Approximate Computing: Contributions to the Design of Arithmetic Circuits and Instruction-Set Architectures

Daniela Luiza Catelan

Advisor: Dr. Ricardo Ribeiro dos Santos

Federal University of Mato Grosso do Sul - UFMS
College of Computing - FACOM
January 15, 2025

Approximate Computing: Contributions to the Design of Arithmetic Circuits and Instruction-Set Architectures

Daniela Luiza Catelan

Advisor: Dr. Ricardo Ribeiro dos Santos

A thesis presented to the Ph.D. Program in Computer Science at the College of Computing of the Federal University of Mato Grosso do Sul, as a partial requirement for attaining the Doctorate degree.

Federal University of Mato Grosso do Sul - UFMS
College of Computing - FACOM
January 15, 2025

Abstract

The growing demand for computing power, coupled with the limitations of the end of the Dennard scale, has challenged designers to find alternative solutions to maintain performance within energy and cost limits. Approximate computing (AC) has emerged as a promising approach to balance performance and energy efficiency in error-tolerant applications. However, many AC techniques focus on specific problems or require much intervention from the programmer. This work identified gaps that were transformed into research opportunities. One is related to approximate arithmetic circuits, which focus on single-bit operations, limiting the analysis of these circuits' physical behavior, accuracy, and performance on real platforms with larger inputs and outputs. There are also limitations in the loop perforation technique since once the perforation degree (*pd*) is established, the application metrics will improve only at the cost of accuracy. Adopting a strategy in which *pd* can use approximate hardware resources would overcome this limitation, and greater flexibility would be obtained without forcing additional compilation steps. There is little exploration of the use of approximate instructions, especially in the context of floating point operations, leaving an implementation gap that can be solved by introducing an additional level of approximation, replacing precise (non-approximate) instructions with approximate instructions, thus offering a hardware-level approximate technique over a source code that is already (or not) approximated by a software-level technique. Thus, this work aims at design space exploration (DSE) at different levels of abstraction, investigating the impact of AC on approximate arithmetic circuits, approximate instructions, loop perforation techniques, and approximate mathematical functions. In addition, extensions of the RISC-V architecture instruction set with support for AC are designed. AC techniques were integrated into a widely used platforms, such as SPIKE and ACCEPT, to provide a flexible and efficient infrastructure for developing of approximate systems. The results demonstrate that AC can significantly improve performance and energy efficiency without substantially compromising the accuracy of the systems. This work contributes to new approximate instructions, arithmetic circuits, and an energy model for approximate instructions. It explores the feasibility of these techniques in mathematical functions and control structures (loop) in applications that demand high performance but tolerate controlled errors.

Keywords: approximate computing, approximate arithmetic circuits, approximate instructions, loop perforation, approximate mathematical functions

Resumo

O crescimento da demanda por poder computacional, aliado às limitações do fim da escala de Dennard, desafiou os projetistas a encontrar soluções alternativas para manter o desempenho dentro dos limites de energia e custo. A computação aproximada (CA) emerge como uma abordagem promissora para equilibrar desempenho e eficiência energética em aplicações que toleram erros. No entanto, muitas técnicas de CA focam em problemas específicos ou exigem muita intervenção do programador. Este trabalho identificou lacunas que foram transformadas em oportunidades de pesquisa. Uma delas está relacionada aos circuitos aritméticos aproximados, que focam em operações de um único bit, limitando a análise do comportamento fisico, da precisão e do desempenho desses circuitos em plataformas reais com entradas e saídas maiores. Há também limitações na técnica de perfuração de loops, visto que uma vez que o grau de perfuração (*pd*) é estabelecido, as métricas do aplicativo melhorarão apenas ao custo da precisão. Ao adotar uma estratégia em que o *pd* possa usar recursos de hardware aproximado, esta limitação seria mitigada, além de obter uma maior flexibilidade sem forçar nenhuma etapa de compilação adicional. Há pouca exploração sobre o uso de instruções aproximadas, especialmente no contexto de operações de ponto flutuante, deixando uma lacuna de implementação, que poder ser resolvida com a introdução de um nível adicional de aproximação, substituindo instruções precisas (não aproximadas) por instruções aproximadas, oferecendo desta forma, uma técnica aproximada de nível de hardware sobre um código-fonte que já é (ou não) aproximado por uma técnica de nível de software. Desta forma, este trabalho tem por objetivo a exploração do espaço de projeto (DSE) em diferentes níveis de abstração, investigando o impacto da CA em circuitos aritméticos aproximados, instruções aproximadas e técnicas de perfuração de loops e de funções matemáticas aproximadas. Além disso, são projetadas extensões do conjunto de instruções da arquitetura RISC-V com suporte à CA. A integração de técnicas de CA foi realizada em plataformas amplamente utilizadas, como SPIKE e ACCEPT, para proporcionar uma infraestrutura flexível e eficiente no desenvolvimento de sistemas aproximados. Os resultados demonstram que a CA pode melhorar significativamente o desempenho e a eficiência energética, sem comprometer substancialmente a precisão dos sistemas. Este trabalho contribui com novas instruções aproximadas, circuitos aritméticos e um modelo energético para instruções aproximadas, além de explorar a viabilidade dessas técnicas em funções matemáticas e estruturas de controle (*loop*) em aplicações que exigem alto desempenho, mas que toleram erros controlados.

Palavras-Chave: computação aproximada, circuitos aritméticos aproximados, instruções aproximadas, loop perforation, funções matemáticas aproximadas

Acknowledgements

I could not begin my thanks without first thanking God. There were countless challenges during my doctorate, but God was always with me, blessing me and covering me with wisdom to overcome each obstacle that appeared.

I thank my children, José Luiz and Maria Luiza, for understanding my moments of absence, lousy mood, and impatience. But I am even more grateful for their encouragement, support, and strength.

I am immensely grateful to my parents, Luiz and Neuza, who were my support network. They found a way to be there even when busy with their tasks, making the path easier.

From the bottom of my heart, I thank Professor Ricardo for his excellent guidance, the weekly meetings that kept me focused, and his patience and dedication to understanding the delays in completing the scheduled activities and for never giving up on guiding me.

I must thank my lab colleagues, such as Samuel, who played a valuable role in part of my project, and Guilherme Gloriano, for the chats to relax and share the worries of doctoral students and for the countless help with Overleaf.

I would also like to thank Professor Liana for her participation and support throughout the entire process of my doctorate.

Finally, so as not to run the risk of forgetting anyone and before the tears flow, I would like to thank all those who, directly or indirectly, were with me on this journey.

Contents

List of Figures

List of Tables

List of Acronyms

AC - Approximate Computing

ALE - Total Logic Elements Approximate

AMA - Approximate Mirror Adder

ANN - Artificial Neural Networks

AO - Approximate Output

AP - Power Approximate Circuits

APDr - Approximate Divider

APE - Approximate Processing Elements

APSC - Approximate Subtractor

AS - ACCEPT-SPIKE

ASIC - Application Specific Integrated Circuits

AXA - Approximate XOR/XNOR - based Adder

AXSC - Approximate Subtractor Cell

BL - Baseline

BO - Output Baseline

CAPE - Completely Accurate Processing Element

CF - Correction Factor

CPU - Central Process Unit

DCT - Discrete Cosine Transform

DP - Dynamic Power

DRAM - Dynamic Random Access Memory

DSE - Design Space Exploration

ED - Error Distance

EDP - Energy Delay Product

ELE - Total Logic Elements Accurate

EO - Accurate Circuit Outputs

EP - Power Accurate Circuits

ER - Error Rate

ESA - Equal Segmentation Adder

FA - Full Adder

FAl - FastApprox library

FP - Floating Point

FPGA - Field Programmable Gate Array

GPU - Graphics Processing Unit

HW/SW - Hardware-Software Interface

I - Innermost

$\mathbf{I_{Approx}}$ - Floating Point Approximate Instructions

IC - Inexact Computation

ILAF - Instruction-Level Approximate Functions

ILLP - Instruction-Level Loop Perforation

InXA - Inexact Adder Cells

IOP - I/O Power

$\mathbf{I_R}$ - Replaced Instructions

ISA - Instruction Set Architecture

LI - Loop Interpolation

LOA - Lower Part or Adder

LP - Loop Perforation

LSB - Least Significant Bit

MAPE - Mixed Accuracy Processing Elements

MLC - Multi-Level Cell Model

MRED - Mean Relative Error Distance

NED - Normalized Error Distance

NN - Nearest Neighbor

O - Outermost

PD - Power Dissipation

$\mathbf{P_{instr}}$ - Percentage Instruction Group

$\mathbf{Power_{ap}}$ - Power Instruction Approximate

$\mathbf{Power_{Diff}}$ - Difference in Percentage

$\mathbf{Power_{imp}}$ - Power Improvement

$\mathbf{Power_{nap}}$ - Non-approximated Instructions

Q - Quotient

QUORA - Quality Programmable Vector Processors for Approximate Computing

R - Remainder

RA - Relative Area

rd - Destination Registers

RE - Relative Error

REACT - Rapid Exploration of Approximate Computing Techniques

REp - Relative Error Percentage

RISC-V - Reduced Instruction Set Computer V

rm - Rounding Mode

RP - Relative Power

$\mathbf{RP_{circuit}}$ - Relative Power Approximate Circuit

$\mathbf{RP_{prog}}$ - Relative Power per Program

rs - Source Registers

SDRAM - Synchronous Dynamic Random Access Memory

SoC - System-on-chip

SP - Static Power

SRAM - Static Random Access Memory

TA - Fully Approximate Circuits

TP - Total Power Dissipation

$\mathbf{T_I}$ - Total of Instructions

$\mathbf{T_{ILAF}}$ - Total ILAF Approximate Instructions

VHDL - Very High Speed Integrated Circuits Language

Chapter 1

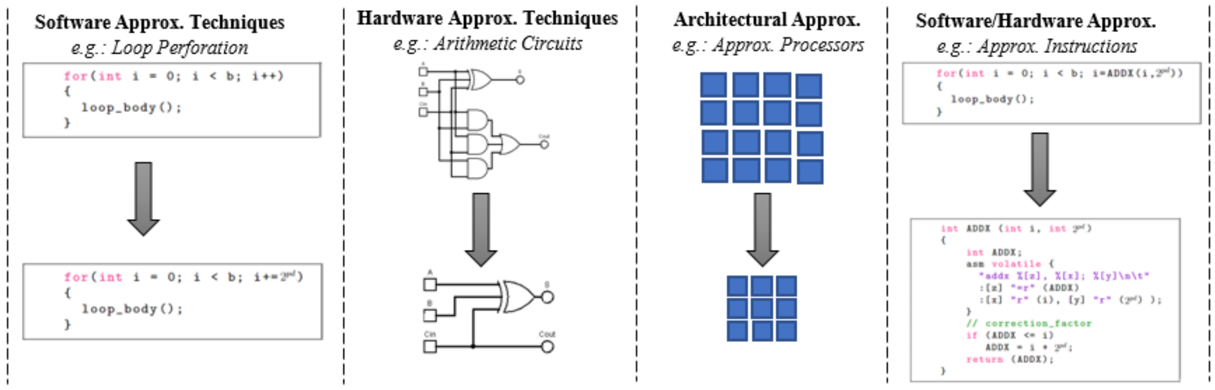
Introduction

The exponential growth in computing power demand and the limitations imposed by the end of the Dennard scale [16], have challenged designers to find alternative solutions to maintain system performance within limited energy and cost budgets. Precise computing requires high resources to ensure the accuracy of results. However, many applications, such as image processing, data mining, and machine learning, can tolerate a certain degree of inaccuracy in calculations without significantly compromising the quality of the final results.

In this context, approximate computing (AC) emerges as a promising approach to balance the need for performance and energy efficiency in applications that tolerate acceptable error margins. Approximate computing encompasses a range of techniques that vary from the circuit level to the application level, enabling the development of hardware and software that perform calculations with reduced precision but with significant gains in energy consumption and performance [79].

Figure 1.1 presents examples of approximate computing approaches. In software, the Loop Perforation technique modifies the loops increment, reducing runtime. In hardware, approximate circuits minimize the area usage, and potentially, the power consumption, by simplifying the number of logical components. At the architectural level, approximations allow the creation of approximate cores. Software/Hardware approximation techniques can be exemplified combining approximate instructions and software techniques offers performance/power consumption gains.

At the circuit level, several proposals for approximate arithmetic circuits, such as adders, subtractors, multipliers, and dividers [30, 24, 31], have been widely studied. These circuits aim to reduce the circuit's area, energy consumption, and delay time at the cost of lower calculation precision. Most current proposals focus on small-scale circuits with single-bit operations, needing deeper analyses of the behavior of such circuits in real platforms and with larger inputs. Furthermore, when evaluating small circuits, these works do not have the opportunity to analyze the circuit accuracy and physical behavior in the presence of long inputs and outputs or even on a real-world design platform.


Software Approx. Techniques
e.g.: Loop Perforation
Hardware Approx. Techniques
e.g.: Arithmetic Circuits
Architectural Approx.
e.g.: Approx. Processors
Software/Hardware Approx.
e.g.: Approx. Instructions

```
for(int i = 0; i < b; i++)
{
  loop_body();
}
```

```
for(int i = 0; i < b; i+=2nd)
{
  loop_body();
}
```

```
for(int i = 0; i < b; i=ADDX(i,2nd))
{
  loop_body();
}
```

```
int ADDX (int i, int 2nd)
{
    int ADDX;
    asm volatile {
      "addx %[z], %[x]; %[y]\n\t"
      :[z] "=r" (ADDX)
      :[x] "r" (i), [y] "r" (2nd) );
    }
    // correction_factor
    if (ADDX <= i)
      ADDX = i + 2nd;
    return (ADDX);
}
```

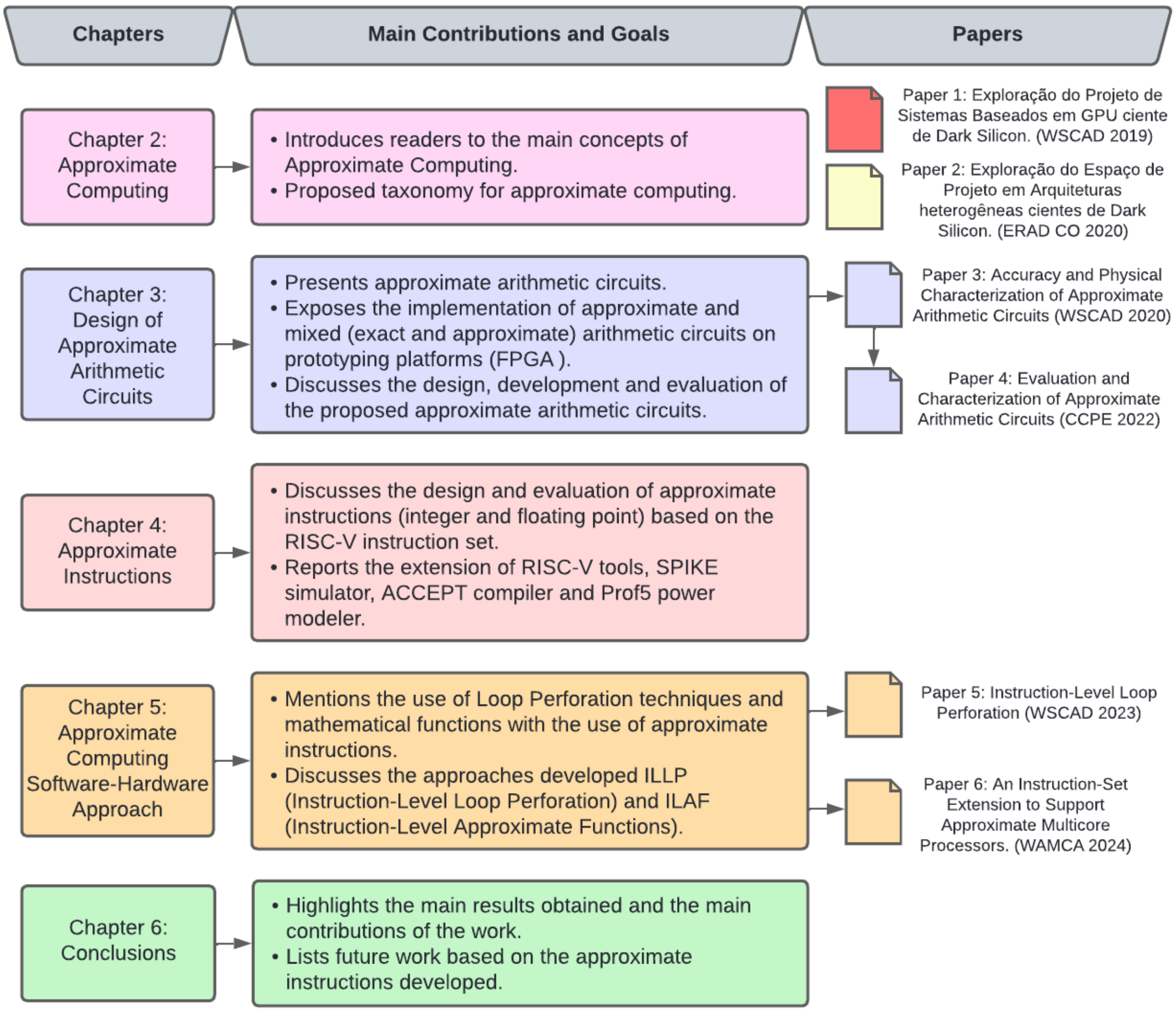
Figure 1.1: Examples of approximate computing techniques: software, hardware, architecture, and software/hardware. Source: author.

On the other hand, at the software level, most AC techniques aim to solve specific problems or require intervention from the programmer, who should identify which application parts are susceptible to approximations [53]. AC techniques also target identifying parts of an application that can be approximated, allowing for resource savings while maintaining the application's proper functionality.

