:3]&b[4:4], a[2:2]&b[5:5], a[1:1]&b[6:6], a[0:0]&b[7:7], s5, c5);
 accurate_compressor_4_2 acc1 (1'b0, a[7:7]&b[1:1], a[6:6]&b[2:2], a[5:5]&b[3:3], a[4:4]&b[4:4], s6, c6, k1);
 fa fa1 (a[3:3]&b[5:5], a[2:2]&b[6:6], a[1:1]&b[7:7], s7, c7);
 accurate_compressor_4_2 acc2 (k1, a[7:7]&b[2:2], a[6:6]&b[3:3], a[5:5]&b[4:4], a[4:4]&b[5:5], s8, c8, k2);
 ha ha3 (a[3:3]&b[6:6], a[2:2]&b[7:7], s9, c9);
 accurate_compressor_4_2 acc3 (k2, a[7:7]&b[3:3], a[6:6]&b[4:4], a[5:5]&b[5:5], a[4:4]&b[6:6], s10, c10, k3);
 fa fa2 (k3, a[7:7]&b[4:4], a[6:6]&b[5:5], s11, c11);
```

Figure 18: TIE implementation for stage 1

```
wire s12, s13, s14, s15, s16, s17, s18, s19, s20, s21;
wire c12, c13, c14, c15, c16, c17, c18, c19, c20, c21;

approximate_ha ha5 (a[2]&b[0], a[1]&b[1], s12, c12);
approximate_compressor ac8 (a[3]&b[0], a[2:2]&b[1:1], a[1:1]&b[2:2], a[0:0]&b[3:3], s13, c13);
approximate_compressor ac9 (s0, a[2:2]&b[2:2], a[1:1]&b[3:3], a[0:0]&b[4:4], s14, c14);
approximate_compressor ac10 (c0, s1, a[1:1]&b[4:4], a[0:0]&b[5:5], s15, c15);
approximate_compressor ac11 (c1, s2, s3, a[0:0]&b[6:6], s16, c16);
approximate_compressor ac12 (c2, c3 ,s4, s5, s17, c17);
wire compensate = ((a[5]&b[2])&(a[4]&b[3])) | ((a[1]&b[6])&(a[0]&b[7]));
accurate_compressor_4_2 acc4 (compensate, c4, c5, s6, s7, s18, c18, k4);
accurate_compressor_4_2 acc5 (k4, c6, c7, s8, s9, s19, c19, k5);
accurate_compressor_4_2 acc6 (k5, c8, c9, s10, a[3:3]&b[7:7], s20, c20, k6);
accurate_compressor_4_2 acc7 (k6, c10, s11, a[5:5]&b[6:6], a[4:4]&b[7:7], s21, c21, k7);
accurate_compressor_4_2 acc8 (k7, c11, a[7:7]&b[5:5], a[6:6]&b[6:6], a[5:5]&b[7:7], s22, c22, k8);
fa fa3 (k8, a[7:7]&b[6:6], a[6:6]&b[7:7], s23, c23);
```

Figure 19: TIE implementation for stage 2

```verilog
wire s22, s23, s24, s25, s26, s27, s28, s29, s30, s31, s32, s33, s34, s35, s36, s37;
wire c22, c23, c24, c25, c26, c27, c28, c29, c30, c31, c32, c33, c34, c35, c36, c37;

approximate_ha ha7 (a[1]&b[0], a[0]&b[1], s24, c24);
approximate_fa fa4 (c24, s12, a[0]&b[2], s25, c25);
approximate_fa fa5 (c25, c12, s13, s26, c26);
approximate_fa fa6 (c26, c13, s14, s27, c27);
approximate_fa fa7 (c27, c14, s15, s28, c28);
approximate_fa fa8 (c28, c15, s16, s29, c29);
approximate_fa fa9 (c29, c16, s17, s30, c30);
fa fa10 (c30, c17, s18, s31, c31);
fa fa11 (c31, c18, s19, s32, c32);
fa fa12 (c32, c19, s20, s33, c33);
fa fa13 (c33, c20, s21, s34, c34);
fa fa14 (c34, c21, s22, s35, c35);
fa fa15 (c35, c22, s23, s36, c36);
fa fa16 (c36, c23, a[7]&b[7], s37, c37);
```

Figure 20: TIE implementation for stage 3

```verilog
wire [15:0] temp = {a[0]&b[0], s24, s25, s26, s27, s28, s29, s30, s31, s32, s33, s34, s35, s36, s37, c37 };
wire [15:0] temp_out;
assign temp_out = {temp[0], temp[1], temp[2], temp[3], temp[4], temp[5], temp[6], temp[7], temp[8], temp[9], temp[10], temp[11], temp[12], temp[13], temp[14], temp[15]};
assign output = (A[7]^B[7]) ? (~temp_out+1) : temp_out;
```

Figure 21: TIE implementation for result

7 Expanding to a 16-bit Approximate Multiplier

To enhance the functionality of the multiplier and enable it to perform 16×16 bit calculations, we will utilize four previously constructed 8×8 bit multiplier components. The systematic process for multiplying two 16-bit numbers with an 8×8 bit multiplier is detailed as follows:

To handle a 16-bit number, we will separate it into its upper 8 bits and lower 8 bits. To ensure the integrity of the number and maintain the correct positioning of each bit, the upper 8 bits must be shifted left by 8 positions. This shifting process preserves the significance of the bits, allowing for accurate multiplication and subsequent operations within the 16-bit framework.

After performing the separation for each 16-bit number, we can represent it as the sum of the upper 8 bits (shifted left by 8) and the lower 8 bits. Thus, the number is effectively expressed as the addition of two 8-bit components.

To multiply two 16-bit numbers, we apply the law of division, breaking the multiplication into four smaller multiplications. These smaller multiplications are then combined to form the final result. However, before combining them, it's essential to align the bits correctly using bit-shifting. Here's how the shifting is applied:

- When both factors are the upper 8 bits, we perform a left shift by 16 (8 positions for each factor).
- When one factor is from the upper 8 bits and the other from the lower 8 bits, we shift left by 8 positions.
- When both factors are the lower 8 bits, no shifting is required.

This shifting ensures the integrity of the multiplication and maintains the correct position of each bit in the final result, preserving accuracy.

While implementing this method, we encountered several challenges, which we will explain and discuss in detail in the following section.

The following figure describes the explanation that appears above

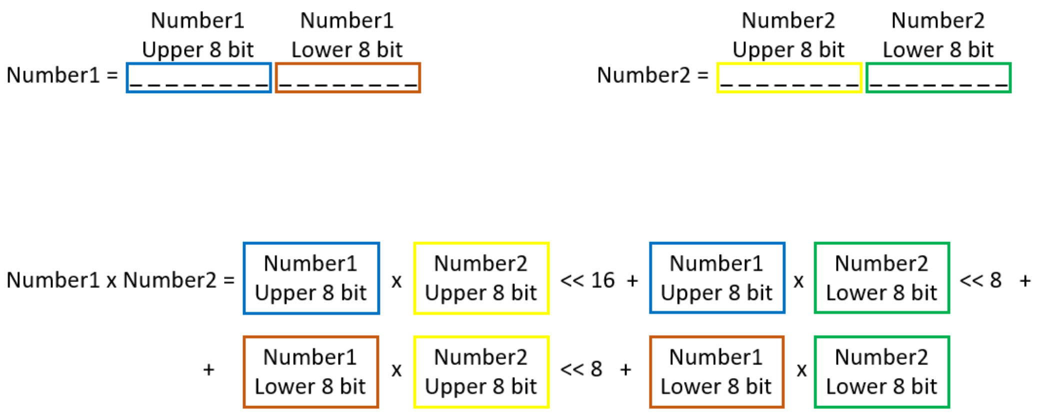


Figure 22: 16 x 16 bit multiplication

8 Difficulties and solutions

In this project, we implemented 8×8 multiplication between signed numbers. Initially, the results did not align with the expected values, as the multiplier was unable to handle negative numbers. Upon identifying this issue, we refined the hardware code to properly handle signed number multiplication (both positive and negative) using the 2's complement method. For negative numbers, we invert all the bits and add 1, thus making the number positive. For positive numbers, no modification is needed (see Figure 23). This approach allows us to work with positive numbers only. We will consider the sign during the calculation of the final result.

The final result is determined by examining the Most Significant Bits (MSBs) of the two numbers. We perform an XOR operation on the MSBs—if the XOR result is 1, it indicates that the numbers had different signs, and the result should be negative. So, to make the result negative, we will use the 2's complement method. If the XOR result is 0, it means both numbers had the same sign, and no further adjustment is necessary (see Figure 24).

