FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Design and Evaluation of Approximate Arithmetic Units for Machine Learning

Miguel José Nunes Almeida

Mestrado em Engenharia Eletrotécnica e de Computadores

Supervisor: João Paulo de Castro Canas Ferreira

October 11, 2023

Abstract

Approximate computing is an alternative to further technological development, allowing faster-performing circuits and saving energy consumption. It is viable to use in error-tolerant applications, such as neural networks and the most common image processing applications, that can tolerate non-significant calculation deviations. It can be performed in multiple different paradigms and forms from the software and hardware domains. This work focuses on the hardware domain and, more specifically, on the logical pruning of circuits and the discovery of circuits that perform similar functions to arithmetic units. Adders and multipliers were chosen as the main focus of the work developed, as they are two of the most vital circuits and the main ones used in neural networks. After the literature review, several circuits of each type are chosen to be implemented and tested in various use cases.

This work presents a simple neural network that uses approximate arithmetic units to perform additions and multiplications. The power consumption and resources utilised, as well as the impact they have on accuracy, are discussed. The modules are also tested before use in the neural network by performing simple additions and multiplications and by calculating the internal product of two vectors. Image processing applications are also tested by performing the sum of two images and a Sobel filtering operation for edge detection. Synthesis and implementation are done using Vivado 2022.2, targeting the Zynq-7000 Field Programmable Gate Array (FPGA).

Resumo

A computação aproximada é uma alternativa viável para o desenvolvimento tecnológico, permitindo circuitos de desempenho mais rápido e poupando no consumo de energia. A sua utilização é viável em aplicações tolerantes ao erro, como as redes neuronais e algoritmos de processamento de imagem, que podem tolerar desvios de cálculo não significativos. Este tipo de computação pode ser realizada em múltiplos paradigmas e de diferentes maneiras, tanto no domínio do software como do hardware. Esta dissertação centra-se no domínio do hardware e, mais especificamente, na redução lógica de circuitos e na descoberta de circuitos que desempenham funções semelhantes às unidades aritméticas. Os somadores e multiplicadores foram escolhidos como o foco principal do trabalho desenvolvido, uma vez que são dois dos circuitos mais vitais e os principais utilizados nas redes neuronais. Após revisão da literatura relevante, vários circuitos de cada uma destas unidade aritméticas foram escolhidos para serem implementados e testados em vários casos de uso.

Esta dissertação apresenta uma rede neuronal simples que utiliza unidades aritméticas aproximadas para efetuar adições e multiplicações. O consumo de energia e os recursos utilizados, bem como o impacto que têm na precisão, são discutidos. Os módulos são também testados antes de serem utilizados na rede neural, efectuando adições e multiplicações simples e calculando o produto interno de dois vectores. Usos em aplicações de processamento de imagem também são testados, efectuando a soma de duas imagens e um filtro Sobel para deteção de arestas. A síntese e a implementação são feitas com recurso ao Vivado 2022.2, com a *"Field Programmable Gate Array (FPGA)"* Zynq-7000 como alvo.

Agradecimentos

Gostaria de agradecer ao Professor João Canas Ferreira, pelo apoio prestado ao longo deste ano na elaboração deste documento e do trabalho nele contido. Queria também agradecer à minha família, em especial aos meus pais, que sempre acreditaram em mim e nas minhas capacidades. Agradeço também a todos os que me apoiaram e ajudaram neste percurso académico, com menção especial aos Cogumelos (vocês sabem quem são) e aos membros do NEEEC, companheiros de trabalho e de festa.

Miguel Almeida

“Queen Marika has high hopes for us.
That we continue to struggle, unto eternity...”

Sir Gideon Ofnir, The All-Knowing

Contents

List of Figures

List of Tables

Symbols and Abbreviations

ACLA	Approximate Carry-Lookahead Adder
AFA	Approximate Full Adder
AMACU	Approximate MAC Unit
AP	Acceptance Probability
ART-MAC	Approximate Rounding and Truncation based MAC
BEAD	Bounded Error Approximate Adder
BRAM	Block Random Access Memory
CBA1	Carry Based Approximate Adder type I
CBA4	Carry Based Approximate Adder type IV
CBAA	Carry Based Approximate Adder
CGP	Cartesian Genetic Programming
CJU	Carry Judge Unit
CLA	Carry Look-Ahead Adder
CORDIC	Coordinate Rotation Digital Computer
CSA	Carry Select Adder
CSPA	Carry Speculative Adder
DRAM	Dynamic Random Access Memory
DSP	Digital Signal Processing
ECU	Exact Carry Unit
ECT	Error Correction Term
ER	Error Rate
ETAIV	Error Tolerant Adder type IV
FA	Full Adder
FF	Flip Flop

FFT	Fast Fourier Transform
FMA	Fused Multiply-Add
FPGA	Field Programmable Gate Array
HA	Half Adder
HPAM	High-Performance Approximate Multiplier
HPARM	High-Performance Approximate Recursive Multipliers
ISA	Instruction Set Architectures
LOA	Lower-part-OR Adder
LPL	Lower Part Length
LSB	Least Significant Bit
LUT	Look-Up Table
MAA	Minimum Acceptable Accuracy
MAC	Multiply and Accumulation
MED	Mean Error Distance
MRE	Mean Relative Error
MRED	Mean Relative Error Distance
MSB	Most Significant Bit
MSE	Mean Square Error
NMED	Normalized Mean Error Distance
OOC	Out-of-Context
PSNR	Peak Signal-to-Noise Ratio
RCA	Ripple Carry Adder
RCPFA	Reverse Carry Propagate Full Adder
RED	Relative Error Distance
RoBA	Rounding Based Approximate Multiplier
SAIF	Switching Activity Interchange Format
SRAM	Static Random Access Memory
VLCSA	Variable Latency Carry Selection Adder
WL	Word Length

Chapter 1

Introduction

1.1 Context

To meet the ever more challenging technical demands, technological products nowadays have to perform better and consume less power, two requirements that typically conflict with each other. In the technological domain, these requirements are translated into the need for circuits that perform more functionalities in less area while operating at faster clock frequencies and consuming less power.

Creating circuits that meet the new technological requirements so far has been achieved by scaling down the transistor and supply voltage. This technology scaling is summarised by the famous *Moore's Law* [1], which states that the number of transistors in a microchip will double roughly every two years (being later restated to be every eighteen months), and to *Dennard's Scaling* [2], that states that with each new transistor generation the transistor length, capacitance and supply voltage is $\frac{1}{\sqrt{2}}$ of the previous one, the area and power consumption is half and the frequency is $\sqrt{2}$ of the previous generation. However, as both these statements tend towards an end, due to the rise of production costs per transistor (effectively annulling *Moore's law*) and due to the increase in power consumption and effect of minor parameter variations and leakage currents, new alternatives to technology scaling need to be found and researched to continue to increase performance while maintaining or reducing power consumption.

Specific applications, like neural networks and image processing algorithms, can tolerate minor errors and deviations in their calculations and intermediate values while still obtaining the correct and desired results and maintaining high performance, making them error-tolerant and resilient. For example, if an image processing algorithm does not output the correct pixel value for every pixel, the calculated values are close to the correct value. In that case, there may not exist a loss of functionality or information. Similarly, suppose the calculations in the different layers of a neural network do not lead to the exact result. However, these approximations do not significantly affect the neural network's accuracy. In that case, there are no losses in making these approximations. Therefore, performing these approximations to save resources and energy or obtain higher performances may be desirable.

Machine Learning algorithms are resource intensive and power-hungry, requiring an ever-increasing amount of energy and space to obtain their results. This problem escalates as Neural Networks with more Neurons and Layers are created in order to achieve higher accuracies or obtain solutions to more complex problems. Using inexact computations that do not significantly affect the results but can save on the used resources and lower power consumption significantly becomes an attractive approach to use these algorithms in Low Power applications, such as IoT devices.

Approximating or "inexactly computing and processing data in order to save power and achieve high performance, while maintaining results at an acceptable accuracy for use" is known as approximate computing [3]. It is considered one of the few approaches that can drastically reduce power consumption, possibly by an order of magnitude. This can be achieved by carefully controlling or inducing small errors after careful previous analysis, ensuring the possibility of measuring the error tolerance and quality of the approximation, usually on the least significant part of the hardware or algorithms, not to compromise the accuracy of the obtained results.

Even though extensive research is being done in approximate computing, some challenges still need to be addressed. Error analysis is typically based on metrics related to the specific application and use case, so no general error analysis method is available. Due to the immense diversity of use cases and techniques, error management becomes daunting without a general error analysis method. In most cases, error analysis of systems is based on simulations and iterative processes that may miss specific edge cases and are not generalisable. Extensive verification and testing of the implemented solutions are needed to guarantee that the results generated are acceptable. As such, verification and testing methodologies must be developed to detect critical faults that significantly affect the accuracy of results. Another aspect to pay close attention to is the reliability of an approximate system, as environmental variations and ageing may affect the systems in an undesirable way, leading to a shorter product lifespan and faster result deterioration than their exact counterparts [3].

Approximate computing can be done on different levels ranging from software to hardware. At the software and algorithm levels, we can apply precision scaling, reducing the operand bit-width to save computational and storage resources, skipping loop iterations can be used to reduce computation times, and energy-aware compiler options can be used to reduce power consumption. At a higher level, skipping tasks or sampling data instead of processing it can also be used to save power and resources while obtaining the desired results. Approximations can be made on devices, circuits and architectures at the hardware level. Possible approaches include voltage scaling, where higher supply voltages are used on critical sections of circuits to ensure accuracy while the supply voltage for the least significant parts is carefully reduced, and logic reduction or pruning, where the approximations are made by carefully removing parts of the logic circuit to consume fewer resources and obtain results faster, without compromising the accuracy of said results. Due to voltage scaling having high implementation cost due to the complex control of the supply voltages required, most approximate circuits are designed based on the logic pruning and reduction method, as it is easier to control and implement. This method has tested different logic unit designs, like the adder and multiplier, and more complex algorithm implementations, like the Coordinate Rotation Digital Computer (CORDIC) and the Fast Fourier Transform (FFT) algorithms. Another vital part of a computer system is memory. As such, approximate memory has been studied as an option to reduce power consumption, using techniques like reducing the refresh rate for Dynamic Random Access Memory (DRAM) for non-critical data, reducing the supply voltage in Static Random Access Memory (SRAM), and using inexact read/write operations for non-volatile memories. At a higher hardware level, approximate architectures have been studied for accelerating specific workloads. Examples are quality-programmable extended Instruction Set Architectures (ISA), reduced precision floating-point units and fixed-point units [3]. All these approaches follow significance-based criteria, where, in order not to negatively impact the results' reliability, the higher significance bits and circuits are either not approximated or suffer from only slight changes.

1.2 Objectives

As referred to in Section 1.1, there are many possible approaches and targets to approximate. However, not all of them are worked on during this dissertation. The work developed focuses on logic pruning and resource usage reduction in arithmetic units for machine learning applications. These include low-power adders, multipliers, and other units, such as Fused Multiply-Add (FMA) units designed at the gate level. They are used to create a small prototype implementation of a simple inference accelerator for digital neural to evaluate performance and accuracy in a real application. This prototype implementation is performed in an Field Programmable Gate Array (FPGA) to permit changing the unit being used frequently, allowing the test of multiple units. All units are characterised by their maximum clock frequency in the circuit, the occupied resources and the power consumption.

1.3 Document Structure

This document is divided into five chapters.

Chapter 2 presents the relevant concepts involved in this work, some exact circuits for some arithmetic units and functions and some approximate variations and approaches. The results reported and presented by the literature are compared and discussed to select the circuits to be implemented and the design flow to follow.

Chapter 3 presents the developed work during the dissertation. Firstly, the chosen arithmetic unit designs to be implemented are characterised by a single operation to perform a pre-selection in terms of accuracy. The methodology used in designing and testing them is also described in this chapter. Afterwards, a simple circuit for testing a simple use case is presented, making parallelisms to neural network inference.

Chapter 4 details the implementation target and the settings used during the synthesis, place and route process. It also has the results of the final vector product circuit, the accuracy obtained using different approximate arithmetic units and the predicted power consumption and resource consumption. Finally, it also presents the performance of the created modules in different applications, showcasing other use cases and scenarios.

Lastly, Chapter 5 concludes the document, presenting a general overview and outlining possible future work to be developed in the field.

Chapter 2

State of the Art

2.1 Introduction

This chapter contextualises the problem further and presents and discusses several key aspects necessary for the development of the dissertation, like detailing examples of error-tolerant applications. The state-of-the-art circuit designs for different arithmetic units are also discussed and reviewed based on different authors' proposals and developed work. Some exact designs are also shown to contextualise the designs being discussed. The main advantages, benefits and problems of each circuit are also presented, mainly regarding resource usage or power consumption. Lastly, some topics and key takeaways are highlighted in a small summary to close the chapter.

2.2 Error Tolerant Applications

As mentioned in section 1.1, some applications can tolerate deviations in their calculations and procedures without compromising the results produced due to their resilience to error nature. This resilience comes from not needing the exact output value to obtain the desired output or small parameter and value changes in the outputs not producing a noticeable change in quality; that is, there is low result degradation. As mentioned in section 1.1 with some simple use cases, two examples of this type of application are Neural Networks and Image Processing algorithms.

2.2.1 Neural Networks

Neural networks are computational models inspired by the complex interconnectedness of the human brain. They consist of interconnected artificial neurons organised into layers, allowing them to extract meaningful patterns and make predictions from a dataset (Figure 2.1). One remarkable characteristic of neural networks is their inherent error tolerance, which allows them to handle noisy or imperfect data and still provide reliable outputs. This means neural networks can effectively learn from data with slight errors or inaccuracies.

This error tolerance is obtained through the learning process and the architecture of neural networks. The learning process allows the networks to adapt to the data and overcome errors in the training set. The architecture of the network further enhances it. The interconnected layers distribute the computational load, preventing errors in individual components from significantly affecting the overall output. As such, using approximate arithmetic units in neural networks can be a promising approach to reducing these models' resource usage and power consumption without compromising their accuracy.

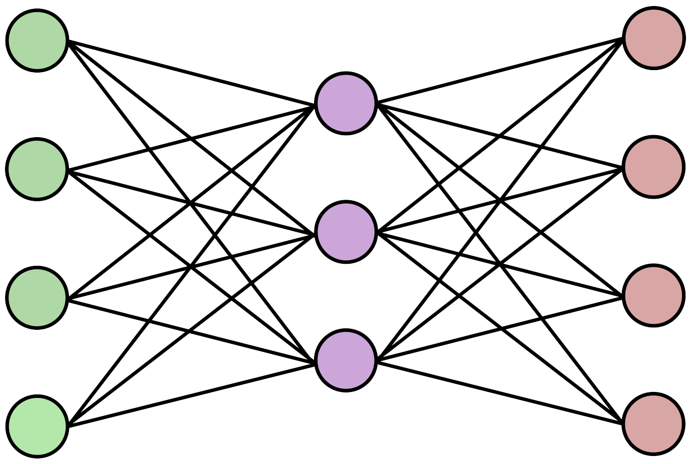


Figure 2.1: Simple Neural Network

2.2.2 Image Processing

Image processing algorithms, such as filters and other operations that apply kernel operations to an image, are error tolerant by nature, as a slight deviation of the output value of the pixel will not produce a visibly noticeable difference in the image and will usually not propagate to other stages of the processing algorithm.

In their simplest form, these operations consist of simple additions of a neighbouring pixel region the operation is being applied to, coupled with a check to verify that the corresponding output does not exceed the maximum value, usually 255. This output saturation deals with any possible overflows that might occur.

More complex kernel operations and filters apply weights to each of the neighbour pixels, usually powers of two. This can be achieved by shifting the values appropriately or being a simple signal change that does not require additional arithmetic units. As such, image processing algorithms and applications effectively test the performance and the errors introduced using approximate adders.

2.3 Error Metrics

In order to evaluate the performance of the different approximate arithmetic units, several error metrics are used to quantify the error introduced by the approximate circuits. These metrics are used to compare the results produced by the approximate circuits with the exact results produced by the exact circuits. Different metrics are used depending on the application and the desired results. The most common metrics are the Error Rate (ER), Mean Error Distance (MED), Normalized Mean Error Distance (NMED), Relative Error Distance (RED), Mean Relative Error Distance (MRED), Mean Square Error (MSE), Peak Signal-to-Noise Ratio (PSNR) and Image Difference.

ER is simply the percentage of cases where the obtained result differs from the exact result, being counted as an error. This error can be of small or large magnitude, so this metric cannot be used in isolation.

MED is the average absolute difference between the obtained and accurate results. NMED normalises this metric to make possible comparisons between different input ranges. Equations 2.1 and 2.2 show how these metrics are obtained.

$$MED = \frac{1}{N}\sum_{i=0}^{N}|Real_i - Approximate_i| \tag{2.1}$$

$$NMED = \frac{1}{N \times MAX}\sum_{i=0}^{N}|Real_i - Approximate_i| \tag{2.2}$$

RED and MRED are relative distance metrics. They incorporate the scale of the result into the calculation to give a better idea of the scale of the errors obtained. They can be obtained following equations 2.3 and 2.4.

$$RED = \sum_{i=0}^{N}\frac{|Real_i - Approximate_i|}{|Real_i|} \tag{2.3}$$

$$MRED = \frac{1}{N}\sum_{i=0}^{N}\frac{|Real_i - Approximate_i|}{|Real_i|} \tag{2.4}$$

MSE is also sometimes used as an error metric in approximate computing. By squaring the difference between the obtained result and the actual value, cases where the ER is low, but the error magnitude is high will have a high MSE. This metric is used mainly on image processing applications and can be determined using equation 2.5, where m and n are the sizes of the images.

$$MSE = \frac{1}{m \times n}\sum_{0}^{m-1}\sum_{0}^{n-1}\left[Real(i,j) - Appr(i,j)\right]^2 \tag{2.5}$$

For image processing applications, PSNR and Image Difference are commonly used metrics. The first one is defined using the MSE and is obtained by using a logarithm, as described in equation 2.6, where MAX represents the maximum pixel value, which for greyscale images is 255. The second one, image difference, can be calculated in a variety of different ways, from simply calculating the differences between image pixels to obtaining the average root-mean-square of the pixel differences of the precise and approximate outputs (Equation 2.7).

$$PSNR = 10 \times \log_{10}\left(\frac{MAX^2}{MSE}\right) \tag{2.6}$$

$$ImgDiff = \sqrt{MSE} \tag{2.7}$$

The state-of-the-art designs for the different arithmetic units also use other error metrics depending on the application and desired comparison points. These metrics will be detailed in the subsection they are mentioned.

Not all of these metrics are used to evaluate the performance of every approximate circuit, as some are more suited to specific applications and use cases. Due to this, a direct comparison between different approximate circuits is not always possible, as the metrics used to evaluate them are not the same and sometimes are even defined differently, as seen in the following sections.

2.4 Adders

Adders are the building blocks of most operations. They are simple circuits that sum two numbers, producing a result with the same length in bits and a possible extra bit as carry.

2.4.1 Ripple Carry Adder

The Ripple Carry Adder (RCA) is the simplest multi-bit adder (Figure 2.2). It consists of a chain of full adders, where the carry output of each adder is connected to the carry input of the next adder. Each adder in the chain receives two input numbers and a carry. The last adder produces the carry output in addition to the sum bit. The RCA is the slowest adder, as the carry propagation has to go through the entire chain of adders. The RCA is also the adder that uses the least resources, as it only uses full adders [4].

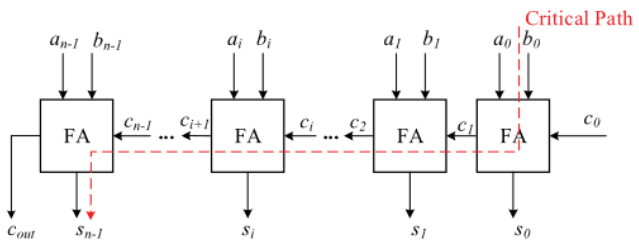


Figure 2.2: 4-bit Ripple Carry Adder from [5]

2.4.2 Carry Select Adder

The Carry Select Adder (CSA) is a parallel adder that consists of two RCAs and a multiplexer (Figure 2.3). The two RCAs receive the two input numbers and produce two possible results, one with a carry of 0 and another with a carry of 1. The multiplexer selects the correct result based on the carry input. The CSA is faster than the RCA, as the carry propagation is split between the two RCAs. However, the CSA uses more resources than the RCA, as it uses two RCAs and a multiplexer [4].

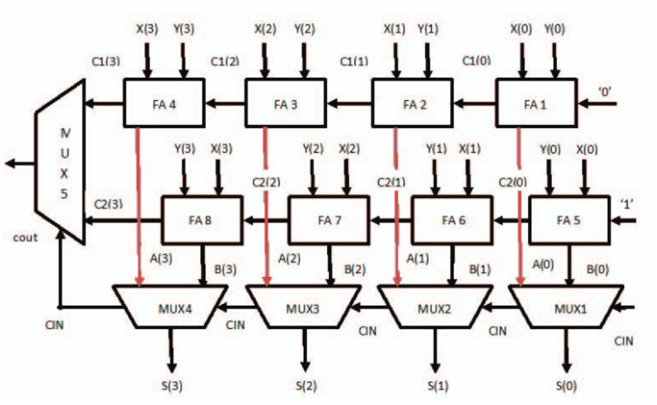


Figure 2.3: 4-bit Carry Select Adder from [4]

2.4.3 Carry Look-Ahead Adder

The Carry Look-Ahead Adder (CLA) is a parallel adder that consists of a chain of full adders and a set of logic gates that are responsible for generating and propagating the carry inputs(Figure 2.4). The CLA is faster than the CSA, as the carry propagation is split between the logic gates and the full adders. However, the CLA uses more resources than the RCA and has a higher power consumption, as it uses a chain of full adders and a set of logic gates [4].

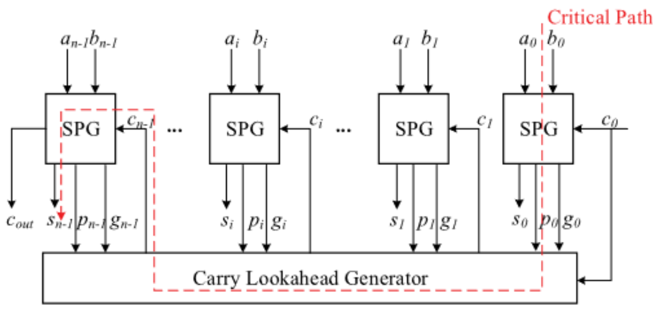


Figure 2.4: 4-bit Carry Look-Ahead Adder from [5]

The propagate and generate logic gates are obtained based on Equations 2.8 to 2.12.

$$
\begin{aligned}
P(i) &= A(i) \text{ xor } B(i) && (2.8)\\
G(i) &= A(i) \text{ and } B(i) && (2.9)\\
S(i) &= P(i) \text{ xor } C(i) && (2.10)\\
C(1) &= G(0) \text{ or } (P(0) \text{ and } C(0)) && (2.11)\\
C(2) &= G(1) \text{ or } (P(1) \text{ and } G(0)) \text{ or } (P(1) \text{ and } (P(0) \text{ and } C(0))) && (2.12)
\end{aligned}
$$

2.4.4 Carry Based Approximate Adders

In order to decrease area and power consumption, the Carry Based Approximate Adder (CBAA) alters the conventional exact Full Adder (FA), removing gates to save resources, leading to errors in the obtained result. Several implementations of CBAA exist, but type I and type IV should be mentioned, as the former leads to the lowest power consumption and the latter to the highest accuracy results.