One opportunity to advance in approximate computing lies in introducing an additional level of approximation with the combined usage of hardware and software techniques. From this perspective, this thesis presents a set of studies and approaches for exploiting hardware/software approximate techniques. Specifically, this research work has investigated and designed a range of approximate arithmetic circuits to work as building blocks of new instructions. These instructions show their applicability in different software techniques such as loop perforation and approximate mathematical functions. To carry out this research, we have extended a toolchain based on the RISC-V processor architecture [56]. Concerning the research goals, this thesis has the following:

- Explore the impact of approximate computing at different levels of abstraction, focusing on the implications of accuracy, power consumption, and performance;
- Investigate and propose innovative approximate computing techniques at different levels of abstraction (software and hardware);
- Integrate AC techniques into widely adopted platforms, such as the RISC-V architecture, providing an infrastructure for developing approximate systems;
- Investigate the feasibility of approximation techniques in mathematical functions and control structures, such as loops, for applications that require high performance but tolerate controlled errors;
- Analyze the impact of approximate computing techniques in real applications, seeking a balance between efficiency and precision.

This manuscript is structured as illustrated in Figure 1.2, which presents the published papers as outcomes of each research step. The papers present the physical characterization and evaluation of approximate arithmetic circuits [7, 8], the instruction-level approach to loop perforation [9], and the instruction-level approach to approximate mathematical functions [10]. Other publications carried out along this work present the exploration of dark silicon-aware GPU-based system designs [62] and design space exploration in dark-silicon-aware heterogeneous architectures [6].



Chapters
Main Contributions and Goals
Papers
Chapter 2: Approximate Computing
• Introduces readers to the main concepts of Approximate Computing.
• Proposed taxonomy for approximate computing.
Paper 1: Exploração do Projeto de Sistemas Baseados em GPU ciente de Dark Silicon. (WSCAD 2019)
Paper 2: Exploração do Espaço de Projeto em Arquiteturas heterogêneas cientes de Dark Silicon. (ERAD CO 2020)
Chapter 3: Design of Approximate Arithmetic Circuits
• Presents approximate arithmetic circuits.
• Exposes the implementation of approximate and mixed (exact and approximate) arithmetic circuits on prototyping platforms (FPGA).
• Discusses the design, development and evaluation of the proposed approximate arithmetic circuits.
Paper 3: Accuracy and Physical Characterization of Approximate Arithmetic Circuits (WSCAD 2020)
Paper 4: Evaluation and Characterization of Approximate Arithmetic Circuits (CCPE 2022)
Chapter 4: Approximate Instructions
• Discusses the design and evaluation of approximate instructions (integer and floating point) based on the RISC-V instruction set.
• Reports the extension of RISC-V tools, SPIKE simulator, ACCEPT compiler and Prof5 power modeler.
Chapter 5: Approximate Computing Software-Hardware Approach
• Mentions the use of Loop Perforation techniques and mathematical functions with the use of approximate instructions.
• Discusses the approaches developed ILLP (Instruction-Level Loop Perforation) and ILAF (Instruction-Level Approximate Functions).
Paper 5: Instruction-Level Loop Perforation (WSCAD 2023)
Paper 6: An Instruction-Set Extension to Support Approximate Multicore Processors. (WAMCA 2024)
Chapter 6: Conclusions
• Highlights the main results obtained and the main contributions of the work.
• Lists future work based on the approximate instructions developed.

Figure 1.2: Overall structural diagram of the thesis chapters. Source: author.

Chapter 2

Approximate Computing

Approximate computing offers techniques ranging from the circuit level to the application level, providing benefits on energy efficiency, faster runtime, and even circuits' area shrinking. This chapter introduces AC concepts and techniques, classifying them into physical, logical, and architectural levels. This chapter also highlights the need to balance precision and performance when adopting AC techniques.

2.1 Contextualization

Approximate computing is an emerging paradigm that proposes the introduction of insignificant and controlled inaccuracies so that significant savings can be achieved in design metrics such as execution time, design area, and energy efficiency [14]. According to Palem and Lingamneni [49], the first idea of approximate computing was used in approximate signal processing, where approximation was approached to design systems with limited resources in the areas of artificial intelligence and real-time computing. The second idea came from algorithmic noise tolerance, allowing circuits to be unreliable in the first instance but to use algorithmic error control schemes based on system statistics and input and output behavior to fix resulting errors. From these ideas, inaccurate design began to be applied at the physical design level, logical design, and architectural layers. Many computing processes and/or tasks may require little precision in their results. Some applications of machine learning, signal processing, database analysis, computer vision, and natural language processing do not require an exact answer but a "good enough" result for their purpose [2].

AC provides greater freedom to explore the design space, as it allows compromising computational precision to obtain new regions of performance that were not possible in the traditional way. However, the challenge is the difficulty of determining how safe an approach is for a system or application and predicting how it behaves in synergy with other systems participating in the collaboration. Although promising, effective use of

AC requires carefully selecting the parts of the system (circuit, hardware, or software) that can be approximated and the approximation strategies. Inappropriate use of AC may lead to an unacceptable quality loss.

The related research presents different AC metrics, such as minimum acceptable precision, amplitude precision, information precision, average error distance, normalized error distance, error rate, error significance, maximum error, and magnitude, among others [63, 20, 79]. Most metrics compare some output items in the approximate calculation with the exact calculation.

AC presents differs characteristic terminology, such as the term *error*, which indicates that the result with AC from the result without AC, differentiating itself from the meaning of *fail* in which it is commonly used, when referring to the fact that the result did not come out as expected. Another characteristic term is *accuracy*, a crucial concept that refers to the distance between the approximate and exact result (this distance is calculated using error metrics, such as Relative Error). The terms *exact/accurate/non-approximate* refer to an application or result without approximation techniques, generally used as a comparison reference for results with approximate techniques.

2.2 Organization of AC Techniques

Many research works focus on organizing the set of AC techniques into specific categories. Palem and Lingamneni [49] divide AC into Physical, Logical, and Architecture (Figure 2.1). The physical category concerns methods that change the operating conditions of a circuit, such as voltage and energy consumption. At the logical category, methods to change the logical functions of a computational block, such as an adder or multiplier, are applied to reduce the complexity of the circuit, area, or energy consumption. Finally, the architectural category refers to methods that somehow change the architecture of a system, removing components and/or connections to save resources.

Shafique et al. [63] classify AC approaches into Software, Hardware/Circuits, and Architecture (Figure 2.2). The software category approach encompasses manual and automatic annotation techniques to skip code snippets and parallels and exchange precise modules for approximate ones. Despite recognizing the potential to use multiple techniques simultaneously at different categories, the work maintains its classification into only three main categories. Approximations at the hardware/circuit category include designing approximate versions of logic circuits using algorithms or designing arithmetic datapaths with approximate adders or multipliers, supply voltage reduction techniques, minimize the number of transistors, and reduction in memory units. Approximations at the architectural category can be made with complex algorithms using Artificial Neural Networks (ANNs) to identify critical neurons with approximate instructions and truncation of critical paths.

Leon et al. [32] proposed a classification of AC approaches divided into software,

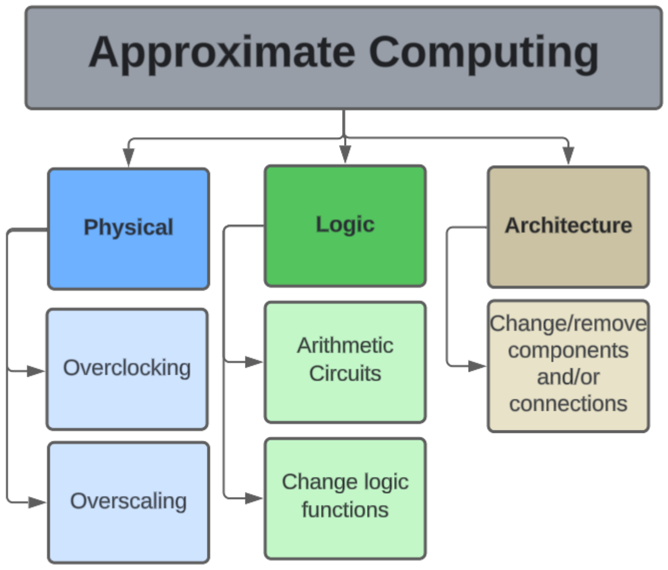


Figure 2.1: Classification of AC proposed by Palem and Lingamneni. Source: [49].

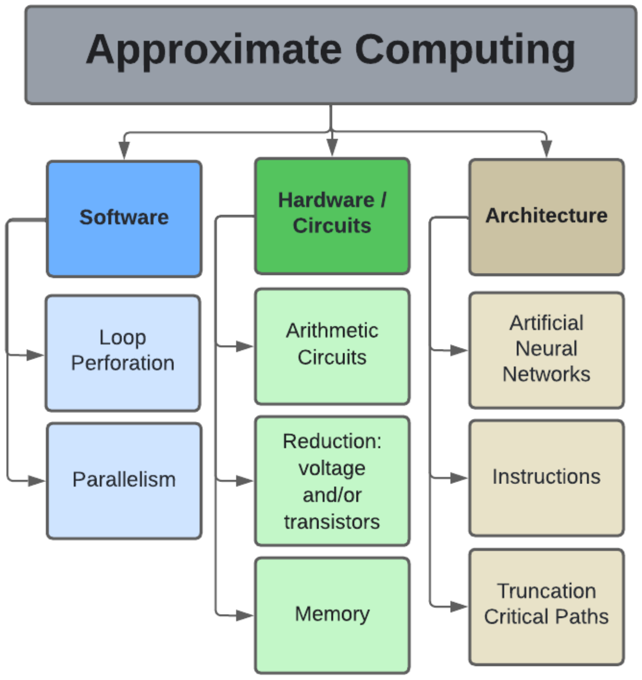


Figure 2.2: Classification of AC techniques proposed by Shafique et al. Source: [63].

hardware, and architecture, considering the techniques of approximate programming languages, loop perforation, and memory access skipping as approximate software methods. Approximate hardware comprises arithmetic circuits (addition, multiplication, and division) and approximate synthesis (automatic approach to generating inexact circuits). Approximate architecture are approximate processors and approximate data storage (Figure 2.3). Xu et. al [79] use the same approach to organize AC techniques but add Instruction Set Architecture (ISA) extensions to the architecture category.

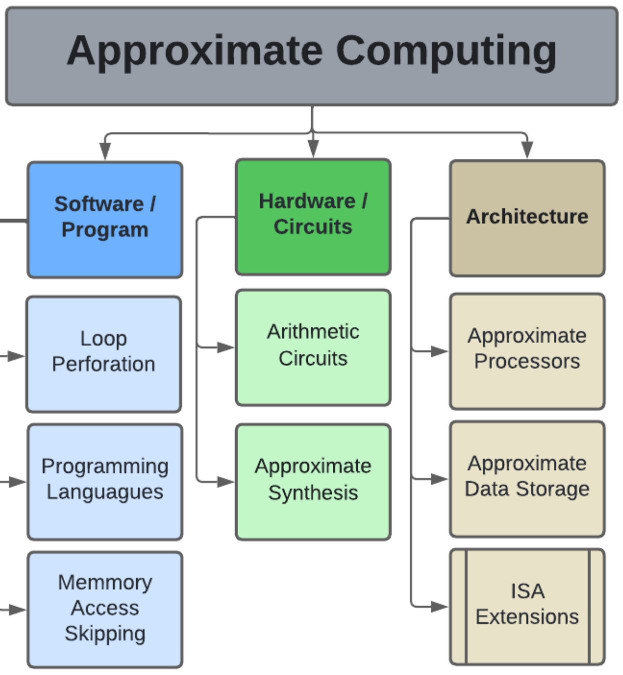


Figure 2.3: Classification of AC techniques proposed by Vasileios et al. and Xu et al. Source: [32, 79].

The scientific work proposing taxonomies for AC, classifies approximate computation reflecting the technological developments of their time and specific contexts [42, 27, 2]. However, as one may notice, the authors adopt the same standard categories (software, hardware, architecture) to organize the techniques. Classifications are expected to cover as many techniques as possible in the least general way; after all, classifying with hardware and software is already a common classification. The classifications should evolve as new challenges and solutions emerge, reflecting the diversity of applications and the interdisciplinary nature of approximate computing. For example, the AC approaches presented in [59, 21, 73] whose proposals simultaneously employ multiple techniques. Regarding new AC techniques that may find difficult to fit into the previous classifications, Figure 2.4 presents a new extended taxonomy for AC techniques. The new techniques are represented in double borders. The novelty is on the Software/Hardware category, which covers software techniques relying on approximate hardware.

The Loop Perforation (LP) technique is widely used in AC, since any modification in the loop increment already makes it approximate. Several software techniques are employed for this purpose; however, the use of hardware techniques to complement LP is still in the development phase. Catelan et al. [9], who exploit approximate instructions in hardware to modify the LP increment. Similarly, approximate mathematical functions have several software-based approximation techniques, but also allow the replacement of exact instructions by approximate instructions in hardware, as demonstrated by Catelan et al. [10].

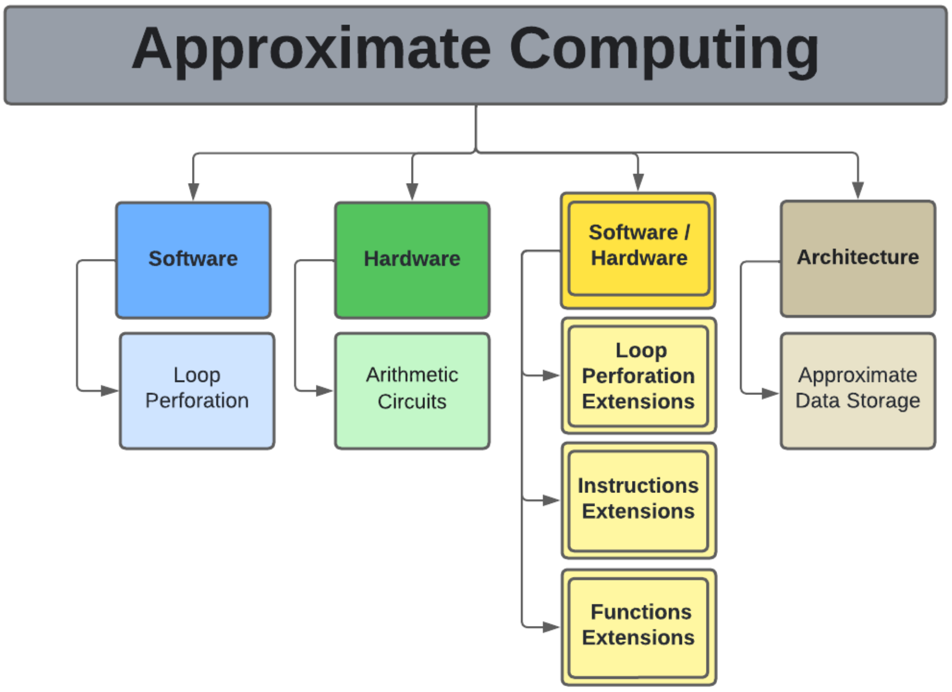


Figure 2.4: Proposed taxonomy for approximate computing. Source: author.

2.3 Software Support for Approximate Computing

The objective of software approximate computing techniques is to improve programs' runtime and/or energy consumption. Typical software techniques are approximated libraries/frameworks, compiler extensions, precision tuning tools, runtime systems, and language annotations [32].

REACT (Rapid Exploration of Approximate Computing Techniques) [77] is a modeling framework that allows researchers to rapidly evaluate approximate computing techniques for achieving energy efficiency and modeling through user-initiated error injection. Techniques that REACT can evaluate include data approximation, which introduces controlled errors to reduce energy consumption; operations approximation, which uses simplified arithmetic operations; and circuit approximation, where deliberate faults are introduced to reduce area or energy consumption; control approximation, which modifies the program control flow to avoid complex computations; and error modeling, which allows controlled injection of errors to study the impact of approximations on the quality of results. REACT employs a custom linear energy model to estimate energy savings. It allows accurate and rapid exploration of the energy-quality trade-off space of approximate computing. The paper lists eight approximate computing techniques, such as DRAM Refresh Rate, Neural Acceleration, and Precision Scaling. The tests were performed using applications with varying AC techniques. The results show an average error of 0.87%, ranging from 0.31% to 1.38%, depending on the AC technique and the application under analysis.

FlexJava [51] features a small set of extensions that reduce annotation effort for approximate programming, allowing programmers to annotate outputs that exhibit approximation tolerance. Programmers need to manually write down all approximate variables, statements, and safe operations to approximate. The FlexJava compiler, equipped with an approximation safety analysis, automatically infers operations and data that affect these outputs and selectively marks them as approximations, giving safety guarantees. The results achieved energy reduction when exchanging precise computation for approximate computation. FlexJava results are based on comparisons to EnerJ [59]. FlexJava reduced the number of annotations between 2 and 17× and a reduction in annotation time between 6 and 12×, keeping the same level of energy savings.

AppSyn [4] is a synthesizer program that can take specific input, an input-output set, or a complete implementation reference. AppSyn searches among the implementations to find a program that meets a given specification. The programmer only needs to provide a reference and a desired precision value. The synthesizer will try to find an approximate implementation that satisfies the given limits. The synthesizer also validates whether the approximate program result is more efficient than the reference. The output is the approximate implementation, and a table with speedup results.

ACCEPT [58] compiler introduces specific annotations to show the programmer which code regions are more suitable for approximation and which techniques would be most promising. The ACCEPT offers a set of techniques to users interested in applying approximations in a program: loop perforation, loop parallelization, neural acceleration, and approximate functions. The compiler evaluates the approximation techniques in the code and presents performance and accuracy results, thus allowing the programmer to analyze the impact of the approximations.

Loop Perforation (LP) is an AC software technique that has been gaining appreciation among designers because it is simple, general purpose, and is widely applicable. LP consists of skipping loop iterations to reduce computational workload and gain performance. A simple change in the loop step variable is enough to change its performance, but manually changing one or more loops of an application sometimes becomes more costly than the loop execution itself. ACCEPT compiler automatizes the process to find suitable loops for perforation. The tool considers only canonical loops fitted for perforation, which have a precise body without early exits and conditionals. The ACCEPT divides the loop into three blocks: the header, the body, and the latch. Loop calculations are performed inside the body, and the header and latch are responsible for changing condition variables and checking jump conditions. Other techniques are also applied to find loops for perforation, such as: perforation space exploration algorithm [64], selective dynamic [34], time redundancy [44], polyhedral compilation [3], Linear Interpolation (LI), and LP with Nearest Neighbor (NN) [57].

