Approximate computing using booth multipliers

A Dissertation

Submitted to the designated
by the Assembly
of the Department of Computer Science and Engineering
Examination Committee

By

Kazantzidis Panagiotis

In partial fulfillment of the requirements for the degree of
DATA AND COMPUTER SYSTEMS ENGINEERING

WITH SPECIALIZATION IN
ADVANCED COMPUTER SYSTEMS

University of Ioannina
School of Engineering
Ioannina 2025

Examination Committee:

- **Aristides Efthymiou**, Assistant Professor, Department of Computer Science and Engineering, University of Ioannina (Supervisor)

- **Yiorgos Tsiatouhas**, Professor, Department of Computer Science and Engineering, University of Ioannina

- **Xrysovalantis Kavousianos**, Professor, Department of Computer Science and Engineering, University of Ioannina

Acknowledgement

First and foremost, i am extremely grateful to my family for their invaluable support both financially and spiritually throughout my academic studies. Furthermore, I would like to thank my supervisor Prof. Aristides Efthimiou and the PhD candidate Ioannis Rizos, for their catalytic help and valuable guidance during this master thesis

Table of Contents

TABLE OF FIGURES

ABSTRACT

In the present master thesis, we shall explore the design and implementation of an approximate Booth multiplier. The primary goal of this research is to investigate how approximate computing techniques can increase the efficiency of power intensive systems that are error tolerant.

With the development of artificial intelligence and big data processing, an unprecedented problem has arisen. These new applications must process massive datasets using increasingly complex computing architectures, creating a critical demand for both energy-efficient systems and highly integrated circuitry. However, high-precision calculations are not always required.

On the contrary, certain small errors can compensate for each other or do not significantly affect the result. Therefore, Approximate Computing (AC) has emerged as a new approach for an energy-efficient design, as well as for increasing the performance of a computing system, with limited loss of accuracy.

Booth encoding is a well-established algorithm used to optimize binary multiplication by reducing the number of partial products generated and thereafter improving computational efficiency. By encoding the multiplier operand, Booth encoding enables the multiplier to handle multiple bits of the multiplicand simultaneously, thereby minimizing the number of operations required for multiplication. This reduction in operations translates to faster computation times, reduced hardware complexity, and lower power consumption, making the combination of Booth encoding with approximate computing an attractive choice for high-performance and error tolerant computing systems.

Finally, the results of this thesis, demonstrate the effectiveness of the approximate Booth multiplier in improving efficiency. By strategically reducing bit precision, we achieved significant improvements in resource utilization, power efficiency, and processing speed, while remaining within acceptable error margins. These findings highlight the potential of approximate computing for designing efficient and high-performance systems, particularly in applications where minor errors are permissible.

ΠΕΡΙΛΗΨΗ

Στην παρούσα διπλωματική εργασία θα διερευνήσουμε τη σχεδίαση και την υλοποίηση ενός προσεγγιστικού πολλαπλασιαστή Booth. Ο πρωταρχικός στόχος αυτής της έρευνας είναι να διερευνήσουμε πώς οι προσεγγιστικές τεχνικές υπολογισμού μπορούν να αυξήσουν την αποδοτικότητα των συστημάτων υψηλής έντασης ισχύος που είναι ανθεκτικά σε σφάλματα.

Με την ανάπτυξη εφαρμογών όπως της τεχνητής νοημοσύνης και της επεξεργασίας δεδομένων μεγάλου όγκου, έχει προκύψει ένα άνευ προηγουμένου πρόβλημα. Οι νέες αυτές εφαρμογές αντιμετωπίζουν έναν τεράστιο όγκο δεδομένων χρησιμοποιώντας ολοένα και πιο πολύπλοκες αρχιτεκτονικές, απαιτώντας ενεργειακά αποδοτικά συστήματα καθώς και ισχυρά ενσωματωμένα κυκλώματα. Ωστόσο στην πράξη δεν χρειάζονται πάντα υπολογισμοί υψηλής ακρίβειας.

Αντιθέτως, ορισμένα μικρά σφάλματα μπορούν εύκολα να αντισταθμίσουν το ένα το άλλο ή δεν επηρεάζουν σημαντικά το αποτέλεσμα. Ως εκ τούτου, οι προσεγγιστικοί υπολογισμοί (Approximate Computing, AC) έχουν αναδειχθεί ως μια νέα προσέγγιση για έναν ενεργειακά αποδοτικό σχεδιασμό, καθώς και για την αύξηση της απόδοσης ενός υπολογιστικού συστήματος, με περιορισμένη απώλεια ακρίβειας.

Η κωδικοποίηση Booth είναι ένας καλά εδραιωμένος αλγόριθμος που χρησιμοποιείται για τη βελτιστοποίηση του δυαδικού πολλαπλασιασμού μέσω της μείωσης του αριθμού των παραγόμενων μερικών γινομένων και στη συνέχεια τη βελτίωση της απόδοσης. Με την κωδικοποίηση της πράξης του πολλαπλασιαστή, η κωδικοποίηση Booth επιτρέπει στον πολλαπλασιαστή να χειρίζεται ταυτόχρονα πολλά bits της πράξης του πολλαπλασιαστή, ελαχιστοποιώντας έτσι τον αριθμό των πράξεων που απαιτούνται για τον πολλαπλασιασμό. Αυτή η μείωση των πράξεων οδηγεί σε ταχύτερους χρόνους υπολογισμού, μειωμένη πολυπλοκότητα υλικού και χαμηλότερη κατανάλωση ενέργειας, καθιστώντας τον συνδυασμό της κωδικοποίησης Booth με τον προσεγγιστικό υπολογισμό μια ελκυστική επιλογή για υπολογιστικά συστήματα υψηλής απόδοσης με υψηλή ανοχή σε σφάλματα.

Τέλος, τα αποτελέσματα αυτής της διπλωματικής εργασίας αποδεικνύουν την αποτελεσματικότητα του προσεγγιστικού πολλαπλασιαστή Booth στην αύξηση της αποδοτικότητας. Αυξάνοντας τον αριθμό των bits που παραλείπονται, πετύχαμε μειώσεις στη χρήση πόρων, στην κατανάλωση ενέργειας και στο χρόνο εκτέλεσης , διατηρώντας μια αποδεκτή ανοχή σε σφάλματα. Τα αποτελέσματα αυτά αναδεικνύουν τις δυνατότητες του προσεγγιστικού υπολογισμού για τη σχεδίαση αποτελεσματικών και υψηλής επίδοσης συστημάτων, ιδίως σε εφαρμογές όπου επιδέχονται μικρά σφάλματα.

CHAPTER 1

INTRODUCTION

1.1 Objective

Over the last decades, the size of transistors has decreased exponentially, as predicted by Moore's law, leading to continuous improvements in the performance and power efficiency of integrated circuits. However, on the nanometer scale, the supply voltage cannot be further reduced, which has led to a significant increase in power density. Hence, a percentage of transistors in a circuit must be turned off to combat thermal problems. These deactivated transistors are called "dark silicon". The area of "dark silicon" can reach more than 50% of the initial area on a 8 nm technology. This indicates a growing challenge to improve circuit performance and power efficiency when using traditional technologies. To address this issue, new design methodologies have been explored, including multicore architectures, external integration, improved arithmetic algorithms and approximate computing(AC).

In digital computing, arithmetic operations form the backbone of virtually all computational tasks, influencing everything from basic calculations to complex algorithmic processes. These operations, including addition, subtraction, multiplication, and division, are fundamental for executing algorithms and processing data. Innovations in arithmetic algorithms aim to streamline these operations, reducing both computation time and resource consumption. One such advancement is the Booth multiplier, which introduces Booth encoding to optimize binary multiplication. Booth's algorithm reduces the number of partial products and simplifies the addition process, significantly enhancing both speed and hardware efficiency.

AC is based on the fact that many applications, such as multimedia, identification, classification and machine learning, are tolerant of certain errors. Some errors do not cause a noticeable degradation in the quality of image, audio and video processing. In addition, the input data to a digital system usually contains noise and is quantized, so there is already a threshold in the accuracy and truth of representation of useful information. In the case of probabilistic computation, they remodel numerical functions into random binary bit streams using simple logic gates, where minor errors do not lead to a critically different result. Finally, many applications, including machine learning, rely on iterative tuning. This process can mitigate or compensate for the effects of trivial errors. Thus, AC is a promising technique that benefits multiple different fault-tolerant applications.

The main objective of this thesis is to design and implement a novel approximate multiplier based on Booth encoding. Initially we evaluate its accuracy based on mean relative error distance, a metric commonly used in AC research. The multiplier is built on a Field-Programmable Gate Array (FPGA) platform and its performance was thoroughly evaluated using metrics such as latency, throughput and power efficiency. We then evaluate its performance in an error-tolerance application, a neural network framework. Specifically, we replace all multiplication operations in the training phase of a Multi-Layer Perceptron model and compare the model's accuracy when using our approximate multiplier and when the exact multiplication is performed. The MLP models were run an an ARM core present in the same FPGA platform used for evaluating the proposed aproximate multiplier and their practical performance is discussed.

1.2 Thesis outline

Chapter 2 provides a foundational background for understanding the key concepts and technologies fundamental to this thesis. It begins with a discussion on compute intensive applications, emphasizing the challenges and opportunities associated with processing and analyzing vast amounts of information. Following this, the chapter delves into multiplication encoders, exploring tools used to

optimize binary multiplication. This section will cover different encoding techniques, with a particular focus on Booth encoding. The chapter then introduces approximate computing, a calculation concept that allows for controlled inaccuracies to achieve faster and more efficient calculations. Then it examines various evaluation metrics, such as mean error and accuracy, which are essential for assessing the performance and reliability of these techniques. The chapter also discusses the rationale behind selecting the Multilayer Perceptron (MLP) as the neural network for this study. Finally, it highlights the necessity of explicitly handling floating-point operations, particularly the extraction of floating-point components to enable accurate multiplication between two float numbers.
Chapter 3 focuses on the development of a novel algorithm designed to enhance computational efficiency. It introduces a new modified Booth encoding method aimed at improving the speed and efficiency of binary multiplication operations. Additionally, it explores the integration of this new Booth encoding with approximate computing, demonstrating how approximations can enable faster calculations and reduce computational demands while maintaining acceptable accuracy levels. It provides a detailed analysis of the new algorithm, explaining its design, implementation, and the thought process behind its development. This includes how the algorithm addresses specific challenges and leverages the strengths of both Booth encoding and approximate computing to achieve efficient and accurate results. Following this, it discusses the process of benchmarking the hardware using Vitis, highlighting the steps taken to evaluate performance and efficiency. The chapter also explores the application of a Multilayer Perceptron (MLP) in the experiments, explaining its role in testing the proposed algorithms
Chapter 4 focuses on the evaluation of the implementation of the proposed algorithm. It begins with a detailed description of the hardware used in this thesis, specifically the Zynq-7000 FPGA on the PYNQ-Z2 board, outlining the components and configurations essential for supporting the new modified Booth encoder and approximate computing. The chapter then introduces the tools utilized for development and testing, providing an overview of their functionalities and relevance to the experiments. The experiments are divided into two main categories: Multiplier testing and MLP testing. Multiplier testing includes an analysis of LUT utilization, schematics complexity, medium error distance of a

multiplication (measured by MRED), and on-chip power consumption, providing insights into the hardware's performance and resource usage. MLP testing focuses on evaluating accuracy, error tolerance , and time taken, demonstrating the impact of the new algorithms on computational efficiency and accuracy. This chapter provides a comprehensive evaluation of the practical implementation and effectiveness of the proposed methods.

Chapter 5 summarizes the findings and contributions of this thesis while outlining potential directions for future research. It begins with conclusions that highlight the key outcomes of the experiments, focusing on the effectiveness of the modified Booth encoding and approximate computing techniques in improving computational efficiency, resource utilization, and accuracy. The chapter then transitions to future thoughts, suggesting opportunities for further exploration, such as testing on larger FPGA boards, investigating higher-radix designs, and integrating the algorithms into more complex neural networks and real-world applications. This chapter emphasizes the significance of the research and its potential to advance the field of approximate computing and hardware-accelerated machine learning.

CHAPTER 2

BACKGROUND

2.1 Approximate computing basics and applications

AC is considered suitable for many error-tolerant applications, such as image processing and machine learning, aiming to increase performance and energy efficiency. [1]

Approximation techniques of algorithm, architecture and circuit level have been used collaboratively in the design of an energy-efficient programmable vector processor for recognition and data mining. These design techniques achieve a 16.7%-56.5% reduction in power consumption compared to a traditional design without any loss of quality, and a 50.0%-95.0% with output quality insignificantly reduced. For example, in mainstream image processing applications, sharpening, smoothing and image multiplication are considered suitable for assessing the quality of approximate adders and unsigned multipliers [2] [3].

Approximate adders and multipliers have been integrated into deep learning accelerators to reduce latency and save energy. In large-scale convolutional neural network (CNN) and deep neural network (DNN) applications, 32-bit floating-point multipliers are used. In many applications these multipliers have now been replaced by 16-bit multipliers with constant error compensation. A reduction of up to 83.6% in circuit area and 86.4% in power consumption is achieved. Also, in some applications the approximate adders and multipliers have been integrated

into a device with varying levels of accuracy for different configurations determined on the fly according to the application requirements. In this way, different performance and power improvements can be achieved by compensating for a variety of levels of processing quality [4] [5] [6].
An approximate computational circuit can be obtained by using the technique of Voltage Overscaling VOS, by redesigning a logic circuit to an approximate one and using a simplification algorithm [7]. Using VOS, a lower supply voltage is provided to reduce the power consumption of a circuit without having to change the circuit structure. However, the reduced voltage increases the critical path delay, potentially resulting in timing errors. Thus, the output may be incorrect due to timing constraints. Moreover, the error characteristics of such an approximate operation are non-deterministic, as they are affected by parametric variations. When the most significant bits (MSB) are affected, the output error could be larger.
Usually, an approximate design is derived from an exact circuit by modifying, removing, or adding certain elements. For example, in a mirror adder, which is a type of full adder known for its symmetrical structure and efficient transistor usage, certain transistors can be removed to create an approximate version. The mirror adder typically consists of two mirrored halves that generate the sum and carry outputs simultaneously, making it faster and more power-efficient than traditional adder designs. By selectively removing transistors, the circuit becomes simpler, reducing power consumption and increasing speed at the cost of introducing minor errors in the output [8].
In addition, an approximate circuit can be obtained by simplifying the truth matrix or Karnaugh map (K-Map). This method leads to circuits with deterministic error characteristics. However, due to the same structure and basic design principles, hardware improvements are minimal, particularly when high accuracy is required.
Compared to addition and subtraction, the multiplication, the division and the calculation of the square root are more complex. Hence, their operations can be converted into some simpler operations. Mitchell's algorithm, based on the binary logarithm, provides an efficient way to implement multiplication and division using only adders and subtractors. The algorithm approximates the logarithm of a number by interpreting its binary representation, allowing multiplication to be

performed as an addition of logarithms and division as a subtraction of logarithms. For multiplication, the algorithm computes the sum of the logarithms of the operands and then converts the result back to a linear value using an antilogarithm approximation. Similarly, for division, it subtracts the logarithms and converts the result back [7].

Mitchell's algorithm is the origin of most current simplification algorithms for approximate design of multipliers and divisors alongside functional iteration-based algorithms for designing divisors. By using algorithmic simplification, the performance and energy efficiency of a numerical circuit can be significantly improved due to the simplification of the basic circuit structure. However, the accuracy of such a design is relatively low. Achieving high accuracy requires many boards, which can limit the efficiency of the hardware. In practice, several approximation techniques are used simultaneously in a hybrid approximation circuit [9].