Type I

The single bit Carry Based Approximate Adder type I (CBA1) [6] is obtained using one AND gate, one OR gate and a single inverter gate as shown in Figure 2.5. The expressions to obtain the carry and the sum bits are simple to obtain (Equations 2.13 and 2.14 from [6]).

$$
\begin{aligned}
Carry1 &= (a.b) + c && (2.13)\\
Sum1 &= \overline{carry1} && (2.14)
\end{aligned}
$$

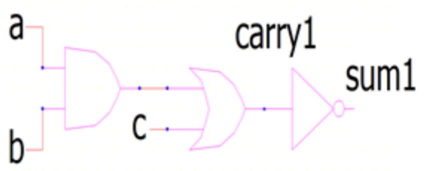


Figure 2.5: Carry Based approximate Adder1 logic circuit from [6]

Type IV

The single bit Carry Based Approximate Adder type IV (CBA4) [6] is obtained using three AND gates, two OR gates and a single inverter gate as shown in Figure 2.6. This configuration leads to no errors in the carry bit generated. The boolean expressions to obtain the carry and sum bits are expressed in Equations 2.15 and 2.16 from [6].

$$Carry4 = a.b + b.c + c.a \tag{2.15}$$

$$Sum4 = \overline{carry4} \tag{2.16}$$

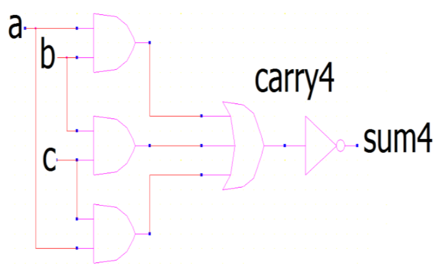


Figure 2.6: Carry Based approximate Adder4 logic circuit from [6]

Analysis

Analysing the logic circuits for each of these configurations, it is possible to construct the truth table for all input possibilities, as seen in Table 2.1. A simple analysis of this truth table shows that when both inputs are 0 and there is no carry, the sum output is 1. This is the primary source of error for these adders and is present in both configurations. When chaining these adders to create a multi-bit adder, this error makes it so that using unsigned numbers of lower magnitude, but with a high bit width will lead to drastically inaccurate results. The most extreme case for this is adding 0 with 0, which results in the maximum value representable for the specific bit width. Using signed numbers with two's complement, these errors are less significant, as shown in Table 2.2.

Table 2.1: Truth Table for Carry Based Approximate Adders adapted from [6]

Inputs			Accurate Outputs		Approximate Outputs			
					CBA1		CBA4	
A	B	Cin	Sum	Cout	Sum	Cout	Sum	Cout
0	0	0	0	0	1	0	1	0
0	0	1	1	0	0	1	1	0
0	1	0	1	0	1	0	1	0
0	1	1	0	1	0	1	0	1
1	0	0	1	0	1	0	1	0
1	0	1	0	1	0	1	0	1
1	1	0	0	1	0	1	0	1
1	1	1	1	1	0	1	0	1

When the bit width increases, the error introduced by these circuits lowers. Both approximate circuits reduce the used area and the consumed power significantly, according to the tests performed by Ramasamy et al. [6]. However, the CBA4 does not obtain any significant speed improvements, even being slower than the conventional adder for higher bit widths.

Table 2.2: CBAA Performance Comparison adapted from [6]

Bit Width	Type	Delay (ns)	Area (μm)	Power(μW)	Error (%)
16	Conventional	1.45	684.15	1940	0
	CBAA - Type I	0.86	262.61	60.79	6.2
	CBAA - Type 4	1.37	403.50	503.14	3.2
32	Conventional	2.52	1368.15	2880	0
	CBAA - Type I	1.69	525.08	94.72	4
	CBAA - Type 4	2.52	806.87	1010	2
64	Conventional	4.61	2736.27	7760	0
	CBAA - Type I	3.46	1049.97	159.14	1.2
	CBAA - Type 4	4.91	1613.56	2050	0.062
128	Conventional	8.87	5472.29	15540	0
	CBAA - Type I	6.78	2099.85	290.88	0.005
	CBAA - Type 4	9.52	3227.02	4070	0.002

2.4.5 Reverse Carry Propagate Full Adders

Reverse Carry Propagate Full Adder (RCPFA) are “a group of adders where the carry is propagated in reverse order to eliminate the dependency of output generation on preceding lower carry bits” [7]. Reversing the carry propagation order provides two advantages: accelerating the Most Significant Bit (MSB) outputs and reversing the error propagation, spreading the error through the Least Significant Bit (LSB).

Reverse adders take one more input signal than normal FAs, F_i, generated from an auxiliary prediction function that uses A_{i-1} and/or B_{i-1} as inputs, to generate outputs sum (S_i) and carry (C_i) bits when (S_i - C_i = 0). Equation 2.17 represents the Reverse adder operation [7].

$$S_i - C_i = A_i + B_i - 2C_{i+1} \tag{2.17}$$

There are different implementations of RCPFAs, but only one will be focused on in detail in this subsection. That is the RA4 implementation from [7]. In this circuit (Figure 2.7), the extra input F_{i+1} is generated through a NAND gate of the two input bits and the sum and carry bits can be obtained through Equations 2.18 and 2.19.

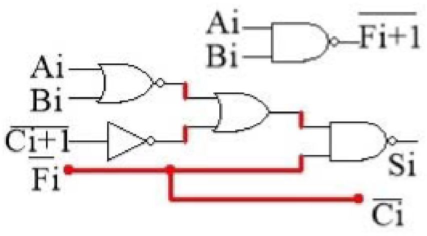


Figure 2.7: Reverse Carry Propagate Full Adder logic circuit from [7]

$$C_i = F_i \tag{2.18}$$

$$S_i = ((A_i \text{ nor } B_i) \text{ or } C_{i+1}) \text{ nand } F_i \tag{2.19}$$

Following these equations, we can construct the truth table for this logic circuit, shown in Table 2.3

Table 2.3: Reverse Carry Propagate Full Adder Truth Table from [7]

A_i	B_i	C_{i+1}	F_i	S_i	C_i	F_{i+1}
1	1	x	1	1	1	1
1	1	1	0	0	0	1
1	1	0	0	1	0	1
1	0	x	1	1	1	0
1	0	1	0	0	0	0
1	0	0	0	1	0	0
0	1	x	1	1	1	0
0	1	1	0	0	0	0
0	1	0	0	1	0	0
0	0	x	1	1	1	0
0	0	x	0	0	0	0

The 1-bit adder design was characterised by Singh et al. by the delay to get each output, power consumption and area used, and these values were compared to the exact 1-bit FA. Table 2.4 shows these values, including the delays for generating Sum (S_i), Carry (C_i), and Auxiliary-signal(F_i). The design shows profound benefits when compared with the exact FA design in all aspects evaluated.

The design was then tested by making a 32-bit approximate adder using the design for the 21 LSBs and calculating the outputs for 250000 random inputs, and the MED and the MRED were evaluated. In both of these metrics, the design had a relatively high value, meaning that the circuit introduces a substantial amount of error that could lead to a more significant loss of information or data quality than desired.

Table 2.4: RCPFA Performance adapted from [7]

Adder	T_s (ps)	T_c (ps)	T_f (ps)	Power (nW)	Area (μm^2)
FA	475	374	0	63.98	5.1
RCPFA	310	200	223	34.49	2.7

2.4.6 Lower-Part-OR Adder

The Lower-part-OR Adder (LOA) divides a p-bit adder into two smaller sub-adders, one with m bits and the other with n bits [8]. One of these sub-adders is a precise adder that accurately computes the upper part of the MSBs. The other one comprises OR gates that obtain an approximate result for the p LSBs. An extra AND gate generates the carry input for the precise adder in the MSB. This helps consider the carries from the lower to upper parts of the LOA adder to decrease imprecision. For a LOA with a constant Word Length (WL) higher Lower Part Length (LPL) values decrease the area, delay, and power consumption of the LOA while at the same time increasing its imprecision [8].

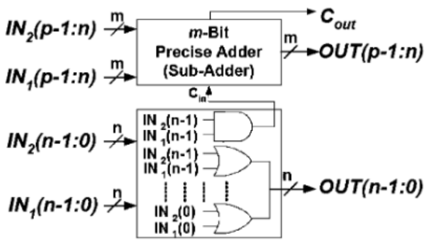


Figure 2.8: Lower-Part-OR Adder structure from [8]

The error probability of a LOA is given by Equation 2.20 and shows that in a LOA, the error is independent of the WL, depending only on the LPL. From the equation, it is also possible to understand that the error probability of this circuit increases rapidly as more bit adders are replaced with bitwise OR gates. However, the magnitude of these errors is contained to the values shown in Equations 2.21 and 2.22. An essential feature of the LOA is that it has a nearly unbiased and symmetric output error range [8], which means chaining these adders will not cause a considerable error accumulation.

$$P_e = 1 - \left(\frac{3}{4}\right)^{LPL} \tag{2.20}$$

$$MAX_{LOA} = 2^{LPL-1} - 1 \tag{2.21}$$

$$MIN_{LOA} = -2^{LPL-1} \tag{2.22}$$

2.4.7 Approximate Carry-Lookahead Adder

The Approximate Carry-Lookahead Adder (ACLA) architecture, illustrated in Figure 2.9, adds two n-bit binary numbers by dividing them into n/k blocks, where k is the size of each block. Each block is fed an input of k bits from an n-bit binary number [9]. The adder's block structure includes three units: an Exact Carry Unit (ECU), a Carry Judge Unit (CJU), and a Sub Adder, as well as an OR gate. As an approximate parallel adder, each of the n/k blocks in the proposed adder does not rely on the previous blocks to calculate C_{out}. Only the C_{out} from the previous block is used as the C_{in} for calculating the block sum. The ECU generates the C_{ECU}, while the CJU predicts the C_{CJU} for its respective block, which is then fed into an OR gate along with the C_{ECU}. The overall equation for C_{out} can be written as shown in equation 2.23.

$$C_{out} = C_{ECU} + C_{CJU} \tag{2.23}$$

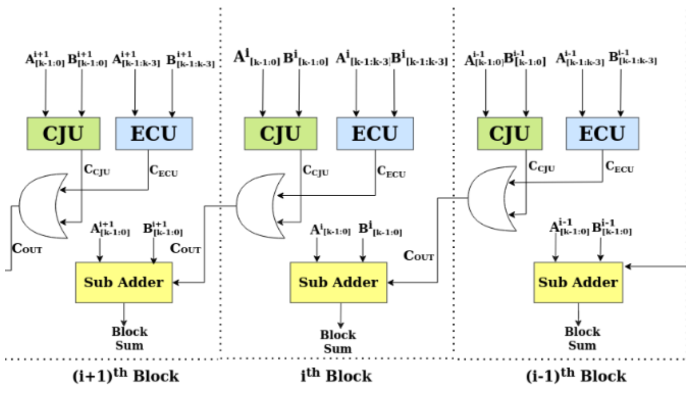


Figure 2.9: Cascaded blocks of Approximate Carry-Lookahead Adder from [9]

The ECU in Figure 2.9 uses only the three most significant bits of the given block size of k bits, as they have the highest likelihood of contributing to $C_{out} = 1$. The ECU employs the Carry-Lookahead Adder principle to compute the C_{ECU}, as the CLA has a low delay in transmitting the carry.

Using equations 2.24 and 2.25, we can calculate the C_{ECU} for the given block:

$$P_i = A_i \oplus B_i \tag{2.24}$$

$$G_i = A_i.Bi \tag{2.25}$$

$$C_{ECU} = G_0 + P_0.G_1 + P_0.P_1.G_2 \tag{2.26}$$

Equation 2.26 presents the C_{ECU} equation using the 3 MSBs of the block. The adder's accuracy can be adjusted by increasing or decreasing the most significant bits used to calculate the C_{ECU}.

The CJU estimates or predicts whether the carry is generated by less significant bits than the upper three bits. It does this by multiplying the carry generation term of bits less significant than the upper three bits with the carry propagation term of the block's upper two most significant bits. An Error Correction Term (ECT) will also be multiplied along with the carry generation term to check whether the third most significant bit of the block propagates the generated carry.

Since ACLA was made considering the high area and power consumption of CLA, it is optimised to use less area and resources. As such, it occupies less area than the CLA, and because it does not use all bits to calculate the carry in most configurations, it also consumes less power. Regarding delay, ACLA is significantly faster than the CLA. This is due to the parallel computation by the two blocks CJU and ECU. However, ACLA is slower than some state-of-the-art approximate adders as it has one extra unit, the CJU, which increases its delay to improve result accuracy.

2.4.8 Carry Speculative Adder

The Carry Speculative Adder (CSPA) (Figure 2.10) contains block adders and carry predictor circuits. Each block adder has size x. An n-bit CSPA has $m = \frac{n}{x}$ block adders and $(m-1)$ carry predictor circuits [10]. Each predictor circuit predicts the carry-out bit of a corresponding block adder. Each block adder has three main components: a carry generator, a sum generator and a multiplexer. The carry generator produces carry signals that can be used in the sum generator. The sum generator calculates the partial sum bits of the block adder. Finally, the multiplexer uses the previous carry predictor output as a selector to select one of the signals generated by the carry generator to be used in the sum generator.

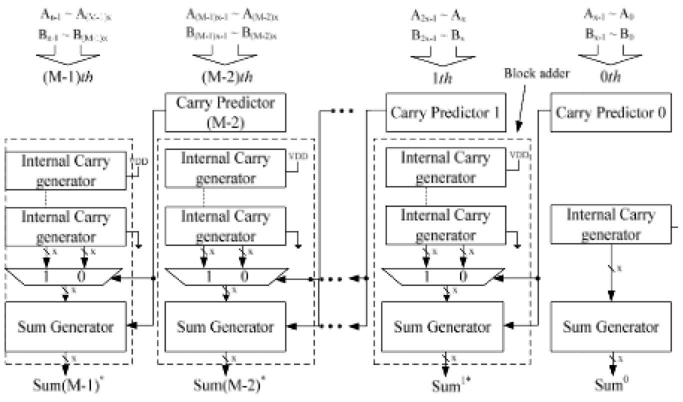


Figure 2.10: Carry Speculative Adder Architecture from [10]

To speed up the carry and sum bits calculation, a modified FA can be used that uses extra logic gates in order to obtain the carry output in two logic stages (as opposed to the three stages a regular FA). However, opting for this implementation leads to higher power consumption. In order to reduce hardware complexity and area overhead, the CSPA carry predictor circuit uses only the input bits near the MSB to predict the carry-out bit of a block adder. This is because the probability that a carry-out bit depends on the k previous bit positions is 1/2k, meaning that the input bits near the LSB have a low probability of affecting the carry-out bit.

Analysis of the occupied area and circuit delay made in [10] show that the CSPA is better than other approximate speculative adders in these aspects, occupying approximately 10% less area and having lower critical path delay. Analysis of errors shows that this design has higher error rates than other speculative adders. However, the error significance (MRED) of the CSPA is low, causing results to still be within acceptable ranges.

2.4.9 Variable Latency Carry Selection Adder

The distribution of inputs affects the performance of speculative and variable latency adders. Mathematical distributions are often employed to predict or emulate the practical inputs. In practice, smaller numbers tend to appear more frequently than large ones. A non-trivial portion of carry chains is as long as the adder size for signed Gaussian inputs. These long carry chains significantly increase the error rate of the speculative addition. As such, creating approximate models considering the type of inputs is essential, as the circuit can be made to perform better specifically on said inputs. In the case of signed Gaussian inputs, the Variable Latency Carry Selection Adder (VLCSA) proposed in [11] (Figure 2.11) is an example of how a circuit can be created having in mind the use case it will have.

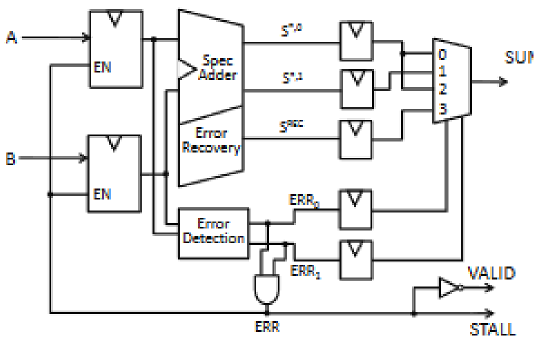


Figure 2.11: Variable Latency Carry Selection Adder Implementation from [11]

The VLCSA attempts to speculate when very long carry chains will occur, which in 2's complement signed numbers usually occurs when adding a small positive number with a small negative number where the MSB position is affected. Two error detection signals exist to detect these scenarios: ERR_0 and ERR_1. These signals can be defined according to Equations 2.27 and 2.28, where P^i is a propagate signal from the ith adder and G^i is the generate signal from the ith adder [11].

$$ERR_0 = \sum_{i=0}^{\lceil \frac{n}{k} \rceil - 2} P^{i+1}_{k-1:0} G^{i}_{k-1:0} \tag{2.27}$$

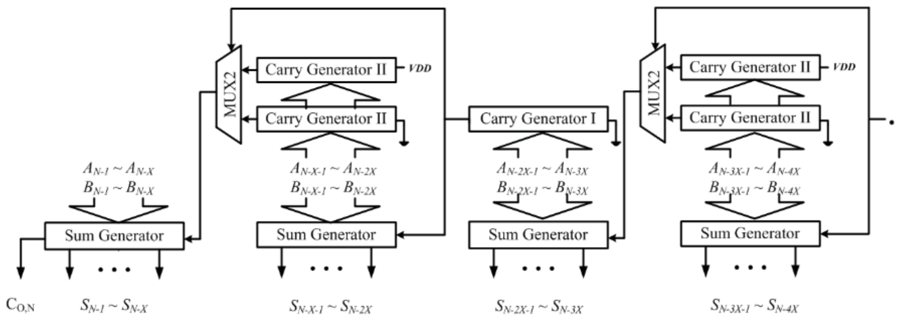
$$ERR_1 = \sum_{i=0}^{\lceil \frac{n}{k} \rceil - 2} \overline{P^{i+1}_{k-1:0}} P^{i}_{k-1:0} \tag{2.28}$$

If $ERR_0 = 1$ and $ERR_1 = 0$, a long carry chain occurred and did not incur errors and the correct result is in signal $S^{*,1}$. In the case where both signals are equal to 1, a carry chain starts and ends before reaching the MSB position, and error detection recognises this as an error. Finally, if $ERR_0 = 0$, the speculative result is correct and found in signal $S^{*,0}$. If an error is detected, an error recovery circuit gives the correct result by using intermediate results and an accurate adder. This circuit adds area overhead and has a higher delay, but it helps improve the accuracy of the results. These error signals are selectors in a final multiplexer to select the correct result, as seen in Figure 2.11.

To estimate the error rates of VLCSA, Monte Carlo simulations for 1 million 2's complement Gaussian inputs were performed [11]. For the Gaussian distribution, the mean is $\mu = 0$, and the standard deviation is $\sigma = 2n$, where n is the adder width. By varying the window size of the speculative adder, accuracies of up to 99.99% were obtained. However, due to the implementation of the error recovery mechanism that allows for these high accuracies, the area occupied by the adder and the delay increases significantly, so relaxing the accuracy requirements is necessary to achieve significant improvements in these aspects.

2.4.10 Error Tolerant Adder type IV

The Error Tolerant Adder type IV (ETAIV) uses a 2-to-1 multiplexer (MUX2) to break carry chains into two stages and use two Carry Generators, type I and II, to generate the carry bits. As illustrated in Figure 2.12, an N-bit ETAIV adder is divided into $\frac{N}{X}$ blocks of size X [12].



Carry Generator II
VDD
Carry Generator II
MUX2
Carry Generator I
Carry Generator II
VDD
Carry Generator II
MUX2
$A_{N-1} \sim A_{N-X}$
$B_{N-1} \sim B_{N-X}$
$A_{N-X-1} \sim A_{N-2X}$
$B_{N-X-1} \sim B_{N-2X}$
$A_{N-2X-1} \sim A_{N-3X}$
$B_{N-2X-1} \sim B_{N-3X}$
$A_{N-3X-1} \sim A_{N-4X}$
$B_{N-3X-1} \sim B_{N-4X}$
Sum Generator
Sum Generator
Sum Generator
Sum Generator
$C_{O,N}$
$S_{N-1} \sim S_{N-X}$
$S_{N-X-1} \sim S_{N-2X}$
$S_{N-2X-1} \sim S_{N-3X}$
$S_{N-3X-1} \sim S_{N-4X}$

Figure 2.12: Error Tolerant Adder type IV Block Diagram from [12]

Both Carry Generator types have X bits and perform their functions simultaneously, calculating the carry outputs with X bits accuracy. The greater the number of bits considered, the higher the chances of calculating the correct carry. However, considering more bits lead to an increase in the time needed to calculate the carry. The type II Carry Generators calculate the carry assuming an input of either '0' or '1'. The MUX2 selects the correct carry based on the value calculated from the type I Carry Generator, selecting the same as this select signal to be used as input in the following sum generator block. This allows for faster operation while maintaining the correct carry signal.

To implement ETAIV, the block size X must be chosen to guarantee accurate results but maintain a low power wastage and high performance. To help in this process, Zhu et al. [12] used two parameters: Minimum Acceptable Accuracy (MAA) and Acceptance Probability (AP). MAA is a minimum accuracy threshold results need to pass to be called acceptable, and AP is the probability that the adder has higher accuracy than the MAA.

In order to test the accuracy of the ETAIV, simulations were conducted using random 32-bit inputs for various MAAs and the corresponding APs. The results can be found in Table 2.5, and it can be verified that using a block size of 4 bits leads to acceptable accuracies.

2.4.11 Bounded Error Approximate Adder

The Bounded Error Approximate Adder (BEAD) structure divides a w-bit adder into M sub-adder segments, each with $l = \frac{w}{M}$ bits. Each segment's last FA is modified to compensate for a possible

Table 2.5: APs of 32 Bit ETAIV with varying MAAs adapted from [12]

	Number of Bits per block				
MAA (%)	X = 1	X = 2	X = 4	X = 8	X = 16
100	0.0430	0.4136	0.9136	0.9985	1.0
99	0.6466	0.8848	0.9917	1.0	1.0
95	0.7101	0.9214	0.9981	1.0	1.0
90	0.7374	0.9249	1.0	1.0	1.0

wrong carry propagation to the next block [13]. The structure for this implementation can be seen in Figure 2.13.