Most software techniques found in the literature seek to find code sections that can be automatically [59, 58] or manually [4] carried out inserting approximate code, variables, loops, and functions. There are also benchmarks capable of approximate computing to ensure fairer and more reproducible comparisons. AxBench [81] has a representative set of applications for a fair evaluation of different approximation techniques. It contains applications for Central Process Unit (CPU) (C/C++), Graphics Processing Unit (GPU) (CUDA), and hardware (Verilog). Table 2.1 presents 21 applications, platforms, domains, and metrics from AxBench.

Table 2.1: Applications, platforms, domains, and metrics of AxBench. Source: [81].

Applications	**Platforms**	**Domains**	**Metrics**
binarization	GPU	Image Processing	Image Difference
blackscholes	CPU, GPU	Finance	Average Relative Error
brent-kung	Hardware	Computational Arithmetic	Average Relative Error
cannel	CPU	Computational Arithmetic	Average Relative Error
convolution	GPU	Machine Learning	Average Relative Error
fastwalsh	GPU	Signal Processing	Image Difference
fft	CPU	Signal Processing	Average Relative Error
fir	Hardware	Signal Processing	Average Relative Error
forwardk2j	CPU, Hardware	Robotics	Average Relative Error
inversek2j	CPU, GPU, Hardware	Robotics	Average Relative Error
jmeint	CPU, GPU	Games 3D	Loss Rate
jpeg	CPU	Image Processing	Image Difference
kmeans	CPU, Hardware	Machine Learning	Image Difference
kogge-stone	Hardware	Computational Arithmetic	Average Relative Error
laplacian	GPU	Image Processing	Image Difference
meanfilter	GPU	Machine Vision	Image Difference
neural network	Hardware	Machine Learning	Average Relative Error
newton-raph	GPU	Numerical Analysis	Average Relative Error
sobel	CPU, GPU, Hardware	Image Processing	Image Difference
srad	GPU	Medical Images	Image Difference
wallace-tree	Hardware	Computational Arithmetic	Average Relative Error

2.4 Hardware Support for Approximate Computing

One of the AC hardware techniques consists of modifying logical functions to reduce the circuits' complexity and thus reduce area and energy consumption. It is essential to highlight that errors will be introduced into the circuit when modifying logical functions. However, these errors occur in just a few combinations, and the designer considers reducing complexity or energy consumption favorable. In that case, these circuits will be advantageous for approximate applications [74].

Various works propose different techniques (voltage over-scaling, speculative carry select addition, and memristor) for the development of approximate adders [38, 83, 43, 82, 36, 33, 18]. The commonly used technique is circuit minimization, which reduces the number of logical components in the circuit. Gupta et al. [26] presented an Approximate Mirror Adder (AMA), which, with the removal of transistors and the insertion of minimal errors in the truth table, derives four different circuits for an approximate Full Adder (FA): AMA1, AMA2, AMA3, and AMA4. Compared to other approximate adders, the Inexact Adder Cells (InXA) [1] have fewer transistors on their three terminals, thus occupying a smaller area, in the three designed models, InXA1, InXA2, and InXA3. The Approximate XOR/XNOR-based Adder (AXA) [80] is based on exact adders with XOR and XNOR gates, with a reduction in the number of transistors and the production of three new approximate adders (AXA1, AXA2, and AXA3).

Table 2.2 shows the number of transistors and the number of errors of AXA, AMA, and InXA adders. Note that AXA2, InXA1, and InXA3 adders have the same number of transistors; however, the number of errors is significantly different.

Table 2.2: Comparison among AXA, AMA, and InXA adders. Source: [80, 67].

Adder	Number of Transistors	Number of errors - SUM
AMA1	16	2
AMA2	14	2
AMA3	11	3
AXA1	8	4
AXA2	6	4
AXA3	8	2
InXA1	6	0
InXA3	6	2

Approximate subtractor circuits also use the circuit minimization technique, introducing controlled errors into the truth table and obtaining the logical expression through the Karnaugh Map. Examples of such circuits are the Approximate Subtractor Cell (AXSC) [12] and Approximate Subtractor (APSC) [24] subtractors. The AXSC presents three derivations of approximate subtractors, with the insertion of errors varying between the SUM and Cout outputs for 2 to 4 errors. APSC has four subtractor derivations, with the insertion of 1 error only in the SUM output. Table 2.3 presents the number of errors inserted in each subtractor and the area (μm^2), using a $45nm$ transistor. It is noted that the AXSC2 subtractor presents the largest insertion of errors in its outputs, with the smallest area size. The APSC4 subtractor only presents one error in its output, and the area is just one unit larger than that of the AXSC2.

Table 2.3: Comparison between AXSC and APSC subtractor. Source: [24].

Subtractor	**Number of errors**	**Area (μm^2)**
AXSC1	2	16
AXSC2	4	8
AXSC3	2	14
APSC4	1	9
APSC5	1	15
APSC6	1	19
APSC7	1	17

In multipliers, the insertion of errors in the truth table is also viable. Kulkarni et al. [31] presented a conversion of a precise 2×2 multiplier to an approximate 2×2 multiplier. A single entry in the multiplier truth table was modified. The change reduced the number of logic gates required to implement the multiplier. The proposed approximate multiplier presented an average energy saving between 31.78% and 45.4%, with an average error variation between 1.39% and 3.35%. The average relative error increases with the number of operand bits but saturates between 3.3% and 3.35%.

Another technique used for approximate arithmetic circuits is to replace part of the operation hardware with approximate components. Gorantla and Deepa [24] replaced the subtraction hardware in the divider circuit with the approximate subtractors (APSC), thus obtaining four types of approximate dividers (APDr4-APDr7). These dividers showed low power consumption (APDr4 = 71%), lower delay (APDr6 = 40%) and smaller area (APDr4 = 29%) compared to the exact divider.

EvoApprox8b [45] is an 8-bit approximate adder designed to improve speed and power consumption efficiency, especially in embedded systems and mobile devices. Instead of providing exact results, EvoApprox8b uses hardware approximation techniques to simplify addition operations, which reduces the number of transistors and signal propagation time. This is achieved by modifying the digital circuit architecture, including reducing the complexity of logic gates and eliminating intermediate operations. This approach results in an adder that sacrifices some accuracy in favor of faster performance and lower power consumption. By employing approximation directly at the hardware level, EvoApprox8b provides an efficient solution for applications where absolute accuracy is not critical, but system efficiency is paramount. EvoApprox8b is a benchmark for circuit approximation methods, having a library with 430 approximate adders and 471 approximate multipliers, both 8 bits, with models in Verilog, Matlab, and C of all circuits approximate, in addition to presenting seven different error metrics, area, delay, and power. The use of EvoApprox8b allows the integration of hardware and software design circuits.

2.5 Software/Hardware Support for Approximate Computing

The category of software/hardware support for AC is comprised of frameworks, such as EnerJ [59], RISK-5 [21], and QUORA [73], which have software for control approximation depending on approximate hardware, such as SRAM, DRAM, and operations units (with integers and floating point numbers).

The EnerJ simulator is a Java extension that allows the addition of approximate data types. The model proposes using qualifiers to declare data that can be subject to approximate calculation. EnerJ automatically maps approximate variables to low-power storage and utilizes low-power operations. Figure 2.5 presents the hardware model assumed in the EnerJ system, where the shaded areas indicate approximate components.

The SRAM cells in the register and data caches reduce the supply voltage. The floating point unit performs approximation using reduced mantissa. The reduction in the update rate makes the main memory (DRAM) approximate.This allows the programmer to explicitly control how information flows from approximate data to precise data.

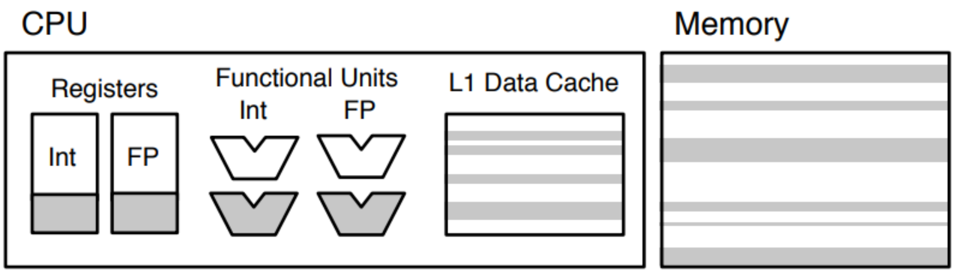


Figure 2.5: EnerJ hardware model. Source: [59].

RISK-5 [21] is an extension of RISC-V ISA for approximate hardware unit control. RISK-5 extends the RISC-V architecture, implementing control mechanisms combining several approximation techniques. The approximate hardware resources are informed to the software, that in turn, has control of what and how much approximation will be used in an application. This control can range from turning approximate hardware on/off to configure allowable error levels and probabilistic error configuration operating parameters, such as the refresh rate for approximate SDRAM. Approximations can be configured dynamically, allowing for simplified design space exploration. Figure 2.6 provides a visual representation of the RISK-5 hardware. The key component is the status register, which is responsible for the approximation control of each unit.

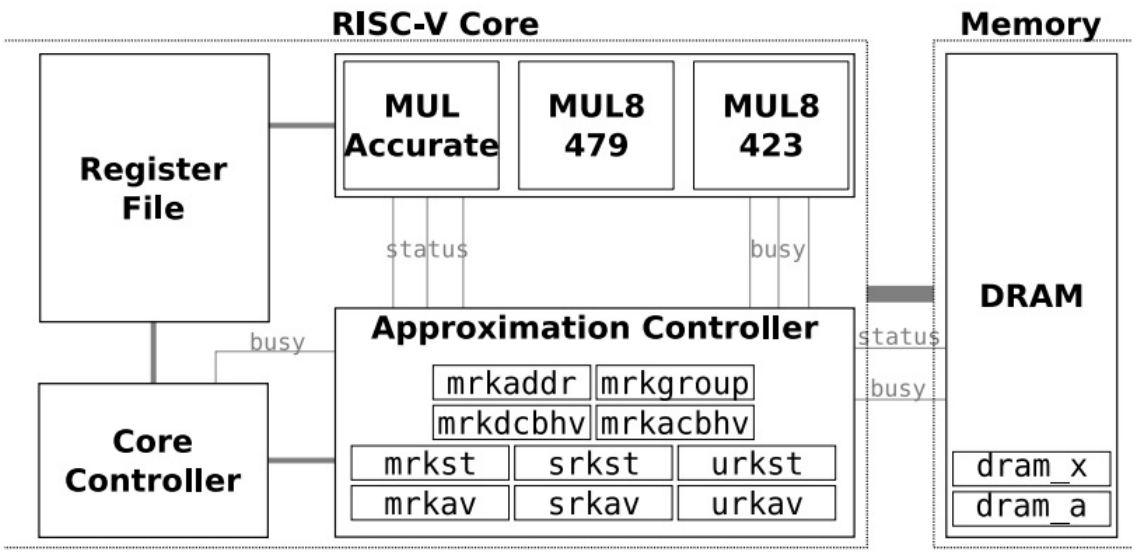


Figure 2.6: RISK-5 hardware schematic. Source: [21].

Quality programmable vector processors for approximate computing (QUORA) [73] is a programmable processor that expresses its tolerance for AC at the hardware-software interface (HW/SW), having as its main components a quality programmable ISA (QP -ISA) and a microarchitecture (QP-uArch). QUORA features precision scaling-based hardware mechanisms with error monitoring and compensation to facilitate configurable quality execution across these processing elements and demonstrate significant energy benefits. The HW/SW interface allows approximate instructions to be identified and specifies the amount of error each instruction can tolerate during its execution. Figure 2.7 presents the QUORA microarchitecture. The QUORA design has three hierarchy levels: APE - Approximate Processing Elements, MAPE - Mixed Accuracy Processing Elements, and CAPE - Completely Accurate Processing Element. These levels provide different trade-offs between power and quality, and the hardware mechanisms are based on precision scaling with monitoring and error compensation.

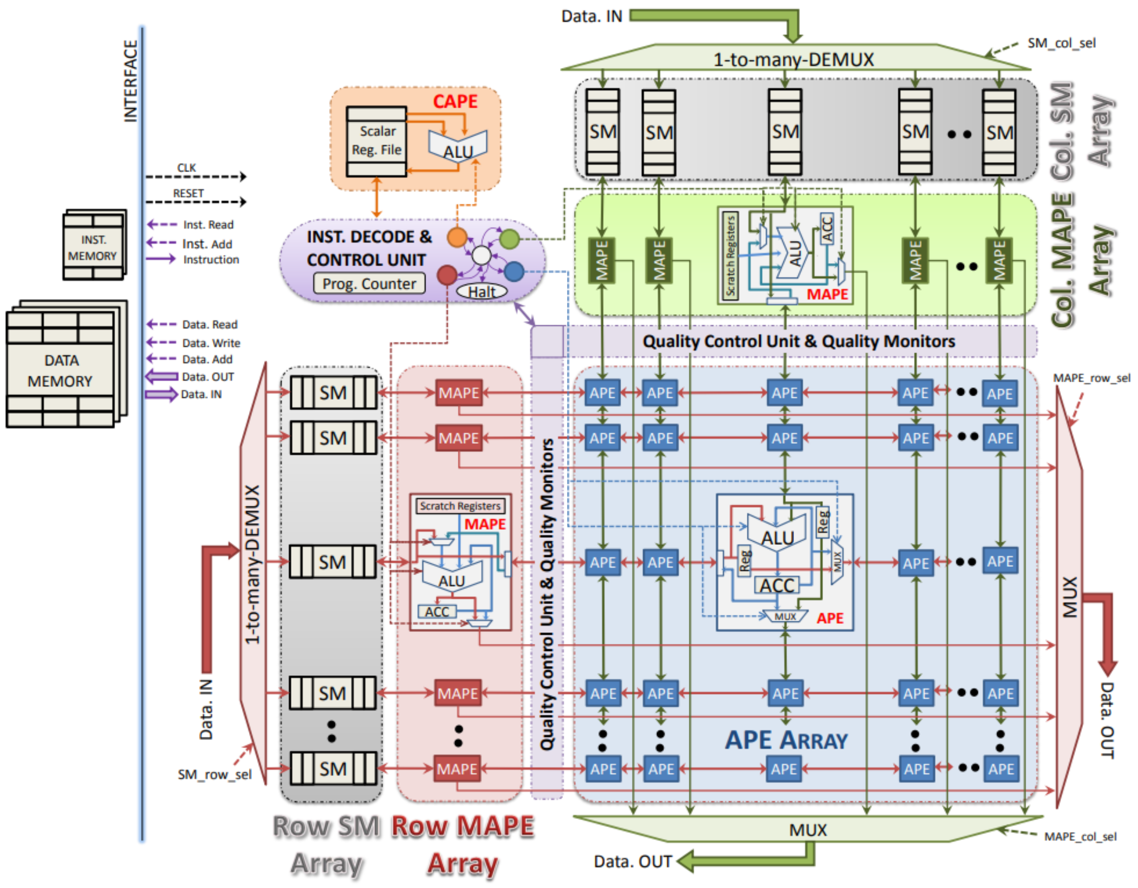


Figure 2.7: QUORA microarchitecture. Source: [73].

2.6 Architectural Support for Approximate Computing

This section presents the architectural support for AC regarding strategies using approximate memories. When an application can tolerate occasional errors in some data, approximate storage is used, and data recovery errors may occur. In either case, software must determine which data can tolerate errors and which needs accuracy. There are several propositions for adopting approximate memories [39]. The Unreliable Retention technique is the most used. This technique is restricted to DRAM and consists of updating at longer intervals than the maximum guaranteed retention of the cells. As a result, the bits that store 1 can lose their stored value due to capacitor discharge.

Sampson et al. [60] propose two approximated memory techniques. The first uses multi-level cells (MLC - Multi-Level Cell Model) to allow greater density or better performance, considering the cost of imprecise errors caused in data recovery. The other technique uses of faulty bit blocks to store approximate data. MLC storage blocks that have faults. When the first uncorrectable block fails, the memory will issue an interrupt and indicate the failed block, which will be used as a temporary storage block. The authors conclude that approximate storage helps mitigate the disadvantages of solid-state systems and non-volatile memories, as there is no significant compromise in storage quality for write acceleration and lifetime extension criteria.

Another approach of approximate memories was demonstrated in [37]. The discrete cosine transform (DCT) was applied to a dimension between the cache and main memory to reduce memory traffic while maintaining acceptable quality degradation using natural sampled data such as sound, sensor data, and images.The DCT performs an orthogonal transformation that preserves energy, and its transformation matrix is orthonormal, thus allowing an approximate DCT matrix to be written. The approximate DCT was used in an approximate memory based on lossy compression, performing transformations of memory blocks. As a result, the authors point out that a DCT of approximately 8 points has better compression performance for approximate memory than a DCT of 16 points.

2.7 Final Remarks

This chapter presented an overview of approximate computing, demonstrating its main techniques and applications in both software and hardware levels. AC is still an underexploited area with diverse research possibilities, given that most existing work is related to specific applications, a limited set of bits, or requires much knowledge from the programmer. Regarding approximate circuits, there is great potential for exploiting mixed exact/approximate circuits that provide a balance between precision and energy efficiency. Another promising field is the development of approximate instruction sets combining integer and approximate floating point instructions, which would allow the execution of complex mathematical operations with less consumption of computational resources. The diversity of AC techniques reflects the interdisciplinary nature of this subject, requiring a constant evolution of classifications and approaches.

Chapter 3

Design of Approximate Arithmetic Circuits

This chapter studies approximate arithmetic circuits. We discuss the design, development, and evaluation of approximate arithmetic circuits developed as part of this thesis. The chapter also presents experiments and results of prototyping those circuits on Field-Programmable Gate Array (FPGA) platforms. The results present the trade-offs between accuracy, resource usage, and energy efficiency.

3.1 Contextualization

The difficulty of accommodating more cores in constrained power budgets threatens even the multicore scaling paradigm. Thus, designers seek new computing sources to deliver better performance with power and cost constraints. The workloads that drive computing demand have also fundamentally changed across the computing spectrum, from mobile devices to the cloud. Such workloads exhibit intrinsic resilience to approximations and an ability to produce acceptable outputs even when some of their calculations are inaccurate [72].