2.2 Evaluation and metrics

Later in this dissertation we will discuss approximate multipliers. Thus we need to analyze basic terminologies where they relate to error and their performance. Two basic error metrics are **error rate** (ER) and **error distance** (ED). ER indicates the probability of producing an incorrect result. ED shows the arithmetic difference between the approximate and the exact result. Given the approximate and the exact result M′ and M, respectively, the ED is calculated as follows: $ED = |M' - M|$ ER is calculated as : $\frac{|M'-M|}{M} \cdot 100$. Furthermore, the **relative error distance** (RED) shows the relative difference compared to the exact result, which is given by $RED = \left|\frac{ED}{M}\right|$. For two sets of inputs that result in the same ED, the one that produces the least accurate result, M, will result in a larger RED. Similar to the mean values of all obtained EDs and REDs, called **mean error distance** (MED) and **mean relative error distance** (MRED) are often used to evaluate the accuracy of an approximate design. Here are the functions for calculating MED and MRED.

$$MED = \sum_{i=0}^{N} ED_i \cdot P(ED_i)$$

$$MRED = \sum_{i=0}^{N} RED_i \cdot P(RED_i)$$

where N is the total number of input combinations for a circuit, and ED_i and RED_i are ED and RED for the i-th input number, respectively. $P(ED_i)$ and $P(RED_i)$ are the probabilities of ED_i and RED_i occurring, which are the probabilities of the i-th input combination. NMED is defined as the normalization of MED to the maximum output of the exact correct circuit.

2.3 Multiplication Primitives

Multiplication is less common than addition, but it is still necessary for microprocessors, digital signal processors and GPUs. The most basic form of multiplication consists of calculating the product of two unsigned (positive) binary numbers. This can be achieved using the conventional technique that is also taught in primary school. For example, the multiplication of two positive 6-bit positive binary integers, 25 and 39, is shown in Figure 1.

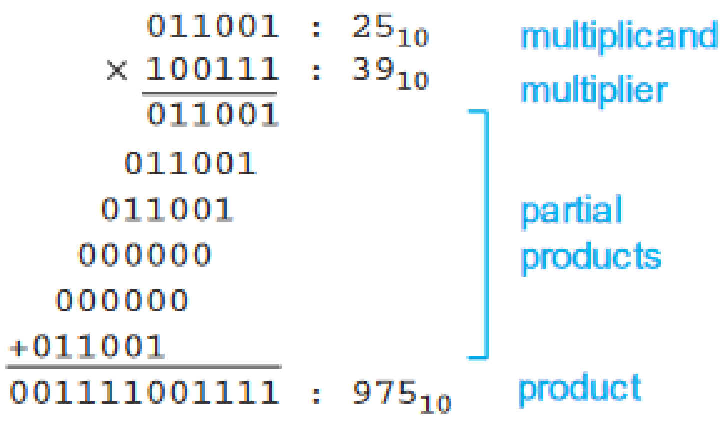


Figure 1 Multiplication of 25 * 39 [10]

The M × N-bit P = Y × X multiplication can be viewed as forming N partial sums of length M bits each, and then summing the appropriately shifted partial sums to produce an outcome P of length M + N bits. Binary multiplication is equivalent to a logical AND operation. Therefore, the creation of partial sums is the logical AND operation of the appropriate bits of the multiplier and the multiplicand. Each column of partial sums must then be added together and, if there are any carry-overs, they must be passed to the next column. We denote the multiplier as

$Y=\{\Upsilon_{M-1},\Upsilon_{M-2},...,\Upsilon_1,\Upsilon_0\}$and the multiplicand $X=\{X_{N-1},X_{N-2},...,X_1,X_0\}$.For unsigned multiplication, the product is given in Equation (2.1). Figure 2 illustrates the creation, shifting, and addition of partial sums in a 6 × 6 bit multiplier.

$$P=\left(\sum_{j=0}^{M-1} y_j \cdot 2^j\right)\cdot\left(\sum_{i=0}^{N-1} x_i \cdot 2^i\right)=\sum_{i=0}^{N-1}\sum_{j=0}^{M-1} x_i y_j 2^{i+j} \qquad (2.1)$$

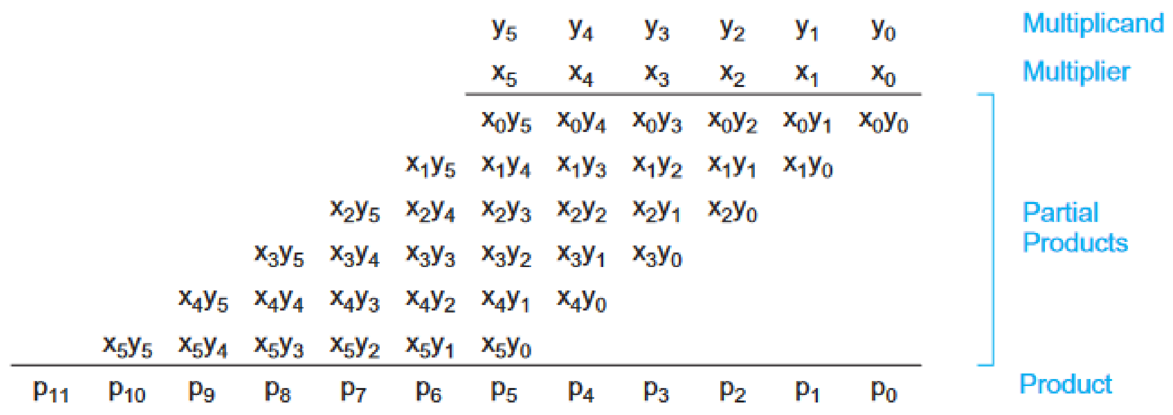

y_5 y_4 y_3 y_2 y_1 y_0
Multiplicand
x_5 x_4 x_3 x_2 x_1 x_0
Multiplier
x_0y_5 x_0y_4 x_0y_3 x_0y_2 x_0y_1 x_0y_0
x_1y_5 x_1y_4 x_1y_3 x_1y_2 x_1y_1 x_1y_0
x_2y_5 x_2y_4 x_2y_3 x_2y_2 x_2y_1 x_2y_0
Partial Products
x_3y_5 x_3y_4 x_3y_3 x_3y_2 x_3y_1 x_3y_0
x_4y_5 x_4y_4 x_4y_3 x_4y_2 x_4y_1 x_4y_0
x_5y_5 x_5y_4 x_5y_3 x_5y_2 x_5y_1 x_5y_0
p_{11} p_{10} p_9 p_8 p_7 p_6 p_5 p_4 p_3 p_2 p_1 p_0
Product

Figure 2 Multiplication of 2 numbers with 6bit length each and the generation of their partial products [10]

Larger multiplications can be better illustrated by using scatter plots. Figure 3 shows a scatter plot of a 16 × 16 multiplier. Each dot represents one bit which can be 0 or 1. Partial products are represented by a horizontal row of cells arranged in a grid pattern, which are shifted according to their significance. The multiplier bits used to generate the partial sums are shown on the right.

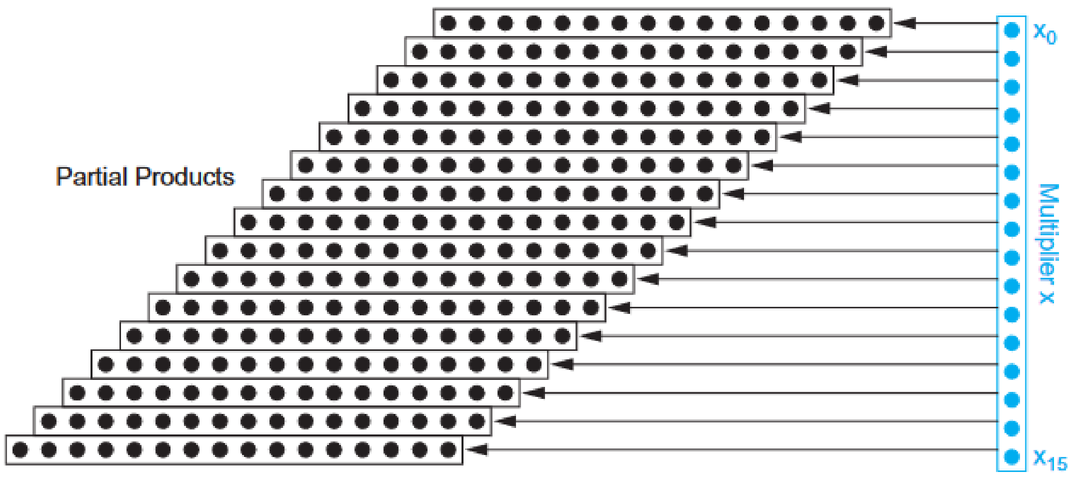


Figure 3 Partial products represented by dots [10]

There are several techniques that can be used for carry propagation. In general, the choice is based on factors such as delay, power, energy, area and complexity of the circuit design. An easy approach is to use an M+1-bit adder (carry-propagate adder CPA) to add the first two partial sums, then another CPA to add the third partial sum to the current sum, and so forth. Such an approach requires N - 1 CPAs and is slow even if a fast CPA is used. There are more efficient parallel approaches available that use some kind of matrix or tree of complete adders to sum the partial products. We start with a simple matrix for unsigned multipliers and then modify the matrix to handle signed numbers of complement of 2 using the Baugh-Wooley algorithm [11]. The number of partial sums being added can be reduced using the Booth encoding, and the number of logical levels required to perform summation can be reduced with Wallace trees (Figure 4). Unfortunately, Wallace trees are complex to design and have long, uneven wires, so hybrid table/tree structures may be more attractive. For the sake of thoroughness, we consider a serial propagation architecture. This was once popular when gates were relatively expensive, but now it is scarcely useful.

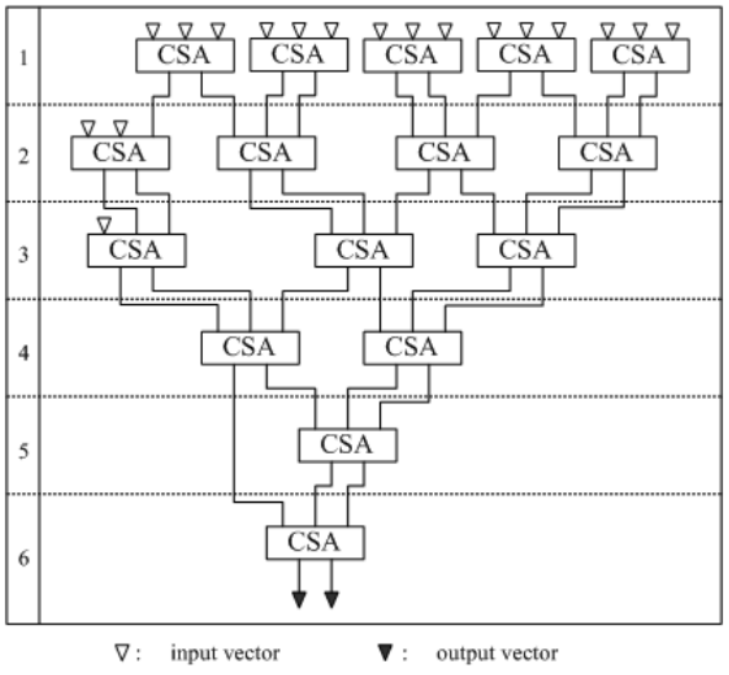


Figure 4 Wallace tree with carry save adders

2.4 Booth multiplication

The original Booth's algorithm, introduced by Andrew D. Booth in his seminal work *"A Signed Binary Multiplication Technique"*, was designed to reduce the number of partial products in signed binary multiplication. The key insight lies in recognizing that consecutive ones in the multiplier do not require individual partial products. Instead, they can be consolidated into fewer operations. For example, a sequence such as 0111...110 can be transformed into 1000...000 - 000...10, effectively minimizing the required computations.

In practice, the algorithm processes the multiplier bits from right to left, examining one bit at a time (radix-2 encoding) and generating partial products of ±Y or 0 based on the current and previous bits(Figure 5). However, it comes with a trade-off, since the number of partial products varies depending on the multiplier's value. This unpredictability presents significant challenges for hardware implementation. When designing critical components like Wallace tree adders, engineers must account for worst-case scenarios since the actual number of partial products cannot be predetermined.

Xi	Xi—1	Operation
0	0	0
0	1	+Y
1	0	-Y
1	1	0

Figure 5 Selection table of partial sums of radix 2

Figure 6 demonstrates a radix-2 Booth encoded multiplication between two signed binary numbers Y=011001 and X=100111. At the right, X is aligned vertically from the LSB to the MSB for demonstration purposes. The multiplication process begins by analyzing pairs of bits starting from the implicit X_{-1} bit and moving through X_6. We must clarify that the X_{-1} and X_6 bit are always set to 0. Then each bit pair [X_i X_{i-1}] determines the new partial product of the multiplicand Y according to Booth's radix 2 encoding rules on Figure 5. Each partial product shifts left according to its bit position while sign extension preserves the correct two's complement representation throughout the calculation The complete set of generated partial products is then summed to produce the final multiplication result.
For instance, when the algorithm detects a transition from 0 to 1 in the multiplier bits, it generates a negative partial product (-Y), while a transition from 1 to 0 produces a positive partial product (+Y).

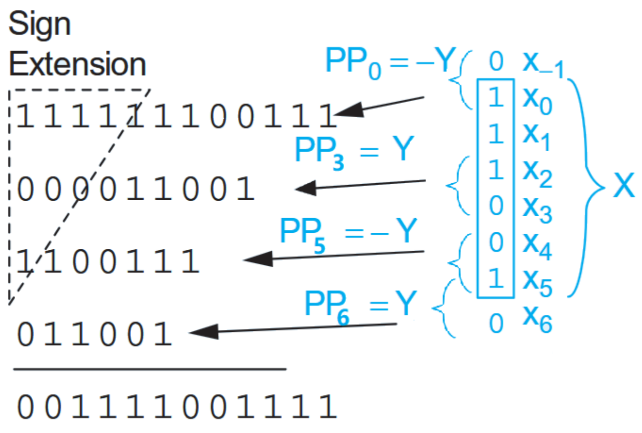


Figure 6 Example of a booth encoder with radix-2

Building upon the fundamental Booth algorithm, higher radix multipliers were introduced to further optimize binary multiplication by reducing the number of partial products. These advanced techniques examine multiple bits of the multiplier simultaneously. The most widely adopted is radix-4, which examines two bits plus an overlapping third bit from the previous step. This modification reduces the number of partial products to approximately N/2, where N is the bit-width of the multiplier. The possible partial products in radix-4 are 0, ±Y, ±2Y, with 2Y easily implemented as a left shift. Crucially, the problematic 3Y multiple is avoided by leveraging signed arithmetic, where 3Y is expressed as 4Y - Y, shifting the complexity to the next higher-weight partial product.
The selection of partial products in radix-4 Booth encoding follows a predefined table based on triplets of multiplier bits(Figure 7). Each partial product is shifted two positions to the left relative to the previous one, reflecting its fourfold increase in weight. Sign extension is applied for negative multiples, ensuring correct two's complement representation.