```
    assign a = (A[7]==1'b1) ? (~A+1) : (A);
    assign b = (B[7]==1'b1) ? (~B+1) : (B);
```

Figure 23: 2's complement method in TIE implementation

```
    assign output = (A[7]^B[7]) ? (~temp_out+1) : temp_out;
```

Figure 24: The result sign in TIE implementation

Another problem we solved, in the initial stages of our project, we encountered unexpectedly high error rates when comparing our approximate results to the exact outputs. These discrepancies prompted a thorough investigation and debugging process, which revealed a critical issue: the significant errors were predominantly occurring when handling lower numerical values, particularly in the range of -256 to 255. Understanding this limitation allowed us to refine our approach and address the challenges associated with this specific numerical range. We will now outline the problem we encountered and describe the steps we took to address it.

This issue arises from our approach of representing a 16-bit number as the sum of two 8-bit numbers, with each 8-bit segment sent to the multiplier. (see above).

When examining the set of numbers from 0 to 255, we note that their binary representations as 16-bit numbers have the upper 8 bits as zeros. The problem occurs when the bit in position 7 is set to 1, causing a positive number to be interpreted as negative due to the signed nature of our system. To address this, if the bit in position 7 is 1, we will send the upper 8 bits (all zeros) as one input to the multiplier, while the second input will consist of the bits in positions 1 to 8, preserving the original sign (where bit 8 remains 0). After performing the multiplication, we will apply a shift left by 1 to reposition the bits correctly.

Similarly, when examining the set of numbers from -256 to 0, we note that their binary representations as 16-bit numbers have the upper 8 bits as ones. The problem occurs when the bit in position 7 is set to 0, causing a negative number to be interpreted as positive due to the signed nature of our system. To address this, if the bit in position 7 is 0, we will send the upper 8 bits (all ones) as one input to the multiplier, while the second input will consist of the bits in positions 1 to 8, preserving the original sign (where bit 8 remains 1). After performing the multiplication, we will apply a shift left by 1 to reposition the bits correctly.

If the two numbers involved in the multiplication fall within the specified range, we will proceed as described above; however, after the multiplication, we will shift the result left by 2.

Attached a segment of the TIE implementation that addresses the issues we outlined above.

```
// FIRST NUMBER
  wire shift_by_1_num1_down = ((x_in[31:24] == one_vector) && (x_in[23] == 1'b0)) || ((x_in[31:24] == zero_vector) && (x_in[23] == 1'b1));
  wire [7:0] num1_up = (x_in[31:24]==one_vector) ? zero_vector : x_in[31:24];
  wire [7:0] num1_down = (x_in[31:24] == one_vector) ? ((x_in[23] == 1'b0) ? x_in[24:17] : x_in[23:16]) :
                  (x_in[31:24] == zero_vector) ? ((x_in[23] == 1'b1) ? x_in[24:17] : x_in[23:16]) :
                  x_in[23:16];


  wire shift_by_1_num2_down = ((x_in[15:8] == one_vector) && (x_in[7] == 1'b0)) || ((x_in[15:8] == zero_vector) && (x_in[7] == 1'b1));
  wire [7:0] num2_up = (x_in[15:8]==one_vector) ? (x_in[15:8]==zero_vector) : x_in[15:8];
  wire [7:0] num2_down = (x_in[15:8] == one_vector) ? ((x_in[7] == 1'b0) ? x_in[8:1] : x_in[7:0]) :
                  (x_in[15:8] == zero_vector) ? ((x_in[7] == 1'b1) ? x_in[8:1] : x_in[7:0]) :
                  x_in[7:0];


  // SECOND NUMBER
  wire shift_by_1_num3_down = ((y_in[31:24] == one_vector) && (y_in[23] == 1'b0)) || ((y_in[31:24] == zero_vector) && (y_in[23] == 1'b1));
  wire [7:0] num3_up = (y_in[31:24]==one_vector) ? (y_in[31:24]==zero_vector) : y_in[31:24];
  wire [7:0] num3_down = (y_in[31:24] == one_vector) ? ((y_in[23] == 1'b0) ? y_in[24:17] : y_in[23:16]) :
                  (y_in[31:24] == zero_vector) ? ((y_in[23] == 1'b1) ? y_in[24:17] : y_in[23:16]) :
                  y_in[23:16];


  wire shift_by_1_num4_down = ((y_in[15:8] == one_vector) && (y_in[7] == 1'b0)) || ((y_in[15:8] == zero_vector) && (y_in[7] == 1'b1));
  wire [7:0] num4_up = (y_in[15:8]==one_vector) ? (y_in[15:8]==zero_vector) : y_in[15:8];
  wire [7:0] num4_down = (y_in[15:8] == one_vector) ? ((y_in[7] == 1'b0) ? y_in[8:1] : y_in[7:0]) :
          (y_in[15:8] == zero_vector) ? ((y_in[7] == 1'b1) ? y_in[8:1] : y_in[7:0]) :
          y_in[7:0];
```

9 Acceleration

In this project, we worked with the AR register, a 32-bit register in the TIE language, to design an approximate multiplier that computes the multiplication of 16-bit numbers. Beyond the area and power savings achieved through approximation, our primary objective was to enhance the speed of the multiplier. Specifically, we aimed to maximize the number of calculations that could be performed within a single clock cycle, ultimately reducing the overall cycle count.

To implement this, we developed a method that efficiently combines two 16-bit numbers into a single 32-bit value without altering the original values. The composition was done as follows: First, we shifted the first number to the left by 16 bits. Then, we applied a bitwise AND operation on the second number using the mask 0xFFFF. Finally, we combined the two numbers using a bitwise OR operation. This resulted in a 32-bit value where the upper 16 bits represent the first number, and the lower 16 bits represent the second number.

Example:

```
32_bit_number = number1 << 16 | (number2 & 0xFFFF)
```

Using this approach, we were able to perform two multiplications of 16 bits within a single clock cycle. **This optimization effectively halved the number of cycles required**, significantly improving the overall efficiency of the multiplier.

For example, if we do 100 iterations, after profiling we got:

Function Name	**Function**	**Children**	**Total (F+C)**	**Called**
approximate_multiplication_TIE	500	0	500	100
multiplication_C	1000	0	1000	200

Table 1: Cycle Analysis Summary

This improvement is further demonstrated in the following assembly code, which illustrates the multiplier's ability to perform two 16-bit multiplications in a single clock cycle as described.