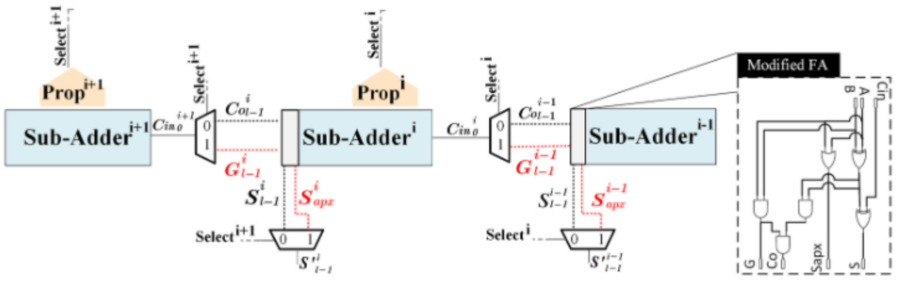


Figure 2.13: Structure of the Bounded Error Approximate Adder from [13]

Unlike other approximate segmented adders that rely on carry prediction schemes to achieve faster accurate results, the BEAD aims to improve the accuracy of its calculations directly in the sum calculation at stages where the critical path is shortened. The equations to obtain the carry predict and approximate sum signals of the *i*th sub-adder block are shown in Equations 2.29 and 2.30, where A^i_{l-1} and B^i_{l-1} are operand bits and G^i is the generate signal of the last FA of the *i*th sub-adder block [13].

$$C^i = G^i \tag{2.29}$$

$$S^i_{apx} = A^i_{l-1} + B^i_{l-1} \tag{2.30}$$

In order to test the accuracy of the design, relative error distance-based metrics were used, namely the MRED, the NMED and the maximum RED. To calculate these metrics, Khaksari et al. [13] applied 10 million uniform random numbers followed by another 10 million log-uniformly distributed random numbers to cover an extensive range of inputs. These metrics are summarised in Table 2.6 for the 16 and 32-bit adder configurations, using three different values of l, the sub-adder size.

Compared with other segmented adder configurations, the MRED and NMED of the BEAD are significantly smaller, leading to about a 40% improvement. Larger sub-adder block sizes lead to a higher accuracy adder and a more minor speed increase. Additionally, the maximum RED never exceeds 21% in the worst-case scenario, achieving results below 5% when using 8-bit sub-adders.

Table 2.6: Error metrics for different sub adder sizes of 16 and 32-bit BEADs adapted from [13]

Block Size	16 bit			32 bit		
	%MRED	%NMED	$\%RED_{max}$	%MRED	%NMED	$\%RED_{max}$
2	4.3	4.0×10^{-2}	21	3.7	3.4×10^{-2}	21
4	1.4	1.1×10^{-2}	21	1.1	0.9×10^{-2}	12
8	0.2	0.2×10^{-2}	0.5	0.1	0.08×10^{-2}	4.9

To study the power, delay and area this adder configuration occupied, the authors of [13] synthesised and compiled it using advanced technology. Configurations using the 8-bit sub-adder blocks are more efficient, and ones using 2-bit sub-adder blocks have the worst results, on average. All configurations achieve a better power-delay-area product than the CLA adder.

2.4.12 EvoApproxLib Adders

The EvoApproxLib library [14] is a collection of hardware and software models of approximate circuits that are designed to be easily used in arbitrary applications. It contains C, Matlab and Verilog descriptions of different circuits generated using multi-objective Cartesian genetic programming.

Cartesian Genetic Programming (CGP) is a form of automatic evolution of computer programs and other computational structures using ideas inspired by Darwin's theory of evolution by natural selection. It uses directed acyclic graphs represented in a two-dimensional grid to achieve these results, unlike traditional genetic programming, where the solution is represented as a tree [15].

The CGP approach utilises a set of processing nodes (gates) arranged in c columns and r rows to represent a target circuit. Each node can perform one function. Connections between the elements give rise to a functional circuit.

As such, each gate can be connected to the output of a gate placed in previous columns or to one of the circuit inputs. The algorithm has a user-defined parameter that can set how many columns back a gate can connect to. For example, if this parameter is 1, the input of a node in column x can only be connected to an arbitrary output of a node in column $x-1$ or one of the circuit's inputs.

The first population is randomly generated and evaluated at the beginning of evolution. In order to create a new population, the chromosome of the highest-scored candidate circuit is selected as the new parent, and by applying a point mutation, offspring are generated in the count needed to fill up the rising population. The steps of the evolution loop are repeated in the next generations until either a circuit is found whose output fulfils the criteria specified or a maximal generation count is reached [16].

The highest-scored candidate is calculated based on fitness functions. These can be as simple as the number of correct bits on the output or more complex like the mean error distance represented in equation 2.31.

$$fit = \sum_{j=1}^{k} |y(j) - t(j)| \tag{2.31}$$

The examples above and early implementations only had one objective to optimise with the algorithm, error. However, other possible objectives can be considered, defining multi-objective Cartesian Genetic Programming.

In multi-objective optimisation, the Pareto front or Pareto curve is the set of all Pareto efficient solutions. These solutions are ones where there is no possibility of improving one parameter without worsening the others. It allows one to analyse only a specific optimal subset of solutions and make unbiased decisions between different trade-offs [17].

Various strategies can be used for multi-objective Genetic Programming. One of them is the Non-dominated Sorting Genetic Algorithm II. This selection algorithm arranges all the individuals within the population into a series of Pareto fronts. It promotes widespread results on the primary fronts and avoids filling the population with closely-clustered individuals, thus maintaining diverse solutions.

The primary fitness measure for each circuit evaluated is the circuit functionality; circuits that are not functionally correct are not evaluated further and are omitted. For circuits that pass this test, four fitness scores are calculated equally weighted within the fitness calculation [17]. These four objectives are:

- number of logic gates used;
- number of transistors used;
- longest gate-level path length between input and output;
- longest approximated delay between input and output

After performing these algorithms, the resulting circuits are presented in a public Github repository that contains different bit-width signed and unsigned adders, grouped by Pareto curve optimisation considered [14].

2.5 Multipliers

Multipliers are digital circuits used to perform the multiplication of two numbers. They are essential to many applications, including signal processing, computer graphics, and scientific computing.

Multipliers typically have three main stages: the partial product generation stage, the partial product accumulation stage, and the final addition stage. In the partial product generation stage, the multiplier generates the partial products for each multiplicand bit. In the partial product accumulation stage, the partial products are accumulated and aligned to form the final product. Finally, the final result is formed by adding the accumulated partial products in the final addition stage. Approximate multipliers usually focus on approximating only one of these stages.

This section will introduce two exact architectures and the most relevant approximate approaches to multiplier design. They will compare their benefits in speed and area and their losses in precision and accuracy.

2.5.1 Dadda Multiplier

The Dadda Multiplier is a tree multiplier that implements the addition of partial products very efficiently using full adders and half adders [18]. The partial products are generated using AND gates on the necessary input bits. Compressors can also be used to optimise the process further. The main advantage of this multiplier configuration is that it reduces the logic in each multiplier layer to the minimum possible, leading to lower power consumption than other exact multipliers.

2.5.2 Wallace Multiplier

The Wallace Multiplier is one of the most efficient tree multipliers frequently used in microprocessor circuits [19]. They are mainly used when performing multiplication with a low delay is desired. The algorithm for this multiplier consists of generating the partial products; using adders to add groups of partial products, generating carries that will be rippled to the next column; and proceeding to group the outputs of these adders with the remaining partial products.

The dot representation of an example of this architecture is shown in Figure 2.14, where '.' indicates partial products and 'o' indicates generated carries.

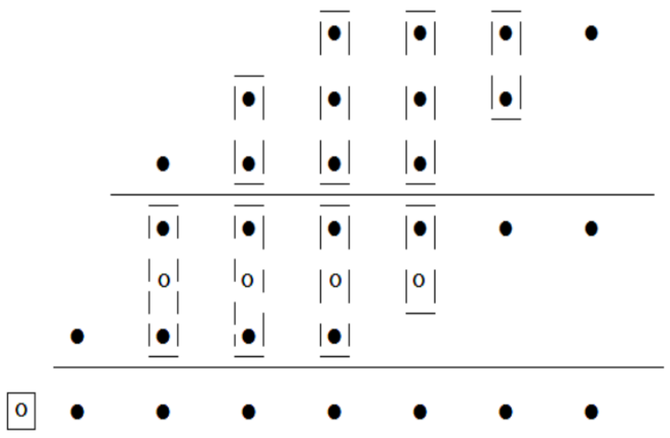


Figure 2.14: 4-bit Wallace Multiplier Dot Representation from [19]

2.5.3 Approximate Compressors

One of the most common approaches to approximate multiplication is to create approximate compressors to speed up the partial product accumulation stage. These compressors can be used in Tree Multipliers to speed up their operation effectively. A conventional four-input, two-output compressor is implemented using two FAs, as seen in Figure 2.15.

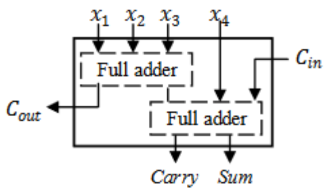


Figure 2.15: Conventional 4:2 Compressor from [20]

Different approaches have been used to create approximate compressors depending on the application and constraints. In the case of Lin and Lin [21], the proposed compressor maintains most of the logic of an exact 4:2 compressor, replacing only one XOR gate with a multiplexer (Figure 2.16), while in the case of Krishna et al. [22] most logic is simplified, using only AND and OR gates, as well as a multiplexer to obtain the results (Figure 2.17). Both compressors count the number of inputs that are logic value '1' and, as such, always fail when all four inputs are '1', as it only uses two bits to represent the result, which happens with a probability of $\frac{1}{256}$, since the probability of a partial product being '1' is $\frac{1}{4}$. The circuit results for both compressors can be found in Table 2.7, where the {Carry,S} bit pair gives the number of logic '1' inputs, and incorrect results appear in red.

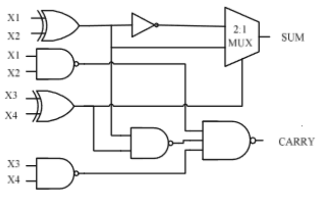


Figure 2.16: Approximate 4:2 Compressor proposed by [21]

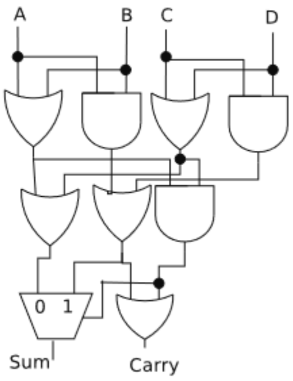


Figure 2.17: Approximate 4:2 Compressor proposed by [22]

As it is possible to see in Table 2.7, the second design has a higher error rate, outputting an incorrect value in $\frac{3}{16}$ths of cases. However, as it does not use XOR gates, this design's delay and

Table 2.7: Approximate 4:2 Compressor Truth Tables adapted from [21], [22]

				Lin and Lin		Krishna et al	
A	B	C	D	Carry	S	Carry	S
0	0	0	0	0	0	0	0
0	0	0	1	0	1	0	1
0	0	1	0	0	1	0	1
0	0	1	1	1	0	1	1
0	1	0	0	0	1	0	1
0	1	0	1	1	0	1	0
0	1	1	0	1	0	1	0
0	1	1	1	1	1	1	1
1	0	0	0	0	1	0	1
1	0	0	1	1	0	1	0
1	0	1	0	1	0	1	0
1	0	1	1	1	1	1	1
1	1	0	0	1	0	1	1
1	1	0	1	1	1	1	1
1	1	1	0	1	1	1	1
1	1	1	1	1	0	1	1

circuit area are better, leading to it being used in applications that require faster results and may tolerate higher error rates.

2.5.4 High-Performance Approximate Multiplier

Most approximate multipliers only apply approximation strategies in one of the three main stages that compose a multiplier circuit. However, the High-Performance Approximate Multiplier (HPAM) applies them to two stages, the partial product accumulation stage and the final addition stage [23]. The workflow of the HPAM can be seen in Figure 2.18.

As seen in the figure, the HPAM truncates the LSB partial products to speed up the accumulation stage and uses a combination of exact and approximate adders, namely a segmented version of the RCA, to compute the final sum. Following the directive that less significant sections of the circuit should be approximated first, this approximate adder is only applied in the LSBs to maintain reasonable accuracy. The partial products are first rearranged in the HPAM to make them look like an upper triangular matrix in the figure. This is done to save the number of resources used in the compression stage.

In the final multiplication stage, the segmented RCA attempts to speed up the calculation process by cutting the carry chain and calculating the result in blocks simultaneously, leading to a lower accuracy result.

In order to test the performance and measure the accuracy of the 8-bit HPAM, Pandey et al. [23] synthesised the design and tested 106 random cases to obtain standard error metrics, namely the MRED. The tests performed showed that, on average, the HPAM is 48.06% more accurate

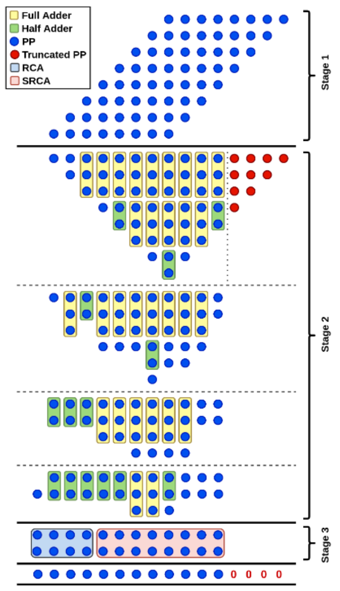


Figure 2.18: High-Performance Approximate Multiplier Computational Workflow from [23]

than other existing designs and has a 21.42% lower power-delay product. This means the HPAM can produce high-accuracy results while improving power consumption.

2.5.5 High-Performance Approximate Recursive Multipliers

The High-Performance Approximate Recursive Multipliers (HPARM) [24] are multipliers that achieve higher bit width implementations based on lower bit width ones. The most straightforward implementation is a 4-bit version. This multiplier performs approximations by altering the structure of the partial product tree based on bit probabilities and using approximate half adders and full adders for the compression and final addition stages.

One of the first simplifications is to use propagate-and-generate terms to reduce the probability of the occurrence of a 1 in specific locations in the partial product tree. The propagate (e) and generate (f) terms can be obtained based on equations 2.32 and 2.33, where c and d are the operands and k and l are bit positions.

$$e_{k,l} = c_k.d_l + c_l.d_k \tag{2.32}$$

$$f_{k,l} = (c_k.d_l).(c_l.d_k) \tag{2.33}$$

The f term has a probability of $\frac{1}{16}$ (assuming a uniform distribution of inputs) of being 1, while the e term has $\frac{7}{16}$. A probabilistic analysis performed on the resulting partial product tree demonstrates that columns 2, 3 and 4, represented by these new terms, perform better. Further analysis reveals that the probability of 2 f terms being one is negligible. Therefore, for column 3, one bit can be reduced by performing an OR operation. For columns 2 and 4, the generate element can be approximated to 0s with no significant impact on the accuracy. A simple comparison between the original PP array and the final one proposed can be seen in figure 2.19.

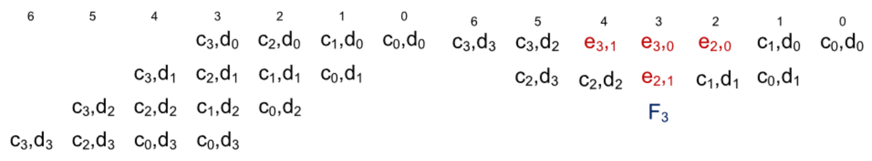


Figure 2.19: Partial Product Tree proposed in [24]

Approximate adders are used in this partial product tree in order to obtain the final result. The half-adder is used in the maximum number of scenarios possible to decrease the resources used.

NOR-Based Approximate Half Adder

Waris et al. [24] present two approximate half-adder designs of variable accuracy, denoted as NxHA1 and NxHA2. These use NOR gates instead of XOR gates to reduce delay and area. For both designs, the differences between the exact and approximate outputs can be seen in table 2.8. The equations that describe these half-adders are equations 2.34 and 2.35.

Table 2.8: NOR Based Approximate Half Adder Truth Table based on [24]

Input		Exact		NxHA1		NxHA2	
A	B	C	S	C	S	C	S
0	0	0	0	0	0	0	0
0	1	0	1	0	1	0	1
1	0	0	1	1	0	0	0
1	1	1	0	1	0	1	0

$$\begin{cases} Sum_{NxHA1} = \overline{a+\bar{b}} \\ CarryNxHA1 = a \end{cases} \tag{2.34}$$

$$\begin{cases} Sum_{NxHA2} = \overline{a+\bar{b}} \\ CarryNxHA2 = a.b \end{cases} \tag{2.35}$$

NOR-Based Approximate Full Adder

A full adder based on NOR gates is also proposed by Waris et al. [24]. It uses three NOR gates and one NOT gate. As the carry affects the next bit (making it a more significant error), effort was focused on getting the maximum number of correct cases for this output. The truth table for this block is shown in table 2.9, and its behaviour can be summarised in equation 2.36.

Table 2.9: NOR Based Approximate Full Adder Truth Table based on [24]

Input			Exact		NxFA	
A	B	Cin	Cout	S	Cout	S
0	0	0	0	0	0	0
0	0	1	0	1	0	0
0	1	0	0	1	0	1
0	1	1	1	0	1	0
1	0	0	0	1	0	1
1	0	1	1	0	1	0
1	1	0	1	0	0	1
1	1	1	1	1	1	0

$$\begin{cases} Sum_{NxFA} = \overline{\overline{a+b}+c} \\ CarryNxFA = \overline{\overline{a+b}+\bar{c}} \end{cases} \tag{2.36}$$

4X4 Block Multiplier

Two variants, the more approximated (MxA) and less approximated (LxA) 4×4 block multipliers, are implemented according to the scheme in figure 2.20. The main difference between the designs is that the MxA uses the first half adder design while the LxA uses the second one. They also differ in the way the S3 bit is generated. Due to this, the LxA achieves better accuracy than MxA while consuming more power.

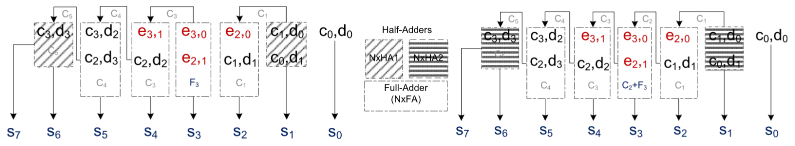


Figure 2.20: 4x4 Multiplier blocks proposed in [24]

Higher Bit Width Multipliers

In order to implement bigger multipliers, a recursive partitioning approach is used, attempting to achieve a faster implementation. With it, four 4×4 modules are used to implement an 8×8 multiplier. This allows the smaller modules to obtain intermediate results simultaneously, speeding up the calculation of the final result.

Due to this modular nature, different combinations of the MxA, LxA and exact blocks can be used depending on the desired results.

2.5.6 Rounding Based Approximate Multiplier

The main idea behind the Rounding Based Approximate Multiplier (RoBA) is to use the ease of operation when the inputs are powers of 2. The multiplication of A and B is given by Equation 2.37, where A_r and B_r are the rounded inputs.

$$A \times B = (A_r - A) \times (B_r - B) + A_r \times B + B_r \times A - A_r \times B_r \tag{2.37}$$

As the rounded inputs are powers of 2, implementing the last three terms can be performed using shift operations, saving resources. The first term, however, is more complex to implement. Since the impact of this term in the final result is typically tiny, [25] proposes to omit it to simplify the process, making the operation require only shifts and additions/subtractions.

For this implementation to work, it is first required to determine the nearest rounded values of A and B to a power of 2. If both possibilities lead to similar approximations, the larger power should be chosen, as it leads to a smaller implementation rounding circuit [25]. The approach developed in RoBA only works for positive inputs, as negative numbers represented in 2's complement cannot be rounded to a power of two like positive numbers. In order to fix this, [25] proposes to use a signal detector to determine the sign of the result and perform the operation with the absolute value of both inputs. The block diagram for this circuit can be seen in Figure 2.21.

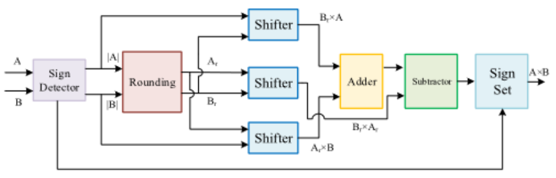


Figure 2.21: Rounding Based Approximate Multiplier Block Diagram from [25]

The inaccuracies of this multiplier come from the omission of one of the terms from the accurate multiplication in Equation 2.37. As such, the error can be determined according to Equation 2.38. The maximum value this error can take is 11% [25].

$$error(A,B) = \frac{(A_r - A)(B_r - B)}{AB} \tag{2.38}$$

2.5.7 EvoApproxLib Multipliers

Similarly to the adders described in subsection 2.4.12, Cartesian Genetic Programming can also be used to implement approximate multipliers. The process is similar, with changes to initial conditions and the number of gates used, as multipliers are more complex circuits than adders. Since the fitness function uses the exact computation to determine the MED, there is no need to alter it further.

The public GitHub repository mentioned in 2.4.12 also contains multipliers with different bit widths. For neural networks, the required multipliers are $N \times N$, that is, with both inputs having the same bit width, but there are also $N \times M$ multipliers available [14].

2.6 Fused Multiply Add

FMA units are digital circuits used to perform multiplication and addition in a single operation. They are an essential part of many applications, including digital signal processing. FMA units are implemented using several architectures for the multiplier and the adder sections, the most common of which is the Carry Save Adder based FMA.

FMA, or Multiply and Accumulation (MAC), units have several advantages over separate multiplication and addition units. They can save on area and power consumption, as they require fewer components than separate units. They also have the potential to reduce latency, as multiplication and addition can be performed in parallel. However, FMA units also have some trade-offs. They may be more complex to design and have a higher error rate than separate units.

The Carry Save Adder based FMA architecture consists of a series of full adders and carry save adders used to perform multiplication and addition. It is a typical architecture due to its simplicity and low cost. However, due to the limited precision of the carry-save adders, it may have a higher error rate.

The following subsections discuss three different approximate FMA approaches, their benefits and trade-offs in accuracy, resources and power consumption.

2.6.1 Approximate Rounding and Truncation based MAC

The Approximate Rounding and Truncation based MAC (ART-MAC) proposed in [26] uses input preprocessing to identify the truncation points and rounding that lead to the maximum efficiency configuration. It also uses a clever approach to rounding so that the loss of accuracy caused by truncation is as mitigated as possible. This is done by modifying the LSB of the truncated inputs.