X_{2i+1}	X_{2i}	X_{2i-1}	Operation
0	0	0	0
0	0	1	+Y
0	1	0	+Y
0	1	1	+2Y
1	0	0	-2Y
1	0	1	-Y
1	1	0	-Y
1	1	1	0

Figure 7 Selection table of partial sums of radix 4

Figure 8 depicts a radix-4 Booth encoded multiplication between two signed binary numbers Y=011001 and X=100111. The multiplication iterates overlapping 3-bit windows of the multiplier [X_{i+1} X_i X_{i-1}] to determine the new partial product of the multiplicand Y .Each window selects from five possible actions according to the radix-4 Booth encoding rules as aforementioned(Figure 7).

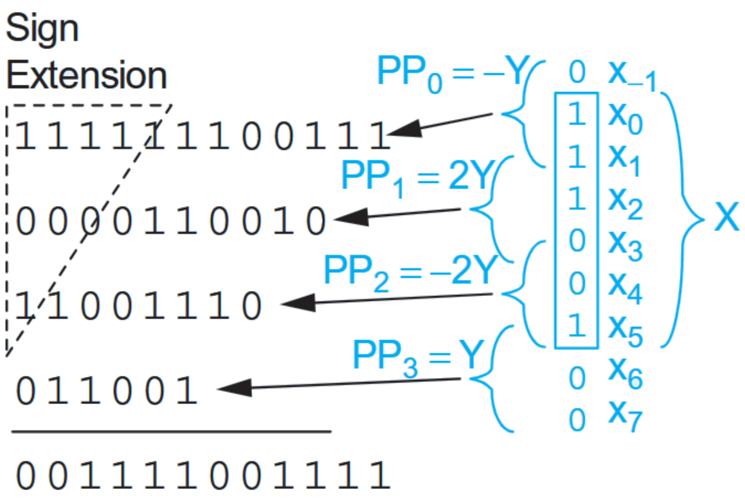


Figure 8 Example of a booth encoder with radix-4

In a common radix-4 Booth encoded multiplier design, each group of 3 bits (a pair, along with the most significant bit of the previous pair) is encoded in selection signals (SINGLE i, DOUBLE i, and NEG i, Figure 9) and connected to the partial product line, as shown in Figure 10.

Inputs			Partial Product	Booth Selects		
x_{2i+1}	x_{2i}	x_{2i-1}	PP_i	SINGLE_i	DOUBLE_i	NEG_i
0	0	0	0	0	0	0
0	0	1	Y	1	0	0
0	1	0	Y	1	0	0
0	1	1	$2Y$	0	1	0
1	0	0	$-2Y$	0	1	1
1	0	1	$-Y$	1	0	1
1	1	0	$-Y$	1	0	1
1	1	1	-0 $(= 0)$	0	0	1

Figure 9 Selection table of partial sums [10]

The Y multiplier is allocated to all rows. The selector signals control the Booth selectors that choose the appropriate multiple of Y for each partial product. The Booth selectors replace the AND gates of a simple matrix multiplier to compute the i-th partial product. Figure 10 shows a classic encoder design with a Booth selector. Y is expanded by M+1 bits with zeros. Depending on SINGLE_i and

DOUBLEᵢ, either 0, Y, or 2Y is selected. The negative partial products must be in the form of complement by two (i.e., invert and add 1). If NEGᵢ is activated, the partial product is inverted. The extra ace can be added to the next row to avoid the need for CPA.

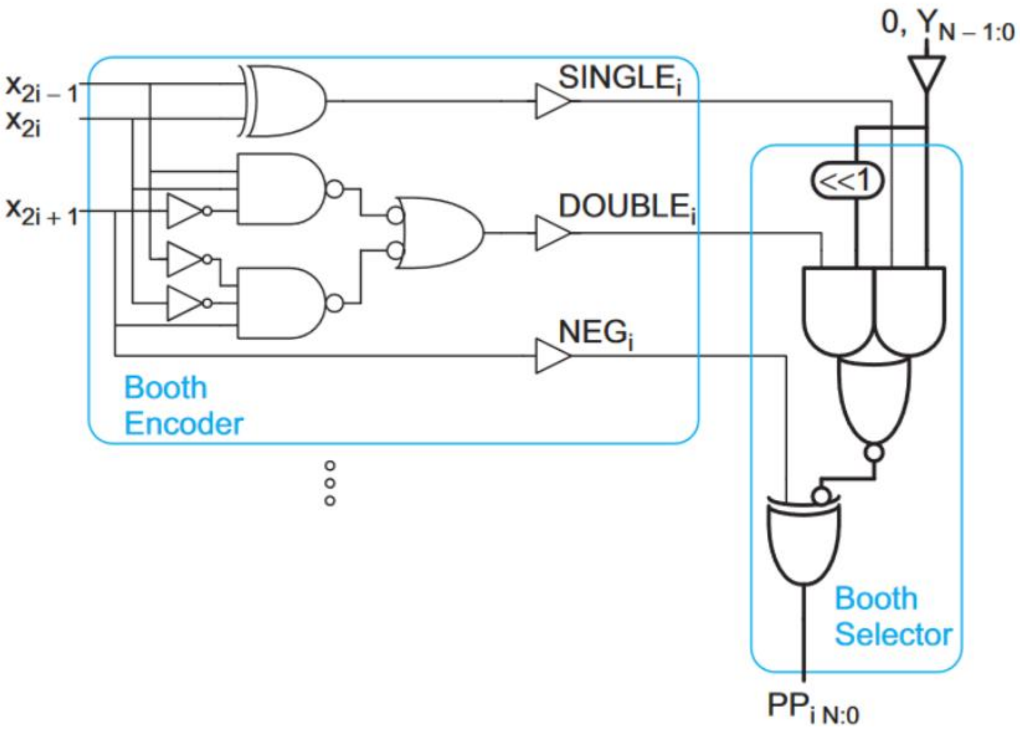


Figure 10 Radix 4 booth encoder with selection signals

Higher-radix versions, such as radix-8 or radix-16, further reduce the number of partial products but introduce more complex multiples (e.g., ±3Y, ±4Y), which often outweigh their benefits due to increased hardware overhead.

In this work, we employ a modified version of the original Booth algorithm (radix-2) rather than radix-4. Given that our primary focus is clarity and predictable hardware behavior, radix-2 provides a more straightforward solution.

2.5 Approximate booth multipliers and error compensation

The modified (or radix-4) Booth encoder is commonly used in the design of approximate multipliers. Initially targeting a fixed-length signed multiplier, a widely used method is to cut off a chunk of the LSB's partial products (PPs) to create an output with the same width as the input. This truncation makes it easier to collect PPs but introduces a large error. Therefore, many error compensation schemes have been proposed.

The Broken booth multiplier (BBM) skips the computational cells of the adder to the right of a VBL line (Vertical breaking line Figure 11), while truncating some k less significant bits of the input results in a k-Truncated booth multiplier (TBM-k). The TBM is considered as a basic design for comparing booth multipliers. In the tests where they were performed, the value of k was approximated experimentally and ranged from 2 to 6. [12]

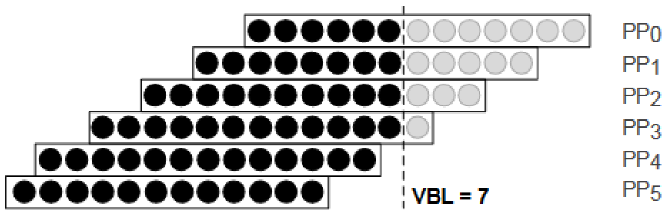
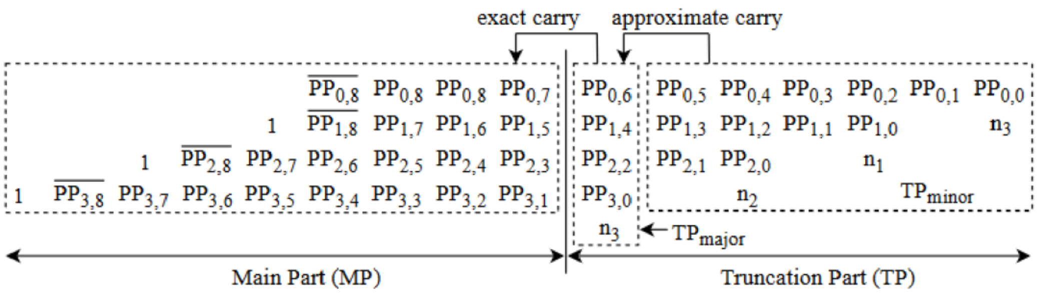


Figure 11 Broken booth multipler [12]

A fixed-width Booth multiplier divides the PP array into two regions: one containing the most significant bits (MSBs) called the Main Part (MP), and the other containing the LSBs called the Truncation Part (TP). The TP is further subdivided into TP_major and TP_minor. The final result is obtained by summing the MP and carry terms generated from the TP.



exact carry
approximate carry
$\overline{PP_{0,8}}$ $PP_{0,8}$ $PP_{0,8}$ $PP_{0,7}$ $PP_{0,6}$ $PP_{0,5}$ $PP_{0,4}$ $PP_{0,3}$ $PP_{0,2}$ $PP_{0,1}$ $PP_{0,0}$
1 $\overline{PP_{1,8}}$ $PP_{1,7}$ $PP_{1,6}$ $PP_{1,5}$ $PP_{1,4}$ $PP_{1,3}$ $PP_{1,2}$ $PP_{1,1}$ $PP_{1,0}$ n_3
1 $\overline{PP_{2,8}}$ $PP_{2,7}$ $PP_{2,6}$ $PP_{2,5}$ $PP_{2,4}$ $PP_{2,3}$ $PP_{2,2}$ $PP_{2,1}$ $PP_{2,0}$ n_1
1 $\overline{PP_{3,8}}$ $PP_{3,7}$ $PP_{3,6}$ $PP_{3,5}$ $PP_{3,4}$ $PP_{3,3}$ $PP_{3,2}$ $PP_{3,1}$ $PP_{3,0}$ n_2 TP_{minor}
n_3 TP_{major}
Main Part (MP)
Truncation Part (TP)

Figure 12 8x8 fixed width modified booth multiplier

To mitigate truncation error, the BM04 architecture introduces a basic error correction mechanism by examining whether a partial product row is entirely zero or not. This is captured by a Boolean signal $\overline{zero}_i$ which equals 1 if the i-th row contains any non-zero bits. The compensation term is generated by summing selected $\overline{zero}_i$ values, enabling lightweight correction without complex adders or predictors. The approximate partial product is generated by the equation:

$$\sigma = \left\lfloor 2^{-1}\left(\sum_{i=0}^{n/2-2} \overline{\text{zero}}_{\,i} + 1\right)\right\rfloor \tag{2.2}$$

To enable tunable accuracy, BM07 allows adjusting the number of most significant PP columns retained via a parameter ω. Based on this, different thresholds are applied to determine whether and how to compensate the error induced by truncation. [13]

BM11 introduces a new approach to error handling: instead of only estimating magnitude, it ensures that errors are centered around zero, eliminating systematic bias. This is achieved using an odd-even merge sorting network, where the carry signals from TP rows are ranked, and a central (e.g., median) signal is selected to represent the correction.

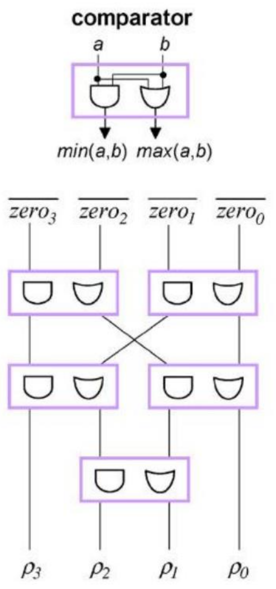


Figure 13 Sorting network odd-even merge

The aforementioned circuit sorts the inputs in ascending order from r3 to r0. According to the authors, utilizing the r2 output suffices to approximate the presence of the carry signal. Following the removal of certain gates and subsequent partial simplification, the circuit assumes the form shown in Network 2. This network is designated as the SC-generator. [14]

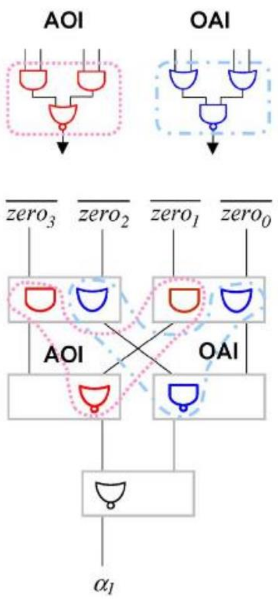


Figure 14 Simplified sorting network with the use of gates

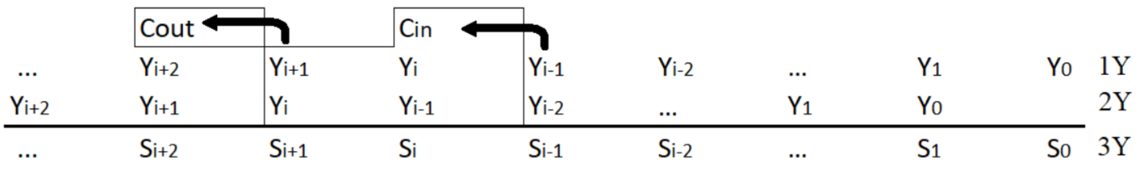
Another Booth multiplier was designed in [15] based on a probabilistic error estimation approach, specifically addressing the discrepancy between estimated and actual values. This multiplier architecture is termed as Probability Estimation Bias Multiplier (PEBM). The number of accumulated partial product columns varies according to the desired hardware-accuracy trade-off ratio. The error compensation formula is derived through probability analysis rather than time-consuming exhaustive simulation. The carry signal generated by the truncated portion (TP) is approximated as follows:

$$\sigma = \left\lfloor 2^{-1}\left(\sum_{i=0}^{n/2-1-\lfloor\omega/2\rfloor} z_i - 1\right)\right\rfloor \tag{2.3}$$

$Z_i = P_{o,n/2-1} + n_{n/2-1}$ when i = n/2 - 1, otherwise $Z_i = \overline{zero}$

Another multiplier variant specifically targets the computation of triple multiples (3Y). To mitigate the additional latency inherent in radix-8 Booth algorithms, the design employs an approximate adder for calculating these triple multiplicands [10]. This is achieved through strategic modifications to the Karnaugh map, which simplifies the logic function (Figure 15-16-17). The architecture subsequently incorporates a Wallace tree structure combined with a truncation technique for partial product (PP) accumulation. [16]

The most efficient approximate radix-8 Booth multiplier, designated ABM2_R15, implements a 15-bit truncation scheme to create a fixed-width multiplier. This optimized configuration is formally recognized and referenced as ABM2.



Cout
Cin
... Yi+2 Yi+1 Yi Yi-1 Yi-2 ... Y1 Y0 1Y
Yi+2 Yi+1 Yi Yi-1 Yi-2 ... Y1 Y0 2Y
... Si+2 Si+1 Si Si-1 Si-2 ... S1 S0 3Y

Figure 15 Addition between 1Y and 2Y to calculate 3Y

$C_{out}S_{i+1}S_i$		y_iy_{i-1}			
		00	01	11	10
$C_{in}y_{i+1}$	00	000	001	100	011
	01	010	011	110	101
	11	011	100	111	110
	10	001	010	101	100

Figure 16 Truth table of a 2 bit adder

$C_{out}S_{i+1}S_i$		y_iy_{i-1}			
		00	01	11	10
$C_{in}y_{i+1}$	00	000	001	100	101
	01	010	011	110	111
	11	011	010	111	110
	10	001	000	101	100

Figure 17 Truth table of an approximate 2 bit adder

Below are the Boolean functions derived from the Karnaugh map optimization shown in Figure 16

$$C_{\text{out}} = \left((y_{i-1} \vee y_{i+1} \vee C_{in}) \wedge y_i \right) \vee (C_{in} \wedge y_{i+1} \wedge y_{i-1}),$$

$$S_{i+1} = \left(\left(\left(\overline{y_i \vee y_{i-1}} \right) \vee \left(\overline{C_{in}} \wedge y_{i-1} \right) \right) \wedge y_{i+1} \right) \vee \left(C_{in} \wedge \overline{y_i} \wedge y_{i-1} \right) \vee \left(\overline{C_{in}} \wedge y_i \wedge \overline{y_{i-1}} \right) \right) \wedge \overline{y_{i+1}} \right),$$

$$S_i = C_{in} \oplus y_i \oplus y_{i-1},$$

And below Is the Karnaugh output for the simplified circuit in Figure 17

$$C_{out} = y_i,$$
$$S_{i+1} = y_{i+1},$$
$$S_i = C_{in} \oplus y_i \oplus y_{i-1}.$$

We can easily observe the simplification through the boolean functions.