```
int multiplication_C(int a, int b) __attribute__((noinline)) {
60001c18:   004136          entry   a1, 32
    return a * b;
60001c1b:   f19f532270fe    { nop; mull a2, a3, a2 }
60001c21:   f01d            retw.n

60001c23 <multiplication_C+0xb>:
    ...

60001c24 <approximate_multiplication_TIE>:
}

MultiplyResult approximate_multiplication_TIE(int a, int b) __attribute__((noinline)) {
60001c24:   004136          entry   a1, 32
    MultiplyResult result;
    approximate_mult_16_x_16(result.out1, result.out2, a, b);
60001c27:   263230          approximate_mult_16_x_16    a2, a3, a2, a3
    return result;
60001c2a:   f01d            retw.n
```

Figure 25: Assembly

We also analyzed the runtime of the function we developed and, as expected, found a reduction in execution times. By utilizing the multiplier we designed, we achieved a 15% improvement in runtime efficiency. The results of this analysis are presented below.

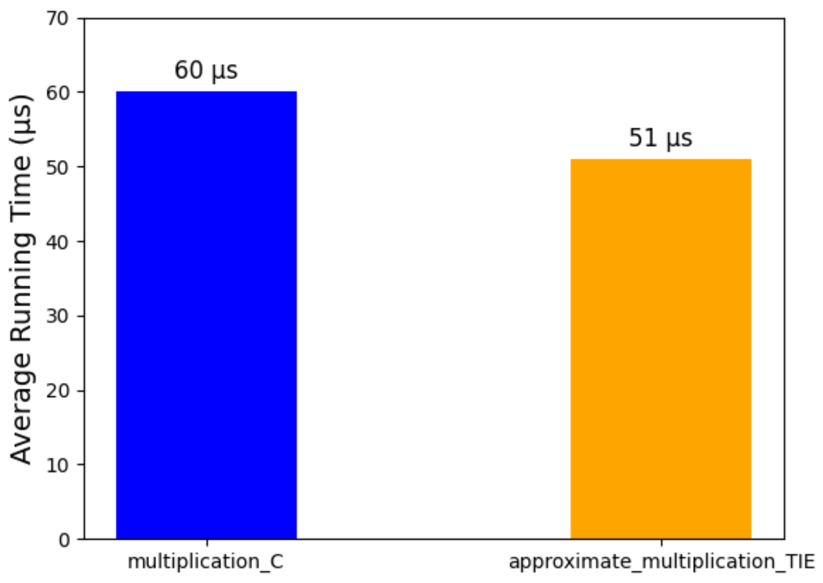


Figure 26: Average Running Time Comparison

10 Results and discussion

In this section, we will present the results obtained from the approximate multiplier we developed.

In this graph, the y-axis denotes the exact product values, and the x-axis represents the approximate product values. Each data point reflects the relationship between an exact value and its corresponding approximate value. The degree of linearity in the plot serves as a key indicator of accuracy: the closer the points are to forming a straight, diagonal line, the more the approximate values match the exact ones. A perfectly linear graph suggests near-identical results, while deviations from this line indicate differences between the exact and approximate products.

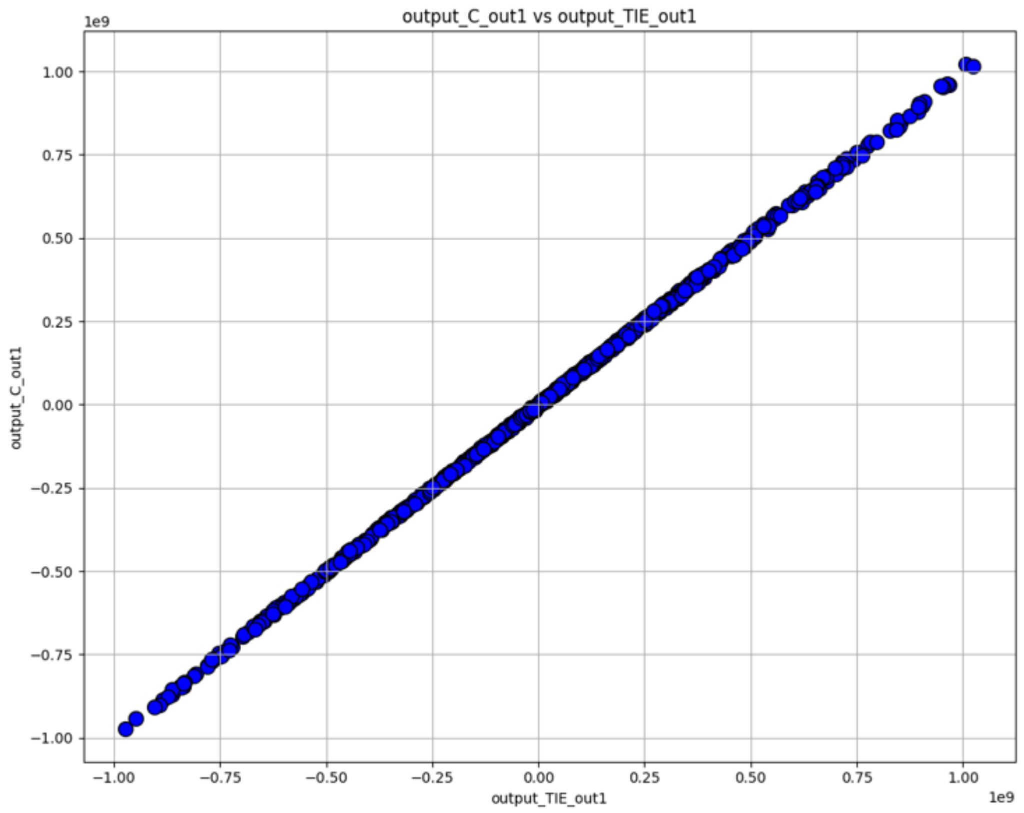


Figure 27: output_C vs output_TIE

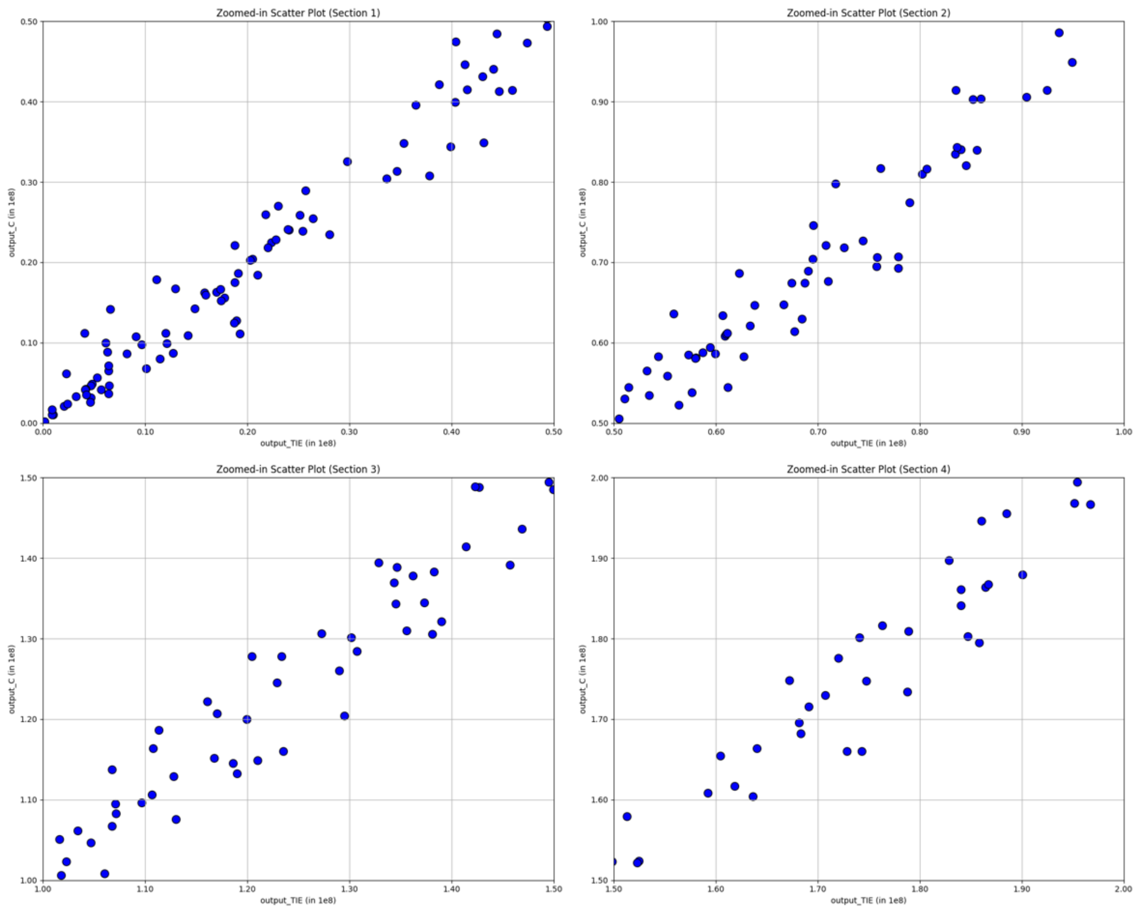


Figure 28: output_C vs output_TIE zoom-in

This graph shows the Cumulative Distribution Function (CDF) of error percentages, where the majority of the data points have low error values, indicated by the sharp rise in cumulative probability at the beginning. Nearly all the data is accounted for by the time the error percentage reaches around 15This suggests that most errors are relatively small, and only a small fraction of data points exhibit large errors beyond this range. The flattening of the curve near the top confirms that higher error percentages are rare.

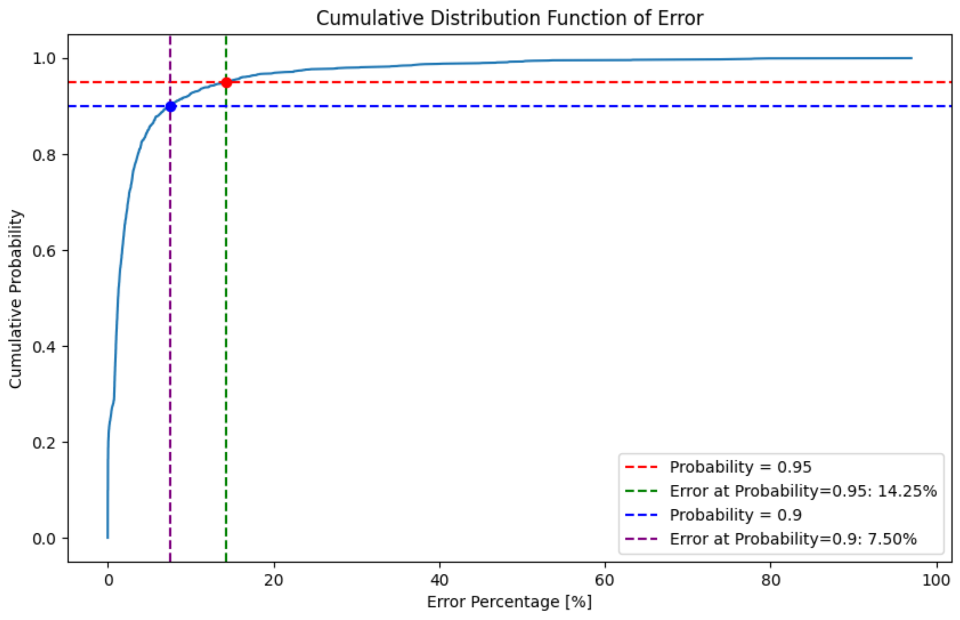


Figure 29: Cumulative Distribution Function of Error