Some authors [27, 49] indicate that AC techniques can be applied at the logical level that cover methods to approximate logic functions in digital circuits. There is a diversity of proposals for approximate circuits [30, 24, 31, 13], mainly for approximate adders, subtractors, multipliers, and dividers. Most of these studies present circuits with just 1-bit, showing details such as error placement, area usage, power, and delay. When evaluating small circuits, these works need more opportunities to analyze how circuit accuracy and physical behavior will perform in the presence of larger inputs and outputs and even in a real-world design platform.

The following sections will present an analysis of the behavior of a wide range of precision-controlled approximate arithmetic circuits. An evaluation of those circuits on flexible prototyping platforms, such as FPGAs, is also presented and discussed.

3.2 Approximate Arithmetic Circuit Designs

There are many proposals of approximate circuit designs [26, 80, 1, 30, 47, 24, 13], mainly focusing on approximate arithmetic circuits such as adders, subtractors, multipliers, and dividers. Some studies present circuits with just 1-bit, evaluating their features regarding error positioning, area usage, power, and delay.

Most research presents only a single circuit to control accuracy. For example, Gorantla and Deepa [24] present four types of subtractors (APSC4-APSC7), where the difference between them is only the position where the error was inserted in the truth table. Although the APSC circuit is visibly larger than the exact subtractor, there are considerable power gains. The APSC4, with $90nm$ transistors, achieves a power of $891nW$ compared to the $3475nW$ of the exact subtractor [24].

Approximate subtractors AXSC1, AXSC2, and AXSC3 are based on XOR and XNOR cells with transistor reduction in each cell. Chen et al. [13] indicated that the precision of the $Cout$ output is more important than the S output. $Cout$ remains unchanged in AXSC1 and AXSC3, and a combination of the outputs S and $Cout$, aiming to reduce the delay, is proposed in AXSC2 and AXSC3. AXSC approximate subtractors have an absolute error distance between 2 and 4. The logic circuits of the AXSC subtractors are smaller than the exact subtractor, which is consistent with the reduction in the size of the area, with $180nm$ transistors of the AXSC2 of $50\mu m^2$ compared to the exact subtractor that has $90\mu m^2$ of area [24].

The approximate adder designs of AMA type (*Approximate Mirror Adder* - AMA) [26] is based on the Mirror Adder. In an AMA design, the transistors of the conventional circuit were removed one by one, always ensuring that for any combination of the inputs A, B and Cin would not result in a short circuit or open circuit. A few errors in the truth table were inserted and a set of approximate adder circuits were derived (AMA1, AMA2, AMA3, and AMA4). However, taking into account the size of the logical expression, only the approximate adders AMA1, AMA3, and AMA4 were used for the tests. Compared to the full adder that has an area of $40.66\mu m^2$, the AMA3 and AMA4 adders have an area, respectively, of $22.56\mu m^2$ and $23.91\mu m^2$ [26].

The AXA adder (*Approximate XOR/XNOR - based Adder*) [80] was designed based on the transistor removal technique. The complete AXA1 adder is an exact adder with XOR gates and its implementation uses 8 transistors. The AXA2 adder is based on XNOR ports requiring 6 transistors. The AXA3 adder was based on the AXA2 adder with two more transistors for better accuracy. The three models were evaluated in this work. Concerning the exact adder, the AXA2 improves 65.45% in static power [80].

The approximate InXA1, InXA2, and InXA3 adder models [1] were developed with inexact adder cells with less circuit complexity compared to other approximate circuits found in the area (AMA, AXA, LOA, and ESA). The InXA1 adder has an approximate design on the *Cout* output. InXA2 and InXA3 have approximate designs on the S output. InXA1 and InXA3 were evaluated in this work. Analyzing delay, energy dissipation, and the energy-delay-product (EDP), the InXA1 and InXA2 adders outperform the AMAs and AXAs adders in all metrics; however, the InXA3, despite dissipating less energy, incurs a higher amount of delay [1].

Table 3.1 presents the logical expressions of the approximate adder circuits. The S output is different among the circuits once it is focused on approximate designs. Regarding the *Cout* output, one may also observe that the AXA1 adder has a more complex logic; AMA1, AMA3, AXA2, and AXA3 have the same expression in the *Cout* output.

Table 3.1: Logical expressions for approximate adders. Source: author.

Circuits	**Output**	**Logical Expression**	**Circuits**	**Output**	**Logical Expression**
AMA1	S	$\overline{A}.\overline{B}.Cin + A.B.Cin$	AXA2	S	$\overline{(A \oplus B)}$
	Cout	$A.Cin + B$		Cout	$(A \oplus B).Cin + A.B$
AMA3	S	$\overline{(A.Cin + B)}$	AXA3	S	$\overline{(A \oplus B)}.Cin$
	Cout	$A.Cin + B$		Cout	$(A \oplus B).Cin + A.B$
AMA4	S	$\overline{A}.Cin + B.Cin$	InXA1	S	$A \oplus B \oplus Cin$
	Cout	A		Cout	Cin
AXA1	S	Cin	InXA3	S	$\overline{Cout}$
	Cout	$\overline{((A \oplus B).Cin + \overline{A}.\overline{B})}$		Cout	$(A \oplus B).Cin + A.B$

Approximate subtractors APSC4-APSC7 have a single error inserted in output S and output *Cout* is the same for all subtractors, so the absolute error distance is 1. Approximate subtractors AXSC1-AXSC3 have the same *Cout* output in AXSC1 and AXSC3, and the same as the S output in AXSC2. Table 3.2 shows the logical expressions of the approximate subtractor circuits.

Table 3.2: Logical expressions for the approximate subtractors. Source: author.

Circuits	**Logical Expression**	**Circuits**	**Logical Expression**
APSC4	$\overline{A}.B + B.Cin + \overline{A}.Cin + A.\overline{B}.\overline{Cin}$	AXSC1	$A \oplus B + Cin$
APSC5	$A.\overline{B} + \overline{B}.Cin + \overline{A}.B.\overline{Cin} + A.Cin$	AXSC2	$A \oplus B \oplus Cin$
APSC6	$A.B.Cin + \overline{A}.B.\overline{Cin} + \overline{A}.\overline{B}.Cin$	AXSC3	$Cout$
APSC7	$A.\overline{B}.\overline{Cin} + \overline{A}.\overline{B}.Cin + A.B.Cin$		

The subtractors APSC4-APSC7, AXSC1, and AXSC3 designs have the same logical expression for the *Cout* output:

$$Cout = \overline{(A \oplus B)}.Cin + \overline{A}.B$$

Kulkarni [31] presents the conversion of a precise 2×2 multiplier to an approximate 2×2 multiplier. A single entry of the multiplier truth table was modified for the conversion, as highlighted (bold) in Table 3.3. The modified entry in the truth table refers to the multiplication of 11_2 by 11_2, whose precise result is 1001_2, and whose approximate result equals 111_2. This change reduced the number of logic gates required to implement the multiplier. The approximate multiplier showed an average energy saving between 31.78% and 45.4%, with the average error variation of 1.39% and 3.35%. The average relative error increases according to the number of operand bits; however, it saturates between 3.3% and 3.35%.

Table 3.3: Approximate 2×2 multiplier truth table. Source: [31].

Input				**Output**		
A1	**A0**	**B1**	**B0**	**Out2**	**Out1**	**Out0**
0	0	0	0	0	0	0
0	0	0	1	0	0	0
0	0	1	0	0	0	0
0	0	1	1	0	0	0
0	1	0	0	0	0	0
0	1	0	1	0	0	1
0	1	1	0	0	1	0
0	1	1	1	0	1	1
1	0	0	0	0	0	0
1	0	0	1	0	1	0
1	0	1	0	1	0	0
1	0	1	1	1	1	0
1	1	0	0	0	0	0
1	1	0	1	0	1	1
1	1	1	0	1	1	0
1	1	1	1	**1**	**1**	**1**

The division process includes subtraction, shifting, and comparison. Gorantla and Deepa [24] replaced the subtraction hardware in the divider circuit with the approximate subtractors (APSCs) they proposed, thus obtaining four types of approximate dividers (APDr4-APDr7). Validation of approximate dividers was implemented in Verilog HDL using transistor libraries with $180nm$, $90nm$, and $45nm$ technologies in the Cadence RTL compiler. The approximate dividers showed low power consumption (APDr4 = 71%), shorter delay (APDr6 = 40%), and smaller area (APDr4 = 29%) compared to the exact divider.

3.3 Experimental Setup, Results, and Discussion

We performed experiments to evaluate precise, approximate, and mixed arithmetic circuits. The approximate arithmetic circuits were built with a reconfigurable FPGA platform and were organized into 4 circuit types: adders, subtractors, multipliers, and dividers with different bit widths (8-, 16-, 32-, and 64-bits). The analyzed circuits were compared in accuracy, area usage, and power dissipation metrics.

The experiments were performed considering precise circuits, approximate circuits, and mixed circuits (precise and approximate). The mixed circuits were designed so that the bit width design was divided into first and second halves. The first half (least significant bits) comprised precise circuits and the second half (most significant bits) of approximate circuits. The results of mixed circuits will be represented with a capital “M” as the first letter.

The experiments used 8 approximate adder circuits and 7 approximate subtractor circuits. The design of the approximate multiplier circuits was built based on the approximate adders as building blocks. Approximate divider circuits were developed based on the approximate subtractors. The complete experimentation with the approximate arithmetic circuits, with detailed information about the procedures used, tools, data for circuit evaluation, complete tables with results, and metrics used, can be found in the paper [7, 8]. This section will present the main results with their respective analyses and discussions.

All circuits were designed in VHDL, synthesized on the Cyclone IV GX FPGA with the ALTERA Quartus II Web Edition 13.1 IDE tool. Simulation scripts and testbenches were simulated using the Altera-ModelSim 10 tool. The area usage results come from the number of FPGA logic elements used by the circuit design. The total power dissipated was obtained by the PowerPlay Power Analyzer tool. PowerPlay estimates the design power dissipation (PD) by the sum of static power (SP), dynamic power (DP), and I/O power (IOP).

Figure 3.1 presents the VHDL code of the exact 1-bit adder. In lines 16 and 17, there are the logical output expressions *Cout* and *S*. Figure 3.2 presents the VHDL code of the approximate adder with the 1-bit InXA3, identified by the entity *Adder_InXA3_1*, where “1” is the number of bits in the adder in question. Lines 16 to 21 express the logical output functions *Cout* and *S*. To change the type of approximate adder, simply replace the code section of these lines with the logical expressions of the adder outputs—desired approximate value, as well as the entity’s name[1].

Figure 3.3 shows the design of the mixed approximate adder with 2 bits InXA3, identified by the entity *Adder_M_InXA3_2*, where “2” refers to 2-bits. Note that the component *Adder_InXA3_1* (line 17) represents the approximate adder and the component *Adder_Accuracy_1* (line 22) is the exact adder. Both make up the structure of the mixed adder. *FA0* (line 28) represents the least significant bit and is composed of the exact adder, while *FA1* (line 30) is the most significant bit and is composed of the InXA3 approximate adder. To increase the number of bits in the mixed adder, increase the number of *FA*. Similarly, the operation of the other proposed approximate arithmetic circuits works.

The accuracy metric is based on the relative error percentage (REp $= 1 - \frac{AO}{EO}$) of the approximate circuits over the accurate circuits. To calculate the relative error percentage, we developed a testbench performing a line-by-line comparison between the

[1]Arithmetic circuits from 1 to 32 bits are available at: https://github.com/danielacatelan/Circuits-Approximate

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity Adder_Accuracy_1 is
port
(
  cin: in std_logic;
  a: in std_logic;
  b: in std_logic;
  cout: out std_logic;
  s: out std_logic
);
end Adder_Accuracy_1;

architecture Adder1Bit of Adder_Accuracy_1 is
begin
  cout <= (a and b) or (a and cin) or (b and cin);
  s <= a xor b xor cin;
end Adder1Bit;
```

Figure 3.1: 1-bit accuracy adder in VHDL code. Source: author.

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity Adder_InXA3_1 is
port
(
  cin: in std_logic;
  a: in std_logic;
  b: in std_logic;
  cout: out std_logic;
  s: out std_logic
);
end Adder_InXA3_1;
architecture Adder1Bit of Adder_InXA3_1 is
  signal c1, c2, c3 : std_logic;
begin
  c1 <= a xor b;
  c2 <= a and b;
  c3 <= c1 and cin;

  cout <= c2 or c3;
  s <= not (c2 or c3);
end Adder1Bit;
```

Figure 3.2: 1-bit approximate adder InXA3 in VHDL code. Source: author.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity Adder_M_InXA3_2 is
port(
        a: in std_logic_vector(1 downto 0);
        b: in std_logic_vector(1 downto 0);
        cin: in std_logic;
        cout: out std_logic;
        s: out std_logic_vector(1 downto 0)
);
end Adder_M_InXA3_2;

architecture Adder2Bits of Adder_M_InXA3_2 is
signal c: std_logic_vector(1 downto 0);

component Adder_InXA3_1
port( cin, a, b : in std_logic;
       cout, s  : out std_logic);
end component;

component Adder_Accuracy_1
port( cin, a, b : in std_logic;
       cout, s  : out std_logic);
end component;