The n-bit ART-MAC consists of three components (Figure 2.22), an n-bit approximate multiplier, a (2n + G)-bit accurate adder, and a (2n + G)-bits accumulation register, where G stands for the number of guard bits. The guard bits exist to protect from overflow [26].

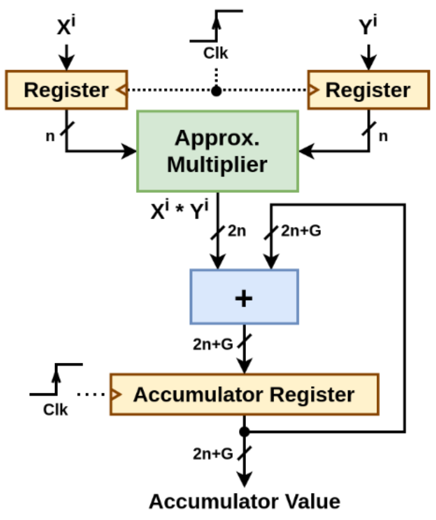


Figure 2.22: ART-MAC Block Diagram from [26]

The computational workflow of the approximate multiplier can be seen in Figure 2.23. In it, the inputs are divided into two parts, X_H and X_L, where X_H are the p MSBs, and X_L are the remaining $n-p$ bits. After this division, X_L is subdivided into two equal size parts, X_{LL} and X_{LH}, to simplify implementing truncation and rounding. X_{LL} are truncated while X_{LH} is rounded. This rounding is done by using an OR gate with the MSB of X_{LL} and the LSB of X_{LH}. Once the rounding and truncation of the input operands are done, their size reduces by $\lfloor (n-p)/2 \rfloor$ bits, simplifying the circuits.

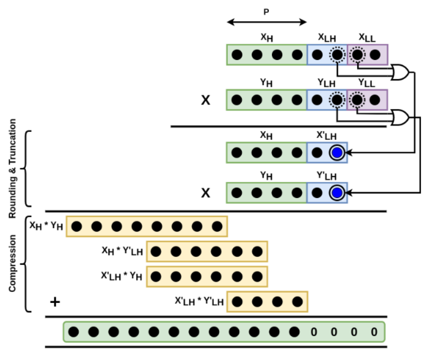


Figure 2.23: Rounding Based Approximate Multiplier Block Diagram from [26]

After generating the product through the approximate multiplier, the (2n + G)-bits value stored in the accumulator register is added to the obtained product, and finally, the accumulator register is updated.

The ART-MAC designs were tested using 106 iterations of 10 random inputs in order to obtain the error rate, MED, MRED. The tests performed by [26] showed that the ART-MAC consumes 35.35% less power and provides a significant speed-up when compared to a conventional MAC unit. On average, the proposed design has 7.44% lesser on-chip area and 13.4% lesser power-delay-product (PDP) when compared to other existing state-of-the-art designs.

2.6.2 Approximate MAC Unit

The Approximate MAC Unit (AMACU) uses an approximate FA to accumulate the multiplier partial products [27]. This approximate FA adder calculates the Approximate Sum (AS) and the Approximate Carry (AC) according to Equations 2.39 and 2.40.

$$AS = A+B+C \tag{2.39}$$

$$AC = \text{any of the inputs} \tag{2.40}$$

Carefully choosing the input bit to be the carry can lead to obtaining the correct result $\frac{6}{8} = 0.75 = 75\%$ of the times, and the AS is correct $\frac{5}{8} = 0.625 = 62.5\%$ of the times.

Instead of combining a multiplier and an adder in order to implement a MAC unit, the AMACU takes advantage of the partial product reduction mechanism present in tree multipliers and incorporates the approximate FA for accumulation. In the process, LSBs are truncated, and exact adders are used to calculate the MSBs, reducing the magnitude of the errors and ensuring accurate results.

The unsigned 8-bit AMACU was tested using 20000 sets of randomly generated inputs to measure error metrics and implemented using Verilog HDL to obtain accurate performance analysis. The test performed by [27] showed that the AMACU has a low delay and minimal power consumption when compared to other equivalent designs, having a lower power-delay product and area-power product.

2.7 Benchmarking

As stated in section 1.1, one of the main challenges of approximate computing is the lack of standardised error testing cases to evaluate and compare different designs. It is a complex procedure to develop standardised loads and benchmarks to run tests of different approximate computing techniques on its various applications and use cases. The AxBench [28] is one of many efforts in creating these benchmarks. It includes different testing scripts and circuits on some of the most common loads where approximate computing can benefit used resources or power consumption. This test suite can benchmark CPU, GPU and hardware implementations. For hardware, the included tests are:

- Brent-Kung 32-bit adder, a parallel form of a CLA;
- Kogge-Stone 32-bit adder, another parallel form of a CLA;
- Wallace Tree multiplier
- Inversek2j, computing the inverse kinematics of a 2-joint robot arm;
- Forwardk2j, performing the direct kinematics of a 2-joint robot arm;
- FIR filter;
- Sobel edge filter;
- K-means clustering algorithm;
- Neural Network that approximates the Sobel filter.

Each of these benchmarks requires different inputs and produces different results. The kinematics tests need integer numbers, RGB images, or even fixed point non-integer numbers. As such, AxBench provides three different-sized datasets for each application. For image applications, the small input dataset consists of ten different 512×512 pixel images, the medium-sized dataset ten different 1024×1024 pixel images and the large-sized dataset ten different 2048×2048 pixel images. In the other applications, the small, medium, and large-sized input datasets include 2^{12}, 2^{18}, and 2^{24} data points [28].

Not all of these benchmarks will be used to test the modules developed during this dissertation, as they are either adders and multipliers themselves, do not use these arithmetic units or require different types of inputs, namely floating point integers. As such, the Sobel edge filter, and the K-means clustering algorithm were selected to be used. The neural network that approximates the Sobel filter would also be selected, as it is the closest scenario to what is the purpose of what is being developed. However, the hardware implementation did not function properly due to the lack of weights and problems with module interconnections.

For evaluating the results obtained by these benchmarks, three different metrics are used depending on the application: average relative error, miss rate, and image difference. However, the hardware applications only use average relative error and image difference.

Chapter 3

Arithmetic Unit Design

This Chapter will present the different implemented arithmetic units and simulation results for the stand-alone modules developed to understand the proposed solutions used in the final design and implementation. This implementation is also presented schematically in its section, where different considerations are considered, and the reasoning for the choices made is exposed. Firstly, the various adder circuits are presented, followed by the multiplier circuits. Finally, a simple implementation of the vector product is exposed as a simple example of operation.

3.1 Adder Circuits

This section presents the implemented adder circuits during this dissertation and the results from simulation and testing the modules on single additions of two numbers. Some designs have variable bit width through a parameter, while others have a fixed width of 8 or 16 bits.

The implemented designs were chosen based on a combination of different criteria: low error rate, promising performance results in the article they were described, and simplicity of implementation, among a few others. While rare, designs with available hardware descriptions in Verilog were preferred to maximise the number of circuits implemented and tested.

3.1.1 Carry Based Approximate Adders

The first approximate adder design implemented was the CBAA [6]. As mentioned in section 2.4.4, this adder changes the typical FA structure to save resources and improve performance. Type I and Type IV designs mentioned in this section were implemented using Verilog. After developing the modules, the 1-bit adders were tested using a simple testbench to verify that both designs were correctly implemented. The results of this simple verification step matched the values derived from equations 2.13 to 2.16, displaying that the Verilog code did not have any errors.

After this verification, simple 8-bit adder modules were created by chaining the 1-bit adders. In this module, the carry output from each adder is the carry input of the next one, precisely like the RCA structure. To test this module, a testbench that tested the adder for all possible input combinations (2^{16}) was created, and the circuit was simulated using Icarus Verilog. Very early on in the simulation, it was noticed that the Mean Relative Error (MRE) or MRED of both types was very large, surpassing 100%. This is because, as mentioned in section 2.4.4 during the analysis of the truth tables, these adders are often in a position where the inputs are all 0 or all 1, which are the cases where the error exists. One case where this happens can be seen in figure 3.1, where adding 4-bit representations of 0 and 1 gives the output of 1111, which is 15 considering unsigned numbers. In the case of signed addition, this would be -1, which is also relatively far from the correct result.

$$\begin{array}{ccccc} & 0 & 0 & 0 & 0 \\ + & 0 & 0 & 0 & 1 \\ \hline & 1 & 1 & 1 & 1 \end{array}$$

Figure 3.1: Error in the CBAA Type IV adder for unsigned numbers

However, unlike what is mentioned in section 2.4.4, these significant errors are not exclusive to unsigned numbers, appearing in the case of signed addition. Figure 3.2 shows an example of this, where the addition of 1111 (-1) and 1001 (-7) gives the result of 0000, assuming that the result is truncated to 4 bits. Even considering the carry output, the result obtained is -16, double the correct result, -8.

$$\begin{array}{ccccc} & 1 & 1 & 1 & 1 \\ + & 1 & 0 & 0 & 1 \\ \hline 1 & 0 & 0 & 0 & 0 \end{array}$$

Figure 3.2: Error in the CBAA Type IV adder for signed numbers

Given these results on a multi-bit adder, it was decided that this design would not be used in the final implementation, as the error is too large to be acceptable, especially when using unsigned numbers. A possible solution to limit the scale of the errors is to only use the approximate adder on some LSB of the inputs and use the exact adder on the remaining MSB. This would limit the error to the LSB of the inputs, which would be acceptable for some applications. However, this solution was not implemented, as the CBAA was not used in the final implementation.

3.1.2 Approximate Full Adder

The second implemented adder was the Approximate Full Adder (AFA) [29]. This design uses OR, AND gates and an inverter to approximate the FA structure. Two types are proposed in [29] and cascaded in alternate order to form a higher bit adder. These implementations can be seen in figure 3.3.

Similarly to the CBAA, firstly, a 1-bit design was created in Verilog and tested to check all possible result combinations. The results of this test can be seen in table 3.1.

The results of the 1-bit implementation immediately demonstrated that this adder was not very accurate, with both types getting the wrong result in 2 out of the eight possible combinations.

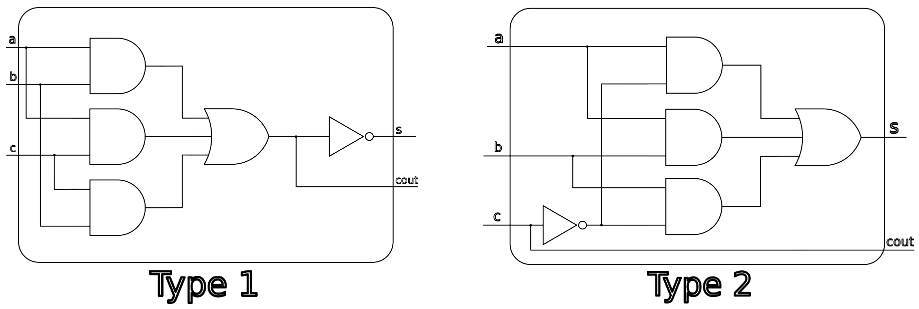


Figure 3.3: AFA 1-bit adder logic circuits

Table 3.1: AFA 1-bit adder Truth Table

Inputs			Exact		Type 1		Type 2	
a	b	c	s	cout	s	cout	s	cout
0	0	0	0	0	1	0	0	0
0	0	1	1	0	1	0	0	1
0	1	0	1	0	1	0	1	0
0	1	1	0	1	0	1	0	1
1	0	0	1	0	1	0	1	0
1	0	1	0	1	0	1	0	1
1	1	0	0	1	0	1	1	0
1	1	1	1	1	0	1	1	1

To create the higher-bit adders, the two types were cascaded by alternating the two designs, starting with Type 2, in order to try to decrease the error propagation. This architecture was implemented in Verilog and tested using a testbench that tested all possible input combinations using Icarus Verilog. Once again, the results were not as expected, with the MRE being higher than 30%, which, while lower than the CBAA, is still too high to be acceptable. As such, this design was also not used in the final implementation.

3.1.3 Reverse Carry Propagate Adders

As the previous two designs showcased, using approximations of a FA to replace all the adders leads to high error rates and undesirable edge cases with large MRE. In light of this, the next approximation of the FA, the RCPFA [7], is only used in a portion of the operation. To be more precise, half the bits of the adder are calculated using the approximate FA and the other half using the exact approach. Two designs, one mentioned in section 2.4.5 and another one not mentioned, were implemented and tested. These are the RA4 (Figure 2.7) and the RA5 (Figure 3.4).

Once again, the first step was to ensure that the 1-bit adder implementations matched the expected results presented[7]. This was done using a testbench that tested all possible input combinations, and the results matched the expected ones. After this confirmation, 8,16,24,32,48 and 64-bit implementations were created and tested using all possible inputs for the 8-bit adders and

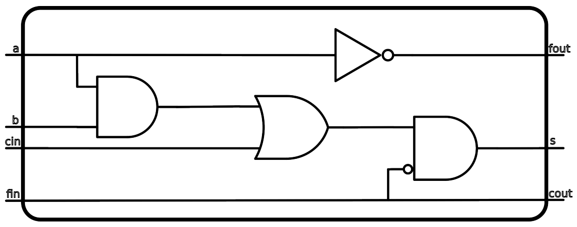


Figure 3.4: RA5 1-bit adder logic circuit

1 million random test cases for all the others. An example schematic of the 8-bit implementation can be seen in figure 3.5.

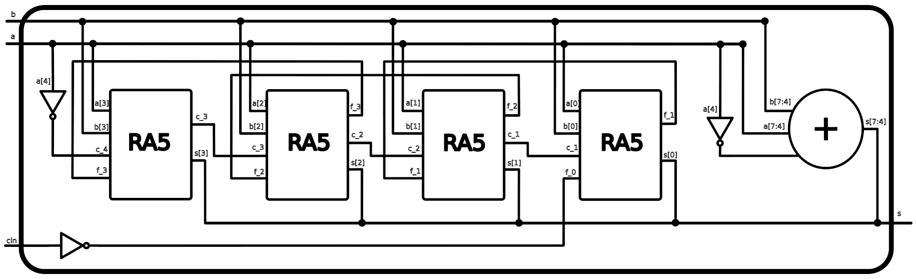


Figure 3.5: RA5 8-bit adder schematic

The MRED presented in the article by Singh et al. [7] for these implementations uses the approximate full adders in 21 bits for a 32-bit implementation. Since the design tested here uses this adder in only 16 bits, the MRED is expected to be lower than the one presented in the article. Table 3.2 confirms these expectations.

Table 3.2: Mean Relative Error for the 8, 16, 24, 32, 48 and 64-bit RA4 and RA5 adders

Adder	Mean Relative Error (MRE)					
	8	16	24	32	48	64
RA4	0.3763	0.0069	0.0004	2.774E-05	1.063E-07	3.251E-10
RA5	0.4160	0.0066	0.0004	2.567E-05	1.001E-07	7.215E-11

As the table shows, the designs yield highly accurate results, especially in higher bit widths, which is expected since the maximum possible relative error decreases. Given this favourable outcome, both designs were chosen to be further tested in the vector product implementation.

3.1.4 Bounded Error Approximate Adder

Unlike the previously implemented adders, where consecutive FAs were replaced by their approximate versions, the BEAD replaces only the MSB of an adder block. This block consists simply of a section of a larger size adder. This separation in smaller blocks of an adder circuit removes latency and carry dependence from the primary circuit, as the carry propagation from the LSB to the MSB is the critical path. However, removing the carry propagation from the MSB to the LSB would lead to a significant error.

To solve this and bind the error to a specific size, the BEAD uses a modified FA at the MSB of each block (Figure 2.13). This, together with some more logic, allows the error to be bounded so that it does not exceed a particular desired value.

As mentioned in section 2.4.11, the size of the sub-adder block, as well as the number of bits used for the estimation of a wrong carry, affect the accuracy of the adder. As such, various combinations of these parameters were tested, leading to the choice of a block size of 8 bits (4 bits if the total adder length is 8 bits) and a number of prediction bits of three.

Similarly to the other adders, the modified FA was implemented and tested for all input combinations first to ensure no errors were committed in the implementation in Verilog. After this, 8, 16, 24, 32, 48 and 64-bit implementations of this architecture were implemented and tested using 1 million random inputs. The results of these tests can be seen in table 3.3.

Table 3.3: Mean Relative Error for the 8, 16, 24, 32, 48 and 64-bit BEAD adders

Adder	Mean Relative Error (MRE)					
	8	16	24	32	48	64
BEAD	0.3991	0.0076	0.0077	0.0077	0.0076	0.0077

Comparing the results obtained with the ones presented in the article [13], it is possible to see that the MRED is higher across the board. This may be due to the number of bits used to determine whether the modified approximate result should be used. However, the error is still bounded and relatively low. As such, this design was chosen to be further tested in the vector product implementation.

One possible disadvantage of this design is that the maximum error can happen closer to the MSB, which can be a problem for specific applications where this error can accumulate and spread to the higher exact MSBs in fewer operations.

3.1.5 EvoApproxLib Adders

Section 2.4.12 mentions that the EvoApproxLib library consists of hardware and software models of approximate circuits designed to be easily used in arbitrary applications. As such, the library contains many different approximate adders and their hardware descriptions in Verilog. These adders are grouped based on different Pareto curves, relating different error metrics with the circuit's power consumption.

The most common error metric used in the reviewed literature is the MRE, and as such, the unsigned adders that are part of the Pareto curve for this metric were chosen to be implemented and tested. A further selection of the designs was performed, by removing designs that had 0.00% MRE, as they were assumed to give similar results to the exact implementations. Cases where the MRE was higher than 10% were also removed, as they were considered too inaccurate. It is important to note that the library only provides 8, 12 and 16-bit adder implementations, so any other bit width adder had to be implemented by either cascading the adders (with appropriate changes to the inputs) or by adding exact FAs to calculate the MSBs.

After these selection steps, six 8-bit adders and eight 16-bit adders were left. They were implemented in Verilog and tested using all possible input combinations in the case of the 8-bit circuits and 1 million random values for the 16-bit ones. The MRE values obtained are presented in tables 3.4 and 3.5, comparing them with the reported values given by the creators of the circuit library.

Table 3.4: Mean Relative Error for the 8-bit EvoApproxLib adders

Adder	3RE	6K6	6P8	6PT	6QU	108
MRE	0.0924	0.0094	0.0040	0.0014	0.0208	0.0428
Reported MRE	0.0924	0.0094	0.0040	0.0014	0.0208	0.0428

Table 3.5: Mean Relative Error for the 16-bit EvoApproxLib adders

Adder	0GX	0HE	0KG	0KU	0NT	0RH	0SD	110
MRE	0.0008	0.0059	0.0220	0.0975	2.161E-04	0.0027	0.1787	8.490E-05
Reported MRE	0.0002	0.0016	0.0055	0.0249	5.400E-05	0.0007	0.0452	1.700E-05

Since the circuits were chosen based on a Pareto curve with the MRE metric, the results should not differ from those presented in the library. In the case of the 8-bit adders, where all possible combinations were tested, the values match perfectly, which leads to believe that the differences in the 16-bit adders are due to the random test cases used. However, the values obtained are still close to the ones presented, even though they tend always to be higher. As such, all designs were chosen to be further tested in the vector product implementation, despite concerns with the high MRE of the 0KU and 0SD 16-bit adders. This choice was made because since the circuits are generated using genetic programming (CGP) it is difficult to predict how the circuit will be implemented in a FPGA. As such, the circuits may still perform well.

3.2 Multiplication Circuits

In this next section, the implemented multiplication modules are detailed and discussed, as well as the simulation results of simple multiplications to obtain simple error metrics, like the MRE.

The choice of the multiplier circuits was based on several criteria, namely the results presented in the literature, the availability of already implemented modules, and logic diagrams representing the adder tree.

Despite being the most common approach to approximating multiplication, not many approximate compressors were implemented. This is due to the difficulty of choosing only one circuit to implement. If multiple were to be implemented, further exploration of the possible combinations would lead to a complex and lengthy process that would prove to be too time-consuming to meet the objectives of this dissertation.

The implemented modules use 4, 8 and 16-bit inputs, resulting in 8,16 and 32-bit outputs. The use of recursive multiplication can obtain other larger sizes, something that was done, for example, on subsection 3.2.1.

3.2.1 Recursive Multipliers

The first multiplier circuit tested and implemented was the recursive multiplier in [24]. This multiplier uses a recursive approach to multiplication, where the inputs are divided into smaller parts and multiplied recursively. The results of these multiplications are then added together to obtain the final result.

The building block for this recursive multiplication is a 4×4 multiplier. After the partial product generation stage, some clever manipulations of the partial product tree (described in subsection 2.5.5) are performed for only one final addition stage to be required to obtain the output. To aid in this addition, approximate NOR-based adders are used. The truth tables for these implementations can be found in tables 2.8 and 2.9, present in subsection 2.5.5.

Similarly to the procedure performed in previous sections, these adders were implemented using Verilog and tested to ensure that their outputs matched the literature. Afterwards, the two proposed variants, the MxA (more approximated) and the LxA (less approximated) 4×4 multipliers (Figure 2.20), were implemented. Together with exact multipliers, these modules can then be used to implement larger multipliers recursively. As such, for an 8×8 multiplier, the literature proposes three different designs: the Ax1, Ax2 and Ax3.

The Ax1 multiplier uses three 4×4 exact multiplication blocks and the MxA block to calculate the LSBs. It is the most accurate implementation, as it uses primarily accurate blocks. The Ax2 multiplier uses two exact blocks, one MxA and one LxA block. Finally, the Ax3 multiplier uses one exact block to calculate the MSBs and then 2 LxAs and one MxA.

These three designs were implemented in Verilog and tested using all possible input combinations. The results of these tests can be seen in table 3.6.

Table 3.6: Mean Relative Error for the three 8-bit Ax Multipliers

Mult	Ax1	Ax2	Ax3
MRE	0.0048	0.0377	0.0708

As expected by their configuration, the MRE of the multipliers increases as the number of approximate blocks increases. The results vary slightly from the ones presented in the article [24], but not significantly, so they can be considered acceptable.

In order to implement higher order multipliers, these 8-bit implementations can be combined similarly to the 4×4 multipliers. As such, the Ax0233, Ax0333, Ax1233 and Ax3333 multipliers were implemented and tested. The numbers in the names represent the 8-bit multiplier used, with 0 being an exact implementation, from the MSB to the LSB. In order to obtain the MRE of these circuits, a testbench containing 1 million random inputs was created. The results of these tests can be seen in table 3.7.

Table 3.7: Mean Relative Error for 16-bit Ax Multipliers

Mult	Ax0233	Ax0333	Ax1233	Ax3333
MRE	0.0015	0.0017	0.0056	0.0697

Similarly to the adders, the MRE decreases when the bit width increases. As such, more approximate implementations can be used without significantly increasing error. The Ax1233, despite being composed of only approximate 8-bit blocks, obtains an accuracy similar to the Ax1 8-bit multiplier, composed of 3 exact blocks. The error of the Ax3333 is the highest, which is expected, as it only uses the most approximated 8-bit design. It should not be used as a 16-bit multiplier, but it can be helpful as a building block for larger multipliers.