The graph offers a comprehensive visualization of the error percentage distribution derived from two datasets in our project. The x-axis denotes the range of error percentages, while the y-axis illustrates the frequency of occurrence for each value. The distribution is notably skewed toward lower error percentages, with a prominent peak at 0%, signifying that the majority of the data exhibits minimal error. As the error percentage rises, the frequency of occurrences sharply declines, highlighting that larger errors are significantly rarer in comparison to smaller ones.

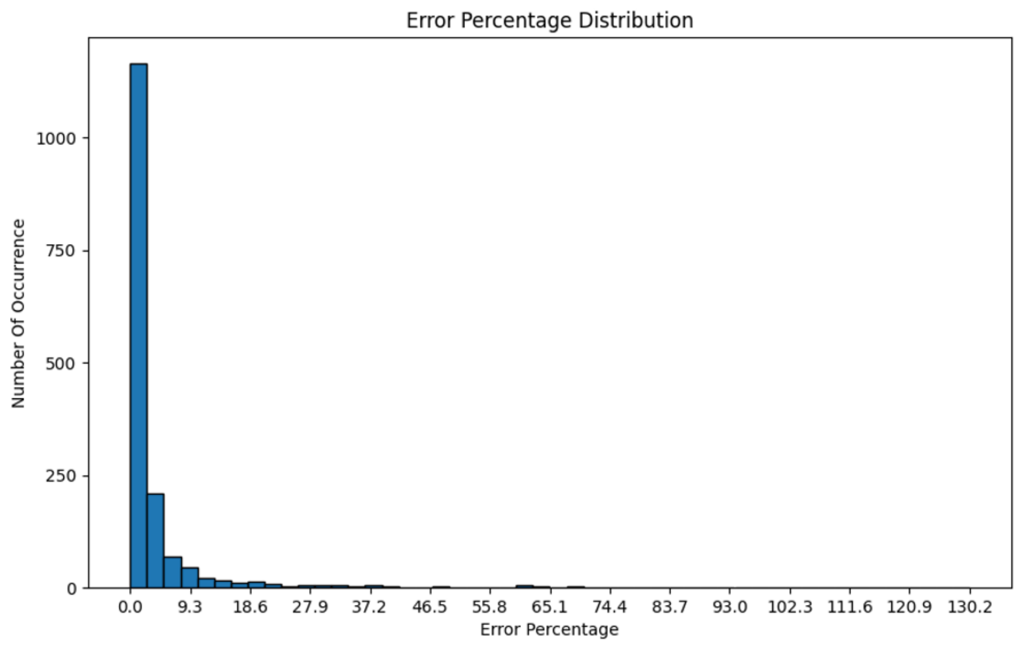


Figure 30: Error Percentage Distribution

11 Summary and conclusions

In this project, we developed a 16-bit approximate multiplier using the TIE language, aimed at exploring the advantages of approximate computation to reduce resource use and enhance processing speed. A component-based method allowed us to adjust the accuracy level as needed for different stages, achieving an efficient balance between performance and error tolerance. This multiplier significantly optimized power consumption and reduced circuit area while maintaining acceptable accuracy for error-tolerant applications, such as signal processing.

Key Findings:

- **Runtime and Clock Cycle Efficiency:** The approximate multiplier demonstrated a 15% reduction in runtime compared to traditional multipliers. It efficiently performs two 16-bit multiplications per clock cycle, effectively halving the required clock cycles and significantly enhancing performance.
- **Error Analysis and Accuracy:** The attached graphs illustrate that the majority of error percentages are minimal, suggesting that the approximate multiplier achieves a high degree of accuracy. This indicates that our approach effectively balances approximation with precision, yielding results that closely align with expected values while benefiting from the efficiency gains of approximation.
- **Space Efficiency:** By reducing the transistor count through approximation, the multiplier design conserves physical circuit area, making it suitable for compact and resource-limited systems.

Potential for FFT Integration: The approximate multiplier created in this project holds promising potential for integration into FFT computations. By incorporating it into the multiplication stages of FFT operations, it could further reduce computational complexity, allowing for faster and more efficient processing of signals in resource-limited and real-time systems. This approach could enhance the multiplier's utility in embedded signal processing and other energy-sensitive applications.

12 Full TIE code