begin
FA0 : Adder_Accuracy_1
    port map (cin=>cin, a=>a(0), b=>b(0), s=>s(0), cout=>c(0));
FA1 : Adder_InXA3_1
    port map (cin=>c(0), a=>a(1), b=>b(1), s=>s(1), cout=>cout);
end Adder2Bits;
```

Figure 3.3: 2-bit mixed circuit in VHDL code. Source: author.

accurate and approximate circuit results. The area usage results were based on the relative area (RA $= \frac{ALE}{ELE}$) metric calculated from the number of total logic elements used by the approximate circuit design (ALE) and the accurate circuit (ELE). Total power was the base metric for calculating the relative power (RP $= \frac{AP}{EP}$) of circuits, considering Approximate Power Circuits (AP) and Precise Power Circuits (EP). Error metrics such as Error Distance (ED), Error Rate (ER), Mean Relative Error Distance ($MRED$), and Normalized Error Distance (NED) [35] were also used to analyze the inaccuracy of the proposed approximate arithmetic circuits.

The results presented in this section are divided into experiments on accuracy, area usage, and power dissipation in approximate circuits.

3.3.1 Accuracy Results

Relative error percentage (REp) for the approximate adders from 8- to 64-bits are presented in Figure 3.4. With the exception of the M_AMA1 and M_AXA3 circuits that showed REp of 99.5%, the other circuits showed REp of 100%. The results on 8-bits circuits present high accuracy (some circuits have small relative errors percentage of 21.9%) when compared to the 64-bits circuits. The ripple carry effect of an error that is propagated to the most significant bits is the reason behind the best performance of short bit-width adder circuits.

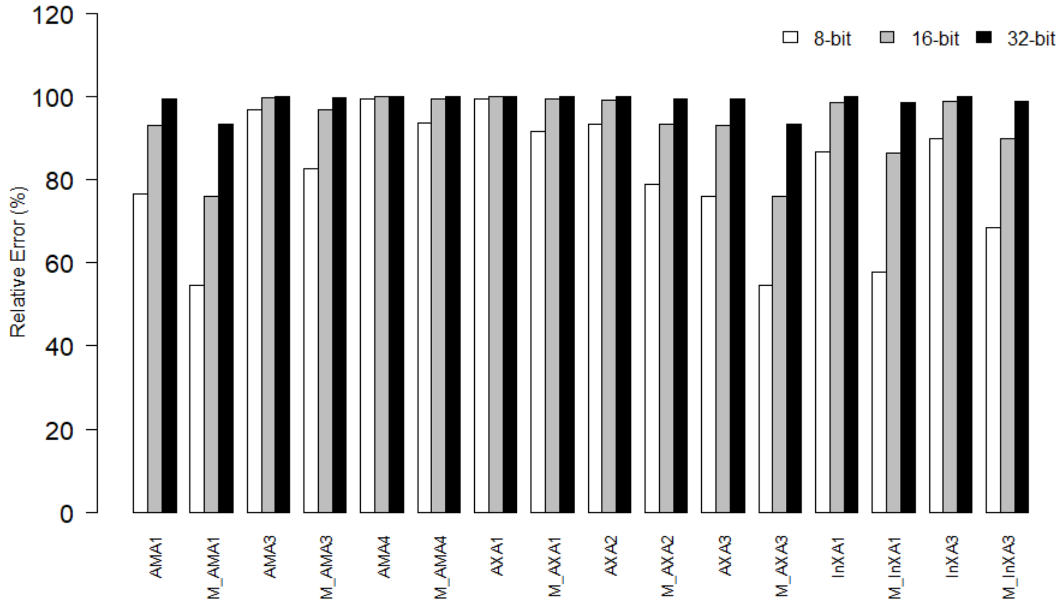


Figure 3.4: Accuracy result (relative error percentage) for adder circuits. Source: author.

Mixed circuits have a smaller number of errors (lesser REp) compared to the approximate circuits of the same type. The 8-bits AMA1 approximate adder circuit has 76% of REp while its mixed type (M_AMA1) has 54.7%. Mixed circuits also show better results for 16-, 32-, and 64-bits circuits. This behavior was expected as the insertion of precise circuits decreases the error propagation and consequently decreases the REp. AMA1 and M_AMA1 circuits achieved NED values of 0.21 while InXA1 and M_InXA1 have 0.29, thus proving the reliability of an approximate adder circuit with approximate cells of the AMA1 and InXA1 type, regardless of the size of the adder circuit.

Figure 3.5 presents the relative error percentage for the subtractor circuits. One may observe that all the mixed subtractor circuits had better results than fully approximated circuits. The 8-bits fully approximated APSC4 subtractor has a relative error percentage of 57.6% while the mixed subtractor has 36.5%. The same behavior is observed for all the subtractor circuits even when circuits bit-width get larger. Circuits APSC4 and M_APSC4 showed low values of NED and MRED.

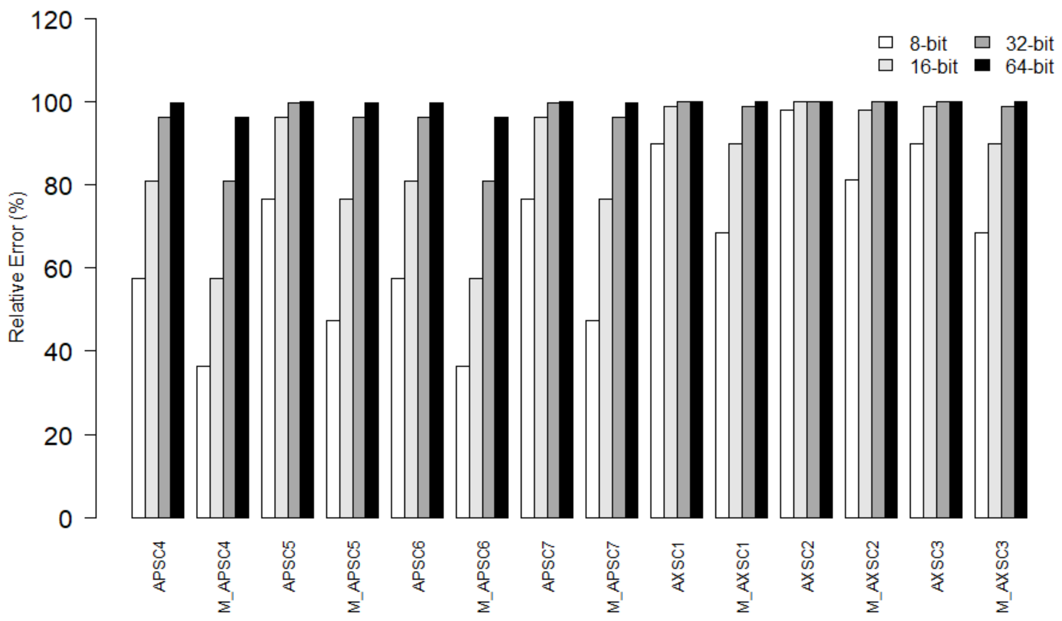


Figure 3.5: Accuracy result (relative error) for subtractor circuits. Source: author.

Figure 3.6 presents the multiplier relative error percentage. The M_InXA1 mixed circuit had the best accuracy result for the 8- and 16-bits circuits. The 64-bits multiplier circuits have 4032 adders, so the error propagation negatively impacts on the multiplier accuracy. In order to analyze the best configuration for the mixed multiplier circuits, we have evaluated them with the least significant adders comprised of exact circuits (M_E/A) and mixed circuits with the least significant adders of the approximate type (M_A/E). Most circuits show better results with the M_E/A 4-bits configuration; only AMA4 and InXA1 circuits show better results in the M_A/E configuration. For 16-bits, the AMA3 and InXA3 circuits have the same result regardless of the order of the accurate and approximate adder designs.

Figure 3.7 shows the relative error percentage for approximate divider circuits. The bars represent the REp of the quotient and the line plot the REp of the remainder. The mixed approximate circuits were designed by substituting the least significant bits of the accurate subtractor circuits by approximate subtractor circuits. Such approximation settings improve the accuracy results in the Q (quotient) output signal, once this is the most affected with the approximation settings. The R (remainder) output presents a greater error due to its dependence on the Q output. The best accuracy results come from the mixed divider circuits, such as the M_APSC4, which presents 21.2% relative error percentage (output Q) for the 8×4 organization. Circuit M_APSC6 achieved the best accuracy result on the R output (59.3%). The REp of the 16×8 organization ranges from 97% up to 99% on the Q output for fully approximated circuits.

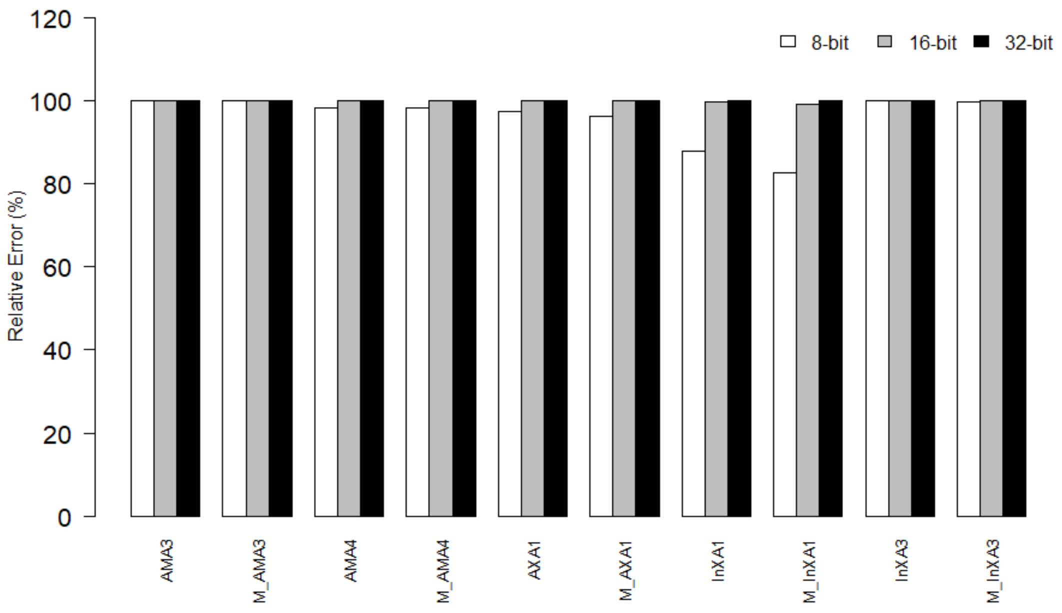


Figure 3.6: Accuracy result (relative error percentage) for multiplier circuits. Source: author.

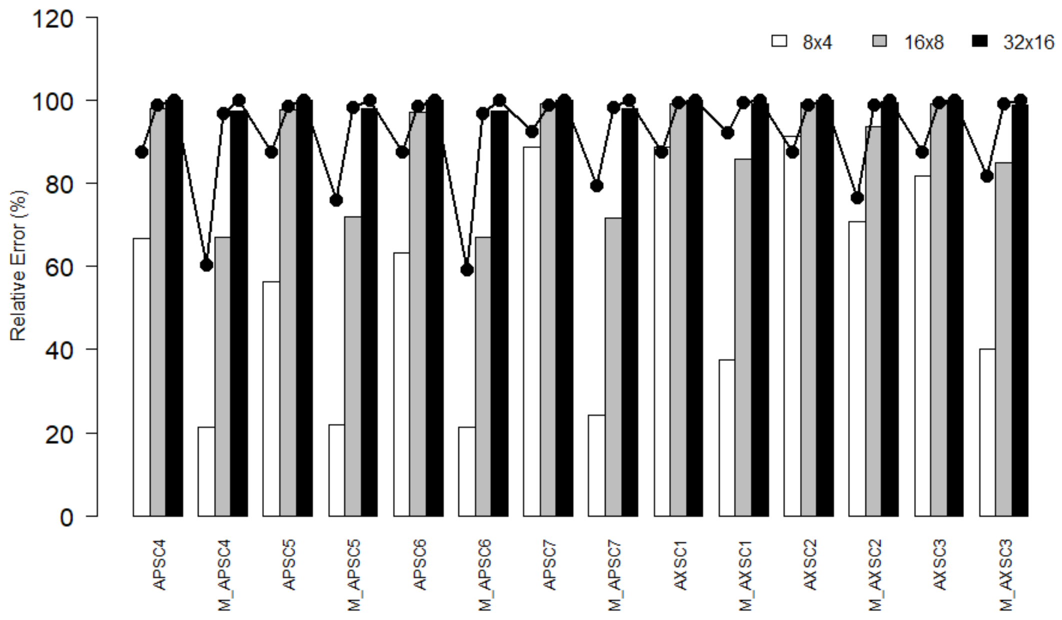


Figure 3.7: Accuracy results (relative error percentage) for divider circuits. Source: author.

Some approximate arithmetic designs have very close relative error percentage values. We carry out the analysis of variance (ANOVA) and statistical tests followed by family pairwise tests (Tukey's test) to verify whether there are a statistically significant difference among those circuits accuracy. The tests were performed with the approximate 16-bits circuits that have the smallest relative error percentage values. The approximate adder circuits M_AMA1, M_InXA1, M_AXA3, and M_InXA1, the subtractors M_APSC4 and M_APSC6, and dividers M_APSC4 and M_APSC6 have $p - values < 0.001$ thus showing a statistically significant difference. The approximate multiplier circuits InXA1 and M_InXA1 have $p - values$ equal to 0.9964, thus showing that both designs have no statistically significant differences. The statistical tests were performed for a significance level $\alpha = 0.05$.

3.3.2 Area Usage

Figure 3.8 shows the RA results of 8-bit adders[2], where the white bars represent approximate adders and the gray bars represent mixed adders. The line on the graph indicates the REp of each adder. It can be seen that there is no direct relationship between a lower RA and a lower REp. For example, the adder AMA4 has the lowest RA but has a high REp. Likewise, the opposite occurs: adders such as AXA3 and AMA1 have a higher RA but a relatively lower REp than the other adders.

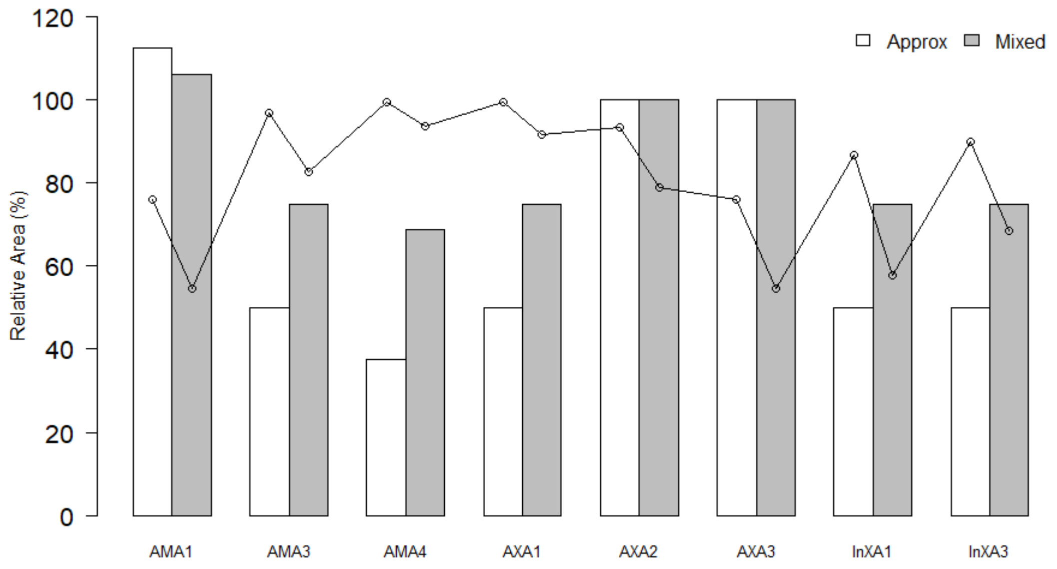


Figure 3.8: Percentage of relative area of approximate circuits for 8-bit adders with relative error. Source: author.

Among all the adder circuits, the AMA4 approximate adder circuit shows better area usage results once output S is formed by just 3 logic gates (S = $\overline{A}.Cin + B.Cin$) and the $Cout$ output is formed only by the A input of the circuit. This fact occurred in all the bit configurations analyzed. At first glance, the InXA1 seemed more promising, due to its configuration being only an XOR gate (S = $A \oplus B \oplus Cin$), but its area results were 50%

[2]The table with the full RA data can be viewed in papers [7, 8].

of the area of the exact adder circuit. Circuits such as AMA1, AXA2, and AXA3 have a larger relative area but with significantly better accuracy than circuit AMA4. Regarding the relative area, the AMA4 and M_AMA4 approximate adder circuits present the best results, with expressive area reductions in FPGA.

Figure 3.9 presents the RA results of the 8-bit subtractors[2]. As in the adding circuits, RA and REp have no direct relationship. It can be seen that the subtractors AXSC2 and AXSC3 have the lowest RA but with a high REp, while APSC6, despite having the second highest RA, has a lower REp.

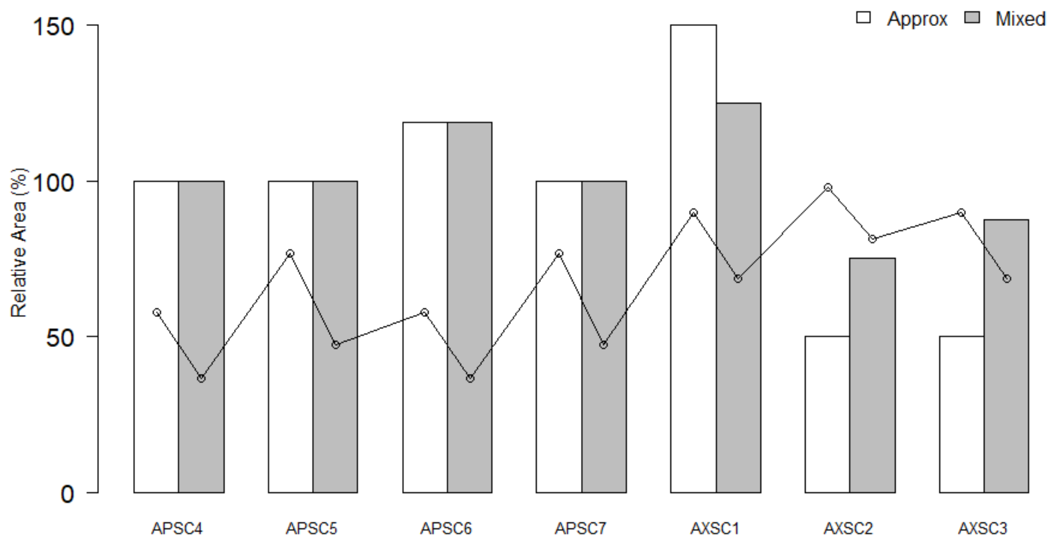


Figure 3.9: Percentage of relative area of approximate circuits for 8-bit subtractor with relative error. Source: author.

Despite the reduction in RA, the multiplier circuits do not present good results in terms of REp. Figure 3.10 shows the graph that relates RA and REp, for 8-bit[3], where it can be seen that, although the multiplier circuits have RA lower than 80%, REp remains high, varying between 80% and 100%.

Figure 3.11 shows the relationship between RA and REp (quotient) for 8-bit divider circuits[3]. Since these circuits are composed of subtractors, RA reaches values above 60% in most dividers. The exception is the divider with the AXSC2 subtractor, which presents a lower RA but maintains high REp values in both configurations.

The trade-off between accuracy and area usage on the approximate designs when prototyped in FPGA. AMA1 and AXA3 achieved good accuracy results but the area usage in FPGA does not provide any benefit compared to the accurate design (100% area usage or more).

[3]The table with the full RA data can be viewed in papers [7, 8].

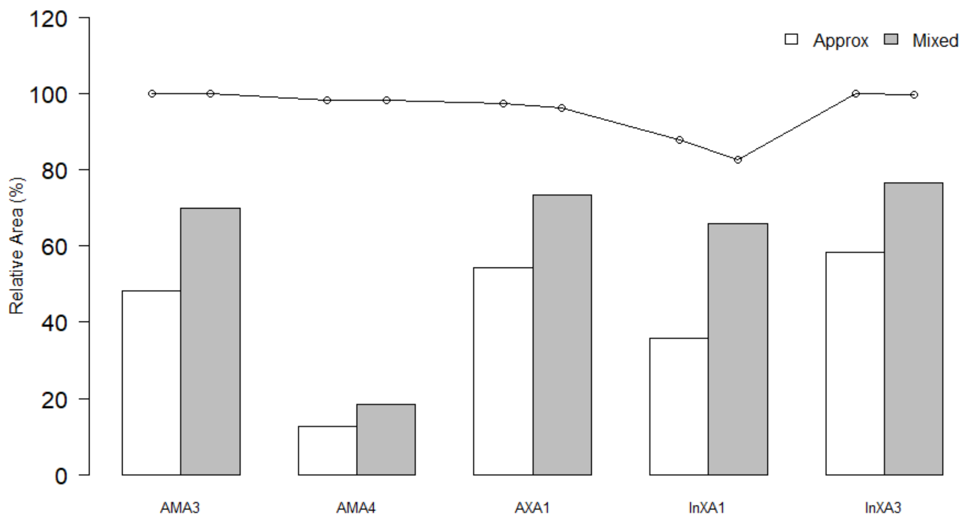


Figure 3.10: Percentage of relative area of approximate circuits for 8-bit multiplier with relative error. Source: author.

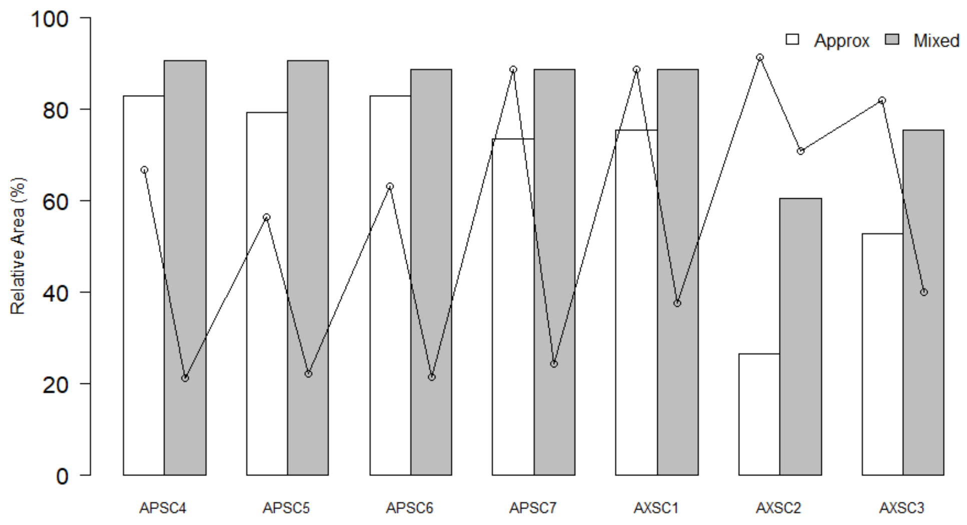


Figure 3.11: Percentage of relative area of approximate circuits for 8-bit divider with relative error. Source: author.

3.3.3 Power Dissipation

Figure 3.12 the relative power of 8-bit adder circuits[4] (bar graph) with their respective REp (line graph). Approximate circuit AMA4 presents better results, which corroborates the use of area, since circuits with small area will have lower static power, thus impacting the total power dissipation of the final circuit.

[4]The table with the full RP data can be viewed in papers [7, 8].

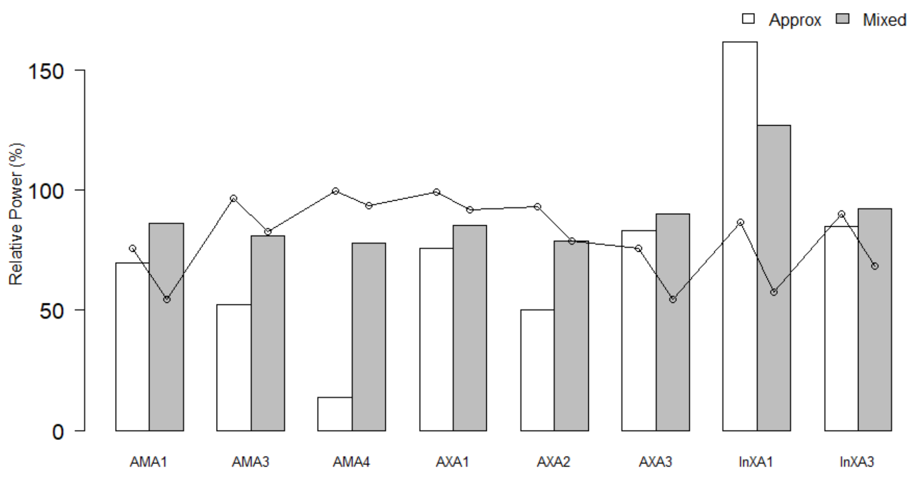


Figure 3.12: Percentage of relative power of approximate circuits for 8-bit adder with relative error. Source: author.

Figure 3.13 presents the relative power of the 8-bit subtractor's approximate[4] experiments with their respective REp values. The approximate subtractor circuits have a larger relative area than the exact subtractor circuits, except for AXSC2 and AXSC3, thus increasing static power dissipation and decreasing the energy-saving benefits of coarse designs. Due to its smaller relative area, the AXSC2 and its respective mixed circuit present the best relative power results. Static and I/O power dissipation were quite akin so that only the average dynamic power has a small difference between the fully approximate circuit (8.10mW) to the mixed circuit (8.89mW).

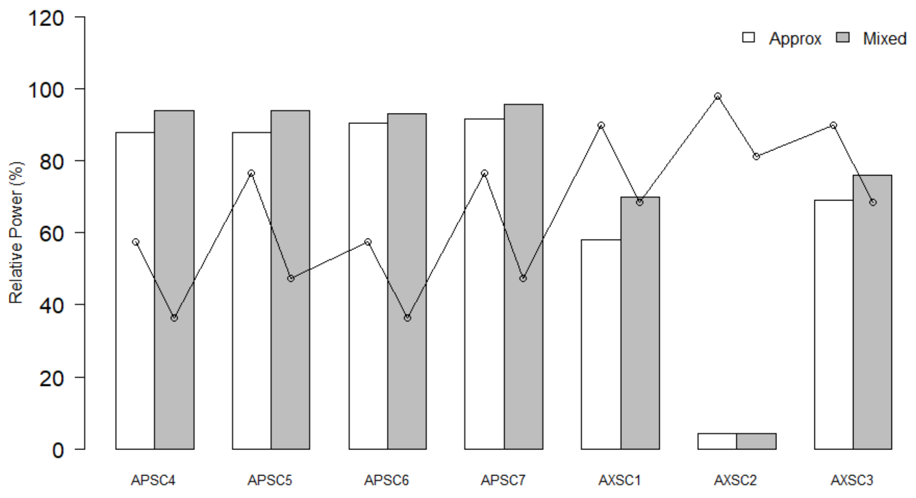


Figure 3.13: Percentage of relative power of approximate circuits for 8-bit subtractor with relative error. Source: author.

Figure 3.14 shows the relative power results of the approximate multiplier circuits[4].

Even being built on the top of approximate adder building blocks, most of the approximate multiplier circuits had better relative power than the accurate multiplier circuit. For complex arithmetic circuits such as multipliers, the power saving can be an issue on the decision to adopt an approximate design.

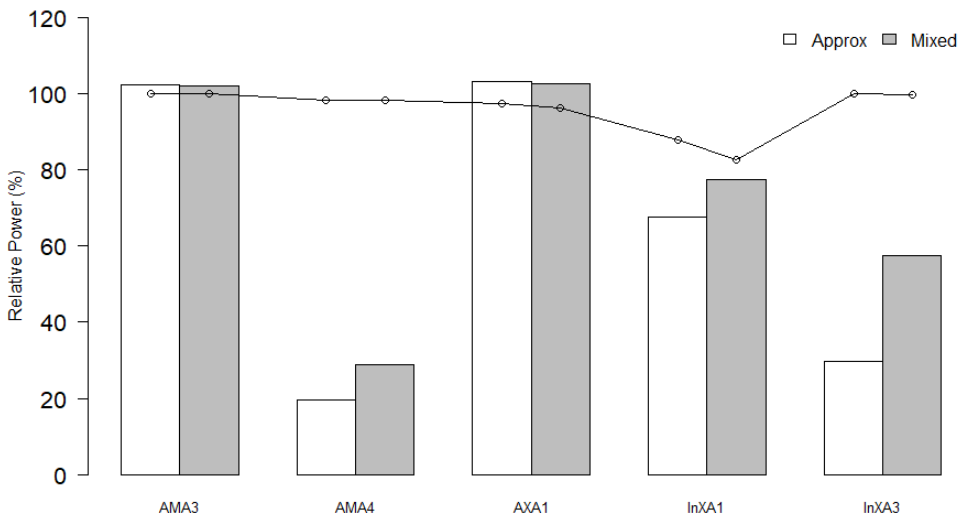


Figure 3.14: Percentage of relative power of approximate circuits for 8-bit multiplier with relative error. Source: author.

One may note that the 8-bits AMA4 relative power is 19.6% while the 32-bits configuration is 98.2%. We observed that the 5× relative power increase was mostly due to the increase in I/O power dissipation. Large circuits designed on an FPGA platform will require more logical blocks and the routing among them, thus increasing of I/O power and its impact on the circuit's total power.

Figure 3.15 presents the approximate relative power of the 8-bit divider circuits[5] and the REp. The logical elements and routing resource usage are the key points to the high relative power for all the circuits. The regularity of the logical blocks and routing available in the FPGA platform did not improve the relative area and power usage for the divider circuits.

As in the RA and REp relationship, there is no relationship between the results in the RP and REp relationship; however, RP has a direct relationship with RA since circuits with smaller areas present lower static power. The adder circuit with AMA4 presents a lower RP but a high REp; the same happens with the AXSC2 subtractor and the multiplier with AMA4. The divider circuits present high RP values and good REp results with the quotient for the mixed circuits.

The accuracy behavior is quite akin among all the circuits. There is a linear increase in relative error percentage according to the circuit size. The relative power

[5]The table with the full RP data can be viewed in papers [7, 8].

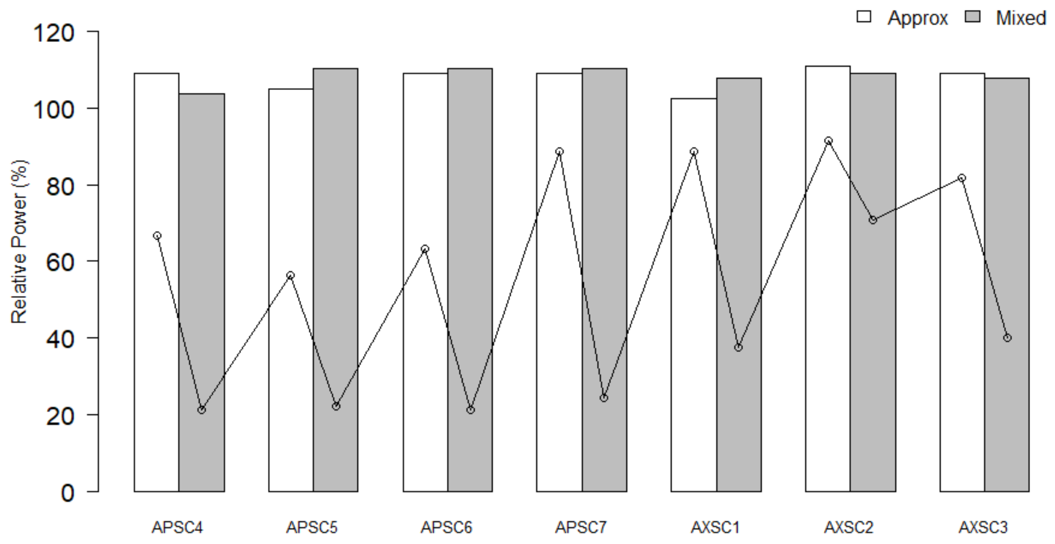


Figure 3.15: Percentage of relative power of approximate circuits for 8-bit divider with relative error. Source: author.

has a different behaviour on divider circuits. For adders, subtractors, and multipliers, the relative power increases following the area usage from 8- to 16-bits. There is not a significant power increase from 16- to 32-bits. For the divider circuits there is a highlighted relative power decrease from 8×4 to 16×8 circuits. The reason is that as the replacement depth increases, the energy consumption (and power dissipation) decreases [13].

There are many proposals presenting circuits and results on power and area benefits greater than 50%. Some of those proposals present results based on simulations or even approximate prototyped circuits. Our results show that general-purpose implementation hardware platform such as FPGAs, despite loosing the flexibility to control or regulate the input voltage and hardware elements to fit and placement, are viable rapid-prototyping technological alternatives for implementing approximate designs.

3.4 Final Remarks

This chapter addressed the evaluation and characterization of approximate and mixed (accuracy and approximate) arithmetic circuits, ranging from 8- to 64-bits. The main contributions include the analysis of the accuracy and physical characterization of these circuits, highlighting the importance of evaluation methods to guarantee the efficiency and reliability of approximate circuits. The circuits have been evaluated on FPGA platforms, which provide significant energy efficiency and resource usage benefits. The results from the proposed approximate circuits show promising error rates, significant area usage, and energy dissipation savings, demonstrating the proposal's potential for practical applications.

Chapter 4

Approximate Instructions

Based on the arithmetic circuits presented in the previous chapter, we have extended the RISC-V instruction set with the design new approximate arithmetic instructions. This chapter presents the design and evaluation of these approximate instructions. We also extend the SPIKE simulator, the Prof5 energy model, and the ACCEPT compiler to design, evaluate, and compile program sources to use the proposed instructions.

4.1 Contextualization

Reduced Instruction Set Computer V (RISC-V) [56] architecture is known for its flexibility and modularity, which facilitate the incorporation of new instructions. Its open-source code makes it easy to adapt, expand, and implement new instructions, allowing the hardware to adapt to the specific needs of an approximation. The SPIKE ISA-SIM [69] simulator is a RISC-V architecture software infrastructure that offers a robust environment for testing and validating the design of new instructions. To evaluate the performance of applications with approximate instructions, the Prof5 [65], a RISC-V profiling tool, has been used. Prof5 allows for detailed profiling of RISC-V programs from the SPIKE log. The outcomes from Prof5 are the number of cycles, instructions, power, energy, and average power per cycle. The ACCEPT [58] compiler is also part of our approximate source-code generation infrastructure. ACCEPT introduces specific annotations to show the programmer which code regions are most suitable for approximation and which techniques would be most promising. ACCEPT offers a set of techniques: loop perforation (LP), loop parallelization, neural acceleration, and approximate functions [53, 58]. The compiler evaluates the techniques and presents performance results (time and accuracy), thus allowing the programmer to analyze the impact of the approximations [54].