In this thesis, we adopt a modified approach based on the Broken Booth Multiplier architecture, deliberately omitting the multiplication of least significant bits as a strategic design choice. While various error compensation and approximation techniques exist in the literature, including probabilistic estimation, truncation methods, and hybrid exact-approximate circuits, our implementation focuses on computational efficiency through controlled precision reduction.

2.6 Basics of neural networks

One of the attractive features of artificial neural networks is their capability to adapt themselves to special environment conditions, by "training" their connection strengths (weights). Especially, feed-forward neural networks with neurons arranged in layers, are widely used in computational or industrial fields. Furthermore, as VLSI technology has developed, the interest in implementing them in hardware is growing.

A Multilayer Perceptron (MLP) is a class of feedforward neural network that processes information through a series of interconnected layers. The network consists of an input layer that receives raw data, one or more hidden layers that transform the data through weighted connections and nonlinear activation functions, and an output layer that produces the final prediction. During operation, input signals propagate forward through the network, where each neuron computes a weighted sum of its inputs, applies an activation function, and passes the result to subsequent layers. The weights are iteratively adjusted during training via backpropagation, which minimizes prediction errors by propagating gradients backward through the network and updating parameters using optimization techniques such as stochastic gradient descent.

Figure 18 shows a multilayer perceptron (simply denoted as an "MLP" in the following). Each neuron in a layer is connected to all neurons in the adjacent layers through uni-directional links (synaptic weights). The first and the last layers are called the input and output layers respectively, and one between them is called a hidden layer. In this dissertation, we deal with only MLPs which have one hidden layer. The output of each neuron (o_i) is given by:

$$o_i = f(X_i) \tag{2.4}$$

$$X_i = \sum_{j=0}^{N_{pre}} w_{ij} \cdot u_j \tag{2.5}$$

where w_{ij} is the value of the synaptic weight from the j-th neuron in the preceding layer to the i-th neuron (i, j) is called to be the index of the weight while N_{pre} is the number of the neurons in the preceding layer connected to the i-th

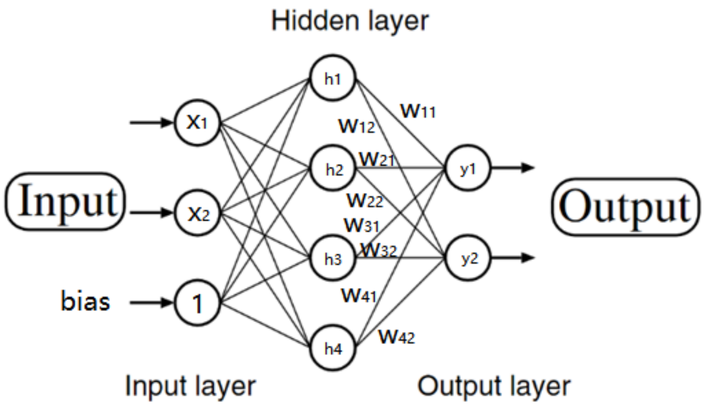


Figure 18 3-layer multilayer perceptron

neuron, u_j is the output of the j-th neuron in the preceding layer (j is called to be the index of the neuron). The number of neurons in each layer is determined by the dimensionality of the input features and the complexity of the desired output, with wider layers typically required to model higher-dimensional transformations while balancing computational efficiency against representational capacity. This architectural choice reflects a trade-off between capturing sufficient nonlinear relationships in the data and avoiding excessive parameterization that could lead to overfitting. Finally, w_{i0} is the synaptic weight connected to the input u_0 = 1 corresponding to the threshold, X_i is called the "inner potential" of the i-th neuron, and f is an activation function. For example, the sigmoid activation function of a neuron is defined by

$$f(x) = \frac{1}{1+exp(-x)} \tag{2.6}$$

The choice of activation function (e.g., ReLU, sigmoid, tanh) determines the network's nonlinear modeling capability and training stability, with selection based on the gradient propagation needs and output range requirements for the specific task and dataset.

The learning process called "back-propagation" is a supervised learning method used to train MLPs by minimizing an error function through gradient descent. Let O be a set of indices of the neurons in the output layer, and let P be a set of indices of the learning input examples. The change of each weight for the p-th learning input example (named w_{ij}^{p}) is done as follows:

$$\Delta w_{ij}^{p} = -\eta \cdot \frac{\partial E_p}{\partial w_{ij}} \tag{2.7}$$

where $E_p = \sum_{i \in O} \left(t_i^p - o_i^p\right)^2/2$ is the error rate function and $t_i^p (= 0 \text{ or } 1)$ is the learning output example of the i-th neuron in the output layer for the p-th learning input example (i ∈ O and p ∈ P), o_i^p is the output of the i-th neuron in the output layer for the p-th learning input example, and η is a parameter of a positive real number (learning rate). Then, the weight modification is repeated until the following condition is satisfied

$$\max_{p \in P, i \in O} \left(t_i^p - o_i^p\right)^2 < e_o^2 \tag{2.8}$$

where e_o is called the output error in learning phase. If an MLP obtained by a learning with P and e_o satisfies this condition, the learning is said to have finished successfully. [17]

2.7 Floating point extraction and multiplication

The IEEE 754 standard is the most widely used standard for floating point computation, and is followed by many CPU implementation. The standard defines formats for representing floating point number (including + zero and denormals) and special values (infinities and NaNs) together with a set of floating point operations specifies four formats for representing floating point values: single-precision (32-bit), double-precision (64-bit), single-extended precision (≥ 43-bit, not commonly used) and double extended precision (≥ 79-bit, usually implemented with 80 bits) [18].

In single-precision number representation, a total of 32 bits is required. To ensure a bias is applied, the value $2^{n-1} - 1$ is added to the actual exponent to obtain the stored exponent. For an 8-bit exponent in the single-precision format, this bias equals 127. Adding this bias allows the exponent to range from -126 to +127. As a result, the single-precision format provides a numerical range from 2^{-126} to 2^{+127}.

Figure 19 illustrates the IEEE 754 single-precision binary format. This format includes a 1-bit sign (S), an 8-bit exponent (E), and a 23-bit fraction (M), also referred to as the mantissa. An additional bit is appended to the fraction to form the significand. When the exponent lies between 0 and 255 and the most significant bit (MSB) of the significand is 1, the number is referred to as a normalized number. In this case, the real value of the number is represented as Equation (2.9).

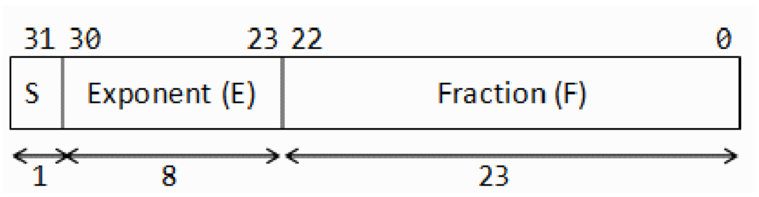


Figure 19 32 bit single precision number format by IEEE

$$Z = (-1^{S}) * 2^{(E-\text{Bias})} * (1.M) \qquad (2.9)$$

Where $M = m_{22}2^{-1} + m_{21}2^{-2} + m_{20}2^{-3} + \cdots + m_{1}2^{-22} + m_{0}2^{-23}$ with Bias=127.

To multiply two numbers in floating-point format, the process involves the following steps:

1. Add the exponents of the two numbers and then subtract the bias from the sum.

2. Multiply the significands (mantissas) of the two numbers.
3. Determine the sign of the result by performing an XOR operation on the signs of the two numbers.

To ensure the result is represented as a normalized number, the most significant bit (MSB) of the result must be a 1 (referred to as the leading one). A more detailed explanation of this process is provided in the following section. The floating point multiplication algorithm is shown in the below flowchart.

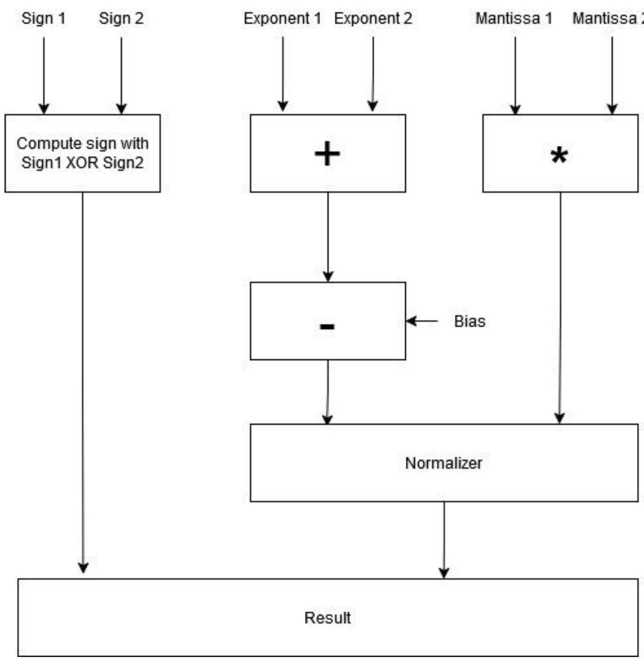


Figure 20 Simplified Floating Point Multiplication

As mentioned in the introduction, normalized floating-point numbers are represented in the format:

$$Z = (-1^{S}) * 2^{(E-\mathrm{Bias})} * (1.M) \tag{2.10}$$

To multiply two floating-point numbers, the following steps are performed:

- Multiply the significands (mantissas), which is represented as “1.M1×1.M2”.
- Adjust the decimal point in the resulting value.
- Add the exponents, calculated as “E1+E2−Bias”.
- Determine the sign of the result by performing an XOR operation on the signs of the two numbers, “S1 XOR S2”.
- Normalize the result so that the most significant bit (MSB) of the significand is 1.
- Round the result to ensure it fits within the allocated number of bits.

Consider a floating-point representation similar to the IEEE 754 single-precision floating-point format, but with a reduced number of mantissa bits (only 4), while still retaining the hidden '1' bit for normalized numbers:

A = 0 10000100 0100 = 40,

B = 1 10000001 1110 = -7.5

To multiply A and B:

1.**Multiply the significand:**

```
      1.0100
  ×    1.1110
  ----------------
       00000
      10100
      10100
     10100
  + 10100
  ----------------
    1001011000
```

2.Place the decimal point:

```
  10.01011000
```

3. Add exponents:

```
  10000100
 +10000001
 ----------------
 100000101
```

The exponent representing the two numbers is already shifted/biased by the bias value (127) and is not the true exponent.

EA = EA_true + bias ,

EB = EB_true + bias ,

EA + EB = EA_true + EB_true + 2*bias

So we should subtract the bias from the resultant exponent otherwise the bias will be added twice

```
  100000101
 - 01111111
 ----------------
```

10000110

Normalize the result so that there is a 1 positioned just before the decimal point (radix point). Shifting the decimal point one place to the left increases the exponent by 1, while shifting it one place to the right decreases the exponent by 1.

For example:

a. Before normalization: 10000110 10.01011000

b. After normalization: 10000111 1.001011000

The final result, excluding the hidden bit, is:

1 10000111 00101100

Since the mantissa contains more bits than the available 4-bit limit, rounding must be applied. After truncation rounding is used, the stored value will be:

1 10000111 0010

CHAPTER 3

BOOTH ENCODING APPLICATION

3.1 Objective

The primary objective of this chapter is to detail the development and integration of a novel algorithm that combines Booth encoding with approximate computing techniques to improve the efficiency of neural network (NN) computations, particularly multiplication operations. The selected neural network for this study is a Multi-Layer Perceptron (MLP), chosen for its simplicity, widespread use, and suitability for foundational experiments. To validate the approach, we will implement the MLP on the PYNQ-Z2 hardware board, which features an embedded CPU and an FPGA [19]. Booth encoding will be applied to optimize the iterative multiplications involved in the weight calculations of the MLP. The goal of this experiment is to enhance the neural network's efficiency by reducing computational overhead, heat generation, power consumption, and overall computation time, while carefully examining the trade-offs between speed, accuracy, and error tolerance

3.2 Booth multiplication of mantissas

As stated in chapter 2.7, to multiply two float numbers we need to multiply their mantissas. A classic Booth multiplier operates by processing the bits of the multiplier in iterations, either one by one (radix-2), three by three (radix-4), and so on, starting from the least significant bit (LSB) and moving toward the most significant bit (MSB). Based on the bit values, it performs additions or subtractions of the multiplicand to compute the final result.

In our implementation, for the sake of simplicity, the Booth multiplier will iterate bit by bit (radix-2). However, unlike the traditional approach, the direction of iteration will be reversed — starting from the most significant bits (MSB) and moving toward the least significant bits (LSB). Additionally, by integrating this Booth multiplier with approximate computing techniques, we will deliberately skip the iteration of the X final bits based on our specific requirements, thus optimizing performance and reducing computation where precision is less critical as outlined in Chapter 2 . The aforementioned description of our algorithm is coded in Verilog. Verilog is used in FPGA development as it allows for precise control over hardware resources. By coding the Booth multiplier in Verilog, we can implement it directly onto the FPGA embedded in the PYNQ-Z2 board. This ensures that the multiplication operations are carried out at hardware-level speeds, leveraging the parallel processing capabilities of the FPGA. Furthermore, Verilog's ability to describe hardware behavior and architecture aligns with the requirements of our design, including bit-level manipulations and iterative computations.