```
/////////////////////////////
//////  Modules Part   //////
/////////////////////////////

module ha ( in [0:0] a, in [0:0] b, out [0:0] sum, out
    [0:0] carry)
{
    assign sum = a ^ b;
    assign carry = a & b;
}

module approximate_ha ( in [0:0] a, in [0:0] b, out
    [0:0] sum, out [0:0] carry)
{
    assign sum = a + b;
    assign carry = a & b;
}

module fa ( in [0:0] a, in [0:0] b, in [0:0] c, out
    [0:0] sum, out [0:0] carry)
{
    wire x = a ^ b;
    wire y = a & b;
    wire z = x & c;

    assign sum = a ^ b ^ c;
    assign carry = y | z;
}

module approximate_fa ( in [0:0] a, in [0:0] b, in [0:0]
    c, out [0:0] sum, out [0:0] carry)
{
    assign sum = (a + b) ^ c;
    assign carry = (a + b) * c;
}

module accurate_compressor_4_2 ( in [0:0] a, in [0:0] b,
    in [0:0] c, in [0:0] d, in [0:0] c_in, out [0:0] sum
    , out [0:0] carry, out [0:0] c_out)
{
    wire i_sum;

    fa full_adder_1 (a, b, c, i_sum, c_out);
    fa full_adder_2 (i_sum, d, c_in, sum, carry);
}

module approximate_compressor ( in [0:0] a, in [0:0] b, in [0:0] c, in [0:0] d, out [0:0] sum, out [0:0] carry)
{
    assign sum = ((a==1 && b==0 && c==1 && d==0) || (a==0 && b==1 && c==0 && d==1)) ? 1 : (a ^ b + c ^ d + a * b * c * d);
    assign carry = (a == 1 && b == 1 && c == 1 && d == 1) ? 1 : (a * b + c * d);
}

module exact_multiplication ( in [7:0] A, in [7:0] B, out [15:0] output)
{
    assign output = TIEmul(A,B,1'b1);
}


// approximate 8x8 bit multiplier
module approximate_mult_8_x_8 ( in [7:0] A, in [7:0] B, out [15:0] output)
{
    wire [7:0] a, b;

    assign a = (A[7]==1'b1) ? (~A+1) : (A);
    assign b = (B[7]==1'b1) ? (~B+1) : (B);

    wire s0, s1, s2, s3, s4, s5, s6, s7, s8, s9, s10, s11;
    wire c0, c1, c2, c3, c4, c5, c6, c7, c8, c9, c10, c11;
    wire k1, k2, k3, k4, k5, k6, k7, k8;

    /////////////////////////
    //////   stage 1   //////
    /////////////////////////

    approximate_ha ha1 (a[4:4]&b[0:0], a[3:3]&b[1:1], s0, c0);
    approximate_compressor ac1 (a[5:5]&b[0:0], a[4:4]&b[1:1], a[3:3]&b[2:2], a[2:2]&b[3:3], s1, c1);
    approximate_compressor ac2 (a[6:6]&b[0:0], a[5:5]&b[1:1], a[4:4]&b[2:2], a[3:3]&b[3:3], s2, c2);
    approximate_ha ha2 (a[2:2]&b[4:4], a[1:1]&b[5:5], s3, c3);
    approximate_compressor ac3 (a[7:7]&b[0:0], a[6:6]&b[1:1], a[5:5]&b[2:2], a[4:4]&b[3:3], s4, c4);
    approximate_compressor ac4 (a[3:3]&b[4:4], a[2:2]&b[5:5], a[1:1]&b[6:6], a[0:0]&b[7:7], s5, c5);
    accurate_compressor_4_2 acc1 (1'b0, a[7:7]&b[1:1], a[6:6]&b[2:2], a[5:5]&b[3:3], a[4:4]&b[4:4], s6, c6, k1);
    fa fa1 (a[3:3]&b[5:5], a[2:2]&b[6:6], a[1:1]&b[7:7], s7, c7);
    accurate_compressor_4_2 acc2 (k1, a[7:7]&b[2:2], a[6:6]&b[3:3], a[5:5]&b[4:4], a[4:4]&b[5:5], s8, c8, k2);
    ha ha3 (a[3:3]&b[6:6], a[2:2]&b[7:7], s9, c9);
    accurate_compressor_4_2 acc3 (k2, a[7:7]&b[3:3], a[6:6]&b[4:4], a[5:5]&b[5:5], a[4:4]&b[6:6], s10, c10, k3);
    fa fa2 (k3, a[7:7]&b[4:4], a[6:6]&b[5:5], s11, c11);