Figure 4.1 sketches the workflow to design and evaluate approximated instructions in this work. We design the instruction format, opcodes, and instruction parameters using the RISC-V simulator, SPIKE ISA-SIM (step 1). Prof5 (step 2) was extended to recognize the new instructions' energy model and evaluate them. ACCEPT (step 3) is responsible for applying built-in approximation techniques.

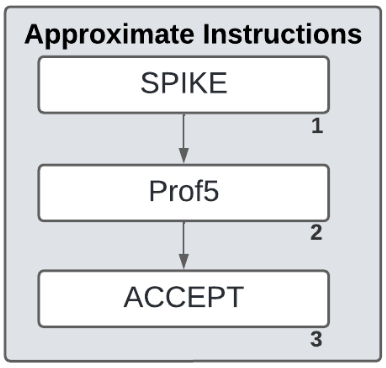


Figure 4.1: Workflow for approximate instructions design and evaluation. Source: author.

4.2 RISC-V Instruction Set

The RISC-V instruction set is divided into six formats: R, I, U, S, SB, and UJ. R-type instructions feature register operations and are comprised of arithmetic instructions, such as add and fadd. I-type instructions use short immediate values and load memory, such as addi and lw (load word). U-type instructions use long immediate values, such as liu and auipc. S-type instructions are for store instructions (sb and sw). The conditional branch and unconditional jump instructions are SB and UJ (beq and jal), respectively.

Table 4.1 shows the format of RISC-V instructions, showing the types, fields, and groups of instructions represented. Regarding that each instruction has 32-bits size, each format has a set of fixed-size fields. The Type-R format, for example, has two data registers (rs1 and rs2) with 5-bits each and a destination register (rd) with 5-bits. The opcode field is the operation code and is the same for all instructions in the format. The joint-usage of funct7 and funct3 fields combined with the opcode represent the operation to be performed. Floating point instructions use rounding mode (rm), with their funct3 field intended for selecting the type of rounding. Instructions not affected by rounding mode use the field with the setting 000.

Table 4.2 shows the layout of some integer and floating point arithmetic instructions of the RISC-V ISA, with the bits of the funct7, funct3, and opcode fields. Note that the opcode is the same for the integer instructions (add, sub, mul, div, and rem), and the floating point instructions (fadd, fsub, fmul, and fdiv) share the same opcode.

RISC-V ISA is open to incorporating new instructions. Adding an instruction not part of the instruction set starts by selecting the format that best adapts to the new

Table 4.1: RISC-V instruction formats. Source: [52].

Type	7 Bits	5 Bits	5 Bits	3 Bits	5 Bits	7 Bits	Group
R	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic
I	immediate		rs1	funct3	rd	opcode	Loads and Immediate Arithmetic
S	immediate	rs2	rs1	funct3	immediate	opcode	Stores
SB	immediate	rs2	rs1	funct3	immediate	opcode	Conditional Branch
UJ	immediate				rd	opcode	Unconditional Jump
U	immediate				rd	opcode	Upper Immediate

Table 4.2: Layout of some RISC-V arithmetic instructions. Source: [52].

Instructions	funct7	rs2	rs1	funct3	rd	opcode
add	0000000	rs2	rs1	000	rd	0110011
sub	0100000	rs2	rs1	000	rd	0110011
mul	0000001	rs2	rs1	000	rd	0110011
div	0000001	rs2	rs1	100	rd	0110011
rem	0000001	rs2	rs1	110	rd	0110011
fadd	0000000	rs2	rs1	rm	rd	1010011
fsub	0000100	rs2	rs1	rm	rd	1010011
fmul	0001000	rs2	rs1	rm	rd	1010011
fdiv	0001100	rs2	rs1	rm	rd	1010011

instruction, choose an opcode that is not in use, or set a new opcode. In addition, the funct7 and/or funct3 fields should be set, since they are used to differentiate the new instruction from the existing ones.

4.3 Approximate Instructions Design

This section presents the design of the approximate instructions. From the design of approximate arithmetic circuits (chapter 3), we designed the approximate integer instructions, named **addx**, **subx**, **mulx**, **divx**, and **remx**. The approximate floating point instructions were designed by replacing parts of their integer operations with the approximate integer instructions and were named **faddx**, **fsubx**, **fmulx**, and **fdivx**.

4.3.1 Approximate Integer Instructions

Choosing the most suitable adder and subtractor for the approximate instruction design is a matter of careful design analysis. The adder/subtractor circuits with the smallest relative area did not bring the best accuracy results, just as the adder/subtractor with the best accuracy did not present good area and/or power results. Therefore, we chose the one that presented the best balance between accuracy and relative area.

Figure 4.2 summarizes the best results from the adder circuits about relative area, relative power, and relative error for sizes of 8-, 16- and 32-bits, highlighting the lowest values of each metric. When analyzing Figure 4.2, considering only RE, the AMA1 and AXA3 adders have the best results for 8-, 16-, and 32-bits. However, both have a larger relative area than the exact adder. The AMA4 adder has the smallest relative area among the adders, followed by the InXA1 and InXA3 adders, which have the same size. However, the InXA1 adder offers the best precision results among the three despite presenting a relative power peak for 8-bits. Given that, the InXA1 adder was chosen to compose the approximate add instruction because it presents a better balance between precision and relative area. However, the idea of composing new approximate instructions with other adders is not ruled out.

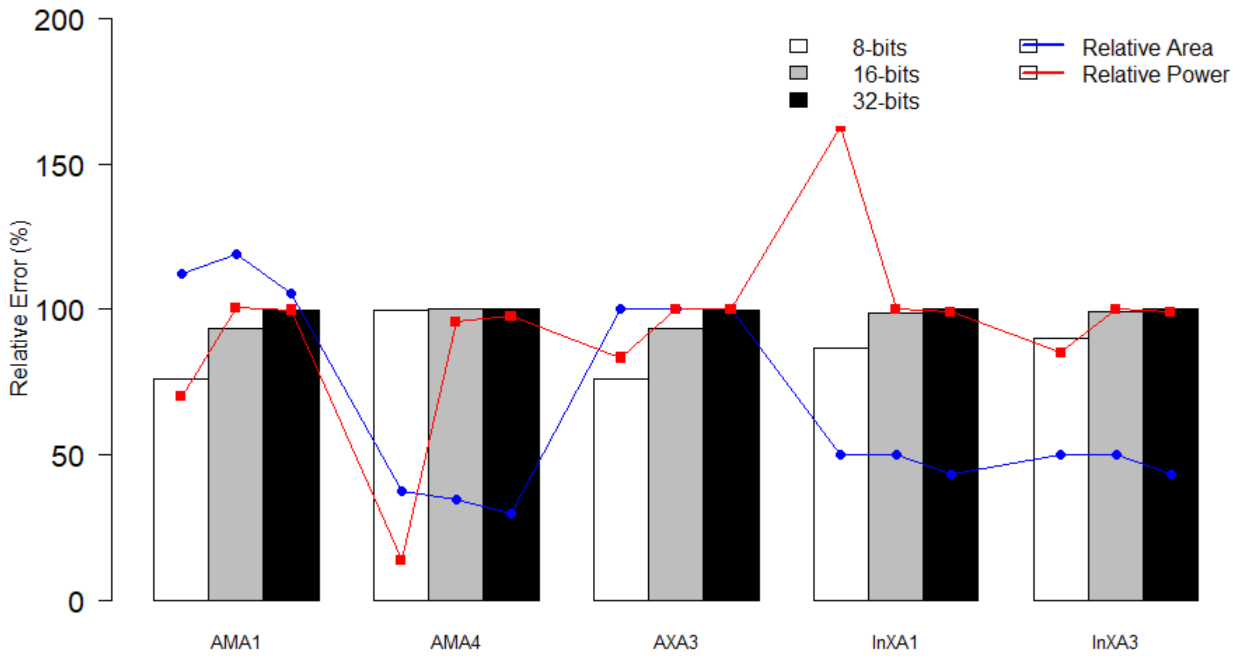


Figure 4.2: Percentages of relative area, relative power, and relative error in approximate adder circuits. Source: author.

The same design context is observed in the subtractors, where the AXSC type subtractors present a smaller relative area than the APSC types; however, APSC circuits present better accuracy results. The APSC4 and APSC6 subtractors present the same accuracy results, with the APSC4 having the smallest relative area, thus being chosen as the building-block for the approximate subtraction instruction. It is known that a 32-bit full adder circuit is comprised of 1-bit full adder (FA) associations, as exemplified in Figure 4.3, therefore the structure of the approximate add instruction is designed with the FA blocks composed of the adder InXA1. The same behaviour happens to the subtractor. The approximate subtractor APSC4 forms the full subtractor blocks.

Regarding the multiplier, the general architecture uses the shift and add algorithms. Depending on the value of the multiplier's least significant bits (LSB), a value of the multiplicand is added and accumulated. At each clock cycle, the multiplier is shifted one bit to the right, and its value is tested. If is “0”, then only one shift operation is performed. If the value is “1”, the multiplicand is added to the accumulator and shifted

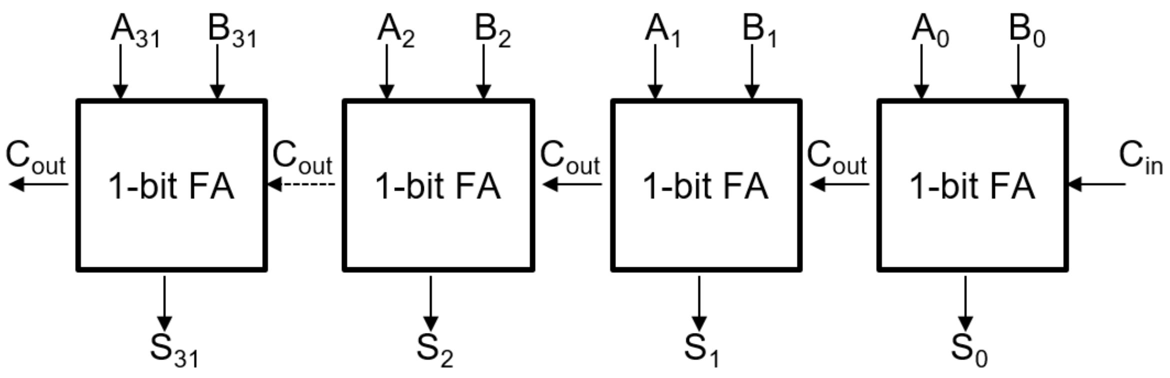
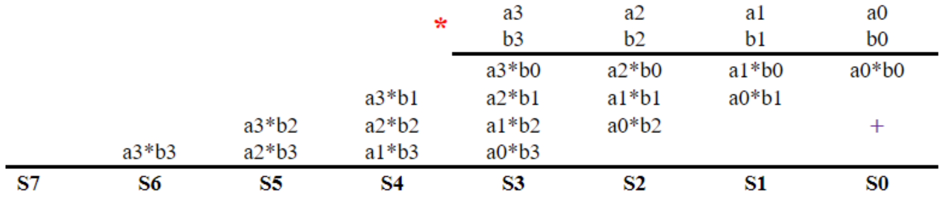


Figure 4.3: 32-bits full adder (FA). Source: Based on [52].

one bit to the right. The product is in the accumulator after all the multiplier bits have been tested [52].

Another way to perform binary multiplication is by following the same procedures as decimal multiplication. Given the multiplicand and the multiplier, we perform the multiplication of each bit of the multiplier by each operand of the multiplicand. Then, we perform the partial sums of each multiplication set and obtain the output results, called the product, where the size of the product is the sum of the multiplicand and the multiplier. Figure 4.4 represents the 4-bit multiplication operation, where the multiplicand is represented by a3-a0, the multiplier by b3-b0, and the product by S7-S0. The approximate multiplication instruction follows the format of decimal multiplication. We used the AND operator (&) to perform multiplications between the multiplicand and multiplier, and the additions were performed with the approximate adder InXA1.



			*	a3	a2	a1	a0
				b3	b2	b1	b0
				a3*b0	a2*b0	a1*b0	a0*b0
			a3*b1	a2*b1	a1*b1	a0*b1	
		a3*b2	a2*b2	a1*b2	a0*b2		+
	a3*b3	a2*b3	a1*b3	a0*b3			
S7	**S6**	**S5**	**S4**	**S3**	**S2**	**S1**	**S0**

Figure 4.4: 4-bits multiplication. Source: author.

The division operation has operands called dividend and divisor, and its result is called quotient, in addition to a second result called the remainder. Division can be expressed by the equation (4.1) and can also be performed in a similar way to decimal division. The division algorithm can be implemented by a series of subtractions and displacements.

$$Dividend = Quotient * Divisor + Remainder \tag{4.1}$$

For the design of the approximate division instruction, we used the division model proposed in Chen et at. [13], as shown in Figure 4.5, using the approximate subtractor APSC4. The proposed divisor has a size restriction. The size of the divisor can be expanded, however, the divisor (represented by Y[3:0]) must have half the size of the dividend (X[7:0]). The quotient (Q[3:0]) and the remainder (R[3:0]) have the same size.

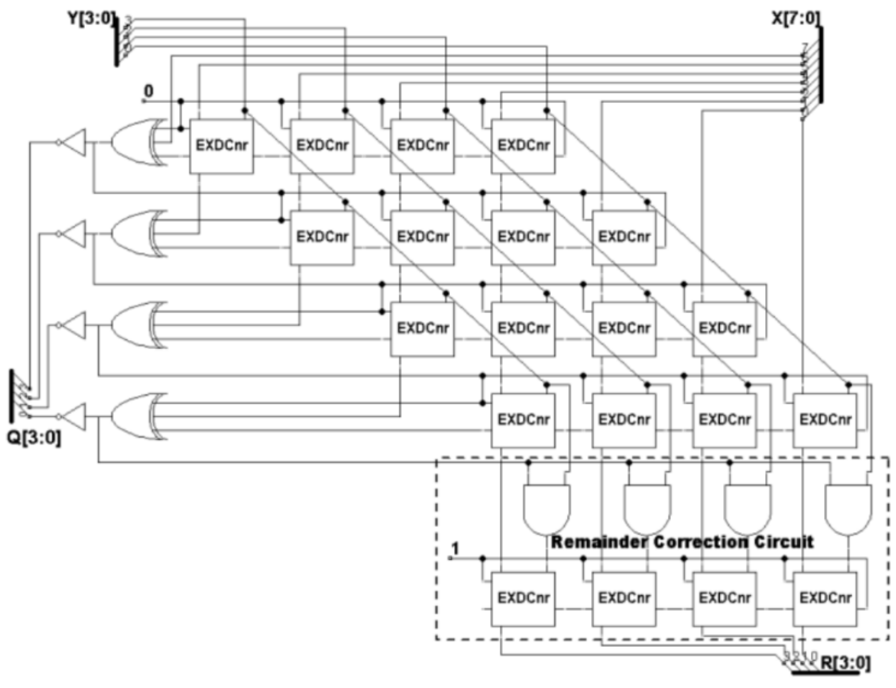


Figure 4.5: 8x4-bits divider. Source: [13].

The authors designed the EXDCnr blocks with exact subtractors and the insertion of an XOR gate, having as input the value of the divisor (Y) and the quotient (Q), as shown in Figure 4.6.The new logical expression of the exact subtractor is in Table 4.3.

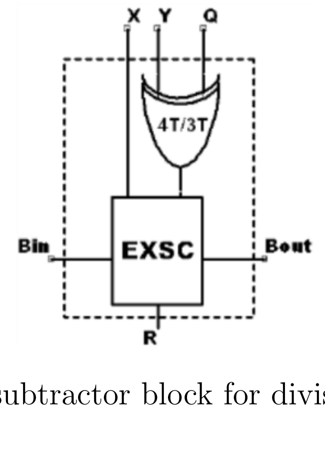


Figure 4.6: Exact subtractor block for division. Source: [13].

Table 4.3: Logical expressions of the subtractor block used in division. Source: author.

Out	**Logical Expression**
R	$X \oplus (Y \oplus Q) \oplus Bin$
Bout	$\overline{(X \oplus (Y \oplus Q))}.Bin + \overline{X}.(Y \oplus Q)$

4.3.2 Approximate Floating Point Instruction

A floating point number can be represented with 32-bits (single precision) or with 64-bits (double precision) and follows a standard called IEEE 754-2008 [68]. Single precision has its 32-bits divided into three parts, with 1-bit representing the sign, 8-bits representing the exponent, and 23-bits representing the fractional part of the number, called the mantissa. Double precision has a representation similar to single precision but with a larger number of bits (1-bit for the sign, 11-bits for the exponent, and 52-bits for the mantissa).

Floating point arithmetic operations involve operations between the mantissa and the exponents. In multiplication, the mantissa is multiplied, and the exponents are added. Division involves subtracting the exponents and dividing the mantissa. Addition and subtraction involve equality between the exponents (equaling the value of the smaller exponent to the larger one) and the addition or subtraction of the mantissa. Each of these operations has its own hardware because despite the simplified explanation of the operations, there are still steps of shifting and rounding the mantissa and exponent.

Figure 4.7 shows the floating point hardware block diagram for addition and subtraction, and the highlighted block (BIG ALU) is where the addition or subtraction operations of the mantissa are performed. Figure 4.8 shows the FP hardware block diagram for multiplication. The highlighted blocks, Adder, Shift left or Shift right, and Rounding hardware, perform, respectively, the sum of the exponents, the shift of the mantissa bits after multiplication, and the rounding of the mantissa and exponent. The FP hardware block diagram for division is shown in Figure 4.9. Similarly to multiplication, the highlighted blocks, Sub, Shift left or Shift right, and Rounding hardware, perform, respectively, the subtraction of the exponents, shifting the mantissa bits after the division, and rounding the mantissa and the exponent.

FP instructions in RISC-V use integer operations to manipulate exponents, mantissa, shift, and rounding. Thus, our strategy for designing new approximate FP instructions was to replace some of the original (precise) integer operations with approximate integer instructions. The decision about which integer operations and instructions to replace was made after performing a design space exploration (DSE) greedy-approach on these operations and evaluating the impact on precision, power, and performance.

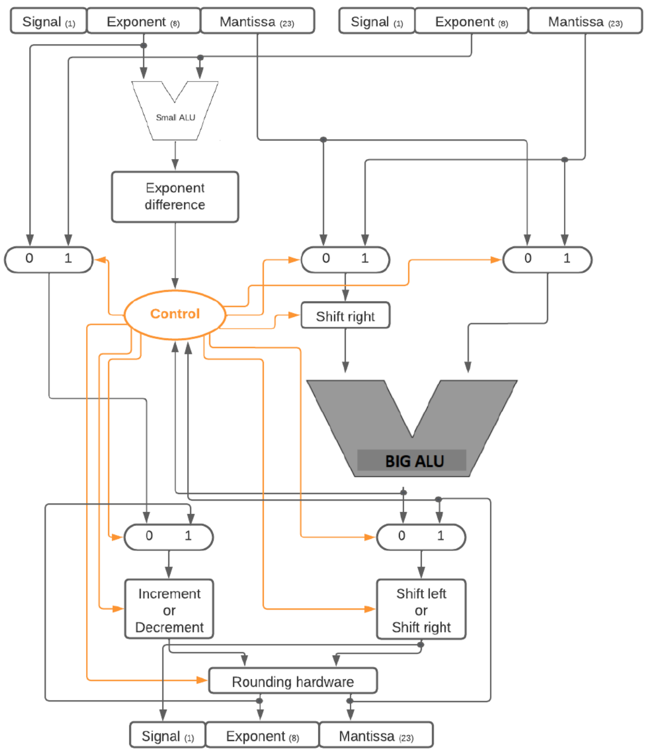


Figure 4.7: Block diagram: FP addition and subtraction hardware. Source: Based on [52].

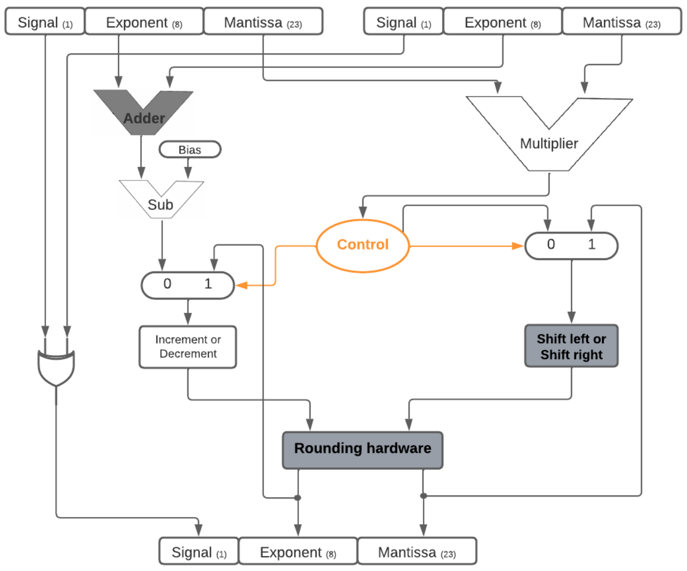


Figure 4.8: Block diagram: FP multiplier hardware. Source: Based on [52].

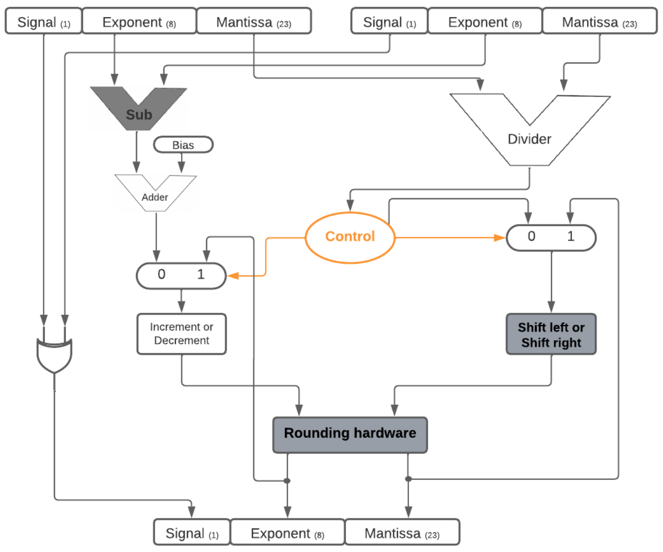


Figure 4.9: Block diagram: FP divider hardware. Source: Based on [52].

4.4 Tools Extension

This section presents the technical details and steps to support approximate insructions into the toolset

4.4.1 SPIKE ISA SIM

Figure 4.10 shows the flowchart used to develop the approximate instructions. In step 1, the format of the instruction to be used is defined. The opcode may be a new opcode class or just an adaptation of it, with the combination of funct7 and funct3 fields. Masks and matches are defined and added to the RISC-V Toolchain. The opcode, mask, and match definition need to be incorporated into the RISC-V PK [55] (RISC-V ELF binary application execution environment), step 2. Step 3 represents the instruction behavior implementation in the SPIKE ISA SIM simulator.

RISC-V arithmetic instructions are Type-R and have the values of the address fields funct7, funct3, and opcodes already defined. To insert new instructions, new opcode values must be set. Table 4.4 shows the values for the fields of the approximate arithmetic instructions. Just as the RISC-V ISA has two instructions for the division operation, we defined the **divx** instruction to the division quotient and the **remx** instruction calculates the remainder of the division.

For the **addx** and **subx** instructions, the funct7 and opcode fields were changed (the changed bits are highlighted in bold). In the **divx** and **remx** instructions, the

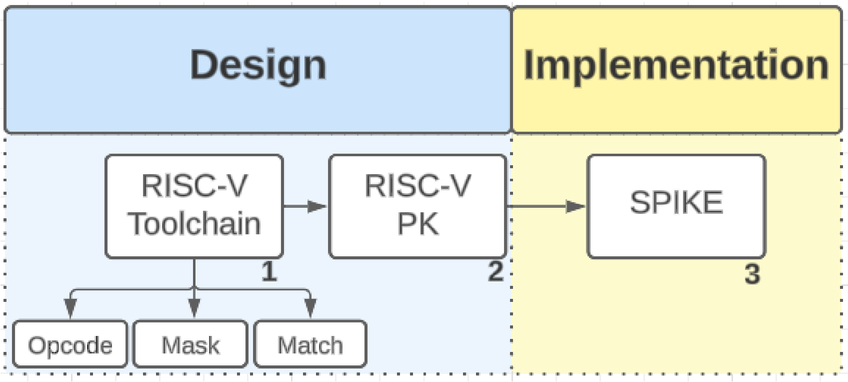


Figure 4.10: Flowchart for the design and implementation of approximate instructions. Source: author.

funct3 and opcode fields were changed (in bold). Only one bit of the opcode field was changed (in bold) in the **mulx** instruction. In the floating point instructions, only the most significant bit of the funct7 field was changed (in bold) when compared with the RISC-V opcodes (Table 4.2).

Table 4.4: Approximate instruction opcodes. Source: author.

Instructions	**funct7**	**rs2**	**rs1**	**funct3**	**rd**	**opcode**
addx	000000**1**	rs2	rs1	000	rd	01**01**011
subx	0**0**0000**1**	rs2	rs1	000	rd	01**011**11
mulx	0000001	rs2	rs1	000	rd	**1**110011
divx	0000001	rs2	rs1	**0**00	rd	**1**110**1**11
remx	0000001	rs2	rs1	**00**0	rd	**1**11**1**011
faddx	**1**000000	rs2	rs1	rm	rd	1010011
fsubx	**1**000100	rs2	rs1	rm	rd	1010011
fmulx	**1**001000	rs2	rs1	rm	rd	1010011
fdivx	**1**001100	rs2	rs1	rm	rd	1010011

With the opcodes of the approximate instructions, the mask and match were defined and inserted into the RISC-V toolchain. Figure 4.11 shows the mask and match of the approximate instructions[6]. In SPIKE, the mask and match are inserted, and the insertion of the files with the functionality of each instruction is required[7].

The **addx** instruction is the simplest, as it uses the approximate adder InXA1 in its design. The difference between the exact adder and InXA1 is only in the output Cout, where in InXA1 Cout receives the value of Cin. In our design, we consider that Cin will always be equal to zero. Thus, since the RISC-V add instruction already uses 32-bit registers, we adapt it with our approximate adder and replace the " + " sign with

[6]The detailed procedures for extending the RISC-V Toolchain are presented at: https://github.com/danielacatelan/Approximate-Instructions

[7]The detailed procedures for the extension of the SPIKE ISA SIM and the files with the approximate instructions designed can be found at: https://github.com/danielacatelan/Approximate-Instructions