The module takes two mul_size-bit inputs, A and B, and produces a 2*mul_size-bit output, out. A parameter "iterations" portrays how many bits of the input A we wish to incorporate into our multiplication. The algorithm iterates through the bits of A (from most significant to least significant), checking for transitions to identify when to add or subtract appropriately shifted versions of B to an accumulated result (temp_output). The first "1" in A triggers an addition, and a transition from "1" to "0" triggers a subtraction. The distinction between the subtraction and the addition will be executed with the help of the variable "first_ace". The final result is assigned to the output wire after the computation. This approach reduces computational effort by ignoring the least significant bits, allowing for an approximate result with a trade-off between precision and efficiency. For a clearer understanding Figure 21 describes the algorithm, then Figure 22 depicts a flow diagram of the approximate booth multiplication between the numbers A and B when skipping X bits.

```
FUNCTION booth_multiplier(A, B)
    SET iterations = mul_size-X
    SET first_ace = 0
    SET temp_output = 0
    SET digit = mul_size

    WHILE (iterations $\neq$ 0) DO
        iterations=iterations-1
        digit=digit-1
        IF A[digit] == 1 THEN
            IF first_ace == 0 THEN
                temp_number = B SHIFT_LEFT (digit + 1)
                temp_output = temp_output + temp_number
                first_ace = 1
            ENDIF
        ELSE IF first_ace == 1 THEN
            temp_number = -B SHIFT_LEFT (digit + 1)
            temp_output = temp_output + temp_number
            first_ace = 0
        ENDIF
    ENDWHILE
    IF first_ace == 1 THEN
        temp_number = -B SHIFT_LEFT (digit)
        temp_output = temp_output + temp_number
    ENDIF
    RETURN temp_output
END FUNCTION
```

Figure 21 Pseudocode of the approximate booth multiplier

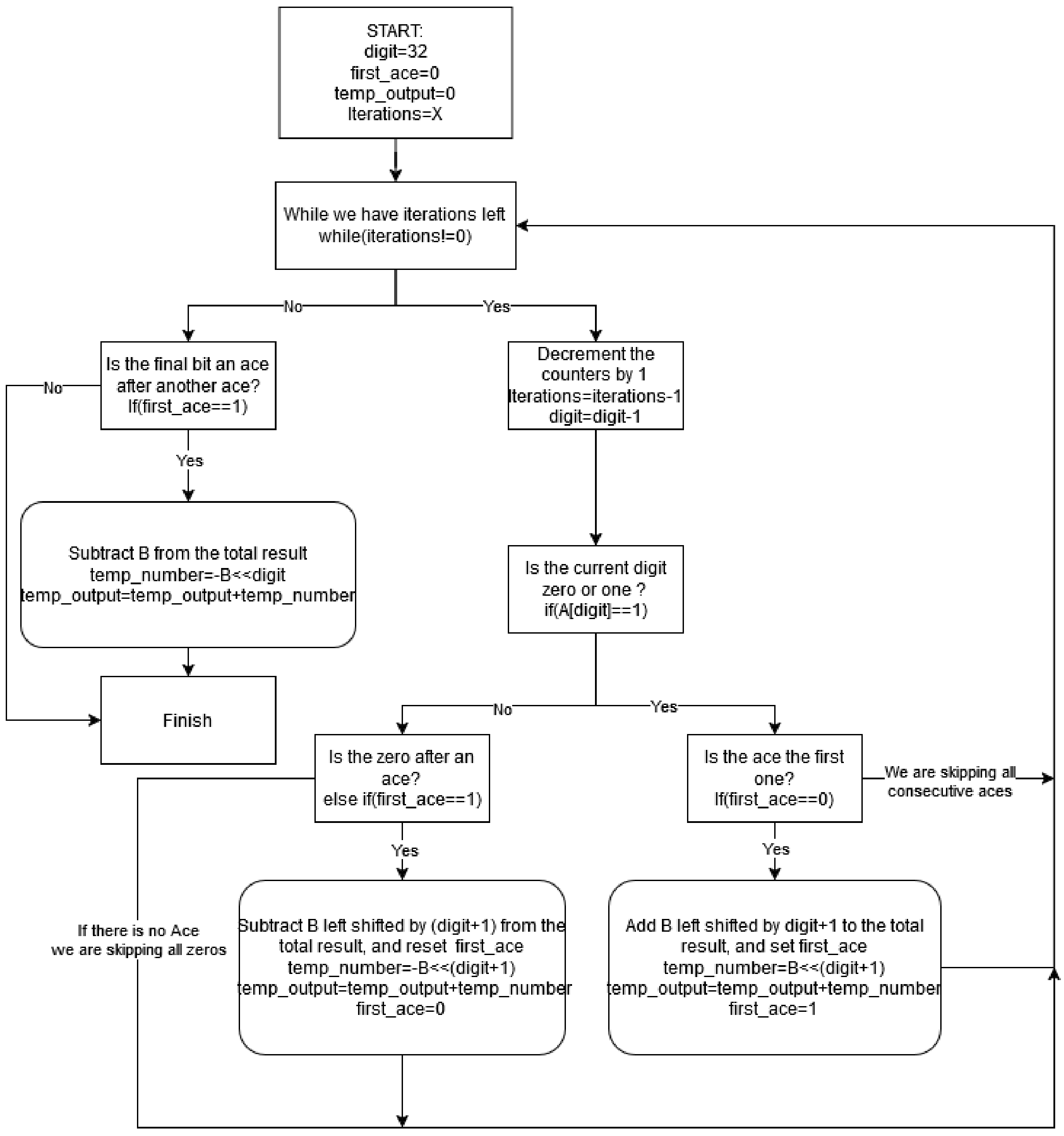

START:
digit=32
first_ace=0
temp_output=0
Iterations=X
While we have iterations left
while(iterations!=0)
No
Yes
Is the final bit an ace after another ace?
If(first_ace==1)
No
Yes
Subtract B from the total result
temp_number=-B<<digit
temp_output=temp_output+temp_number
Finish
Decrement the counters by 1
Iterations=iterations-1
digit=digit-1
Is the current digit zero or one ?
if(A[digit]==1)
No
Yes
Is the zero after an ace?
else if(first_ace==1)
Is the ace the first one?
If(first_ace==0)
We are skipping all consecutive aces
Yes
Yes
If there is no Ace we are skipping all zeros
Subtract B left shifted by (digit+1) from the total result, and reset first_ace
temp_number=-B<<(digit+1)
temp_output=temp_output+temp_number
first_ace=0
Add B left shifted by digit+1 to the total result, and set first_ace
temp_number=B<<(digit+1)
temp_output=temp_output+temp_number
first_ace=1

Figure 22 Flow diagram

3.3 Implementation and design

We implemented our Booth multiplier using Verilog, developed within the Vivado 2024 environment. The verilog code contains a variable which represents the amount of least significant bits that are going to be skipped during the multiplication of the two inputs. Therefore, we can easily create all 32 possible multipliers (skipping 1 bit , 2 bits etc) and so simplifying the hardware design and the area occupation of the multiplier on an FPGA. Once we have all of the

32 multipliers synthesized, the hardware design was exported from Vivado as xsa files which the fpga can translate into hardware. These files were subsequently handled through Vitis 2024 and transferred to the FPGA via serial communication which was executed with the use of a usb to micro-usb cable. We must note that the exported design includes all the 32 versions of the multiplier, each corresponding to a different number of bits skipped during multiplication. Consequently, all 32 variants of the multiplier are simultaneously available on the FPGA and they can be used at anytime.
The target platform for this implementation is the PYNQ-Z2 board. The CPU of this board, which supports the C programming language, includes a set of pre-installed libraries that facilitate development. The next phase of the project involved developing a multilayer perceptron (MLP) within Vitis 2024. The MLP was deployed to the CPU of the board, and its performance was monitored via a serial interface. During operation, the MLP communicates with the FPGA, which houses the full set of Booth multipliers, through a wrapper(see Chapter 3.4).
During the training phase of the MLP, every multiplication, whether related to output computation of a neuron or weight adjustment of an edge, is processed on the FPGA sequentially. Although a more optimized design would allow batch processing and concurrent weight updates, a sequential approach was adopted to maintain simplicity. The primary performance metric under investigation is the effect of bit-skipping during the training phase, particularly during feedforward and backpropagation stages of the neural network. The initial configuration uses no bit-skipping, with subsequent tests progressively increasing the number of skipped bits. Each experiment run corresponds to a fixed number of skipped bits and the same number of epochs, however, a dynamic bit-skipping approach which varies each epoch may be explored in future work.
It is important to note that the testing phase of the MLP does not employ approximation, as this phase involves only a few multiplications and using approximation is not going to have an affect at the speed. All weights within the neural network are randomly initialized at the start of each run, and the number of training epochs and many other variables are all user-configurable(see Chapter 3.5). The datasets used for training and testing is sourced from open sources.

The MLP is designed to address both classification and regression problems through supervised learning. Example classification problems include handwritten digit recognition, sentiment analysis, medical diagnosis based on symptom vectors and many more. The inputs to the network are supplied as static arrays, selected for their efficiency in memory usage and execution speed. This method leverages the stack, which offers optimal performance for handling large volumes of data rapidly. All classification outputs are represented as integer values, such as [0, 1, 2, 3, ...], depending on the class labels assigned in the dataset. Additionally, each attribute of the dataset was normalized to the interval [-1, 1] using the following normalization formula:

$$x_{\text{norm}} = \frac{2(x - x_{\text{min}})}{x_{\text{max}} - x_{\text{min}}} - 1 \tag{3.1}$$

where x is the number we are normalizing and x_{min} and x_{max} are the minimum and maximum values in each attribute.

Custom serial printing functions were developed to accommodate the limitations of the lightweight Vitis libraries available on the CPU. These functions enable the display of floating-point results through serial monitoring. The final output includes the MLP performance metrics such as classification accuracy, execution time, and mean relative error (MRED), as detailed in earlier chapters.

To efficiently perform floating-point operations, a custom casting method was employed. This method allows for the extraction of the sign bit, exponent, and mantissa from each floating-point number.

However, a challenge emerged related to the clocking speed of the FPGA. The maximum allowable clock frequency of the FPGA is constrained on certain values and cannot be altered. So higher and more optimal clocking speeds could not be achieved.

Notably, the transmission of data between the CPU and FPGA through the wrapper did not achieve sufficient speed to yield a noticeable performance benefit. While the use of Direct Memory Access (DMA) could resolve that issue, it was not incorporated into the current design for as we prioritise simplicity. Instead, a fallback approach involved executing the MLP multiplications on the CPU instead of the FPGA. This adjustment did not affect the error tolerance of the training

phase, and the results remained consistent with what would be obtained if the calculations were executed on the FPGA.
A more sophisticated design would involve parallel execution, in which multiple FPGA multipliers with different bit-skip configurations would process arrays of multiplications concurrently. These arrays would be generated in batches by the CPU during parallelized weight adjustments in the MLP. This proposed enhancement, along with other optimizations, is left for future exploration.

3.4 Benchmarking hardware with vitis

In the context of deploying our design onto the Zynq-Z2 board, it is essential to create an HDL wrapper for the block diagram within the Vivado Design Suite. This wrapper serves as the top-level module, encapsulating the entire design and facilitating its integration into the FPGA fabric.
The design comprises several interconnected components, each fulfilling a specific role in implementing the Booth multiplier on the FPGA. The ZYNQ7 Processing System serves as the main controller, integrating an ARM Cortex-A9 processor to manage software execution, system configuration, and communication with the FPGA fabric. It offers essential interfaces such as DDR for memory access, USB for external communication, and clock and reset signals for system synchronization, ensuring efficient operation and management of computational tasks.
The Processor System Reset module ensures that all components initialize in a defined state by providing necessary reset signals, preventing errors due to uninitialized registers.
The AXI Interconnect facilitates data transfer between the ARM processor and the programmable logic by efficiently managing multiple AXI peripherals, allowing seamless communication between software and hardware components.
The AXI GPIO blocks provide a flexible interface for general-purpose input and output, enabling the processor to send and receive control signals to and from the FPGA. They are particularly useful in setting input values for the Booth multiplier and retrieving computed results.
The Booth Multiplier module is the core computational unit, performing binary multiplication using the Booth algorithm, which optimizes signed multiplication

by reducing the number of partial products, allowing efficient hardware acceleration.

The Slice blocks extract specific bit ranges from data signals, ensuring efficient data handling and precise bit-level manipulation, which is necessary for organizing and processing inputs and outputs within the FPGA.

The DDR and FIXED IO interfaces enable access to external memory and peripheral communication, allowing the processing system to handle large datasets efficiently while ensuring stable data exchange with external components through USB.

Here is a picture of the entire design block.

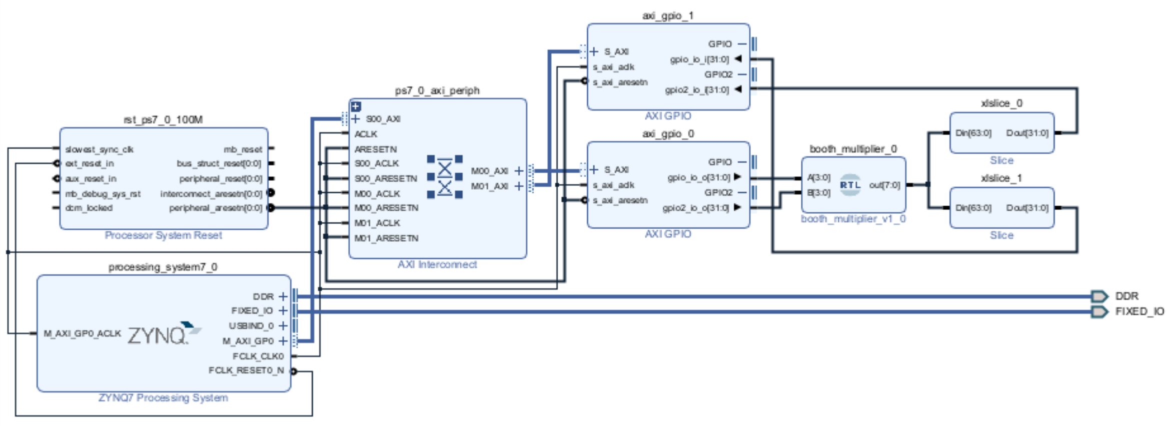


Figure 23 Block design inside vivado’s inteface

3.5 Using a Multilayer perceptron

At this part of the thesis, we will describe some key components of the implementation of a Multilayer Perceptron (MLP) on the PYNQ-Z2 board. The file main.c implements a set of flexible and adjustable parameters that define the architecture and behavior of the network. These parameters can be customized by the user to tailor the model for specific requirements, such as optimizing accuracy, computational efficiency, or adapting to hardware constraints or data size. The following parameters define the structure and operation of the MLP:

#define __num_hidden_layers 3

This parameter defines the number of hidden layers in the MLP. The user can adjust this number to modify the depth of the network, which in turn affects its capacity to learn from the data. A deeper network might capture more complex patterns but could also increase computational costs.

#define __hidden_layers_neurons "12,10,8"

This defines the number of neurons in each hidden layer. In this case, the first hidden layer has 12 neurons, the second has 10 neurons, and the third has 8 neurons. Increasing the number of neurons in each layer lead to more accurate and distinct results but can cause the network to become more complex and therefore demand more computational resources

#define __hidden_activ_func "tanh,tanh,tanh"

This parameter specifies the activation functions used for each of the hidden layers. Many activation functions are available each with unique characteristics and use cases. The identity function outputs the input unchanged and is typically used in regression tasks. The sigmoid function squashes the input into a range between 0 and 1 and is commonly used in binary classification but suffers from the vanishing gradient problem. The tanh function outputs values between -1 and 1, helping with faster convergence but still experiences vanishing gradients in deep networks. The ReLU function outputs the input if positive or zero if negative and is widely used in hidden layers for its computational efficiency and ability to avoid vanishing gradients, although it can suffer from the dying ReLU problem. The softmax function is used in the output layer for multi-class classification tasks, converting raw outputs into a probability distribution where the sum of all outputs is 1.

#define __output_size 7

This defines the size of the output layer. In this case, the model has 7 output units, which is suitable for classification tasks with 7 possible categories. The user can modify this value for tasks with different numbers of classes. The classes on the dataset must be integers with the lowest value of 1.

#define __out_activ_func "softmax"

This parameter defines the activation function for the output layer. Users can replace this with any other activation functions that was mentioned previously depending on the classification problem.

#define __learning_rate 0.01

This parameter controls the learning rate during training. The learning rate determines the step size taken in the gradient descent optimization process. A higher learning rate can speed up convergence, but it may also cause instability, while a smaller learning rate will converge more slowly but more steadily.

#define __skipped_bits 0

This defines the number of bits that can be skipped in the computation for hardware optimizations. The user can adjust this parameter for trade-offs between computation precision and efficiency, particularly when implementing the model on embedded systems or FPGAs.