3.2.2 High-Performance Approximate Multiplier

The second multiplier design implemented was the HPAM [23]. This design performs approximations in two main multiplier stages: partial product tree reduction and final addition. By truncating the LSBs in the partial products, the number of calculations required to obtain the final result is reduced. However, this truncation leads to a loss of accuracy, as the final bits of the result will always be 0. In the final addition stage, a segmented RCA is used to add the non-truncated LSBs.

This approximated adder was the first module to be implemented and tested in Verilog to ensure correct results. The multiplier module was implemented using FAs and Half Adder (HA) for the partial product reduction stage, according to the diagram presented in the article [23]. The module was then tested using all possible input combinations, obtaining a MRE of $1.438E-02$, which matches the results presented in the article.

Unfortunately, this implementation is only possible for 8-bit input numbers, and there are few uses in the vector product tests. Implementing larger multipliers using the recursive multiplier technique is possible, but this was not done due to time constraints.

3.2.3 EvoApproxLib Multipliers

Similarly to the adder case, the EvoApproxLib presented in section 2.5.7 also contains Verilog files with the hardware descriptions of different multipliers grouped in different Pareto Curves. Again, in the same light as the adders, the selection for the multipliers was circuits belonging to the MRE vs Power curve, in which this metric was lower than 10%. For cases where both inputs have the same number of bits ($N \times N$), the library provides eight and 16-bit unsigned adder implementations. After the pre-selection performed above, four 8-bit and eight 16-bit candidate circuits remained.

Like the adders from this library, the Verilog implementations for these circuits were tested by multiplying two input numbers. All possible inputs were tested for the 8-bit ones, while for the 16-bit ones, 100 thousand random inputs were used. This reduction in the number of test cases was made to speed up the process, as the multipliers are composed of numerous 2-input logic gates and performing 1 million tests with all of them would take several days. The results of these tests can be seen in tables 3.8 and 3.9.

As it is possible to observe in the tables, the results obtained are very close to the ones presented in the library. In general, except for the 2NDH multiplier, the MRE of the 8-bit configurations is lower than 1%, which is an excellent result. Three of the 16-bit designs also present an

Table 3.8: Mean Relative Error for the 8-bit EvoApproxLib multipliers

Mult	2NDH	12YX	XFM	ZDF
MRE	0.0499	0.0002	0.0081	0.0013
Reported MRE	0.0503	0.0002	0.0082	0.0013

Table 3.9: Mean Relative Error for the 16-bit EvoApproxLib multipliers

Mult	1UG	2KD	3CF	3H1	3PM	5A2	7QP	HDT
MRE	0.0133	5.883E-05	5.314E-08	3.007E-07	4.575E-06	0.0016	2.319E-05	0.0915
Reported MRE	0.0134	6.700E-05	4.400E-08	3.800E-07	3.400E-06	0.0017	2.400E-05	0.0915

extremely low MRE, which might mean there are few possible gains in other metrics, like power consumption, for these designs.

3.3 Vector Product

This section describes the Vector Product circuit implementation in greater detail. In order to test the circuit designs on a more straightforward problem that would still be significant for the problem to solve, an implementation of a circuit that performs the internal product of two vectors was made.

This internal product calculation performs multiplications and additions similarly to the calculations performed in a Neural Network node, where the node inputs can be seen as one of the vectors and the different weights can be seen as the other vector. As such, it was considered an excellent simple implementation that would allow us to test the performance of the arithmetic unit modules before moving to tests using a Neural Network implemented in hardware. These similarities are also a reason to believe that the gains obtained by using approximate circuits in this implementation will be similar to the ones obtained in a Neural Network.

The vector product module comprises three main blocks: a multiplier, an adder and an accumulator. The multiplier receives two inputs from each vector and outputs the product of these two values with their combined bit width. This value is then added to the accumulator, which stores the sum of all the products. This process is repeated every clock cycle until all values in the vectors have been multiplied and added to the accumulator. The final value in the accumulator is then the result of the vector product. Considering constraints with reading and writing to memory and data transfer limitations, multiple multipliers can be used in parallel to speed up the process, requiring an adder tree to combine the results outputted by the multipliers. This adder tree is then connected to the accumulator, storing the final result. The maximum number of multipliers that can be used in parallel is limited by the maximum number of bits that can be transferred in a single clock cycle and the number of inputs that can be read from memory in a single clock cycle.

Due to these limitations, the implemented modules use different parameters to create the appropriate circuit. The parameters are the input bit width, the number of multipliers used in parallel and the size of the input vectors. The first one can be 8, 16, 24 or 32, and it determines the size of the modules used and limits the number of multipliers used in parallel. This number can be 1, 2 or 4. The size of the input vectors determines how long the circuit will run and can be any power of 2 or multiple of the number of multipliers. Various values were tested, but the one that ended up being used was 32. This value was chosen to consider the possible range of inputs used in order not to cause overflow in the adders or accumulators. A simplified circuit version can be seen in figure 3.6, using two multipliers in parallel.

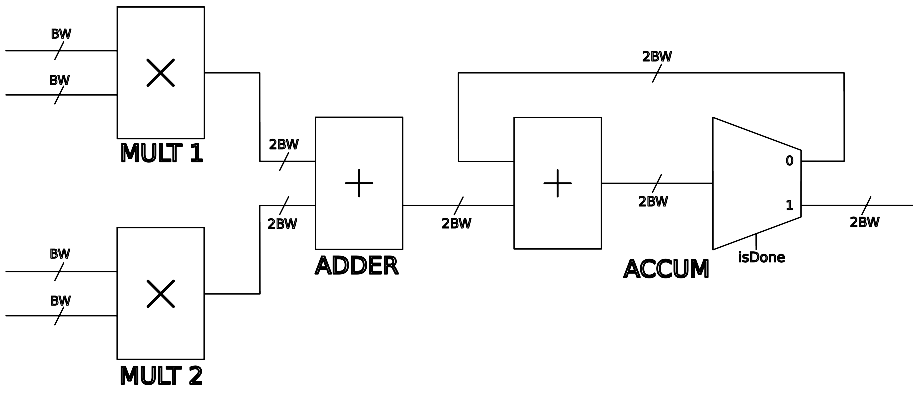


Figure 3.6: Vector Product circuit

The vector product module that implements this operation instantiates the multiplier, adder and accumulator modules and the necessary interconnections between them. In order to control the operation of the circuit, a simple state machine is constructed that provides the correct inputs to the multipliers and determines the start and end of the operation using the appropriate control signals.

Chapter 4

Evaluation and Results

In this chapter, the main results are presented and discussed. The modules described in the previous chapter were used to build more complex modules that perform various operations, namely the internal product of two vectors calculation mentioned in section 3.3, the sum of two images, the application of a Sobel edge filter to a picture and a simple Neural Network implementation to predict handwritten digits. The modules were tested using different adders and multipliers and were synthesised using Vivado 2022.2.

4.1 Simulation

Before moving into module synthesis for the FPGA, the behaviour of the Verilog modules for different implementations was tested using Icarus Verilog to simulate. This was done to quickly obtain result accuracy predictions for many inputs and pre-emptively remove adders or multipliers with poor performance (more than 15% MRE) from the synthesis process. The tests were performed for 50 thousand random inputs. The range of these inputs differs in each case, depending on the function being tested and will be specified in the respective section of the use case. The results obtained from the simulation are presented in the tables of the respective sections in this chapter, as well as in Appendix A, under the column "Relative Error".

4.2 Implementation Target and Settings

As mentioned in the introduction for this chapter, Vivado 2022.2 was the software chosen to perform the synthesis and implementation of the modules. As such, a project was created targetting the Zedboard (Zynq-7000) FPGA. This choice was made based on the available board selection in Vivado, as it is a relatively small board in terms of resources, where savings in used resources may mean the possibility of implementing more complex modules that were previously unable to fit. It was also chosen because it is a board available at FEUP, and previous experience with the board existed from other curricular units.

Several settings were changed in Vivado for the Simulation and Synthesis processes, while the Implementation process was left with the default settings. For Simulation, all signals were logged in order to make it simpler to detect errors, and the Switching Activity Interchange Format (SAIF) file was enabled in order to be able to obtain more accurate power consumption estimations. This file also logged all signals used in the implemented modules.

In the case of Synthesis settings, the goal was to disable as many optimisations as possible because it was assumed that these would skew the results in favour of the exact implementations of the arithmetic units. As such, Digital Signal Processing (DSP) blocks were disabled, as multipliers implemented using them would perform immensely better. The resource-sharing option was turned off, and the keep_equvalent_registers was turned on to minimise hardware simplification that would compress or remove modules. The first solution to prevent these simplifications was to use an Out-of-Context (OOC) synthesis flow. This flow would guarantee that none of the adder and multiplier modules would be simplified, as they would be synthesised separately from the rest of the design. However, this flow was not used because Vivado uses default synthesis settings for OOC synthesis, which are not the same as the ones used in the main synthesis flow and would cause 16-bit or higher multipliers to be implemented in the DSP blocks. As such, the OOC flow was not used. In the tests for the multipliers, DSP blocks were re-enabled temporarily to test if the assumption that they would benefit the exact implementations was correct. The results of these tests are presented in subsection 4.3.2.

4.3 Synthesis and Implementation

This section presents the Synthesis and Implementation results for the vector product module. These include accuracy, power consumption, clock frequency and used resources (represented through Look-Up Table (LUT) and Flip Flop (FF) counts). In order to determine the power consumption, a Post-Implementation Timing Simulation of the module was performed, where 100 random pairs of vectors are generated, and the module performs the internal vector product calculation for each pair in a row. Due to the switching activity being saved in the SAIF file, the power report presented after this simulation, while not 100% accurate, will be a good approximation of the module's power consumption. The clock frequency was obtained by combining the timing report after implementation with several implementations with stricter clock constraints.

Firstly, implementations with only approximate adders will be discussed, followed by ones with approximate multipliers. Lastly, the designs considered to have the best results will be used to perform tests where both circuits are approximated.

4.3.1 Vector Product with Approximate Adders

As mentioned previously, adders were the first approximated circuits used. Using a module parameter, the exact adder was replaced with inexact designs. Using a combination of bash, Python and tcl scripts, the designs were synthesised and simulated to obtain the maximum clock frequency, the used resources and the power consumption. This process was repeated multiple times for different input bit widths (8, 16, 24 and 32 bits) and different numbers of parallel multipliers (1, 2 and 4), to see the impacts of the input size and parallelisation. Not all of these combinations were used, considering limitations on data transfers on a single clock cycle and the maximum number of inputs/outputs possible of the FPGA, which are 200. As such, the 24 and 32-bit input implementations were not run using four parallel multipliers, as they would exceed these limitations.

To simplify the result presentation, only a small portion of the result tables are presented here in this subsection. The tables for the remaining cases can be seen in Appendix A.1.

Table 4.1: Vector Product Results for 16-bit adders with one multiplier

Adder	Clock (ns)	Relative Error	Total Power (W)	Static (W)	Dynamic (W)	LUT	FF	Dynamic*(LUT+3FF)
Exact	10	0.0000	0.109	0.104	0.005	122	83	1.855
RA4	10	0.0244	0.109	0.104	0.005	121	83	1.850
RA5	9	0.0245	0.108	0.104	0.004	113	67	1.256
BEAD	11	0.1047	0.109	0.104	0.005	126	83	1.875
0GX	8	0.0016	0.111	0.104	0.007	135	79	2.604
0HE	8	0.0107	0.109	0.104	0.005	125	76	1.765
0KG	8	0.0501	0.108	0.104	0.004	120	75	1.380
0KU	-	0.4098	-	-	-	-	-	-
0NT	8	0.0026	0.111	0.104	0.007	136	81	2.653
0RH	8	0.0057	0.109	0.104	0.005	125	78	1.795
0SD	-	0.9191	-	-	-	-	-	-
110	9	0.0003	0.111	0.104	0.007	134	82	2.660

Table 4.1 presents the results for the simplest scenario, using 8-bit inputs to the vector product module and using only one multiplier. As the adder is used after multiplication, the adder bit width is double the inputs, so 16-bit adders are used. Two key aspects immediately are noticed glancing at the table; the first is that 0KU and 0SD adders have no implementation results. This is because after running the simulation using Icarus Verilog mentioned in section 4.1, the relative error of the internal product calculation was too high, being 40% for the 0KU and a staggering 91% for the 0SD. With errors this high, the result degradation caused by using these modules would outweigh any possible gain in the other parameters. As such, they were not synthesised. The second key aspect is that Static Power consumption represents a significant portion of the total power consumption. This type of consumption is inherent to the FPGA, as to be reconfigurable, the interconnections are done with the help of extra transistors that have leakage currents that accumulate. This problem is especially significant in smaller designs, like the 8-bit input 1 multiplier vector product module, where the static power consumption represents upwards of 90% of total power consumption.

Another slight problem is the sensibility of the Vivado power reports. While the report can detect smaller consumptions, the minimum presented power is 0.001 W. This is not much of a problem in more extensive designs, as power consumptions rise closer to the Watts. However, for minor cases like the one presented in table 4.1, this representation accuracy can lead to all approximate adders having virtually the same power consumption and the possibility of minor inaccuracies existing due to rounding.

With respect to the relative error, the majority of the nine designs implemented have errors between 1 and 5%. The BEAD is the worst-performing implemented adder in this metric, achieving close to 10.5% error. On the positive side, the 110 adder achieved the results closest to the exact values, with 0.026% relative error.

As for power consumption, as mentioned earlier, all designs have small dynamic power consumption, and rounding errors in the main contributors to these values (clock, logic, I/O and signals) can explain the differences between most of them. However, the 0GX, the 0NT and the 110 adders performed the worst in this metric, with 40% more consumption than the exact adder.

As for resources utilised, most adders, except for the RA4, RA5 and 0KG adders, used more LUTs than the exact implementation. This can be explained by the use of specialised structures for carry propagation, which are better utilised by the exact implementation. All adders use the same or fewer FFs as the exact one, with special mention to the RA5 that uses the least, using 80% of the FFs.

Lastly, when it comes to clock period and frequency, the exact adder worked at 100 MHz. The only adder that performed worse than this was the BEAD, with a period of 11ns, corresponding to roughly 91 MHz. The RA4 obtained the same frequency as the exact implementation. The RA5 and 110 adders could operate with a period of 9ns (frequency of 111 MHz) while the remaining adders with 8ns (frequency of 125 MHz). The main limitation to going any faster is multiplication, not addition.

A simple metric to represent the power-area product was created to select the best possible designs for the implementation with both arithmetic units approximated. This metric is simply the dynamic power consumption multiplied by the sum of LUTs and three times the FFs. This factor of 3 was chosen based on the proportion of FFs used in the designs after all syntheses were performed to balance saving resources of both types. Following this metric, the best designs are RA5 and 0KG. Coupling this metric with the relative error obtained for each adder and the maximum clock frequency, the 0HE was also chosen. This choice might not seem to make much sense according to table 4.1, as the 0RH adder presents similar results to the 0HE in clock, power and resources while having $10\times$ lower relative error, but when the number of parallel multipliers is increased (as possible to see in tables A.1 and A.2), the 0HE adder becomes more advantageous as it uses slightly fewer resources and has lower power consumption than the 0RH. The increase in the number of parallel multipliers also affects the accuracy of the internal product calculation. As the number of parallel multipliers rises, the relative error increases. This is due to using approximate adders to implement a small adder tree after multiplication. In this adder tree, the errors accumulate differently as the number of multipliers increases. While the inputs never cause overflow using exact arithmetic due to precise input scaling, no overflow protection exists in the approximate modules that can overflow from accumulated errors.

4.3.2 Vector Product with Approximate Multipliers

After obtaining the results for approximate adders, the adder module was set to the exact adder to test the approximate multipliers. Similarly to the adder case, the multiplier was altered using a module parameter, and a combination of bash, Python and tcl scripts was used to obtain the results. Since the circuits obtained are more complex and higher bit width multipliers require recursive multipliers for most of the modules tested, only 8 and 16-bit inputs were used, meaning both configurations could be synthesised with 1, 2 or 4 parallel multiplications.

Much like the adder case, only one synthesis case is presented for the 8 and 16-bit multipliers to simplify the result presentation. All other cases can be consulted in Appendix A.2.

Table 4.2: Vector Product results for 8-bit multipliers

Mult	Clock (ns)	Relative Error	Total Power (W)	Static (W)	Dynamic (W)	LUT	FF	Dynamic*(LUT+3FF)
Exact	12	0.0000	0.121	0.105	0.016	378	129	12.24
Ax1	13	0.0043	0.118	0.105	0.013	373	129	9.88
Ax2	12	0.0507	0.118	0.105	0.013	360	129	9.711
Ax3	13	0.1029	0.115	0.105	0.01	340	129	7.27
HPAM	11	0.0575	0.121	0.105	0.016	397	129	12.544
2NDH	11	0.0820	0.109	0.104	0.005	256	129	3.215
12YX	12	0.0005	0.123	0.105	0.018	445	129	14.976
XFM	12	0.0056	0.116	0.105	0.011	380	129	8.437
ZDF	12	0.0013	0.121	0.105	0.016	428	129	13.04

Table 4.2 contains the results for the internal product of two vectors using 8-bit multipliers and performing four simultaneous multiplications. Once again, static power consumption represents a large majority of total power consumption, hovering around 80 to 85%.

The relative error of the modules is higher than the tests performed in Section 3.2 due to accumulation from repetitive calculations. Unlike the adders, the relative error stayed constant with the number of simultaneous multiplications, which aligns with the explanation given in the previous subsection for the increase when approximate adders are used. The 8-bit multiplier with the lowest relative error is the 12YX, achieving 0.052%. On the other hand, the Ax3 multiplier registered the most significant error, at 10%. This result was expected, especially after the simulations performed on simple multiplications.

All multipliers use the same number of FFs. This is because FF usage primarily depends on the adder implementation from the accumulation unit. LUT-wise, four of the eight multipliers use more resources than the exact implementation. They are the HPAM, the 12YX, the XFM and the ZDF multipliers. In the case of HPAM, this resource usage can be explained by the fact that the compression tree was manually built in the hardware description. As such, any possible optimisations the FPGA could have used for implementation of the FAs and HAs that compose it are discarded. In the other cases, the extra resources can be related to the limitations of the CGP algorithm used, where the number of 2-input gates is pre-defined. As such, after the FPGA optimises the exact implementation, fewer LUTs are required to describe the circuit's behaviour. The multiplier that uses the fewest resources is the 2NDH using 68% of the LUTs of the exact implementation. It also is one of the multipliers with the highest error rate.

The 2NDH has the lowest dynamic power consumption by far, having less than $\frac{1}{3}$ of the consumption. On the other end of the spectrum, the 12YX has the highest power consumption, even higher than the exact multiplier. This can be explained by the larger number of resources required to implement this multiplier.

Multiplying the dynamic consumption by the used resources similarly to what was performed for the adders, it is possible to see that the 2NDH has a significant lead in this metric. However, the high relative error raises whether this multiplier will lead to high result degradation. This is also the case for the second multiplier according to this metric, the Ax3. As such, combining these metrics, the XFM multiplier is the most likely to obtain good results in a neural network.

The minimum clock period for the exact multiplier is 12ns, corresponding to 83 MHz. Half of the implemented modules achieved this same result. The Ax1 and Ax3 multipliers could only go to 77 MHz. In the first one, the high amount of exact blocks paired with the additions required to complete the result caused this increase in delay, while in the second one, the use of two less approximate submodules might lead to inefficiencies in the optimisation process of the FPGA. Achieving a maximum clock frequency of 91 MHz, the HPAM and 2NDH perform better than the exact counterpart. It is important to note that using more multipliers in parallel leads to decreased clock frequency. This is due to two factors: the first one is the increase in the number of adders required to obtain the result, and the second one is the need for more interconnections and routing of signals to feed the inputs to the multipliers.

Table 4.3: Vector Product results for 16-bit multipliers

Mult	Clock (ns)	Relative Error	Total Power (W)	Static (W)	Dynamic (W)	LUT	FF	Dynamic*(LUT+3FF)
Exact	13	0.0000	0.12	0.105	0.015	351	131	11.16
Ax0233	14	0.0028	0.119	0.105	0.014	341	131	10.276
Ax0333	13	0.0030	0.122	0.105	0.017	341	131	12.478
Ax1233	15	0.0040	0.119	0.105	0.014	333	131	10.164
Ax3333	14	0.1026	0.119	0.105	0.014	329	131	10.108
1UG	12	0.0311	0.109	0.104	0.005	142	119	2.495
2KD	13	1.807E-05	0.129	0.105	0.024	385	131	18.672
3CF	13	7.099E-09	0.147	0.105	0.042	502	131	37.59
3H1	14	3.475E-08	0.144	0.105	0.039	490	131	34.437
3PM	14	4.099E-07	0.138	0.105	0.033	463	131	28.248
5A2	12	0.0013	0.115	0.105	0.01	270	129	6.57
7QP	14	8.674E-06	0.13	0.105	0.025	391	131	19.6
HDT	14	0.1226	0.107	0.104	0.003	93	111	1.278

Table 4.3 contains the results for the internal product calculation using 16-bit multipliers and only performing one multiplication at a time. As with every other case, static power consumption represents most of the total power consumption.

The relative error of most modules used for the 16-bit multiplication was low, below 1%. There are only three exceptions: the Ax3333, the 1UG and the HDT multipliers. The inaccuracy of the first design was already expected, as mentioned at the end of subsection 3.2.1. The relative error of the other two multipliers was also high in the tests performing a single multiplication between two numbers. Furthermore, the error accumulation natural to the calculations performed in this module increased the MRE. The 16-bit adders also contain the design with the lowest MRE, the 3CF.

In operation frequency, only two modules performed better than the exact approach, the 1UG and the 5A2, achieving frequencies up to 83 MHz compared to the 77 MHz achieved by the exact multiplier. All other modules performed similarly or worse than it, with the worst one being the

Ax1233 multiplier, capping at 67 MHz. These values can be explained by the implementation of the exact multiplier chosen by the FPGA being an extremely efficient design instead of a conventional tree multiplier. Another possible explanation is that the inexact multipliers do not take advantage of optimisations provided by using a FPGA or by the logical expressions not being well translated to the blocks used.

As for used resources, the most inexact EvoApproxLib multipliers stand out from the rest, requiring less LUTs and FFs to be implemented. In the case of the 1UG, less than half of the LUTs are used, while the HDT uses less than one-third. The 5A2 comes next regarding used resources, using 270 LUTs and 129 FFs. It is the third and last adder to use fewer FFs than the exact module. All others use the same amount, 131. The Ax recursive multipliers provide small saves in resource usage, ranging from 2.8% with the Ax0233 to 6.3% with the Ax3333. The price to pay for the highly accurate results presented by the other multipliers is, unfortunately, the need for more resources than the exact circuit.