```
#define MATCH_ADDX 0x200002b
#define MASK_ADDX  0xfe00707f
#define MATCH_SUBX 0x200002f
#define MASK_SUBX  0xfe00707f
#define MATCH_MULX 0x2000073
#define MASK_MULX  0xfe00707f
#define MATCH_DIVX 0x2000077
#define MASK_DIVX  0xfe00707f
#define MATCH_FADDX_S 0x80000053
#define MASK_FADDX_S  0xfe00007f
#define MATCH_FSUBX_S 0x88000053
#define MASK_FSUBX_S  0xfe00007f
#define MATCH_FMULX_S 0x90000053
#define MASK_FMULX_S  0xfe00007f
#define MATCH_FDIVX_S 0x98000053
#define MASK_FDIVX_S  0xfe00007f
```

Figure 4.11: Mask and match approximate instructions. Source: author.

the XOR gate sign “ ˆ ”, thus the functionality of the approximate **addx** instruction is defined as shown in Figure 4.12.

```
WRITE_RD(sext_xlen(RS1 ^ RS2 ^ 00000000000000000000000000000000));
```

Figure 4.12: Functionality of the approximate **addx** instruction. Source: author.

The functionality of the approximate instruction **subx** was designed based on the association of subtractors (similar to that shown in Figure 4.3), where each subtractor block operates a bit with the approximate subtractor APSC4. Figure 4.13 presents the functionality of the instruction **subx**. The block named B0 (Bit 0, lines 6 to 12) represents the complete approximate subtractor block with the least significant APSC4, and the block B31 (Bit 31, lines 28 to 34) represents the block with the most significant bit. The variable S (line 36) represents the final result of the subtraction and is composed by the union of the 32-bits of the outputs of each subtractor block.

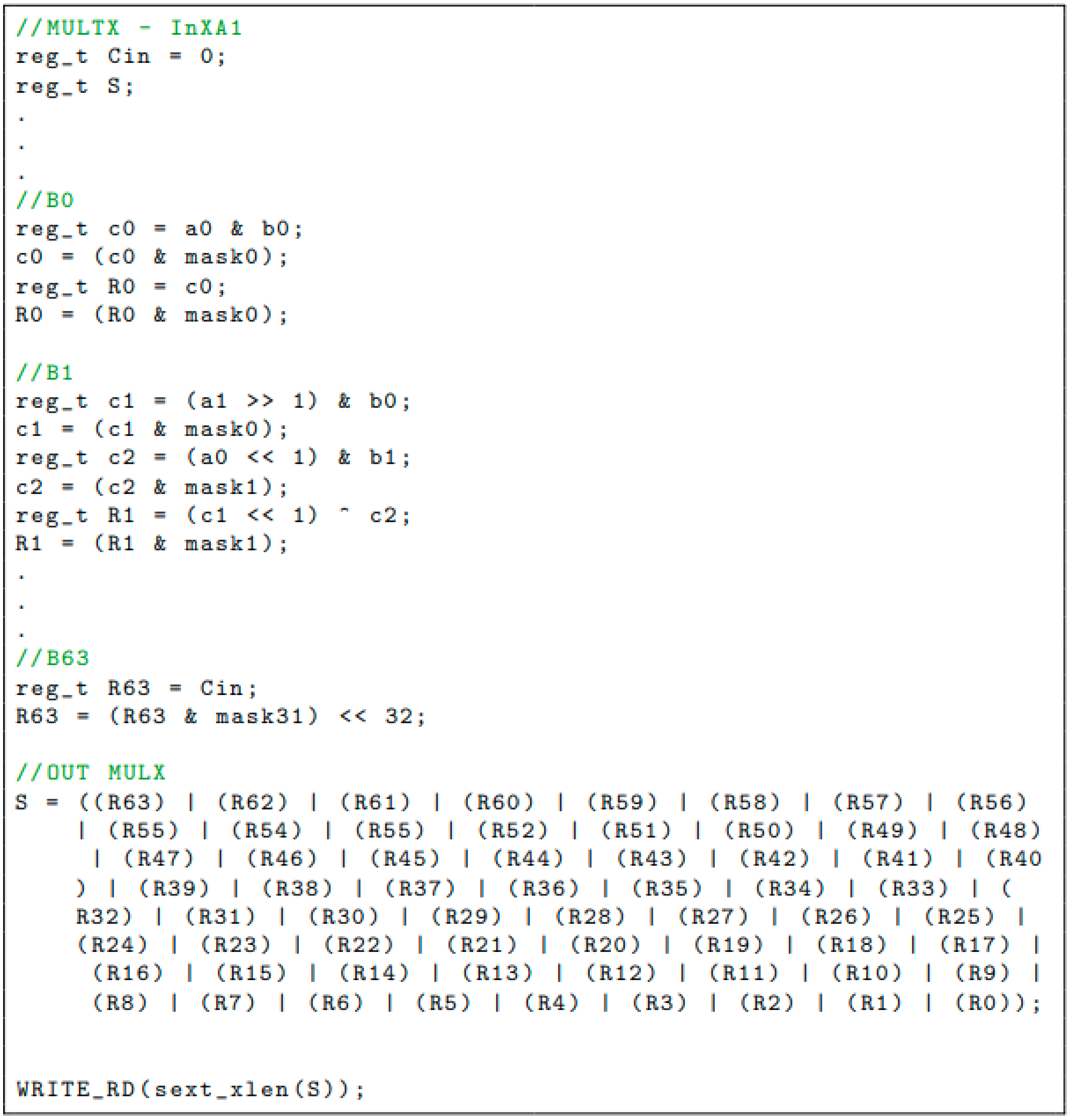
The functionality of the approximate multiplication instruction, **mulx**, required several steps because it was performed similarly to the decimal multiplication operation, as shown in Figure 4.4 and it has 32-bit operators with a 64-bit product. Figure 4.14 shows a code excerpt of the functionality of the approximate instruction **mulx**. The blocks named B0, B1, and B63 (lines 7, 13, and 23) represent the result of the product of bit 0, bit 1, and bit 63. It is noted that in block B1, the multiplication between the multiplicand and the multiplier (lines 14 and 19) is first performed using the AND gate (&), and the sum of these multiplications is performed by the XOR operator, coming from the approximate adder InXA1 (line 18). The variable S (line 28) represents the