    /////////////////////////
    //////    stage 2   //////
    /////////////////////////

    wire s12, s13, s14, s15, s16, s17, s18, s19, s20, s21;
    wire c12, c13, c14, c15, c16, c17, c18, c19, c20, c21;

    approximate_ha ha5 (a[2]&b[0], a[1]&b[1], s12, c12);
    approximate_compressor ac8 (a[3]&b[0], a[2:2]&b[1:1], a[1:1]&b[2:2], a[0:0]&b[3:3], s13, c13);
    approximate_compressor ac9 (s0, a[2:2]&b[2:2], a[1:1]&b[3:3], a[0:0]&b[4:4], s14, c14);
    approximate_compressor ac10 (c0, s1, a[1:1]&b[4:4], a[0:0]&b[5:5], s15, c15);
    approximate_compressor ac11 (c1, s2, s3, a[0:0]&b[6:6], s16, c16);
    approximate_compressor ac12 (c2, c3 ,s4, s5, s17, c17);
    wire compensate = ((a[5]&b[2])&(a[4]&b[3])) | ((a[1]&b[6])&(a[0]&b[7]));
    accurate_compressor_4_2 acc4 (compensate, c4, c5, s6, s7, s18, c18, k4);
    accurate_compressor_4_2 acc5 (k4, c6, c7, s8, s9, s19, c19, k5);
    accurate_compressor_4_2 acc6 (k5, c8, c9, s10, a[3:3]&b[7:7], s20, c20, k6);
    accurate_compressor_4_2 acc7 (k6, c10, s11, a[5:5]&b[6:6], a[4:4]&b[7:7], s21, c21, k7);
    accurate_compressor_4_2 acc8 (k7, c11, a[7:7]&b[5:5], a[6:6]&b[6:6], a[5:5]&b[7:7], s22, c22, k8);
    fa fa3 (k8, a[7:7]&b[6:6], a[6:6]&b[7:7], s23, c23);


    //////////////////////////
    //////    stage 3    //////
    //////////////////////////

    wire s22, s23, s24, s25, s26, s27, s28, s29, s30, s31, s32, s33, s34, s35, s36, s37;
    wire c22, c23, c24, c25, c26, c27, c28, c29, c30, c31, c32, c33, c34, c35, c36, c37;

    approximate_ha ha7 (a[1]&b[0], a[0]&b[1], s24, c24);
    approximate_fa fa4 (c24, s12, a[0]&b[2], s25, c25);
    approximate_fa fa5 (c25, c12, s13, s26, c26);
    approximate_fa fa6 (c26, c13, s14, s27, c27);
    approximate_fa fa7 (c27, c14, s15, s28, c28);
    approximate_fa fa8 (c28, c15, s16, s29, c29);
    approximate_fa fa9 (c29, c16, s17, s30, c30);
    fa fa10 (c30, c17, s18, s31, c31);
    fa fa11 (c31, c18, s19, s32, c32);
    fa fa12 (c32, c19, s20, s33, c33);
    fa fa13 (c33, c20, s21, s34, c34);
    fa fa14 (c34, c21, s22, s35, c35);
    fa fa15 (c35, c22, s23, s36, c36);
    fa fa16 (c36, c23, a[7]&b[7], s37, c37);


    //////////////////////////
    //////    output    //////
    //////////////////////////

    wire [15:0] temp = {a[0]&b[0], s24, s25, s26, s27, s28, s29, s30, s31, s32, s33, s34, s35, s36, s37, c37 };
    wire [15:0] temp_out;
    assign temp_out = {temp[0], temp[1], temp[2], temp[3], temp[4], temp[5], temp[6], temp[7], temp[8], temp[9], temp[10], temp[11], temp[12], temp[13], temp[14], temp[15]};
    assign output = (A[7]^B[7]) ? (~temp_out+1) : temp_out;
}


// approximate 16x16 bit multiplier
operation approximate_mult_16_x_16{ out AR out_16_x_16_1, out AR out_16_x_16_2, in AR x_in, in AR y_in } {}
{
    wire [31:0] temp_output_1_new;
    wire [31:0] temp_output_2_new;

    wire [7:0] zero_vector = {1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0};
    wire [7:0] one_vector = {1'b1,1'b1,1'b1,1'b1,1'b1,1'b1,1'b1,1'b1};

    // FIRST NUMBER
    wire shift_by_1_num1_down = ((x_in[31:24] == one_vector) && (x_in[23] == 1'b0)) || ((x_in[31:24] == zero_vector) && (x_in[23] == 1'b1));
    wire [7:0] num1_up = (x_in[31:24]==one_vector) ? zero_vector : x_in[31:24];
    wire [7:0] num1_down = (x_in[31:24] == one_vector) ? ((x_in[23] == 1'b0) ? x_in[24:17] : x_in[23:16]) :
                         (x_in[31:24] == zero_vector) ? ((x_in[23] == 1'b1) ? x_in[24:17] : x_in[23:16]) :
                         x_in[23:16];


```

```verilog
    wire shift_by_1_num2_down = ((x_in[15:8] == one_vector) && (x_in[7] == 1'b0)) || ((x_in[15:8] == zero_vector) && (x_in[7] == 1'b1));
    wire [7:0] num2_up = (x_in[15:8]==one_vector) ? (x_in[15:8]==zero_vector) : x_in[15:8];
    wire [7:0] num2_down = (x_in[15:8] == one_vector) ? ((x_in[7] == 1'b0) ? x_in[8:1] : x_in[7:0]) :
                        (x_in[15:8] == zero_vector) ? ((x_in[7] == 1'b1) ? x_in[8:1] : x_in[7:0]) :
                        x_in[7:0];


    // SECOND NUMBER
    wire shift_by_1_num3_down = ((y_in[31:24] == one_vector) && (y_in[23] == 1'b0)) || ((y_in[31:24] == zero_vector) && (y_in[23] == 1'b1));
    wire [7:0] num3_up = (y_in[31:24]==one_vector) ? (y_in[31:24]==zero_vector) : y_in[31:24];
    wire [7:0] num3_down = (y_in[31:24] == one_vector) ? ((y_in[23] == 1'b0) ? y_in[24:17] : y_in[23:16]) :
                        (y_in[31:24] == zero_vector) ? ((y_in[23] == 1'b1) ? y_in[24:17] : y_in[23:16]) :
                        y_in[23:16];


    wire shift_by_1_num4_down = ((y_in[15:8] == one_vector) && (y_in[7] == 1'b0)) || ((y_in[15:8] == zero_vector) && (y_in[7] == 1'b1));
    wire [7:0] num4_up = (y_in[15:8]==one_vector) ? (y_in[15:8]==zero_vector) : y_in[15:8];
    wire [7:0] num4_down = (y_in[15:8] == one_vector) ? ((y_in[7] == 1'b0) ? y_in[8:1] : y_in[7:0]) :
                        (y_in[15:8] == zero_vector) ? ((y_in[7] == 1'b1) ? y_in[8:1] : y_in[7:0]) :
                        y_in[7:0];