Dynamic power consumption follows the same trend as resource utilisation, with 1UG, HDT and 5A2 having significantly lower consumptions than the exact scenario. The HDT multiplier has the lowest value, with only 3mW, one-fifth of the exact module. On the other end of the spectrum, the 3CF multiplier uses almost three times more dynamic power than the accurate one.

Similarly to the previous tests, a metric for the power-area product was calculated to determine possible practical circuits to implement in a scenario where both the adder and the multiplier modules were approximated. Despite the higher error rate, the HDT and 1UG performed fantastically in this metric and were selected for use. The 5A2 and the Ax1233 were also chosen as more accurate alternatives in the case the error accumulated by the other two modules caused the results to be unusable.

Synthesis Directives

As previously mentioned, almost all 16-bit multipliers used more resources than the exact implementation. Optimisation strategies and directives were used during the synthesis process to study possible extra optimisations that could lead to decreases in used resources and power consumption. Two directives were tested, AreaOptimized_High and PowerOptimized_High. The synthesis, implementation and simulation to obtain power results were rerun using these directives. Tables 4.4 and 4.5 present the results for the AreaOptimized_High directive. The results for the PowerOptimized_High directive were highly similar, being identical in most cases and differing in one LUT or one mW in power consumption at most, so the tables for these runs are available in Appendix A.3, as the conclusions obtained are the same.

Table 4.4 can only be adequately interpreted with the default directive run table, table 4.2. Comparing these two tables, it is possible to see that the maximum operation frequency of the circuit decreased as the period increased by one or two nanoseconds.

The resources used for all designs decreased, and only the 12YX multiplier used more resources than the exact implementation. The decrease happened exclusively to the LUTs, and the

Table 4.4: Vector Product results for 8-bit multipliers using AreaOptimized_High directive

Multiplier	Clock (ns)	Total Power (W)	Static (W)	Dynamic (W)	LUT	FF	Dynamic*(LUT+3FF)
Exact	13	0.116	0.105	0.011	360	129	8.217
Ax1	14	0.115	0.105	0.01	346	129	7.33
Ax2	12	0.113	0.104	0.009	330	129	6.453
Ax3	13	0.113	0.104	0.009	318	129	6.345
HPAM	13	0.114	0.105	0.009	346	121	6.381
2NDH	12	0.11	0.104	0.006	245	129	3.792
12YX	13	0.116	0.105	0.011	366	129	8.283
XFM	14	0.112	0.104	0.008	320	129	5.656
ZDF	14	0.116	0.105	0.011	360	129	8.217

variation is design-dependent, varying from 4.3% in the 2NDH to 17.8% in the 12YX module. Dynamic power consumption also decreased by about one-third for most designs. The only exception is the 2NDH multiplier, whose power consumption increased by 1mW. However, this design is still the most energy-efficient one.

Table 4.5: Vector Product results for 16-bit multipliers using AreaOptimized_High directive

Multiplier	Clock (ns)	Total Power (W)	Static (W)	Dynamic (W)	LUT	FF	Dynamic*(LUT+3FF)
Exact	18	0.166	0.105	0.061	1270	225	118.645
Ax0233	18	0.163	0.105	0.058	1206	225	109.098
Ax0333	18	0.165	0.105	0.06	1194	225	112.14
Ax1233	18	0.162	0.105	0.057	1169	225	105.108
Ax3333	18	0.162	0.105	0.057	1132	225	102.999
1UG	16	0.117	0.105	0.012	418	149	10.38
2KD	20	0.187	0.106	0.081	1165	223	148.554
3CF	20	0.223	0.106	0.117	1581	225	263.952
3H1	20	0.22	0.106	0.114	1536	225	252.054
3PM	20	0.197	0.106	0.091	1379	225	186.914
5A2	17	0.131	0.105	0.026	796	201	36.374
7QP	20	0.189	0.106	0.083	1275	223	161.352
HDT	11	0.108	0.104	0.004	225	105	2.16

Comparing with the default directive synthesis results presented in table A.13, it is possible to conclude that the dynamic power consumption in all designs decreased by between 25% and 35%, even in the HDT multiplier that already had a small dynamic power consumption of 6mW.

The maximum operation frequency varied very differently according to the design. In the case of 1UG, it maintained the same value in both synthesis scenarios; in the Exact and the HDT modules, the minimum operation period decreased by two and one nanoseconds with the AreaOptimized_High directive, respectively. All other multipliers performed worse, requiring a clock frequency as low as 50 MHz.

As for used resources, the synthesis with the directive decreases the usage of both LUTs and FFs, unlike the 8-bit multiplier case. The FF reductions occurred for the 1UG, 2KD, 5A2, 7QP and HDT designs and ranged from a less than 1% decrease for 2KD and 7QP to 27.6% in the HDT design. Regarding the LUTs, the synthesis was more advantageous for approximate designs, as the exact multiplier only used 2.4% less of this resource.

DSP Usage

In section 4.2, it is mentioned that in the synthesis settings, the use of DSP blocks was disabled as the multiplication being implemented in them was believed to be heavily beneficial for the exact design. To test this hypothesis, these blocks were enabled, and the process was rerun. Table 4.6 contains the results for this run, using 16-bit multipliers and four parallel operations.

Table 4.6: Vector Product Results using DSPs for 16-bit multipliers

Multiplier	Total Power (W)	Static (W)	Dynamic (W)	LUT	FF	DSP	Dynamic*(LUT+ +3FF+100DSP)
Exact	0.11	0.104	0.006	36	65	5	4.386
Ax1233	0.163	0.105	0.058	1220	225	0	109.91
1UG	0.117	0.105	0.012	445	149	0	10.704
2KD	0.186	0.106	0.08	1285	223	0	156.32
3CF	0.228	0.106	0.122	1647	225	0	283.284
3H1	0.224	0.106	0.118	1670	225	0	276.71
3PM	0.211	0.106	0.105	1645	225	0	243.6
5A2	0.134	0.105	0.029	911	201	0	43.906
7QP	0.195	0.106	0.089	1410	223	0	185.031
HDT	0.109	0.104	0.005	240	105	0	2.775

As expected, the use of DSP blocks only benefitted the exact multiplication, as the only circuit the FPGA recognises that can be implemented using these blocks. The accurate multipliers in the Ax multipliers smaller submodules are not implemented in the DSPs due to their size. The FPGA only implements 16×16 multipliers or larger in them, requiring an extra flag to implement 8×8 or 4×4 multipliers. This flag is required because the gains in speed by using these blocks are offset by the need to move data to these blocks and back to the circuit. It is considered that for these smaller multiplier sizes, the negatives outweigh the benefits.

The exact implementation using DSP blocks performs a lot better than its counterpart that does not. The comparison can be made between this table and table A.13. Using five blocks, the use of LUT and FF is cut dramatically, and power consumption is lowered. The number of LUTs decreases by 97% and the number of FFs by 71%. The dynamic power consumption also decreases from 82mW to just 6mW. The maximum clock frequency also more than doubles.

This leads to the conclusion that no approximate multiplier implementation can beat the efficiency of exact multipliers implemented on a FPGA with DSP blocks available.

4.3.3 Vector Product with Approximate Multipliers and Adders

After performing synthesis and implementation with approximate adder modules and approximate multiplier modules, the best implementations mentioned in the previous subsections were selected based on a combination of relative error, power-area product and maximum operation frequency/minimum operation period. After this choice, synthesis and implementation of the vector product module were run using both approximate arithmetic units. To do this, two parameters were used in the hardware description of the module, one for each circuit type. With the help of

Python, bash and tcl scripts, the results were obtained and are presented in tables 4.7 and 4.8 for 8 and 16-bit inputs, respectively. In an attempt to use the largest circuit possible, both runs use four parallel multipliers.

Table 4.7: Vector Product results for 8-bit inputs with four multipliers

Mult	Adder	Clock (ns)	Relative Error	Total Power (W)	Static (W)	Dynamic (W)	LUT	FF	Dynamic*(LUT+3FF)
Exact	Exact	12	0.0000	0.121	0.105	0.016	378	129	12.24
2NDH	0HE	11	0.0759	0.109	0.104	0.005	216	122	2.91
2NDH	0KG	12	0.1209	0.108	0.104	0.004	192	107	2.052
2NDH	RA5	11	0.2549	0.108	0.104	0.004	206	105	2.084
XFM	0HE	14	0.0385	0.115	0.105	0.01	304	122	6.7
XFM	0KG	13	0.0867	0.111	0.104	0.007	284	118	4.466
XFM	RA5	13	0.0841	0.11	0.104	0.006	300	113	3.834
HPAM	0HE	12	0.0527	0.116	0.105	0.011	360	122	7.986
HPAM	0KG	13	0.1022	0.11	0.104	0.006	314	118	4.008
HPAM	RA5	12	0.1917	0.11	0.104	0.006	360	113	4.194
Ax1	0HE	14	0.0116	0.118	0.105	0.013	332	122	9.074
Ax1	0KG	14	0.0468	0.111	0.104	0.007	323	118	4.739
Ax1	RA5	14	0.0987	0.111	0.104	0.007	330	113	4.683

For 8-bit inputs, the multiplier modules were the 2NDH, the XFM, the HPAM and the Ax1. The 16-bit adders chosen were the 0HE, the 0KG and the RA5. Icarus Verilog simulations were run for 50000 random vectors with 32 numbers each, between 0 and 32, to avoid overflow. The MRE of each combination of adder and multiplier were taken. All MRE were between 1% from the Ax1-0HE pair and 25% from the 2NDH-RA5 pair. The 0HE adder presented the lowest relative error when combined with different multipliers. The 2NDH multiplier had the highest relative error of the multipliers when combined with different approximate adders.

Regarding power consumption, once again and maintaining the theme, static power consumption represents most of the total power. All pairings obtain lower values than the exact implementation regarding the dynamic power consumption. This is to be expected, as all designs with higher consumption were excluded from this test. The lowest power consumption came from using the 2NDH multiplier, as all three pairs have the lowest consumption. Regarding adders, the 0HE adder uses more power than the other two designs, going from marginally using more when paired with the 2NDH multiplier to consuming almost double the power when paired with the HPAM or the Ax1 modules.

Seven of the 12 pairs achieve lower maximum operation frequency than the exact module. All pairs that include the Ax1 multiplier achieve the lowest maximum frequency, at roughly 71 MHz. This is because the Ax1 module uses a lot of exact multiplier submodules and 12-bit adders to obtain its result, which takes longer than other multipliers. The two pairs faster than the exact implementation are the 2NDH-0HE and the 2NDH-RA5. They achieve 91 MHz as their maximum operational frequency.

Finally, for resource utilisation, all designs achieve significant savings when compared to the exact implementation in both LUTs and FFs. Once again, the 2NDH multiplier is the multiplier that uses the least amount of resources, with its combo with the RA5 adder achieving the lowest amount with 46% less LUTs and 19% less FFs. On the other end of the spectrum, the HPAM and

Ax1 multipliers lead to the highest resource utilisation, which is close to 10% less than the exact implementation.

Calculating the power-area product designs using the 2NDH multiplier is the clear winner in that metric. However, these cases' MRE is also the highest, so there is a trade-off between the accuracy and the power consumption/area utilised.

Ultimately, the pair chosen for implementation will depend on the use case required, the criteria utilised, and the application requirements. If the objective is to get the lowest possible power consumption, the 2NDH-0KG pair could be used. If it is operation frequency, the 2NDH-0HE one could be used instead. If strict accuracy requirements exist, the Ax1-0HE pair could be used, among other uses. Different combos can solve different problems, depending on what is valued more.

Table 4.8: Vector Product results for 16-bit inputs with four multipliers

Mult	Adder	Clock (ns)	Relative Error	Total Power (W)	Static (W)	Dynamic (W)	LUT	FF	Dynamic*(LUT+3FF)
Exact	Exact	20	0.0000	0.188	0.106	0.082	1301	225	162.032
1UG	RA5	16	0.0320	0.118	0.105	0.013	440	145	11.375
2KD	RA5	18	0.0004	0.161	0.105	0.056	1227	193	101.136
5A2	RA5	17	0.0018	0.135	0.105	0.03	893	185	43.44
HDT	RA5	12	0.1237	0.109	0.104	0.005	260	111	2.965
Ax1233	RA5	19	0.0041	0.139	0.105	0.034	1140	193	58.446

As 32-bit adder implementations do not exist in the EvoApproxLib and no tests have been performed on chaining the adders to obtain a higher bit width adder, the only one used in the 16-bit input implementation was the RA5. The multipliers chosen were the 1UG, 2KD, 5A2, HDT and Ax1233, for reasons mentioned in subsection 4.3.2.

With the exception of the HDT-RA5 pairing, all pairs of modules showed a MRE below 10%. The accuracy concerns with using the HDT multiplier existed since the tests with only approximate multipliers. However, the power consumption and resource utilisation benefits were too big to dismiss this multiplier for this test. The MRE of the 2KD-RA5 pair was especially good, being only 0.04%.

Regarding resource utilisation, all pairs used fewer resources than the exact implementation. The 2KD and Ax1233 saved the least amount of resources, while the HDT-RA5 pair achieved the lowest utilisation, with 80% less LUTs and 49% less FFs than the exact module. This resource-saving is massive, so this would be the pair to use if the goal of using approximate computing was only to save resources.

For power consumption, as per usual, static power is the majority of total consumption. The HDT multiplier's immense power-saving capabilities are shown once again here, as the module has a dynamic power consumption of only 5mW, around 6% of the dynamic power consumption of the exact implementation. All pairs of modules perform better in this metric as well. The worst-performing pair, the 2KD-RA5 pair, has a dynamic power consumption of 56mW, which is still 32% less power consumption.

All pairs can achieve a higher operation frequency than the original exact implementation. This could operate at a maximum of 50 MHz. The various module pairs improve differently on this frequency, going from the Ax1233-RA5 with 53 MHz to the HDT-RA5 pair with 83 MHz. The faster operation frequency enables the possibility of obtaining the results faster for the internal product of two vectors, indirectly leading to lower total power consumption as the static power consumption, which is the majority of total power, decreases.

In the case where accuracy requirements rule out the usage of the HDT multiplier module, two other pairs can be used, the 1UG-RA5 and the 5A2-RA5. These two modules have worse maximum operating frequencies, dynamic power consumptions and resource utilisations than the HDT-RA5 pair but provided lower MRE and better accuracy, allowing for the use of approximate computing in uses where the accuracy requirements are stricter than 10% in the case of the 1UG, or than 1% in the case of the 5A2.

4.4 Image Sum

After performing tests using the vector product module, it was decided to test the arithmetic units in other scenarios where result degradation would have a clear noticeable impact beyond the numerical error. As such, two simple image processing applications were developed and tested: the Image Sum and the Sobel filter. This section presents the results obtained for the Image Sum application.

The image sum application is a straightforward algorithm that takes two images and sums them together pixel by pixel. It has no practical use, but it is an example where errors in the process have apparent visual effects on the result when they are large enough. This simple circuit can be achieved using only one adder that sums up one pixel each clock cycle.

The images used for this experiment are shown in Figure 4.1. The first image is the Lena image, a standard image used in image processing. The second is a landscape image, and the third is a cat image. These images were resized to be the same size and then converted to greyscale. In greyscale, each pixel is represented using an 8-bit number, so the possible values range from 0 to 255, where 0 is black and 255 is white. Adding two values while maintaining the ability to observe the result in an image means that either a saturation post-addition needs to be performed or changes to the input values need to be performed to prevent overflow. The first approach would lead to images that generally have high pixel values, appearing two bright, so a simple solution to the problem was to divide all input numbers by two by performing a right shift. This would guarantee that the result would never overflow using exact addition.

Since three images were chosen to be used, the images were added in three different scenarios. The first scenario is the sum of the cat and the landscape images (Figure 4.2a), the second scenario is the sum of the cat and the Lena images (Figure 4.2b), and the third scenario is the sum of the landscape and the Lena images (Figure 4.2c).

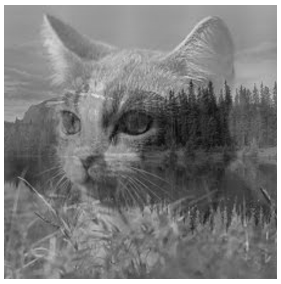
The module that performs the image sum was implemented using Verilog, using a parameter to change the adder being used to operate. This parameter was changed using a Python script that rewrote the appropriate lines in the hardware description files. After performing the synthesis and implementation and obtaining the resource utilisation and power consumption, another Python

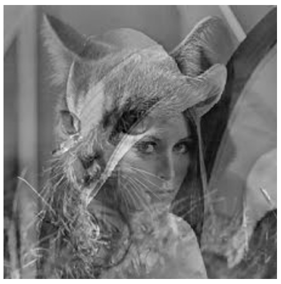
(a) Lena

(b) Landscape

(c) Cat

Figure 4.1: Images used for the Image Sum experiment


(a) Sum of cat and landscape (Scenario 1)


(b) Sum of cat and Lena (Scenario 2)

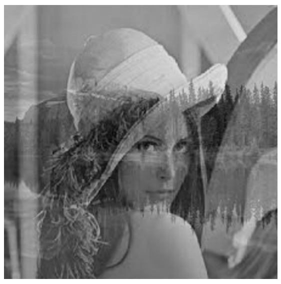
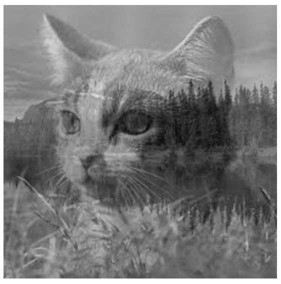

(c) Sum of landscape and Lena (Scenario 3)

Figure 4.2: Exact Results for the Image Sum experiment

script would better reconstruct the output using the output from the module to visualise the impact of the approximation on the image. Figure 4.3 has two examples of these images, one using the 110 adder and another from the 0KG adder, both from the EvoApproxLib. The second adder's result is subpar, making it unable to extract useful information from the operation. One important thing to note is that, even though the inputs were altered so that overflow never happened in the exact scenario, some approximate modules still led to overflow cases, making the addition of a saturation step at the end of the operation, where any value above 255 would become 255. In some cases, this led to drastic changes, where a large portion of the results was between 255 and 300.


(a) Scenario 1 results using the 110 adder

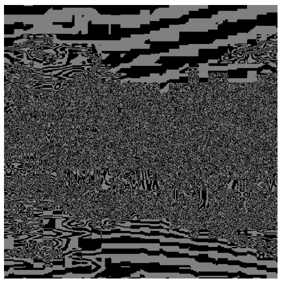

(b) Scenario 1 results using the 0KG adder

Figure 4.3: Images obtained from the Image Sum experiment using different adders

In an effort to better visualise the error each module committed, a simple function was added to the Python script in order to create the difference image between the exact result from figure 4.2 and the results obtained from the approximate adders (Figure 4.3). This image is obtained by obtaining the absolute difference between each pixel from these two images. An example from one of the previous adders, the 0KG, can be seen in Figure 4.4. If the results are perfect, this image will appear primarily black, as the differences are low. When the differences are considerable, this difference image can look like the result image, while usually, it is just almost random noise. These differences are primarily due to the nature of the errors of the adder, which are not studied in depth due to time constraints and the use of only the MRE error metric.

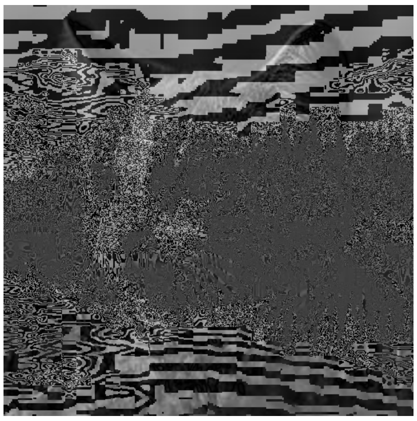


Figure 4.4: Example of difference image for the 0KG adder

After running all three scenarios with all adders, the image difference metric (defined in section 2.3) was calculated for all cases. The geometric mean of these three scenarios was also calculated. These values can be seen in table 4.9. Results below 5, like those obtained from the 6K6, 6P8, 6PT, 0NT and 110 adders, are acceptable results, which are very close to the exact image and differences are only noticed upon close inspection. At the ten mark, the quality of the images obtained is degraded to a point where it is no longer acceptable. At 50 and above, the image does not resemble the correct result and is considered terrible. As such, adders 0HE, 0KG, 0KU and 0SD have too much of a loss for any gains in area and power to compensate for it. It is important to note that the inputs and outputs are 8-bit, so 16-bit adders might perform worse, as the errors they commit might be more significant for lower inputs, being balanced by more accurate MSB calculations, which do not happen in this scenario.

The power consumption and area used by these circuits can be seen in table 4.10. As with the vector product case, the static power consumption is a significant part of the total power consumption, sometimes being more than 80%. Once again, this is due to the characteristics of a FPGA and the way interconnections are performed. Of the remaining dynamic power consumption, the great majority of it comes from I/O operations, as the module performs one addition per pair of inputs, and the images used were 256×256, meaning that summing two images requires changing the inputs 65536 times. All the adders were implemented using only LUTs. Unlike the vector product implementation, the RA5 adder uses more LUTs than the exact adder. As expected, the modules

Table 4.9: Image Sum Image Difference

Adder	Image Difference			
	Scenario 1	Scenario 2	Scenario 3	Geometric Mean
Exact	0.000	0.000	0.000	0.000
RA4	6.697	7.307	7.111	7.034
RA5	7.343	7.440	7.164	7.315
3RE	19.662	19.457	23.597	20.822
6K6	2.493	2.513	2.493	2.500
6P8	1.231	1.224	1.226	1.227
6PT	0.501	0.501	0.497	0.500
6QU	5.324	5.363	5.458	5.381
108	13.041	11.069	12.153	12.060
0GX	10.608	10.641	9.680	10.300
0HE	59.129	54.473	61.494	58.292
0KG	89.972	87.979	92.997	90.293
0KU	104.718	111.159	108.373	108.051
0NT	3.347	3.334	3.325	3.335
0RH	37.268	35.451	33.625	35.417
0SD	88.156	85.242	91.679	88.320
110	1.226	1.226	1.220	1.224

with the highest error and image difference use the lowest amount of LUTs, with three modules even using none. In these cases, Vivado optimises the circuit to directly connect some bits of the inputs to an output, immediately connecting bits that are always considered 0 to ground.

To better understand if this is not a mistake, the image sum module was altered in order to use eight adders simultaneously, parallelising the operation and using more resources. The results for these synthesis attempts can be found in table 4.11. As it is possible to observe in the table, the area usage increased by 8×, and the adders previously mentioned continue to be implemented using no LUTs.

4.5 Sobel Filter

The second image process application implemented was a Sobel filter. This filter is used in edge detection algorithms, and its results are images where edges and transitions are emphasised. It consists of applying two kernels to determine gradients, one for the horizontal values and one for the vertical values. These are then combined to form one single image. The application of a kernel to an image is a simple convolution process. An example of an image after applying the Sobel filter can be seen in Figure 4.5.