#define __max_iterations 200

This parameter defines the maximum number of iterations for the training process of the model. This value dictates how many times the model will cycle through the entire training dataset during the learning process. The user can adjust this value depending on the complexity of the task, the size of the dataset, and the desired training time. Additionally, to prevent overfitting, the training process involves shuffling the data at the beginning of each epoch. Shuffling the data ensures that the model is not exposed to the data in a fixed sequence, which helps to avoid learning spurious patterns that may arise from the order of the dataset.

To conduct an experiment , the user must also modify the data files, specifically the data.c and data.h files. The data used for training and testing are inserted into the neural network as arrays, which are defined in the data.c file. In this file, the arrays represent the input data, with each row containing the feature values for a given sample. Each row is separated by commas, and the last column in the array represents the class to which the sample is assigned. The class values must be integers starting from 1 (for example, 1, 2, 3, etc.), with each integer corresponding to a specific class label.

In addition to modifying the data array in data.c, the data.h file contains the sizes of the training set(#define __train_sample_size) and test sets(#define __test_sample_size) as well as the number of features of the dataset including the class label(#define __features_size). These values must be updated accordingly to reflect the number of samples in the dataset. It is important that the user ensures the data dimensions, the number of features and the class labels are correctly defined to match the structure expected by the neural network. This allows the model to properly interpret the data and proceed with training and evaluation.

CHAPTER 4

EVALUATION

4.1 Introduction

This chapter covers experiments conducted on the PYNQ-Z2 board, focusing on the implementation and evaluation of our proposed design. The experiments involve deploying the Verilog-based Booth multiplier onto the FPGA embedded in the board, allowing us to analyze its performance in terms of efficiency, speed, and power consumption. Additionally, we explore the impact of approximate computing techniques on multiplication accuracy and resource utilization. Through these experiments, we aim to validate the effectiveness of our approach and demonstrate its advantages in hardware-based neural network acceleration.

4.2 Tools

In this study, Vivado 2023.2 was used for the design, synthesis, and analysis of the Booth multiplier. This tool enables precise evaluation hardware parameters, including circuit area, power consumption, and schematic visualization, which encompasses key processing components such as shifters, multiplexers, and adders. The hardware description language used for implementation is Verilog, facilitating efficient hardware modeling and synthesis for FPGA deployment. To integrate the Booth multiplier into a Multilayer Perceptron (MLP) network, Vitis 2023.2 was employed. In Vitis, we materialized our MLP network using the C programming language and parameterized it to allow dynamic modifications of the Booth multiplier as well as the MLP. This flexibility enabled an evaluation of different multiplier configurations, allowing adjustments based on performance requirements. The bitstream file generated from Vivado was incorporated into

Vitis, where each multiplication operation was replaced with Booth multiplication. For output monitoring and analysis, serial monitoring was utilized. While various serial monitoring tools are available, Visual Studio Code was chosen due to its robust debugging capabilities, versatility, and reliability in handling serial data communication.

Finally, we utilized the ZYNQ XC7Z020-1CLG400C which integrates a 650MHz ARM® Cortex®-A9 dual-core processor with programmable logic. The programmable logic consists of 13,300 logic slices, each containing four 6-input LUTs and 8 flip-flops, along with 630 KB of block RAM and 220 DSP slices. The FPGA programming and configuration were carried out using a USB Type-A to Type-B cable. Additionally, the system includes 512MB DDR3 memory with a 16-bit bus, operating at 1050 Mbps. The ARM processor runs bare-metal (no OS) for deterministic low-latency control, avoiding the overhead of a full Linux stack. (Alternatively, if Linux were used, an SD card would be required for booting.) [19]

4.3 Experiments

4.3.1 Multiplier testing

In this experiment, we evaluate the impact of approximate computing on the Booth multiplier using three key metrics: Look-Up Table (LUT) usage, schematics complexity, and on-chip power consumption. These metrics provide a comprehensive understanding of the trade-offs between computational precision and resource efficiency in FPGA-based designs. LUT usage reflects the logic resource utilization, schematics illustrate the structural complexity of the design, and on-chip power consumption quantifies the energy efficiency of the implementation. Together, these metrics offer valuable insights into the effectiveness of bit-skipping as an approximation technique for optimizing Booth multipliers in resource-constrained environments.

Finally we must note that comparing our multiplier to the one embedded within Vivado and its compiler is not meaningful, as it is highly optimized through hardware-level enchantments like DSP slices and not LUTs. Therefore, we conduct our comparison against a manually implemented multiplier which follows the conventional long multiplication algorithm which we were taught in elementary school. In all figures below this kind of multiplier is mentioned as normal multiplication.

LUT utilization

Look-Up Tables (LUTs) are the fundamental building blocks of FPGA logic, used to implement combinational functions. In the context of Booth multipliers, LUT usage is directly influenced by the complexity of partial product generation and accumulation. By skipping bits, we reduce the number of partial products, thereby simplifying the logic and decreasing the number of LUTs required. This metric is critical for assessing the resource efficiency of the design, as lower LUT usage allows for more compact implementations and frees up resources for other tasks. The following results demonstrate how LUT usage scales with the number of bits skipped, highlighting the potential for resource savings in approximate computing. The amount of LUTs on the board "stands at" 53,200. Here is a plot with the LUT utilization per bits skipped as well as a percentile representation.

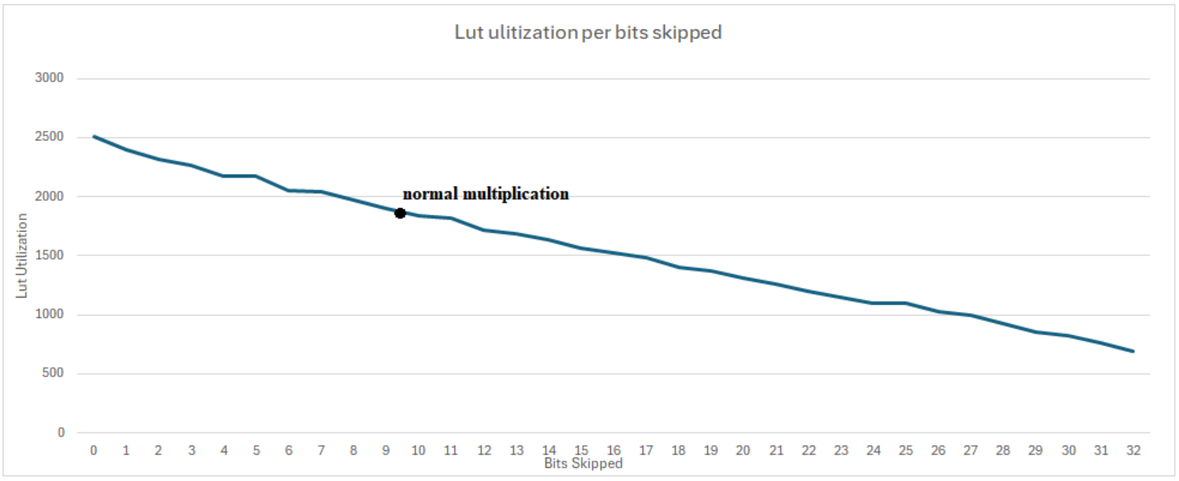


Figure 24 Lut utilization per bit skipped

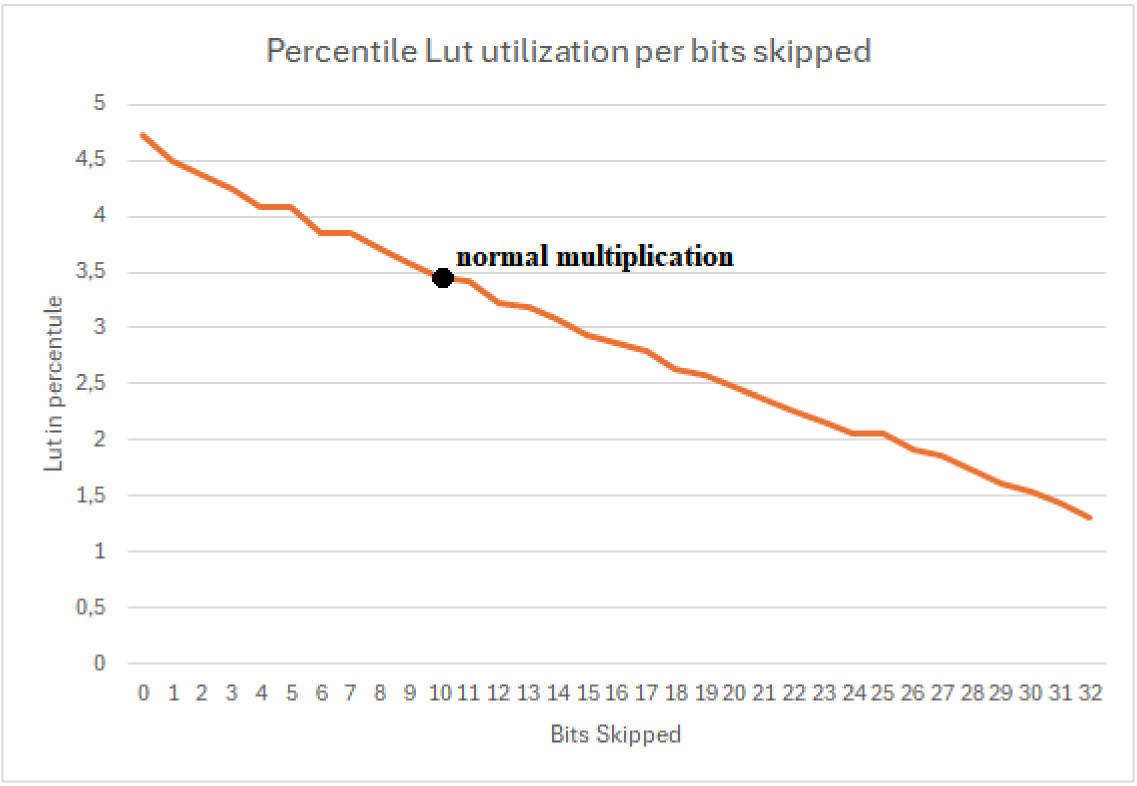


Figure 25 Percentile Lut utilization per bits skipped

Error tolerance/Mred

Error tolerance in the context of approximate computing can be quantified using the Mean Relative Error Distance (MRED), which measures the average deviation of the approximate results from the exact results, normalized by the magnitude of the exact values. As the number of skipped bits increases, the precision of the computation decreases, leading to a corresponding incrementation in MRED. In this experiment we are generating 100000 random multiplications and then calculating the MRED as referenced in section 2.2. The following results demonstrate how MRED scales with the number of skipped bits, providing insights into the error tolerance of the system under varying levels of approximation. It is important to note that due to rounding, an average of 11 or 12 bits (half the size of the mantissas) are cut off from the final output . As a result, the MRED for the first 15 bits appears to have minimal error, as the rounding effect dominates during the initial stages of approximation. Even though some bits are lost in single-precision mantissa multiplication, the mean relative error remains very small in most cases, ensuring high reliability.

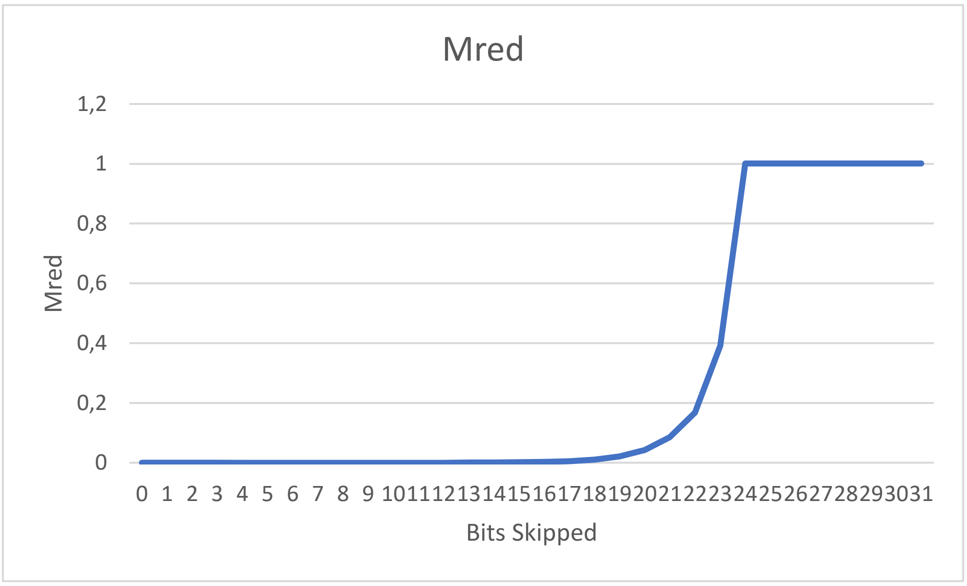


Figure 26 MRED of 100000 random multiplications

Schematics

The schematics of a design provide a visual representation of its structural complexity, including the interconnections between logic elements and the overall organization of the circuit. In the case of Booth multipliers, bit-skipping reduces the number of partial products and simplifies the adder tree, leading to a less complex schematic. This reduction in complexity not only makes the design easier to analyze and debug but also improves routing efficiency on the FPGA. By examining the schematics at different levels of bit-skipping, we can observe how approximation techniques streamline the design, making it more suitable for applications where area and routing resources are limited. The following schematics illustrate the implementation of a approximate Booth multiplier designed to multiply two 4-bit numbers, A and B. The design consists of interconnected components, including shifters, multiplexers, and adders. For reference, the schematic of a traditional 4x4 multiplier is included to provide a baseline comparison with the approximate Booth multiplier designs