    //Implementation of 16x16 bit Multiplication Using Four 8x8 bit Elements
    wire [15:0] out1, out2, out3, out4;
    wire [15:0] out5, out6, out7, out8;

    exact_multiplication am_1(num1_up, num2_up, out1);
    approximate_mult_8_x_8 am_2(num1_up, num2_down, out2);
    approximate_mult_8_x_8 am_3(num1_down, num2_up, out3);
    approximate_mult_8_x_8 am_4(num1_down, num2_down, out4);

    exact_multiplication am_5(num3_up, num4_up, out5);
    approximate_mult_8_x_8 am_6(num3_up, num4_down, out6);
    approximate_mult_8_x_8 am_7(num3_down, num4_up, out7);
    approximate_mult_8_x_8 am_8(num3_down, num4_down, out8);


    // FIRST RESULT
    wire [31:0] t1 = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,out1[0],out1[1],out1[2],out1[3],out1[4],out1[5],out1[6],out1[7],out1[8],out1[9],out1[10],out1[11],out1[12],out1[13],out1[14],out1[15]};

    wire [31:0] TEMP_out1 = {t1[0], t1[1], t1[2], t1[3], t1[4], t1[5], t1[6], t1[7], t1[8], t1[9], t1[10], t1[11], t1[12], t1[13], t1[14], t1[15], t1[16], t1[17], t1[18], t1[19], t1[20], t1[21], t1[22], t1[23], t1[24], t1[25], t1[26], t1[27], t1[28], t1[29], t1[30], t1[31]};



    wire [15:0] t2 = {out2[0],out2[1],out2[2],out2[3],out2[4],out2[5],out2[6],out2[7],out2[8],out2[9],out2[10],out2[11],out2[12],out2[13],out2[14],out2[15]};

    wire out2_sign = (out2[15]==1'b1) ? 1'b1 : 1'b0;

    wire [31:0] TEMP_out2_8shift = {out2_sign,out2_sign,out2_sign,out2_sign,out2_sign,out2_sign,out2_sign,out2_sign, t2[0], t2[1], t2[2], t2[3], t2[4], t2[5], t2[6], t2[7], t2[8], t2[9], t2[10], t2[11], t2[12], t2[13], t2[14], t2[15],1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0};

    wire [31:0] TEMP_out2_9shift = {out2_sign,out2_sign,out2_sign,out2_sign,out2_sign,out2_sign,out2_sign, t2[0], t2[1], t2[2], t2[3], t2[4], t2[5], t2[6], t2[7], t2[8], t2[9], t2[10], t2[11], t2[12], t2[13], t2[14], t2[15],1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0};

    wire [31:0] TEMP_out2 = (shift_by_1_num2_down==1'b1) ? TEMP_out2_9shift : TEMP_out2_8shift;



    wire [15:0] t3 = {out3[0],out3[1],out3[2],out3[3],out3[4],out3[5],out3[6],out3[7],out3[8],out3[9],out3[10],out3[11],out3[12],out3[13],out3[14],out3[15]};

    wire out3_sign = (out3[15]==1'b1) ? 1'b1 : 1'b0;

    wire [31:0] TEMP_out3_8shift = {out3_sign,out3_sign,out3_sign,out3_sign,out3_sign,out3_sign,out3_sign,out3_sign, t3[0], t3[1], t3[2], t3[3], t3[4], t3[5], t3[6], t3[7], t3[8], t3[9], t3[10], t3[11], t3[12], t3[13], t3[14], t3[15],1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0};

    wire [31:0] TEMP_out3_9shift = {out3_sign,out3_sign,out3_sign,out3_sign,out3_sign,out3_sign,out3_sign, t3[0], t3[1], t3[2], t3[3], t3[4], t3[5], t3[6], t3[7], t3[8], t3[9], t3[10], t3[11], t3[12], t3[13], t3[14], t3[15],1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0};

    wire [31:0] TEMP_out3 = (shift_by_1_num1_down==1'b1) ? TEMP_out3_9shift : TEMP_out3_8shift;



    wire [15:0] t4 = {out4[0],out4[1],out4[2],out4[3],out4[4],out4[5],out4[6],out4[7],out4[8],out4[9],out4[10],out4[11],out4[12],out4[13],out4[14],out4[15]};

    wire out4_sign = (out4[15]==1'b1) ? 1'b1 : 1'b0;
    wire [31:0] TEMP_out4_0shift = {out4_sign,out4_sign,out4_sign,out4_sign,out4_sign,out4_sign,out4_sign,out4_sign,out4_sign,out4_sign,out4_sign,out4_sign,out4_sign,out4_sign,out4_sign,out4_sign, t4[0], t4[1], t4[2], t4[3], t4[4], t4[5], t4[6], t4[7], t4[8], t4[9], t4[10], t4[11], t4[12], t4[13], t4[14], t4[15]};

    wire [31:0] TEMP_out4_1shift = {out4_sign,out4_sign,out4_sign,out4_sign,out4_sign,out4_sign,out4_sign,out4_sign,out4_sign,out4_sign,out4_sign,out4_sign,out4_sign,out4_sign,out4_sign, t4[0], t4[1], t4[2], t4[3], t4[4], t4[5], t4[6], t4[7], t4[8], t4[9], t4[10], t4[11], t4[12], t4[13], t4[14], t4[15], 1'b0};

    wire [31:0] TEMP_out4_2shift = {out4_sign,out4_sign,out4_sign,out4_sign,out4_sign,out4_sign,out4_sign,out4_sign,out4_sign,out4_sign,out4_sign,out4_sign,out4_sign,out4_sign, t4[0], t4[1], t4[2], t4[3], t4[4], t4[5], t4[6], t4[7], t4[8], t4[9], t4[10], t4[11], t4[12], t4[13], t4[14], t4[15], 1'b0,1'b0};

    wire [31:0] TEMP_out4 = (shift_by_1_num1_down==1'b1 && shift_by_1_num2_down==1'b1) ? TEMP_out4_2shift :
              ((shift_by_1_num1_down==1'b1 && shift_by_1_num2_down==1'b0) || (shift_by_1_num1_down==1'b0 && shift_by_1_num2_down==1'b1)) ? TEMP_out4_1shift :
              TEMP_out4_0shift;




    // SECOND RESULT
    wire [31:0] t5 = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,out5[0],out5[1],out5[2],out5[3],out5[4],out5[5],out5[6],out5[7],out5[8],out5[9],out5[10],out5[11],out5[12],out5[13],out5[14],out5[15]};

    wire [31:0] TEMP_out5 = {t5[0], t5[1], t5[2], t5[3], t5[4], t5[5], t5[6], t5[7], t5[8], t5[9], t5[10], t5[11], t5[12], t5[13], t5[14], t5[15], t5[16], t5[17],
 t5[18], t5[19], t5[20], t5[21], t5[22], t5[23], t5[24], t5[25], t5[26], t5[27], t5[28], t5[29], t5[30], t5[31]};




    wire [15:0] t6 = {out6[0],out6[1],out6[2],out6[3],out6[4],out6[5],out6[6],out6[7],out6[8],out6[9],out6[10],out6[11],out6[12],out6[13],out6[14],out6[15]};

    wire out6_sign = (out6[15]==1'b1) ? 1'b1 : 1'b0;

    wire [31:0] TEMP_out6_8shift = {out6_sign,out6_sign,out6_sign,out6_sign,out6_sign,out6_sign,out6_sign,out6_sign, t6[0], t6[1], t6[2], t6[3], t6[4], t6[5], t6[6], t6[7], t6[8], t6[9], t6[10], t6[11], t6[12], t6[13], t6[14], t6[15],1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0};