The Verilog modules for this operation are part of the benchmarking suite AxBench [28] (Section 2.7) and is one of the two planned circuits to be used from this suite, the other being the K-means clustering algorithm. After some simple adjustments to the modules provided so that a parameter could define the adders to be used, the modules were ready for testing.

Table 4.10: Image Sum Area and Power Results

Adder	LUTs	Static Power (W)	Total Power (W)
Exact	7	0.105	0.132
RA4	7	0.105	0.119
RA5	10	0.105	0.123
0GX	2	0.104	0.109
0HE	0	0.104	0.106
0KG	0	0.104	0.106
0KU	1	0.104	0.105
0NT	4	0.104	0.114
0RH	1	0.104	0.109
0SD	0	0.104	0.109
110	6	0.105	0.124
3RE	1	0.104	0.110
6K6	6	0.105	0.115
6P8	7	0.105	0.121
6PT	7	0.105	0.126
6QU	5	0.104	0.111
108	5	0.104	0.111

Table 4.11: Image Sum Area and Power with eight adders

Adder	LUTs	Static Power (W)	Total Power (W)
Exact	56	0.106	0.188
RA4	56	0.105	0.149
RA5	80	0.105	0.167
0GX	16	0.105	0.124
0HE	0	0.104	0.107
0KG	0	0.104	0.107
0KU	8	0.104	0.109
0NT	32	0.105	0.139
0RH	8	0.105	0.121
0SD	0	0.105	0.116
110	48	0.105	0.163
3RE	8	0.105	0.125
6K6	48	0.105	0.143
6P8	56	0.105	0.155
6PT	56	0.105	0.166
6QU	40	0.105	0.132
108	40	0.105	0.128

The hardware description for the Sobel operation uses different bit width adders, ranging from 8 to 11 bits. However, most adders being tested do not have implementations for these widths. As such, simple wrapper modules were created that could work with any bit width and would appropriately adapt the input number to be used in an 8 or 16-bit adder. In the 8-bit case, the MSBs were calculated using exact adders, which reduces error magnitude. For the 16-bit case,

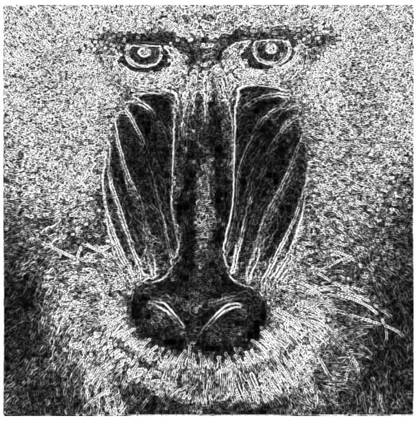
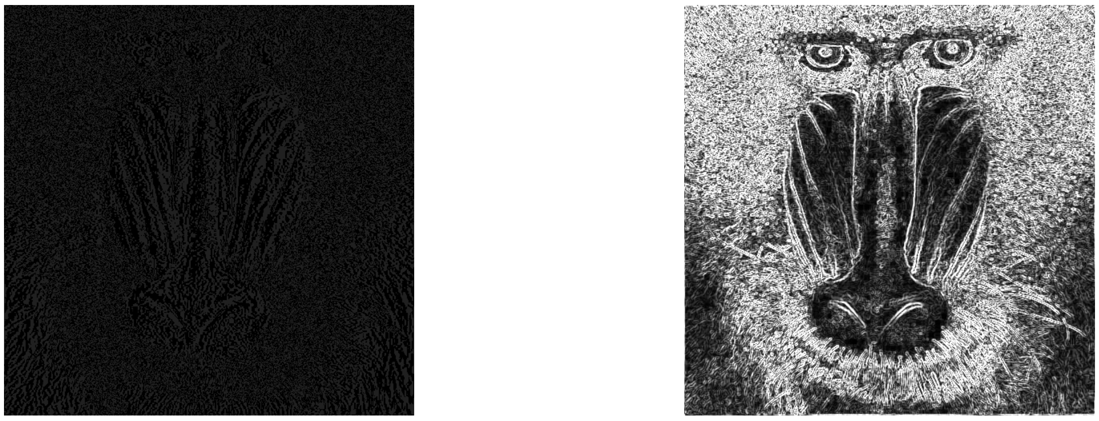


Figure 4.5: Sobel Filter Exact Result

the inputs were padded using 0s until they could be used in the adders. Due to these different implementations, the adders are grouped into three distinct options: 8-bit, 16-bit, and variable width.

In order to obtain the image difference metric, a simulation using Icarus Verilog was performed. Afterwards, using simple Python scripts, the images were reconstructed from the outputs of the modules. In most cases, the error appeared as a difference in the result pixel intensity, as the image shape could still be confirmed, albeit sometimes with some difficulty. Two images containing the results can be seen in Figures 4.6a and 4.6b.



(a) Sobel Filter 0KU Result

(b) Sobel Filter 6PT Result

Figure 4.6: Images obtained after applying Sobel Filter

Similarly to the image sum scenario, the difference image was obtained by performing the absolute difference between the exact result and the values obtained from the approximate modules being tested. For most cases where the error is significant enough for this image not to be completely black, the original image can be perceived without much distortion, further confirming that the differences happen mostly in intensity. Figure 4.7 shows the difference image for the 0KU adder, which is almost the intended result image, showcasing how poorly this adder performs this operation.

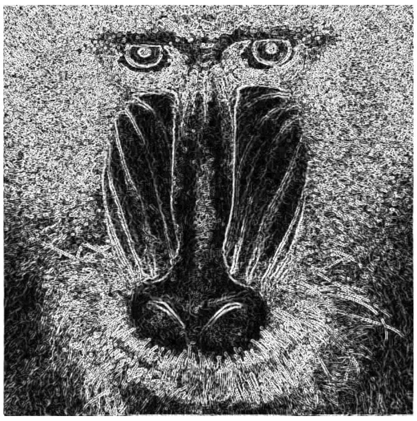


Figure 4.7: Example of difference image for the 0KU adder

After creating the difference image, obtaining the image difference metric is trivial by performing the square root of the mean square error of the values in this image. The obtained values are shown in tables 4.12 to 4.14.

Table 4.12: Sobel Filter Image Difference for adders with variable bit width

Adder	Exact	RA4	RA5
ImgDiff	0.000	34.250	47.277

Table 4.13: Sobel Filter Image Difference for 8-bit adders

Adder	Exact	3RE	6K6	6P8	6PT	6QU	108
ImgDiff	0.000	37.463	7.786	3.328	1.290	17.686	36.257

Table 4.14: Sobel Filter Image Difference for 16-bit adders

Adder	Exact	0GX	0HE	0KG	0KU	0NT	0RH	0SD	110
ImgDiff	0.000	20.662	84.883	126.660	135.783	9.147	119.456	125.457	2.061

Comparing these image difference metrics with the ones obtained in the image sum test scenario, it is possible to see that for the Sobel operation, the metric is worse for all modules, with significant changes for the ones that had medium values previously (between 5 and 50). Only three adders, the 8-bit 6P8 and 6PT, and the 16-bit 110 obtained values lower than five in this metric. Not even calculating the MSBs using exact adders caused this trend to shift in the 8-bit adders.

The increase in the image metric difference can be explained by the simple fact that, unlike the image sum operation, the Sobel filter does not consist of only one addition, and as such, errors can accumulate, causing the result to deviate further.

Unfortunately, due to this application not being the main focus of the dissertation and the time constraints, synthesis and implementation were not performed for the Sobel filter operation, so there are no values for the power consumption and the used resources.

Earlier in this section, it was mentioned that the K-means clustering algorithm from the AxBench benchmarking suite would also be used and implemented. However, simulations performed using it could not obtain any meaningful result, as the hardware description module does not output any values. Since there is no user manual or guide to troubleshoot the problems correctly, and time constraints were already a concern, it was chosen not to use this benchmark.

4.6 Neural Network

To fulfil the objectives outlined in Section 1.2, namely the implementation of a simple inference accelerator for digital neural, it is necessary to choose and implement an adequate Neural Network. As the implementation of said network is not the primary goal of the dissertation but only a means to demonstrate the effectiveness and performance of the implemented designs, a complex problem should not be used. As such, the MNIST database of handwritten digits, whose problem to be solved is the correct identification of greyscale handwritten digits, was chosen as the problem to solve.

This database comprises 70000 28×28 greyscale images divided into two datasets. Sixty thousand compose the training set, and the remaining 10000 the testing set. The images are also labelled with the correct digit they represent, which can be used during training for backpropagation and testing to determine the accuracy achieved by the model.

The implemented network should be simple and not occupy an extensive area, as it will be implemented in a FPGA. Due to this, a simple approach to the network, composed only of fully connected layers and with few nodes and layers, was chosen as the target implementation.

Having defined the target network, a software version was implemented using PyTorch, a well-known Python library for machine learning applications. This version used an input layer of size 784 (28×28), which required reshaping of the inputs but would simplify the hardware implementation, and two hidden layers with 30 nodes each between this layer and the ten-node output layer, where each output would represent the likelihood of the input being each of the digits from 0 to 9. After training and evaluation, the accuracy achieved by this model was 94.46%.

These results are obtained using floating-point 32-bit numbers, the default assumed by PyTorch. However, the arithmetic units to be designed and implemented operate with integer numbers (or fixed point, which can be done using integer operations, taking into account some considerations with the number of bits that represent the fractional part of the number), so this network had to be quantised. In this way, the network weights and biases would be converted to 8-bit integers between 0 and 255, and calculations in the layers would be performed using integers, allowing the use of the modules designed. It would also benefit from using fewer Block Random Access Memory (BRAM) of the FPGA, allowing for more efficient hardware implementation. To quantise the network, the Quantization API from PyTorch was used. After that, a new test was performed to check if the quantisation operation significantly impacted the network's performance. An accuracy of 94.22% was achieved, proving that the operation did not affect the results significantly.

The weights and biases from this implementation were saved to be possible to use them for the hardware implementation using Verilog. A pre-made implementation was preferred to creating one from scratch, as that would have taken a long time. As such, an implementation under the MIT Licence was found and used.

The Neural Network module, zyNet, contains a simple, fully connected network of 784 input nodes, and three hidden layers, the first two with 30 nodes and the last with 10 nodes and 10 output nodes. The module also contains AXI interface for data transfers and a submodule to determine the maximum value to guess the correct output. The activation function for the nodes can be set to Sigmoid or ReLU. The latter was chosen as the Sigmoid implementation required a table of possible values to generate, representing another challenge. While the ReLU has slightly worse accuracy results in the tests performed by the author of the hardware descriptions, Vipin Kizheppatt, the structure is easier to implement.

The weights and biases previously saved are used to create memory implementation files that will initialise the BRAM of the FPGA with the correct values. In order to test the accuracy of the module, 100 test cases were chosen to be used as inputs to the neural network. This number is relatively low compared to the maximum number of possible test cases, 10000. However, it is enough to determine if there is a significant loss of network accuracy when approximate arithmetic units are introduced.

After some changes to the modules in order to be able to use parameters to change the adder and multiplier circuit being used, as well as the creation of modules that would allow the use of the 8-bit adders by chaining two together to create a 16-bit adder, synthesis and implementation were performed. However, due to turning off the DSP blocks and other synthesis options, the resources used to implement the network exceeded the targeted FPGA capacity. To solve this problem, a new project was created targetting a different FPGA of the same family, the Artix-7, while trying to maintain the maximum amount of similarities, such as the speed grade. The new project targeted the XC7A200T, which was capable of handling the neural network and more in terms of resources. Unfortunately, the Post-Implementation Timing Simulations required to accurately determine power consumption would not run in the time left for the project. To solve this, the synthesis setting that forced to keep equivalent registers was disabled, and the resource-sharing option was turned back on, drastically reducing the resources used and allowing Post-Synthesis Functional Simulations to be run on time. Because of these setting changes, the synthesis could now be performed on the Zedboard (Zynq-7000) again.

With these setbacks now solved, the synthesis and implementation process was performed for the neural network and the post-implementation simulation to obtain the power consumption. The results of this process for adders are presented in table 4.15 while table 4.16 contains the results for the multipliers.

Performing an analysis of the obtained accuracy for the 100 test cases, it is possible to see that only six adders out of 16 could maintain network accuracy. All the other ones lead to no matter the input image, the predicted digit being always the same, being 6 for the RA5, 3 for the 0KG, 1 for the 6QU and 0 for all others.

Table 4.15: Neural Network Area and Power Results for Approximate Adders

Adder	LUTs	FFs	Static Power (W)	Total Power (W)	Accuracy (%)
Exact	25023	9657	0.132	0.204	91
RA4	24717	9657	0.132	0.203	91
RA5	26871	9657	0.132	0.210	11
0GX	24599	8937	0.132	0.203	90
0HE	23401	8537	0.132	0.202	86
0KG	22513	8137	0.132	0.199	17
0KU	21579	7382	0.132	0.199	8
0NT	24188	8920	0.132	0.204	88
0RH	23320	8360	0.132	0.202	88
0SD	20435	6885	0.132	0.198	8
110	25179	9497	0.132	0.206	91
3RE	20431	8062	0.132	0.201	8
6K6	24036	8681	0.132	0.202	8
6P8	25897	9177	0.132	0.205	8
6PT	26399	9417	0.132	0.205	8
6QU	22695	7942	0.132	0.199	2
108	22592	8428	0.132	0.199	8

Out of the six adders where the model did not lose the ability to predict different digits, the RA4 and the 110 adders obtained the same accuracy as the exact implementation, 91%. The 0GX adder obtained 90%, getting only one more case wrong. The 0NT and 0RH adders achieved a prediction accuracy of 88%, while the 0HE got 86%.

These results are surprising, as some of the adders that got above 80% did not perform very well in other applications. This is the case of the 0HE adder, which resulted in the bottom half of adders in image processing applications. At the other end of the spectrum, the RA5 adder that showed promising results during vector product, even better than the RA4, performed terribly in the neural network. Similarly, the 6K6, 6P8 and 6PT 8-bit adders predicted the same digit for all cases, even though they were the most accurate in the image sum and Sobel filter tests.

These discrepancies may be due to the large number of computations performed in the neural network that allow errors to accumulate. This causes wild result variations in cases where the error profile of the adder is not balanced, leading to a loss of model accuracy.

Regarding power consumption, all cases had the same static power consumption that represented around 65% of total power. The 0RH and the 0HE have the lowest dynamic power consumption of the six adders. However, the 0RH uses fewer resources in both LUTs and FFs while having 2% more accuracy, making it a better adder for this network if the goal is the lowest power consumption possible.

As Table 4.16 shows, all approximate multipliers caused the network only to predict 0 for all the 100 test cases. It was hypothesised that this could be caused by two factors. The first one was the required changes to the network to use the approximate multipliers. These changes sometimes lead to needing one more clock cycle to perform calculations, which is not accounted

Table 4.16: Neural Network Area and Power Results for Approximate Multipliers

Multiplier	LUTs	FFs	Static Power (W)	Total Power (W)	Accuracy (%)
Exact	25023	9657	0.132	0.204	91
Ax0233	28323	10014	0.132	0.211	8
Ax0333	27984	10012	0.132	0.211	8
Ax1233	27989	10019	0.132	0.210	8
Ax3333	26873	10006	0.132	0.207	8
1UG	11727	6437	0.132	0.176	8
2KD	29861	9506	0.132	0.211	8
3CF	37548	9845	0.132	0.226	8
3H1	36975	9913	0.132	0.227	8
3PM	36169	9660	0.132	0.230	8
5A2	21445	8402	0.132	0.195	8
7QP	30942	9599	0.132	0.216	8
HDT	7300	4965	0.132	0.166	8

for in the control signal circuitry. However, this was ruled out as a cause because performing exact multiplication through the same changes used for approximate multipliers led to obtaining the same result as the original network. The second reason is that both inputs are cast to signed values before being multiplied, even though the result goes to an unsigned register. However, this makes it so that Vivado interprets the multiplication as a signed multiplication, while all approximate modules perform unsigned multiplication. This leads to incorrect results. A possible solution to this is to use the unsigned multipliers with double the input width, extending the signals of the inputs, and after the multiplication is done, use only the least significant half of the result. Since the multipliers have a fixed width, this process would need to be achieved through recursive multiplication, leading to a rise in used resources and consumption. Another solution would be to use signed multipliers, which would need to be tested and go through all the steps mentioned in the previous Chapter and this one.

Since the accuracy is unreliable, the power and area analysis cannot be performed only on the modules that obtained satisfactory results. However, with the current setup, more resources would need to be used to obtain the actual accuracy values, making the exact multiplier always need fewer resources and consume less power. This is already the case for 9 of the 12 designs, except for 1UG, the 5A2 and the HDT multipliers. The results presented in subsection 4.3.2 show that the 1UG and the HDT multipliers introduce a more significant error than the 5A2. They also do use much fewer resources and have the lowest power consumption of them as well. With the HDT multiplier using $\frac{1}{3}$ of the LUTs and $\frac{1}{2}$ of the FFs, as well as having half the dynamic power consumption.

Chapter 5

Conclusions and Future Work

To conclude, the use of approximate computing, more specifically through approximate arithmetic units, leads to gains in area and power consumption while sacrificing accurate results. Depending on the use case, these gains can be variable and the choice of circuit to use needs to be made considering application requirements. In the case of a small, fully connected neural network, it is shown that even adders with similar performance for single additions can lead to vastly different results for the model's accuracy. In the studied implementation, the power consumption reduction was barely noticeable, largely due to the high static power consumption values.

Unfortunately, the network accuracy results when multipliers were so poor due to a problem with implementation and not because of the approximate modules themselves. Because of it, no conclusions can be drawn regarding the modules that can be used for the implemented network and how they affect the model's accuracy. Only general resource and power consumption conclusions can be made, which may not be valid due to the modules affecting the network's accuracy. Unfortunately, due to time constraints, it was not possible to solve this problem on time, but solving it and obtaining the correct results is a top priority for the future.

Implementing the arithmetic units and testing them in a FPGA, while allowing for a more extensive test of solutions, proved to have some consequences on the results obtained. The major ones are the excessive static power consumption and the use of specialised structures to optimise the implementation. High static power consumption is inherent to the FPGA structure and represented more than 50% of total consumption, reaching upwards of 90% for the more straightforward scenarios. This caused total power consumption to be an unreliable metric to verify if the implemented modules achieved significant savings in power, and the dynamic power consumption had to be used instead. Regarding specialised structures, like DSPs, it was proven that approximate designs do not benefit from most of them, giving an edge to the exact implementations. If they are available, the choice to use them with exact implementations will, most of the time, be the most resource and power-efficient option. Other specialised structures, like carry chains, can be partially used by approximate designs, so the differences are not as significant for adder implementations. However, the presence of these structures may lead to a unit design with them in mind and trying to optimise them. Some designs specifically targeting FPGA implementation exist, but they were not studied as it was believed they would not be comparable to the presented designs.

As seen throughout chapter 4, approximate designs' accuracy varies wildly depending on the use case. This is because different applications use different amounts of calculations, chain them in different ways, and use different input ranges, among other differences. As such, no definitive approximate module can be used for all applications. Even in similar use cases, such as the internal product of two vectors calculation and the operations in a node in a neural network, the results can be different, as was proven for the adder case. With this, it is safe to assume that if the network configuration changes significantly, for example, if a convolutional neural network is used instead, the most appropriate approximate module will change again. This changing nature of the best module to use for each application is tied to the spread of errors in the module, which needs to be further studied in order to be able to predict what application type a module suits best.

5.1 Future Work

As briefly mentioned earlier, further developments could be made to improve and build upon the results presented. The first and most important one would be to solve the implementation problems that led to poor results for the use of approximate multipliers in the neural network application. Further testing could also be done using this neural network configuration, combining both approximate arithmetic units or using different bit widths, which would require obtaining new weights and biases. Similar tests could also be done to approximate signed multipliers to compare them with the unsigned ones implemented in this dissertation. Lastly, in an attempt to remove the large static power consumption, the better-performing circuits could be implemented in an application-specific integrated circuit (ASIC) using the appropriate software and synthesis flow.

References

[1] Gordon E. Moore. Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff. *IEEE Solid-State Circuits Society Newsletter*, 11(3):33–35, September 2006. Conference Name: IEEE Solid-State Circuits Society Newsletter. URL: https://ieeexplore.ieee.org/document/4785860, doi:10.1109/N-SSC.2006.4785860.

[2] R.H. Dennard, F.H. Gaensslen, Hwa-Nien Yu, V.L. Rideout, E. Bassous, and A.R. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, October 1974. Conference Name: IEEE Journal of Solid-State Circuits. URL: https://ieeexplore.ieee.org/document/1050511, doi:10.1109/JSSC.1974.1050511.

[3] Weiqiang Liu, Fabrizio Lombardi, and Michael Shulte. A Retrospective and Prospective View of Approximate Computing [Point of View}. *Proceedings of the IEEE*, 108(3):394–399, March 2020. URL: https://ieeexplore.ieee.org/document/9024190/, doi:10.1109/JPROC.2020.2975695.

[4] Bhavani Koyada, N. Meghana, Md. Omair Jaleel, and Praneet Raj Jeripotula. A comparative study on adders. In *2017 International Conference on Wireless Communications, Signal Processing and Networking (WiSPNET)*, pages 2226–2230, March 2017. doi:10.1109/WiSPNET.2017.8300155.

[5] Honglan Jiang, Cong Liu, Leibo Liu, Fabrizio Lombardi, and Jie Han. A Review, Classification, and Comparative Evaluation of Approximate Arithmetic Circuits. *ACM Journal on Emerging Technologies in Computing Systems*, 13(4):1–34, August 2017. URL: https://dl.acm.org/doi/10.1145/3094124, doi:10.1145/3094124.

[6] Manickam Ramasamy, G. Narmadha, and S. Deivasigamani. Carry based approximate full adder for low power approximate computing. In *2019 7th International Conference on Smart Computing & Communications (ICSCC)*, pages 1–4, June 2019. doi:10.1109/ICSCC.2019.8843644.

[7] Shalini Singh, Pavan Kumar Pothula, and Madhav Rao. Design and Evaluation of On-chip DCT accelerators based on Novel Approximate Reverse Carry Propagate Adders. In *2022 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 8–13, July 2022. ISSN: 2159-3477. doi:10.1109/ISVLSI54635.2022.00015.

[8] H. R. Mahdiani, A. Ahmadi, S. M. Fakhraie, and C. Lucas. Bio-Inspired Imprecise Computational Blocks for Efficient VLSI Implementation of Soft-Computing Applications. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 57(4):850–862, April 2010. Conference Name: IEEE Transactions on Circuits and Systems I: Regular Papers. doi:10.1109/TCSI.2009.2027626.