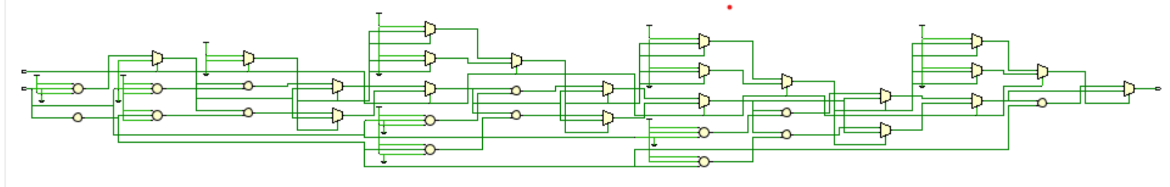


Figure 27 4x4 Approximate booth multiplier, 0 bits skipped

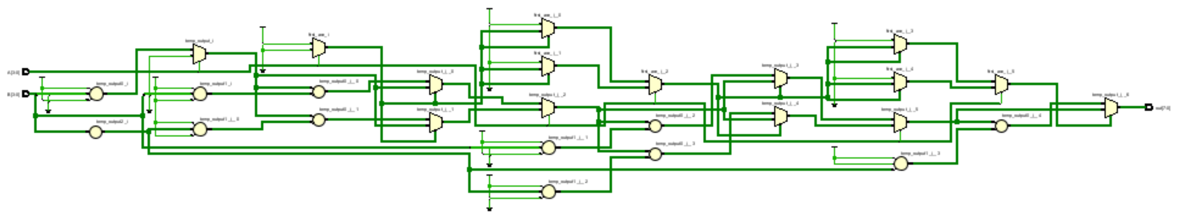


Figure 28 4x4 Approximate booth multiplier, 1 bit skipped

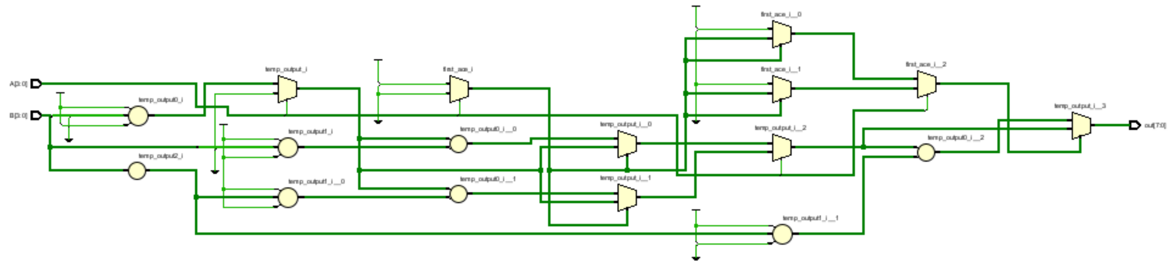


Figure 29 4x4 Approximate booth multiplier, 2 bits skipped

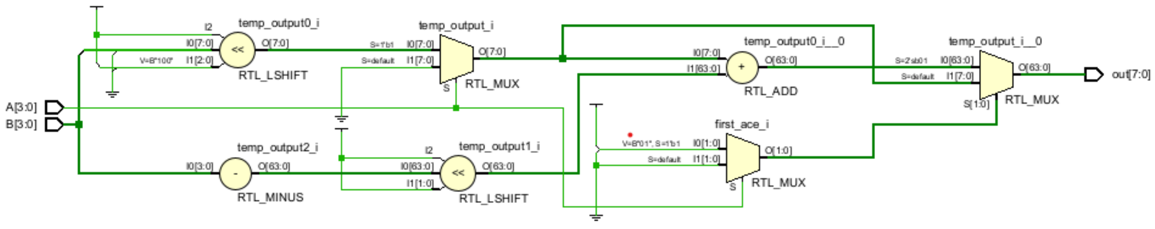


Figure 30 4x4 Approximate booth multiplier, 3 bits skipped

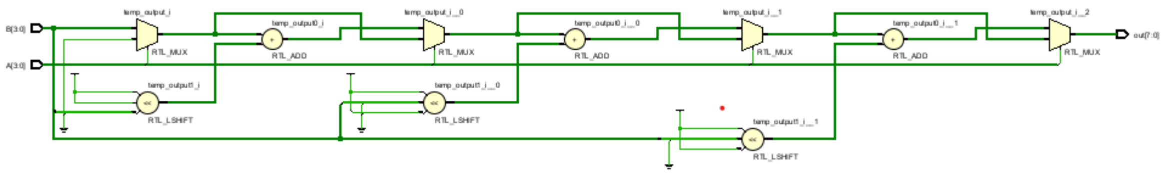


Figure 31 4x4 multiplier

Power on chip

On-chip power consumption is a critical metric for evaluating the energy efficiency of FPGA-based designs. It is influenced by both the dynamic power (due to switching activity) and static power (due to leakage currents). In Booth multipliers, skipping bits reduces the number of logic operations and the switching activity, thereby lowering dynamic power consumption. Additionally, the reduction in LUT usage and routing complexity contributes to lower static power. This metric is particularly important for energy-constrained applications, such as edge computing or IoT devices, where minimizing power consumption is a key design objective. The following analysis quantifies the power savings achieved through bit-skipping, demonstrating the potential of approximate computing for energy-efficient implementations.

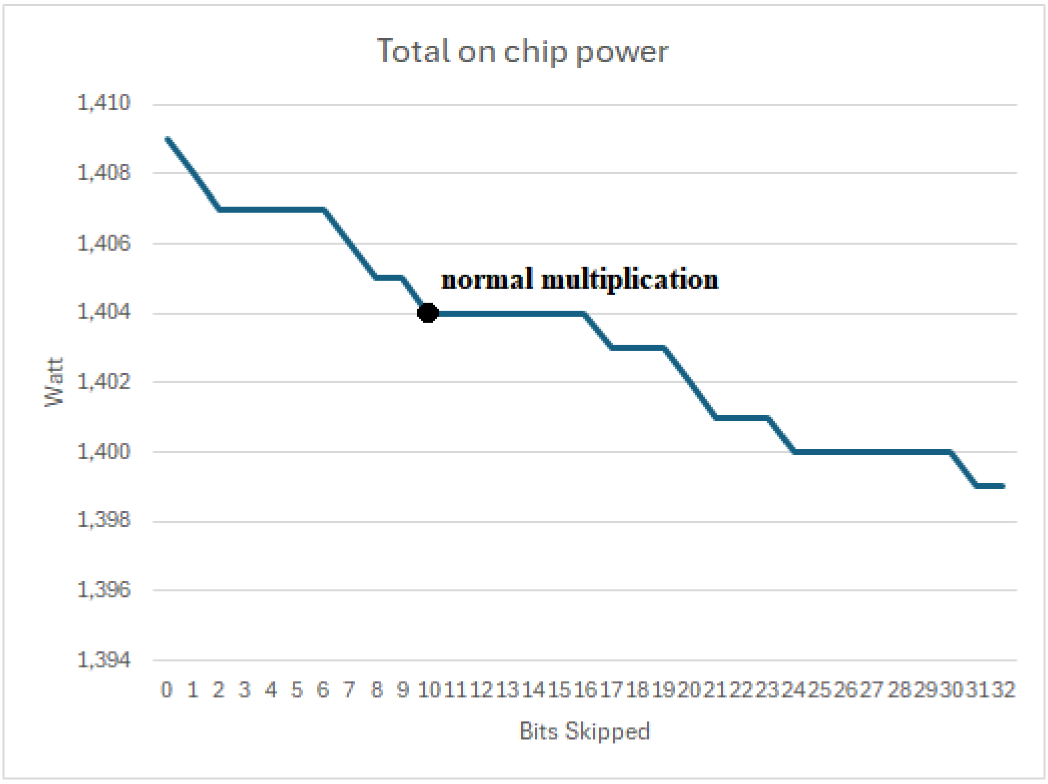


Figure 32 Total on chip power

Time taken per multiplication

The time taken for each multiplication is a critical metric for evaluating the efficiency of the proposed approximate Booth multiplier. In this experiment, we aim to calculate the average time per multiplication using Vivado timing simulation. Vivado provides precise timing analysis, allowing us to measure the latency of each multiplication operation under varying levels of approximation. By generating a large number of random multiplications and analyzing their timing, we can determine how the time taken scales with the number of skipped bits. This analysis provides valuable insights into the trade-offs between computational speed and precision.

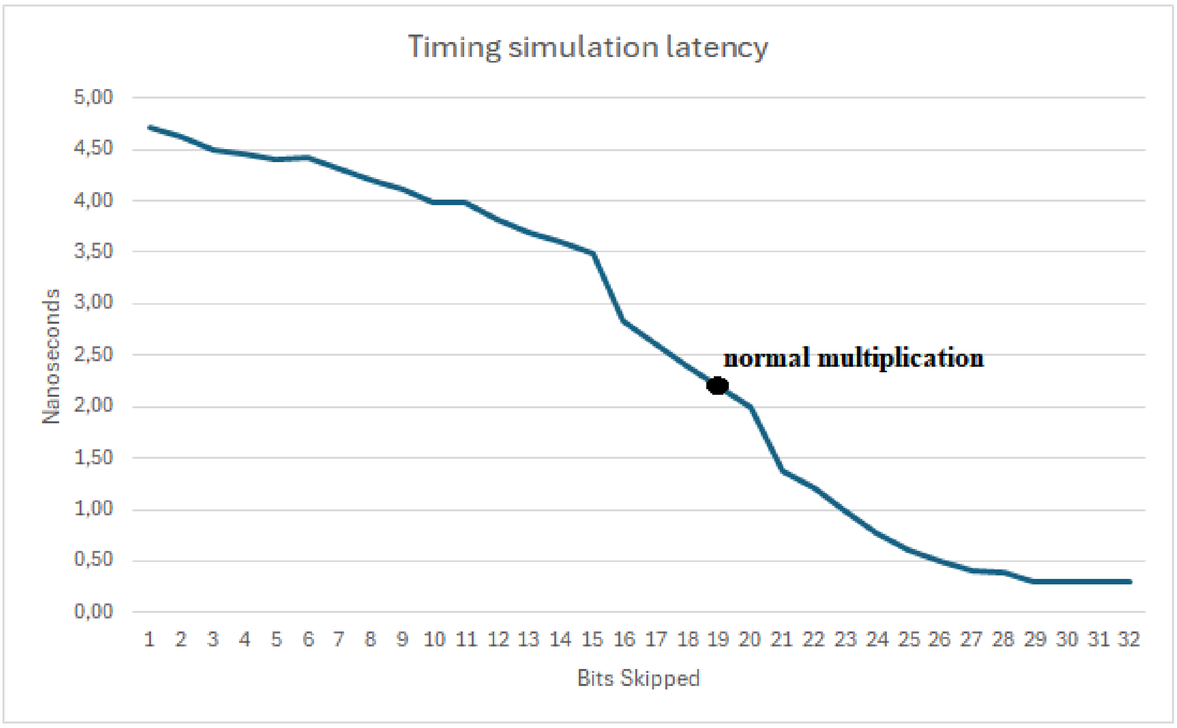


Figure 33 Timing simulation latency

From Figure 33 we can produce the corresponding clocking speeds. These speeds represent the optimal clocking speeds of the FPGA as long as the available hardware supports them.

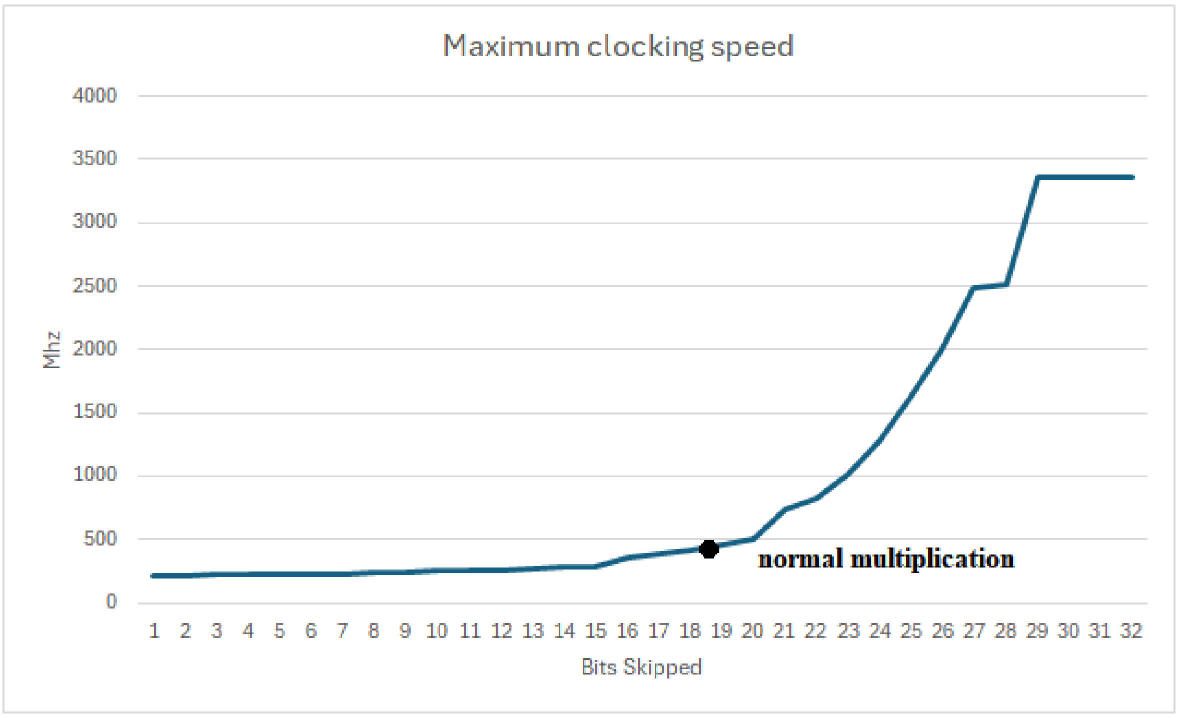


Figure 34 Maximum clocking speed

4.3.2 MLP testing

In the following section of the thesis, we will explore the integration of a Multi-Layer Perceptron (MLP) with hardware-accelerated approximate computing on an FPGA. By replacing all multiplications contained inside the MLP with approximations using booth multipliers, we aim to accelerate the computation while leveraging the inherent error tolerance of neural networks to maintain acceptable accuracy. The MLP's ability to tolerate such errors makes it an ideal candidate for this approach, as it can often produce good results even with reduced computational precision due to its iterative nature. This experiment evaluates the trade-offs between speed, accuracy, and error tolerance when using approximate multiplication in an MLP, providing insights into the potential for hardware-accelerated machine learning in resource-constrained environments. Unfortunately, these tests cannot be conducted with complete exactitude, as the FPGA clock speed does not align with the ideal operating speed. Consequently, all tests will be performed using the board's CPU.

Finally we must note that comparing our multiplier to the one embedded within a C compiler compiler is futile, as it is highly optimized through software-level enchantments and loop accelerations. Therefore, we conduct our comparison against a manually implemented multiplier which follows the conventional long multiplication algorithm which we were taught in elementary school. In all figures below this kind of multiplier is mentioned as normal multiplication.

Accuracy

The accuracy of a classification Multilayer Perceptron (MLP) is a fundamental metric for assessing its performance in tasks such as pattern recognition, image classification, and decision-making. Accuracy measures the proportion of correctly classified instances out of the total number of instances, reflecting the MLP's ability to learn and generalize from training data after a certain amount of epochs. In the context of approximate computing, where precision is intentionally reduced to enhance computational efficiency, maintaining high accuracy becomes a significant challenge. However, the MLP's inherent error tolerance allows it to produce reliable classification results even with approximate computations, making it a strong candidate for hardware-accelerated implementations. This section examines how the accuracy of the classification MLP is influenced by the integration of approximate computing techniques, such as the modified Booth multiplier, and evaluates the trade-offs between computational efficiency and classification performance. The following figures represents the accuracy of an MLP with the according characteristics.