    wire [31:0] TEMP_out6_9shift = {out6_sign,out6_sign,out6_sign,out6_sign,out6_sign,out6_sign,out6_sign, t6[0], t6[1], t6[2], t6[3], t6[4], t6[5], t6[6], t6[7], t6[8], t6[9], t6[10], t6[11], t6[12], t6[13], t6[14], t6[15],1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0};

    wire [31:0] TEMP_out6 = (shift_by_1_num4_down==1'b1) ? TEMP_out6_9shift : TEMP_out6_8shift;



    wire [15:0] t7 = {out7[0],out7[1],out7[2],out7[3],out7[4],out7[5],out7[6],out7[7],out7[8],out7[9],out7[10],out7[11],out7[12],out7[13],out7[14],out7[15]};

    wire out7_sign = (out7[15]==1'b1) ? 1'b1 : 1'b0;

    wire [31:0] TEMP_out7_8shift = {out7_sign,out7_sign,out7_sign,out7_sign,out7_sign,out7_sign,out7_sign,out7_sign, t7[0], t7[1], t7[2], t7[3], t7[4], t7[5], t7[6], t7[7], t7[8], t7[9], t7[10], t7[11], t7[12], t7[13], t7[14], t7[15],1'b0,1'b0,1'b0,1'b0,1'b0,1'b0
```

```
,1'b0,1'b0};

    wire [31:0] TEMP_out7_9shift = {out7_sign,out7_sign,out7_sign,out7_sign,out7_sign,out7_sign,out7_sign, t7[0], t7[1], t7[2], t7[3], t7[4], t7[5], t7[6], t7[7], t7[8], t7[9], t7[10], t7[11], t7[12], t7[13], t7[14], t7[15],1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0};

    wire [31:0] TEMP_out7 = (shift_by_1_num3_down==1'b1) ? TEMP_out7_9shift : TEMP_out7_8shift;



    wire [15:0] t8 = {out8[0],out8[1],out8[2],out8[3],out8[4],out8[5],out8[6],out8[7],out8[8],out8[9],out8[10],out8[11],out8[12],out8[13],out8[14],out8[15]};

    wire out8_sign = (out8[15]==1'b1) ? 1'b1 : 1'b0;

    wire [31:0] TEMP_out8_0shift = {out8_sign,out8_sign,out8_sign,out8_sign,out8_sign,out8_sign,out8_sign,out8_sign,out8_sign,out8_sign,out8_sign,out8_sign,out8_sign,out8_sign,out8_sign,out8_sign, t8[0], t8[1], t8[2], t8[3], t8[4], t8[5], t8[6], t8[7], t8[8], t8[9], t8[10], t8[11], t8[12], t8[13], t8[14], t8[15]};

    wire [31:0] TEMP_out8_1shift = {out8_sign,out8_sign,out8_sign,out8_sign,out8_sign,out8_sign,out8_sign,out8_sign,out8_sign,out8_sign,out8_sign,out8_sign,out8_sign,out8_sign,out8_sign, t8[0], t8[1], t8[2], t8[3], t8[4], t8[5], t8[6], t8[7], t8[8], t8[9], t8[10], t8[11], t8[12], t8[13], t8[14], t8[15], 1'b0};

    wire [31:0] TEMP_out8_2shift = {out8_sign,out8_sign,out8_sign,out8_sign,out8_sign,out8_sign,out8_sign,out8_sign,out8_sign,out8_sign,out8_sign,out8_sign,out8_sign,out8_sign, t8[0], t8[1], t8[2], t8[3], t8[4], t8[5], t8[6], t8[7], t8[8], t8[9], t8[10], t8[11], t8[12], t8[13], t8[14], t8[15], 1'b0,1'b0};

    wire [31:0] TEMP_out8 = (shift_by_1_num3_down==1'b1
↪ && shift_by_1_num4_down==1'b1) ? TEMP_out8_2shift :
               ((shift_by_1_num3_down==1'b1 &&
↪ shift_by_1_num4_down==1'b0) || (shift_by_1_num3_down
↪ ==1'b0 && shift_by_1_num4_down==1'b1)) ?
↪ TEMP_out8_1shift :
               TEMP_out8_0shift;



    assign temp_output_1_new = TEMP_out1 + TEMP_out2 +
↪ TEMP_out3 + TEMP_out4;
    assign temp_output_2_new = TEMP_out5 + TEMP_out6 +
↪ TEMP_out7 + TEMP_out8;

    assign out_16_x_16_1 = temp_output_1_new;
    assign out_16_x_16_2 = temp_output_2_new;
}
```

Listing 4: Full TIE implementation

References

[1] M. Ghanatabadi, B. Ebrahimi, and O. Akbari, "Accurate and Compact Approximate 4:2 Compressors with GDI Structure," *Circuits, Systems, and Signal Processing*, vol. 42, pp. 4148–4169, 2023.

[2] S. Venkatachalam and S.-B. Ko, "Design of Power and Area Efficient Approximate Multipliers," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 5, pp. 1782-1786, May 2017.

[3] Karri Manikantta Reddy, M.H. Vasantha, Y.B. Nithin Kumar, Devesh Dwivedi, "Design and analysis of multiplier using approximate 4-2 compressor," *AEU - International Journal of Electronics and Communications*, vol. 107, pp. 89-97, 2019, ISSN 1434-8411.

[4] F. Salmanpour, M. Moaiyeri, and F. Sabetzadeh, "Ultra-Compact Imprecise 4:2 Compressor and Multiplier Circuits for Approximate Computing in Deep Nanoscale," *Circuits, Systems, and Signal Processing*, vol. 40, 2021.

[5] T. Abtahi, C. Shea, A. Kulkarni, and T. Mohsenin, "Accelerating Convolutional Neural Network With FFT on Embedded Hardware," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 9, pp. 1737-1749, Sept. 2018.

[6] J. Du, K. Chen, P. Yin, C. Yan, and W. Liu, "Design of An Approximate FFT Processor Based on Approximate Complex Multipliers," in *2021 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, Tampa, FL, USA, 2021, pp. 308-313.

[7] P. Lohray, S. Gali, S. Rangisetti, and T. Nikoubin, "Rounding Technique Analysis for Power-Area & Energy Efficient Approximate Multiplier Design," in *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*, Las Vegas, NV, USA, 2019, pp. 0420-0425.

[8] Z. Yang, J. Han, and F. Lombardi, "Approximate compressors for error-resilient multiplier design," in *2015 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS)*, Amherst, MA, USA, 2015, pp. 183-186.

[9] A. Momeni, J. Han, P. Montuschi, and F. Lombardi, "Design and Analysis of Approximate Compressors for Multiplication," in *IEEE Transactions on Computers*, vol. 64, no. 4, pp. 984-994, April 2015.

[10] S. Asif and Y. Kong, “Performance analysis of Wallace and radix-4 Booth-Wallace multipliers,” in *2015 Electronic System Level Synthesis Conference (ESLsyn)*, San Francisco, CA, USA, 2015, pp. 17-22.