[9] Shobhit Belwal, Rajat Bhattacharjya, Kaustav Goswami, and Dip Sankar Banerjee. ACLA: An Approximate Carry-Lookahead Adder with Intelligent Carry Judgement and Correction. In *2021 22nd International Symposium on Quality Electronic Design (ISQED)*, pages 1–7, April 2021. ISSN: 1948-3287. doi:10.1109/ISQED51717.2021.9424329.

[10] Ing-Chao Lin, Yi-Ming Yang, and Cheng-Chian Lin. High-Performance Low-Power Carry Speculative Addition With Variable Latency. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 23(9):1591–1603, September 2015. Conference Name: IEEE Transactions on Very Large Scale Integration (VLSI) Systems. doi:10.1109/TVLSI.2014.2355217.

[11] Kai Du, Peter Varman, and Kartik Mohanram. High performance reliable variable latency carry select addition. In *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1257–1262, March 2012. ISSN: 1558-1101. doi:10.1109/DATE.2012.6176685.

[12] Ning Zhu, Wang Ling Goh, Gang Wang, and Kiat Seng Yeo. Enhanced low-power high-speed adder for error-tolerant application. In *2010 International SoC Design Conference*, pages 323–327, November 2010. doi:10.1109/SOCDC.2010.5682905.

[13] Afshin Khaksari, Omid Akbari, and Behzad Ebrahimi. BEAD: Bounded error approximate adder with carry and sum speculations. *Integration*, 88:353–361, January 2023. URL: https://www.sciencedirect.com/science/article/pii/S016792602200147X, doi:10.1016/j.vlsi.2022.10.015.

[14] Vojtech Mrazek, Radek Hrbacek, Zdenek Vasicek, and Lukas Sekanina. EvoApprox8b: Library of Approximate Adders and Multipliers for Circuit Design and Benchmarking of Approximation Methods. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 258–261, Lausanne, March 2017. IEEE. URL: https://ieeexplore.ieee.org/document/7926993/, doi:10.23919/DATE.2017.7926993.

[15] Lukas Sekanina and Zdenek Vasicek. Approximate circuit design by means of evolvable hardware. In *2013 IEEE International Conference on Evolvable Systems (ICES)*, pages 21–28, April 2013. doi:10.1109/ICES.2013.6613278.

[16] Jiri Petrlik and Lukas Sekanina. Multiobjective evolution of approximate multiple constant multipliers. In *2013 IEEE 16th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*, pages 116–119, April 2013. doi:10.1109/DDECS.2013.6549800.

[17] James Hilder, James A. Walker, and Andy Tyrrell. Use of a multi-objective fitness function to improve cartesian genetic programming circuits. In *2010 NASA/ESA Conference on Adaptive Hardware and Systems*, pages 179–185, June 2010. doi:10.1109/AHS.2010.5546262.

[18] Saurav Chanda, Koushik Guha, Santu Patra, Anupam Karmakar, Loukrakpam Merin Singh, and Krishna Lal Baishnab. A 32-bit Energy Efficient Exact Dadda Multiplier. In *2019 IEEE 5th International Conference for Convergence in Technology (I2CT)*, pages 1–4, March 2019. doi:10.1109/I2CT45611.2019.9033535.

[19] G. Challa Ram, D. Sudha Rani, R. Balasaikesava, and K. Bala Sindhuri. Design of delay efficient modified 16 bit Wallace multiplier. In *2016 IEEE International Conference on*

Recent Trends in Electronics, Information & Communication Technology (RTEICT), pages 1887–1891, May 2016. doi:10.1109/RTEICT.2016.7808163.

[20] Xuan Wang and Weikang Qian. MinAC: Minimal-Area Approximate Compressor Design Based on Exact Synthesis for Approximate Multipliers. In *2022 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 677–681, May 2022. ISSN: 2158-1525. doi:10.1109/ISCAS48785.2022.9938008.

[21] Chia-Hao Lin and Ing-Chao Lin. High accuracy approximate multiplier with error correction. In *2013 IEEE 31st International Conference on Computer Design (ICCD)*, pages 33–38, October 2013. ISSN: 1063-6404. doi:10.1109/ICCD.2013.6657022.

[22] L Hemanth Krishna, J Bhaskara Rao, Sk Ayesha, Sreehari Veeramachaneni, and Sk Noor Mahammad. Energy Efficient Approximate Multiplier Design for Image/Video Processing Applications. In *2021 IEEE International Symposium on Smart Electronic Systems (iSES)*, pages 210–215, December 2021. doi:10.1109/iSES52644.2021.00056.

[23] Divy Pandey, Vishesh Mishra, Saurabh Singh, Sagar Satapathy, Babita Jajodia, and Dip Sankar Banerjee. HPAM: An 8-bit High-Performance Approximate Multiplier Design for Error Resilient Applications. In *2022 23rd International Symposium on Quality Electronic Design (ISQED)*, pages 1–5, April 2022. ISSN: 1948-3295. doi:10.1109/ISQED54688.2022.9806220.

[24] Haroon Waris, Chenghua Wang, Weiqiang Liu, Jie Han, and Fabrizio Lombardi. Hybrid Partial Product-Based High-Performance Approximate Recursive Multipliers. *IEEE Transactions on Emerging Topics in Computing*, 10(1):507–513, January 2022. Conference Name: IEEE Transactions on Emerging Topics in Computing. doi:10.1109/TETC.2020.3013977.

[25] Reza Zendegani, Mehdi Kamal, Milad Bahadori, Ali Afzali-Kusha, and Massoud Pedram. RoBA Multiplier: A Rounding-Based Approximate Multiplier for High-Speed yet Energy-Efficient Digital Signal Processing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(2):393–401, February 2017. Conference Name: IEEE Transactions on Very Large Scale Integration (VLSI) Systems. doi:10.1109/TVLSI.2016.2587696.

[26] Vishesh Mishra, Divy Pandey, Saurabh Singh, Sagar Satapathy, Kaustav Goswami, Babita Jajodia, and Dip Sankar Banerjee. ART-MAC: Approximate Rounding and Truncation based MAC Unit for Fault-Tolerant Applications. In *2022 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1640–1644, May 2022. ISSN: 2158-1525. doi:10.1109/ISCAS48785.2022.9937437.

[27] P Thejaswini, John Jose, and Sukumar Nandi. Energy Efficient Approximate MACs. In *2021 IEEE 18th India Council International Conference (INDICON)*, pages 1–6, December 2021. ISSN: 2325-9418. doi:10.1109/INDICON52576.2021.9691492.

[28] Amir Yazdanbakhsh, Divya Mahajan, Hadi Esmaeilzadeh, and Pejman Lotfi-Kamran. AxBench: A Multiplatform Benchmark Suite for Approximate Computing. *IEEE Design & Test*, 34(2):60–68, April 2017. Conference Name: IEEE Design & Test. doi:10.1109/MDAT.2016.2630270.

[29] Poojitha Lagidi, Aenugula Iswarya, Gangarapu Rajesh, and A.S. Keerthi Nayani. Design of 16-Bit and 32-Bit Approximate Full Adder Using Majority Logic. In *2021 2nd Global*

Conference for Advancement in Technology (GCAT), pages 1–5, October 2021. doi:10.1109/GCAT52182.2021.9587782.

Appendix A

Synthesis and Implementation Result Tables

This Appendix displays the Synthesis and Implementation Results for the scenarios not present in Section 4.3. The cases these tables present are discussed in this section, but the tables are not included in the main text to simplify reading the document. The conclusions drawn from these tables are mostly the same as those drawn from the tables in Section 4.3, with notable exceptions mentioned in that section.

A.1 Vector Product using Approximate Adders

Table A.1: Vector Product Results for 16-bit Adders with two simultaneous operations

Adder	Clock (ns)	Relative Error	Total Power (W)	Static (W)	Dynamic (W)	LUT	FF	Dynamic*(LUT+3FF)
Exact	11	0.0000	0.111	0.104	0.007	216	114	3.906
RA4	11	0.0245	0.111	0.104	0.007	214	114	3.892
RA5	11	0.0245	0.109	0.104	0.005	198	98	2.460
BEAD	12	0.1041	0.116	0.105	0.011	236	114	6.358
0GX	10	0.0017	0.117	0.105	0.012	218	110	6.576
0HE	10	0.0107	0.112	0.104	0.008	218	107	4.312
0KG	10	0.0513	0.111	0.104	0.007	203	103	3.584
0KU	-	0.4154	-	-	-	-	-	-
0NT	11	0.0026	0.119	0.105	0.014	237	112	8.022
0RH	11	0.0215	0.115	0.105	0.01	205	109	5.320
0SD	-	0.9193	-	-	-	-	-	-
110	11	0.0003	0.115	0.105	0.01	240	113	5.790

Table A.2: Vector Product Results for 16-bit Adders with four simultaneous operations

Adder	Clock (ns)	Relative Error	Total Power (W)	Static (W)	Dynamic (W)	LUT	FF	Dynamic*(LUT+3FF)
Exact	12	0.0000	0.121	0.105	0.016	378	129	12.240
RA4	12	0.0248	0.117	0.105	0.012	372	129	9.108
RA5	12	0.0245	0.113	0.104	0.009	341	113	6.120
BEAD	13	0.1041	0.132	0.105	0.027	414	129	21.627
0GX	12	0.0017	0.131	0.105	0.026	378	125	19.578
0HE	12	0.0107	0.122	0.105	0.017	356	122	12.274
0KG	12	0.0513	0.113	0.104	0.009	340	118	6.246
0KU	-	0.4154	-	-	-	-	-	-
0NT	12	0.0026	0.14	0.105	0.035	420	127	28.035
0RH	12	0.0215	0.124	0.105	0.019	354	124	13.794
0SD	-	0.9193	-	-	-	-	-	-
110	12	0.0003	0.141	0.105	0.036	433	128	29.412

Table A.3: Vector Product Results for 32-bit Adders with one simultaneous operation

Adder	Clock (ns)	Relative Error	Total Power (W)	Static (W)	Dynamic (W)	LUT	FF	Dynamic*(LUT+3FF)
Exact	13	0.0000	0.12	0.105	0.015	351	131	11.160
RA5	13	8.189E-05	0.118	0.105	0.013	333	99	8.190
BEAD	12	0.1038	0.131	0.105	0.026	402	131	20.670

Table A.4: Vector Product Results for 32-bit Adders with two simultaneous operations

Adder	Clock (ns)	Relative Error	Total Power (W)	Static (W)	Dynamic (W)	LUT	FF	Dynamic*(LUT+3FF)
Exact	16	0.0000	0.145	0.105	0.04	682	178	48.640
RA5	14	8.189E-05	0.133	0.105	0.028	649	146	30.436
BEAD	15	0.1032	0.171	0.105	0.066	744	178	84.348

Table A.5: Vector Product Results for 32-bit Adders with four simultaneous operations

Adder	Clock (ns)	Relative Error	Total Power (W)	Static (W)	Dynamic (W)	LUT	FF	Dynamic*(LUT+3FF)
Exact	16	0.0000	0.188	0.106	0.082	1301	225	162.032
RA5	16	8.189E-05	0.153	0.105	0.048	1221	193	86.400
BEAD	17	0.1032	0.267	0.107	0.16	1454	225	340.640

Table A.6: Vector Product Results for 48-bit Adders with one simultaneous operation

Adder	Clock (ns)	Relative Error	Total Power (W)	Static (W)	Dynamic (W)	LUT	FF	Dynamic*(LUT+3FF)
Exact	16	0.0000	0.159	0.105	0.054	713	203	71.388
RA5	16	5.128E-06	0.148	0.105	0.043	687	155	49.536
BEAD	17	0.1038	0.17	0.105	0.065	752	203	88.465

Table A.7: Vector Product Results for 48-bit Adders with two simultaneous operations

Adder	Clock (ns)	Relative Error	Total Power (W)	Static (W)	Dynamic (W)	LUT	FF	Dynamic*(LUT+3FF)
Exact	18	0.0000	0.204	0.106	0.098	1403	290	222.754
RA5	18	1.903E-06	0.189	0.106	0.083	1354	242	172.640
BEAD	19	0.1032	0.272	0.107	0.165	1460	290	384.450

Table A.8: Vector Product Results for 64-bit Adders with one simultaneous operation

Adder	Clock (ns)	Relative Error	Total Power (W)	Static (W)	Dynamic (W)	LUT	FF	Dynamic*(LUT+3FF)
Exact	19	0.0000	0.197	0.106	0.091	1310	261	190.463
RA5	19	1.248E-09	0.195	0.106	0.089	1275	197	166.074
BEAD	20	0.1038	0.245	0.106	0.139	1344	261	295.653

Table A.9: Vector Product Results for 64-bit Adders with two simultaneous operations

Adder	Clock (ns)	Relative Error	Total Power (W)	Static (W)	Dynamic (W)	LUT	FF	Dynamic*(LUT+3FF)
Exact	21	0.0000	0.294	0.107	0.187	2588	374	693.770
RA5	21	1.248E-09	0.254	0.107	0.147	2521	310	507.297
BEAD	22	0.1032	0.473	0.11	0.363	2656	374	1371.414

A.2 Vector Product using Approximate Multipliers

Table A.10: Vector Product Results for 8-bit Multipliers with one simultaneous operation

Mult	Clock (ns)	Relative Error	Total Power (W)	Static (W)	Dynamic (W)	LUT	FF	Dynamic*(LUT+3FF)
Exact	10	0.0000	0.109	0.104	0.005	122	83	1.855
Ax1	10	0.0043	0.108	0.104	0.004	121	83	1.48
Ax2	10	0.0506	0.108	0.104	0.004	118	83	1.468
Ax3	10	0.1029	0.108	0.104	0.004	113	83	1.448
HPAM	8	0.0575	0.108	0.104	0.004	126	83	1.5
2NDH	8	0.0820	0.108	0.104	0.004	91	83	1.36
12YX	8	0.0005	0.11	0.104	0.006	152	83	2.406
XFM	8	0.0056	0.108	0.104	0.004	130	83	1.516
ZDF	8	0.0013	0.11	0.104	0.006	146	83	2.37

Table A.11: Vector Product Results for 8-bit Multipliers with two simultaneous operations

Mult	Clock (ns)	Relative Error	Total Power (W)	Static (W)	Dynamic (W)	LUT	FF	Dynamic*(LUT+3FF)
Exact	11	0.0000	0.111	0.104	0.007	216	114	3.906
Ax1	12	0.0043	0.111	0.104	0.007	216	114	3.906
Ax2	11	0.0507	0.11	0.104	0.006	208	114	3.3
Ax3	11	0.1029	0.11	0.104	0.006	198	114	3.24
HPAM	9	0.0575	0.111	0.104	0.007	227	114	3.983
2NDH	9	0.0820	0.108	0.104	0.004	156	114	1.992
12YX	10	0.0005	0.114	0.105	0.009	264	114	5.454
XFM	10	0.0056	0.111	0.104	0.007	221	114	3.941
ZDF	10	0.0013	0.111	0.104	0.007	250	114	4.144

Table A.12: Vector Product Results for 16-bit Multipliers with two simultaneous operations

Mult	Clock (ns)	Relative Error	Total Power (W)	Static (W)	Dynamic (W)	LUT	FF	Dynamic*(LUT+3FF)
Exact	16	0.0000	0.145	0.105	0.04	682	178	48.64
1UG	13	0.0311	0.116	0.105	0.011	268	154	8.03
2KD	16	1.807E-05	0.148	0.105	0.043	685	178	52.417
3CF	16	7.099E-09	0.175	0.105	0.07	933	178	102.69
3H1	16	3.475E-08	0.183	0.106	0.077	934	178	113.036
3PM	16	4.099E-07	0.173	0.105	0.068	881	178	96.22
5A2	15	0.0013	0.126	0.105	0.021	511	174	21.693
7QP	16	8.674E-06	0.156	0.105	0.051	755	178	65.739
HDT	11	0.1226	0.109	0.104	0.005	168	138	2.91

Table A.13: Vector Product Results for 16-bit Multipliers with four simultaneous operations

Mult	Clock (ns)	Relative Error	Total Power (W)	Static (W)	Dynamic (W)	LUT	FF	Dynamic*(LUT+3FF)
Exact	20	0.0000	0.188	0.106	0.082	1301	225	162.032
1UG	16	0.0311	0.121	0.105	0.016	475	177	16.096
2KD	18	1.807E-05	0.2	0.106	0.094	1313	225	186.872
3CF	18	7.099E-09	0.244	0.106	0.138	1766	225	336.858
3H1	18	3.475E-08	0.242	0.106	0.136	1721	225	325.856
3PM	18	4.099E-07	0.22	0.106	0.114	1623	225	261.972
5A2	16	0.0013	0.144	0.105	0.039	913	217	60.996
7QP	16	8.674E-06	0.209	0.106	0.103	1441	225	217.948
HDT	12	0.1226	0.11	0.104	0.006	290	145	4.35

A.3 Vector Product Results with PowerOptimized_High Directive

Table A.14: Vector Product Results for 8-bit Multipliers using PowerOptimized_High Directive

Mult	Clock(ns)	Total Power (W)	Static (W)	Dynamic (W)	LUT	FF	Dynamic*(LUT+3FF)
Exact	14	0.115	0.105	0.01	360	129	7.47
Ax1	15	0.115	0.105	0.01	346	129	7.33
Ax2	13	0.113	0.104	0.009	330	129	6.453
Ax3	14	0.113	0.104	0.009	318	129	6.345
HPAM	14	0.114	0.105	0.009	346	121	6.381
2NDH	12	0.11	0.104	0.006	245	129	3.792
12YX	14	0.116	0.105	0.011	366	129	8.283
XFM	14	0.112	0.104	0.008	320	129	5.656
ZDF	14	0.116	0.105	0.011	360	129	8.217

Table A.15: Vector Product Results for 16-bit Multipliers using PowerOptimized_High Directive

Mult	Clock(ns)	Total Power (W)	Static (W)	Dynamic (W)	LUT	FF	Dynamic*(LUT+3FF)
Exact	18	0.166	0.105	0.061	1270	225	118.645
Ax0233	18	0.163	0.105	0.058	1206	225	109.098
Ax0333	18	0.165	0.105	0.06	1194	225	112.14
Ax1233	18	0.162	0.105	0.057	1169	225	105.108
Ax3333	18	0.162	0.105	0.057	1132	225	102.999
1UG	16	0.117	0.105	0.012	418	149	10.38
2KD	20	0.187	0.106	0.081	1165	223	148.554
3CF	20	0.222	0.106	0.116	1581	225	261.696
3H1	20	0.22	0.106	0.114	1536	225	252.054
3PM	20	0.1538	0.106	0.0478	1379	225	98.1812
5A2	17	0.132	0.105	0.027	796	201	37.773
7QP	20	0.189	0.106	0.083	1275	223	161.352
HDT	11	0.108	0.104	0.004	222	105	2.148

A.4 Vector Product Result with Approximate Multipliers and Adders

Table A.16: Vector Product Results for 8-bit inputs with one multiplier

Mult	Adder	Clock (ns)	Relative Error	Total Power (W)	Static (W)	Dynamic (W)	LUT	FF	Dynamic*(LUT+3FF)
Exact	Exact	10	0.0000	0.109	0.104	0.005	122	83	1.855
2NDH	0HE	8	0.0759	0.107	0.104	0.003	82	76	0.93
2NDH	0KG	8	0.1209	0.106	0.104	0.002	78	73	0.594
2NDH	RA5	7	0.3003	0.106	0.104	0.002	79	65	0.548
XFM	0HE	8	0.0384	0.108	0.104	0.004	115	76	1.372
XFM	0KG	8	0.0842	0.108	0.104	0.004	114	75	1.356
XFM	RA5	8	0.1789	0.107	0.104	0.003	113	67	0.942
HPAM	0HE	8	0.0529	0.109	0.104	0.005	120	75	1.725
HPAM	0KG	8	0.1022	0.107	0.104	0.003	120	72	1.008
HPAM	RA5	7	0.2052	0.107	0.104	0.003	119	67	0.96
Ax1	0HE	9	0.0116	0.108	0.104	0.004	115	76	1.372
Ax1	0KG	9	0.0464	0.107	0.104	0.003	113	75	1.014
Ax1	RA5	9	0.1389	0.107	0.104	0.003	112	67	0.939

Table A.17: Vector Product Results for 8-bit inputs with two multipliers

Mult	Adder	Clock (ns)	Relative Error	Total Power (W)	Static (W)	Dynamic (W)	LUT	FF	Dynamic*(LUT+3FF)
Exact	Exact	11	0.0000	0.111	0.104	0.007	216	114	3.906
2NDH	0HE	10	0.0760	0.108	0.104	0.004	136	107	1.828
2NDH	0KG	10	0.1209	0.106	0.104	0.002	126	98	0.84
2NDH	RA5	10	0.2526	0.107	0.104	0.003	132	94	1.242
XFM	0HE	13	0.0386	0.109	0.104	0.005	180	107	2.505
XFM	0KG	12	0.0867	0.108	0.104	0.004	172	103	1.924
XFM	RA5	11	0.1037	0.108	0.104	0.004	178	98	1.888
HPAM	0HE	11	0.0529	0.111	0.104	0.007	208	107	3.703
HPAM	0KG	12	0.1022	0.108	0.104	0.004	192	103	2.004
HPAM	RA5	10	0.1735	0.108	0.104	0.004	208	98	2.008
Ax1	0HE	13	0.0116	0.111	0.104	0.007	194	107	3.605
Ax1	0KG	13	0.0468	0.109	0.104	0.005	190	103	2.495
Ax1	RA5	12	0.0915	0.108	0.104	0.004	194	98	1.952

Table A.18: Vector Product Results for 16-bit inputs with one multiplier

Mult	Adder	Clock (ns)	Relative Error	Total Power (W)	Static (W)	Dynamic (W)	LUT	FF	Dynamic*(LUT+3FF)
Exact	Exact	13	0.0000	0.12	0.105	0.015	351	131	11.16
1UG	RA5	11	0.0319	0.108	0.104	0.004	138	87	1.596
2KD	RA5	13	0.0006	0.124	0.105	0.019	363	99	12.54
5A2	RA5	13	0.0018	0.116	0.105	0.011	252	97	5.973
HDT	RA5	7	0.1236	0.106	0.104	0.002	98	77	0.658
Ax1233	RA5	13	0.0042	0.116	0.105	0.011	318	99	6.765

Table A.19: Vector Product Results for 16-bit inputs with two multipliers

Mult	Adder	Clock (ns)	Relative Error	Total Power (W)	Static (W)	Dynamic (W)	LUT	FF	Dynamic*(LUT+3FF)
Exact	Exact	16	0.0000	0.145	0.105	0.04	682	178	48.64
1UG	RA5	14	0.0319	0.11	0.104	0.006	248	122	3.684
2KD	RA5	16	0.0003	0.131	0.105	0.026	641	146	28.054
5A2	RA5	14	0.0017	0.12	0.105	0.015	473	142	13.485
HDT	RA5	10	0.1237	0.107	0.104	0.003	158	104	1.41
Ax1233	RA5	16	0.0041	0.121	0.105	0.016	598	146	16.576