Feature size=5, Training set size =120, Test set size = 30, Neurons Layers, 6,8,8, Activation functions: Tanh, Tanh, Tanh, Softmax , Epochs=100

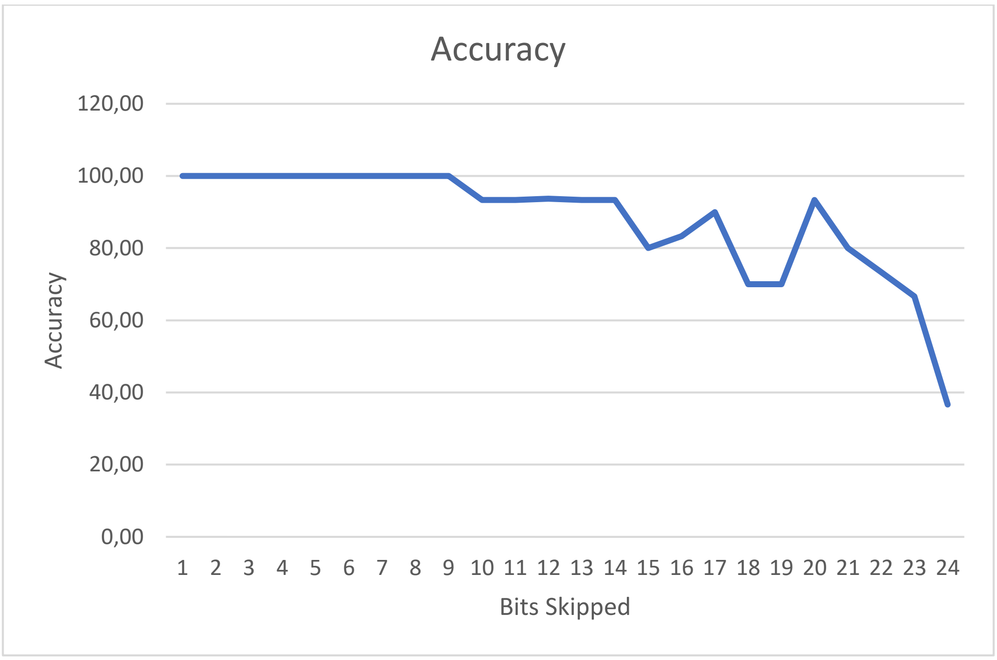


Figure 35 First dataset, accuracy per bit skipped

The results from Figure 35 as observed, accuracy remains nearly constant and close to 100% for up to approximately 9 bits skipped. Beyond this threshold, a gradual decline in accuracy becomes evident, with more pronounced degradation occurring after skipping more than 14 bits. Notably, while some fluctuations appear, due to randomness, such as temporary increases at 18 and 20 bits, the general trend shows that accuracy diminishes as the level of approximation increases. At 24 bits skipped, the network's accuracy drops significantly to below 40%.

Feature size=6, Training set size =200, Test set size = 26, Neurons Layers, 12,10,8, Activation functions: Tanh, Tanh, Tanh, Softmax, 300 epochs

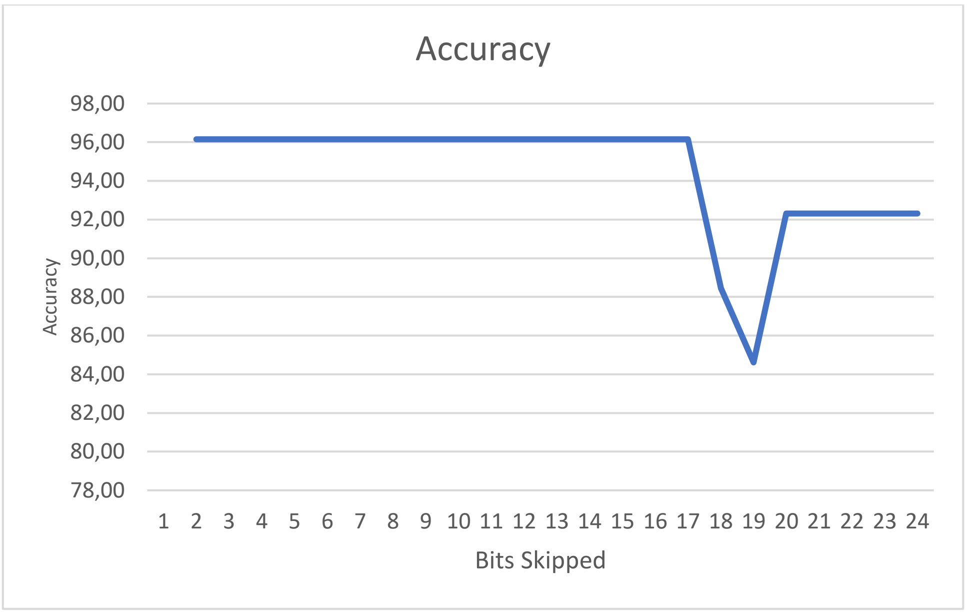


Figure 36 Second dataset, accuracy per bit skipped

The results presented in Figure 36 depict the accuracy of the multilayer perceptron on the second dataset as a function of the number of bits skipped during multiplication. It is evident that accuracy remains consistently high, approximately 96%. However, the sharp drop in accuracy beyond 16 bits skipped suggests a critical threshold, beyond which the model begins to lose essential information from the training data. This degradation may also be influenced by overfitting, where the model becomes too tightly adapted to the training set and fails to generalize under altered numerical conditions.

Training time taken

The time taken for all multiplications to be executed during the training phase of the MLP is significantly reduced when using Booth approximation compared to classic multiplication. Booth approximation simplifies the classic multiplication process by skipping bits and reducing the number of partial products, thereby decreasing latency of each multiplication and so the duration of the training phase. The following results depict the time taken of the NN to be trained and tested using the classic and Booth-approximated multiplication, demonstrating the potential for faster processing in hardware-accelerated MLPs. It is also important to emphasize that the comparison was conducted between our custom Booth multiplier and the straightforward long multiplication algorithm, the same method typically taught in elementary school. As also mentioned before, using the highly optimized multiplication operations embedded within the C compiler would have rendered the comparison invalid. This baseline multiplier, like our Booth designs, was executed on the CPU (and is shown as "normal multiplication") to ensure a fair and consistent evaluation environment.

Feature size=5, Training set size =120, Test set size = 30, Neurons Layers, 6,8,8, Activation functions: Tanh, Tanh, Tanh, Softmax , Epochs=100

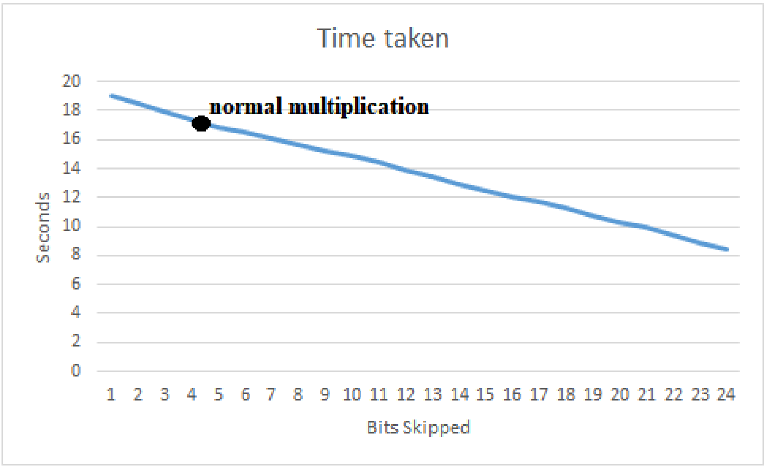


Figure 37 First dataset, time taken

Feature size=6, Training set size =200, Test set size = 26, Neurons Layers, 12,10,8, Activation functions: Tanh, Tanh, Tanh, Softmax, 300 epochs

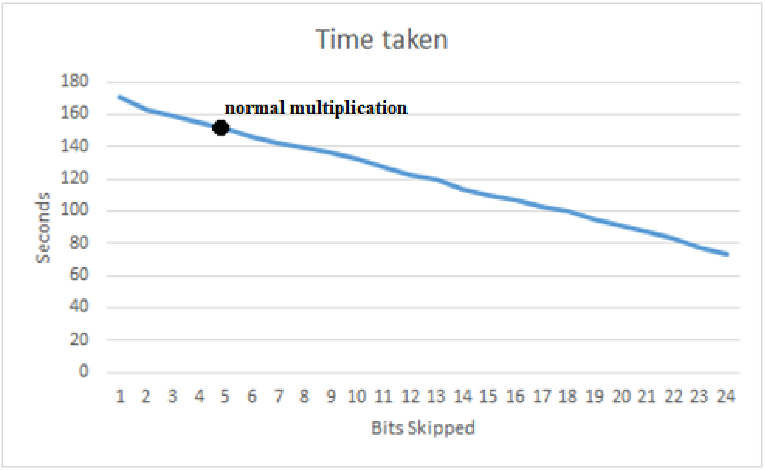


Figure 38 Second dataset, time taken

CHAPTER 5

Conclusions and future thoughts

5.1 Conclusions

This thesis presented the development and evaluation of a new approximate Booth multiplier designed to handle floating-point numbers and process bits from the most significant bit (MSB) to the least significant bit (LSB). Unlike traditional Booth multipliers, which process bits sequentially and focus on reducing the number of partial products through Booth encoding, our approach prioritizes the most significant bits first, enabling early termination and reducing computational complexity. This design choice, combined with approximate computing techniques, significantly improves efficiency while maintaining acceptable accuracy. The multiplier was implemented on the PYNQ-Z2 board, and its performance was evaluated across several key metrics, including LUT utilization, schematics complexity, on-chip power consumption, accuracy, error tolerance, and time taken.

The approximate Booth multiplier demonstrated significant reductions in LUT utilization as the number of skipped bits increased. By simplifying the logic required for partial product generation and accumulation, the multiplier freed up valuable FPGA resources, enabling more compact designs and leaving room for additional functionality. This reduction in LUT usage is particularly beneficial for resource-constrained environments, where efficient resource allocation is critical. The schematics complexity of the multiplier also decreased as the number of skipped bits grew. By reducing the number of partial products and simplifying the adder tree, the design became more streamlined and easier to route on the FPGA. This reduction in complexity not only improved the scalability of the design but also made it more suitable for applications where area and routing resources are limited.

In terms of on-chip power consumption, the multiplier showed a decrease as the number of skipped bits increased, although the reduction was not as significant as expected. This is likely due to the limited resources and small scale of the PYNQ-Z2 board, which constrained the potential power savings. Nevertheless, the results suggest that approximate computing techniques can contribute to energy-efficient designs, particularly when implemented on larger FPGAs with more resources.

Furthermore as the number of skipped bits increases, the computational complexity decreases, allowing for faster multiplication operations. This reduction in complexity also enables the FPGA to operate at higher clocking speeds, as long as the rest of the system supports those speeds.

Despite the reduction in precision caused by bit-skipping, the multiplier maintained acceptable accuracy levels, especially when integrated into the MLP. The MLP's inherent error tolerance allowed it to produce reliable results even with approximate computations, demonstrating the feasibility of using approximate multipliers in machine learning applications.

The multiplier exhibited strong error tolerance, as quantified by the Mean Relative Error Distance (MRED). The results suggest that the approximate Booth multiplier is a suitable choice for applications where error tolerance is critical. Finally, the time taken for computations was significantly reduced, making it a viable option for real-time applications where speed is critical.

5.2 Future thoughts

To fully realize the potential of this approximate Booth multiplier, future work should focus on testing it on a larger FPGA board with more resources, such as increased LUTs, DSP slices, and memory bandwidth. A larger board would allow for more extensive experimentation, including the implementation of higher-radix designs (e.g., radix-2 or radix-4) and the evaluation of more complex neural networks. Additionally, a larger board would provide a better environment for analyzing power consumption, as the current results were limited by the small scale of the PYNQ-Z2.

Another promising direction is the parallelization of the hardware, which could enable even faster speeds by processing multiple operations simultaneously. This would be particularly beneficial for real-time applications where latency is critical. Furthermore, a larger board with fewer clocking restrictions would allow for experiments with faster clocks, enabling true hardware acceleration and providing a more accurate assessment of the multiplier's performance.

Exploring error compensation techniques could further improve the multiplier's accuracy without sacrificing efficiency. Extending the multiplier's capabilities to handle larger datasets and more complex models, such as convolutional neural networks (CNNs) or recurrent neural networks (RNNs), would also provide valuable insights. Testing the multiplier in real-world applications, such as edge computing or IoT devices, would help assess its performance under practical constraints.

By addressing these areas, future work can build on the foundation laid by this thesis to advance the field of approximate computing and hardware-accelerated machine learning, ultimately enabling more efficient and scalable implementations for a wide range of applications. The contrast between traditional Booth multipliers and our new approximate approach highlights the potential for innovation in this space, paving the way for future research and development.

SHORT BIOGRAPHY

Panagiotis Kazantzidis was born on September 1, 1999, in Kavala, Greece. In 2017, he began his undergraduate studies in Computer Science and Engineering in the University of Ioannina. After five years of dedicated study, he completed his undergraduate program in 2022 with a thesis titled " Encoding of transmission data in communication systems via visible trichromatic LED light" showcasing his interest in embedded systems and data encoding and communication.

In 2023, Panagiotis pursued a postgraduate degree in "Data and Computer Systems Engineering" with a specialization in "Advanced Computer Systems", further specializing in digital systems and computer engineering. His current research focuses on approximate computing using Booth multipliers, exploring energy-efficient and high-performance computing techniques. With a strong background in hardware design and computer architecture, his research interests lie at the intersection of digital circuit optimization, energy-efficient computing, and emerging hardware paradigms.

References

[1] V.K. Chippa, H. Jayakumar, D. Mohapatra, K. Roy, and A. Raghunathan. Energy-efficient recognition and mining processor using scalable effort design. In CICC, 2013.

[2] F.S. Snigdha, D. Sengupta, J. Hu, and S.S. Sapatnekar. Optimal design of jpeg hardware under the approximate computing paradigm. In DAC, page 106, 2016.

[3] H.A.F. Almurib, T.N. Kumar, and F. Lombardi. Approximate DCT image compression using inexact computing. TC, 67(2):149–159, 2017.

[4] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In ACM Sigplan Notices, volume 49, pages 269–284, 2014.

[5] M. Brandalero, A.C.S. Beck, L. Carro, and M. Shafique. Approximate on-the-fly coarse-grained reconfigurable acceleration for general-purpose applications. In DAC, pages 1–6, 2018.

[6] M.S. Ansari, V. Mrazek, B.F. Cockburn, L. Sekanina, Z. Vasicek, and J. Han. Improving the accuracy and hardware efficiency of neural networks using approximate multipliers. TVLSI, 28(2):317–328, 2020.

[7] J.N. Mitchell. Computer multiplication and division using binary logarithms. IRE Transactions on Electronic Computers, (4):512–517, 1962.

[8] V. Gupta, D. Mohapatra, A. Raghunathan, and K. Roy. Low-power digital signal processing using approximate adders. TCAD, 32(1):124137, January 2013..

[9] Approximate Arithmetic Circuits: A Survey,Characterization and Recent Applications Honglan Jiang, Member, IEEE, Francisco Javier Hernandez Santiago, Hai Mo, Leibo Liu∗,senior Member, IEEE, and Jie Han∗, Senior Member, IEEE.

[10] Neil Weste and David Harris. 2010. CMOS VLSI Design: A Circuits and Systems Perspective (4th. ed.). Addison-Wesley Publishing Company, USA..

[11] C. R. Baugh and B. A. Wooley. 1973. A Two's Complement Parallel Array Multiplication Algorithm. IEEE Trans. Comput. 22, 12 (December 1973).

[12] F. Farshchi, M.S. Abrishami, and S.M. Fakhraie. New approximate multiplier for low power digital signal processing. In CADS, pages 25–30, October 2013..

[13] M.A. Song, L.D. Van, and S.Y. Kuo. Adaptive low-error fixed-width Booth multipliers..

[14] J. Wang, S. Kuang, and S. Liang. High-accuracy fixed-width modified Booth multipliers.

[15] Y. Chen and T. Chang. A high-accuracy adaptive conditional-probability estimator for.

[16] H. Jiang, J. Han, F. Qiao, and F. Lombardi. Approximate radix-8 Booth multipliers for.

[17] Gavrilova, M., & Tan, C. J. K. (2014). Transactions on Computational Science XXII (1st ed.). Springer Berlin Heidelberg : Imprint: Springer..

[18] IEEE Standard for Floating-Point Arithmetic," in IEEE Std 754-2019 (Revision of IEEE 754-2008) , vol., no., pp.1-84, 22 July 2019.

[19] 2018, PYNQ-Z2 Reference Manual v1.0 17 May.

[20] K. Cho, K. Lee, J. Chung, and K.K. Parhi. Design of low-error fixed-width modified.