```
//SUBX - APSC4
reg_t Cin = 0;
reg_t S;

//B0
reg_t mask = 1;
reg_t x0 = (RS1 & mask);
reg_t y0 = (RS2 & mask);
reg_t Cout0 = (((x0 ^ ~y0) & Cin) | (~x0 & y0));
Cout0 = (Cout0 << 1);
reg_t S0 = ((~x0 & y0) | (y0 & Cin) | (~x0 & Cin)
            | (x0 & ~y0 & ~Cin));

//B1
mask = 2;
x0 = (RS1 & mask);
y0 = (RS2 & mask);
reg_t Cout1 = (((x0 ^ ~y0) & Cout0) | (~x0 & y0));
Cout1 = (Cout1 << 1);
reg_t S1 = ((~x0 & y0) | (y0 & Cout0) | (~x0 & Cout0)
            | (x0 & ~y0 & ~Cout0));

.
.
.

//B31
mask = 2147483648;
x0 = (RS1 & mask);
y0 = (RS2 & mask);
reg_t Cout31 = (((x0 ^ ~y0) & Cout30) | (~x0 & y0));
Cout31 = (Cout31 << 1);
reg_t S31 = ((~x0 & y0) | (y0 & Cout30) | (~x0 & Cout30)
             | (x0 & ~y0 & ~Cout30));
//OUT SUBX
S = ((S31) | (S30) | (S29) | (S28) | (S27) | (S26) | (S25)
    | (S24) | (S23) | (S22) | (S21) | (S20) | (S19) | (S18)
    | (S17) | (S16) | (S15) | (S14) | (S13) | (S12) | (S11)
    | (S10) | (S9) | (S8) | (S7) | (S6) | (S5) | (S4) | (S3)
    | (S2) | (S1) | (S0));

WRITE_RD(sext_xlen(S));
```

Figure 4.13: Functionality of the approximate **subx** instruction. Source: author.

final result of the approximate multiplication.

Figure 4.15 shows the code snippet of the approximate instruction functionality **divx**. The snippet presents the functionality of the approximate divisor of size 64×32 (this size is derived from the approximate divisor model used in Figure 4.5). Note that



```
//MULTX - InXA1
reg_t Cin = 0;
reg_t S;
.
.
.
//B0
reg_t c0 = a0 & b0;
c0 = (c0 & mask0);
reg_t R0 = c0;
R0 = (R0 & mask0);

//B1
reg_t c1 = (a1 >> 1) & b0;
c1 = (c1 & mask0);
reg_t c2 = (a0 << 1) & b1;
c2 = (c2 & mask1);
reg_t R1 = (c1 << 1) ^ c2;
R1 = (R1 & mask1);
.
.
.
//B63
reg_t R63 = Cin;
R63 = (R63 & mask31) << 32;

//OUT MULX
S = ((R63) | (R62) | (R61) | (R60) | (R59) | (R58) | (R57) | (R56)
    | (R55) | (R54) | (R55) | (R52) | (R51) | (R50) | (R49) | (R48)
     | (R47) | (R46) | (R45) | (R44) | (R43) | (R42) | (R41) | (R40
    ) | (R39) | (R38) | (R37) | (R36) | (R35) | (R34) | (R33) | (
    R32) | (R31) | (R30) | (R29) | (R28) | (R27) | (R26) | (R25) |
    (R24) | (R23) | (R22) | (R21) | (R20) | (R19) | (R18) | (R17) |
     (R16) | (R15) | (R14) | (R13) | (R12) | (R11) | (R10) | (R9) |
     (R8) | (R7) | (R6) | (R5) | (R4) | (R3) | (R2) | (R1) | (R0));


WRITE_RD(sext_xlen(S));
```

Figure 4.14: Functionality of the approximate **mulx** instruction. Source: author.

the output “R0”, “R31”, “R992,” and “R1023” (lines 7, 15, 26, and 32) represents the remainder of the division (in partial steps of the approximate divisor), having the logical expression of the output “S” of the approximate subtractor APSC4, where the term “B” is replaced by the expression $Y \oplus Q$. The same replacement occurs in the output Cout, in the divisor called Bout (lines 6, 14, 25, and 31). The instruction output will present the result of the quotient of the operation (line 40).

The functionality of the **remx** instruction is shown in the Figure 4.16. The **remx** instruction has all the functionality of the **divx** instruction, but only the part related to remainder correction is highlighted. The remainder correction uses an AND gate, receiving the values of the remainders and the dividend (lines 8 - 13) as input. After the remainder correction, the instruction shows the result of the remainder of the division operation (line 32).

In order to design the functionality of the floating point instructions, it was necessary, in order to make the decision, to perform a design space exploration (DSE) on the blocks of the arithmetic operations hardware diagram to decide which and how many integer instructions would be replaced by approximate integer instructions, evaluating the impact on precision, power, and performance[8].

In the highlighted block, BIG ALU, in Figure 4.7, six integer addition operations were replaced by approximate integer operations, thus becoming a BIG ALU APPROX block. The approximate add FP instruction that uses this new hardware is named **faddx**. The design of the new **fsubx** instruction has also changed the BIG ALU hardware block, using three approximate integer operations (two add and one sub).

The new six approximate integer addition operations used in the BIG ALU of the approximate FP addition hardware block are shown in Figure 4.17. Lines 1, 4, 7, and 10 present the original (non-approximate) integer addition operations (represented by the sign " + "), and lines 2, 5, 8, and 11 present the approximate operations addition sign (" ˆ "). A similar procedure was performed with the **fsubx** instruction, as can be seen in Figure 4.18, where lines 1, 5, and 8 present integer operations, and lines 2, 6, and 9 feature the exchange for approximate instructions. It should be noted that the original integer subtraction operation (" **-** ") is replaced by the logical function of the APSC4 subtractor.

The new approximate FP multiplication **fmulx** instruction has three approximate add operations (Figure 4.19, lines 2, 13, and 17) and one approximate subtraction operation (Figure 4.19, line 6). The Adder block (add the 2 exponents) has one approximate add operation, becoming an approximate exponent adder block. The other approximate integer operations are on the mantissa side (Shift left or Shift right block) and rounding hardware block.

The new approximate FP instruction **fdivx**, features three integer instruction swaps per approximate instruction. The exponent subtraction block (Sub), the mantissa shift block (Shift left or Shift right), and the rounding block have two approximate addition operations and one approximate subtraction operation, as shown in Figure 4.20 (lines 2, 5, and 9).

[8]Due to the development of the RISC-V FP instructions and the results of the DSE, several files needed to be modified. Description of files and changes can be found at: https://github.com/danielacatelan/Approximate-Instructions

```
//Divx - 64X32
//Line 0 - Quoc 31
//Block 0
reg_t Bin = 0;
reg_t Q = 0;
reg_t Bout0 = ((~(x31 ^ (y0 ^ Q)) & Bin) | (~x31 & (y0 ^ Q)));
reg_t R0 = ((~x31 & (y0 ^ Q)) | ((y0 ^ Q) & Bin) | (~x31 & Bin) | (
    x31 & ~(y0 ^ Q) & ~Bin));
.
.
.
//Block 31
reg_t Bin = Bout30;
reg_t Q = 0;
reg_t Bout31 = ((~(x62 ^ (y31 ^ Q)) & Bin) | (~x62 & (y31 ^ Q)));
reg_t R31 = ((~x62 & (y31 ^ Q)) | ((y31 ^ Q) & Bin) | (~x62 & Bin)
    | (x62 & ~(y31 ^ Q) & ~Bin));

reg_t Quoc31 = ~(x63 ^ Bout31 ^ Q);
reg_t Q = (x63 ^ Bout31 ^ Q);
reg_t Bin = Q;
.
.
.
//Line 31 - Quoc 0
//Block 992
reg_t Bout992 = ((~(x0 ^ (y0 ^ Q)) & Bin) | (~x0 & (y0 ^ Q)));
reg_t R992 = ((~x0 & (y0 ^ Q)) | ((y0 ^ Q) & Bin) | (~x0 & Bin) | (
    x0 & ~(y0 ^ Q) & ~Bin));
.
.
.
//Block 1023
reg_t Bout1023 = ((~(R990 ^ (y31 ^ Q)) & Bin) | (~R990 & (y31 ^ Q))
    );
reg_t R1023 = ((~R990 & (y31 ^ Q)) | ((y31 ^ Q) & Bin) | (~R990 &
    Bin) | (R990 & ~(y31 ^ Q) & ~Bin));

reg_t Quoc0 = ~(R991 ^ Bout1023 ^ Q);
.
.
.
Quoc = ((Quoc31) | (Quoc30) | (Quoc29) | (Quoc28) | (Quoc27) | (
    Quoc26) | (Quoc25) | (Quoc24) | (Quoc23) | (Quoc22) | (Quoc21)
    | (Quoc20) | (Quoc19) | (Quoc18) | (Quoc17) | (Quoc16) | (
    Quoc15) | (Quoc14) | (Quoc13) | (Quoc12) | (Quoc11) | (Quoc10)
    | (Quoc9) | (Quoc8) | (Quoc7) | (Quoc6) | (Quoc5) | (Quoc4) | (
    Quoc3) | (Quoc2) | (Quoc1) | (Quoc0));

WRITE_RD(sext_xlen(Quoc));
```

Figure 4.15: Functionality of the approximate **divx** instruction. Source: author.

4.4.2 Prof5

The performance and power results were acquired from the Prof5 [65] tool. Prof5 is a RISC-V profiling that uses the SiFive E24 RV32IMAFBC microcontroller. The SiFive

```
//Remx - 64X32
.
//Instruction Divx
.
.
.
//Remainder Correction Circuit
correct0 <= y0 & Quoc0;
correct1 <= y1 & Quoc0;
.
.
.
correct31 <= y31 & Quoc0;

reg_t Bin = 1;
reg_t Q = 1;

//Line Correction - Remainder
//Block 1024
reg_t Bout1024 = ((~(R992 ^ (correct0 ^ Q)) & Bin) | (~R992 & ((
    correct0  ^ Q)));
reg_t Rem0 = ((~R992 & ((correct0  ^ Q)) | (((correct0  ^ Q) & Bin)
     | (~R992 & Bin) | (R992 & ~((correct0  ^ Q) & ~Bin));
reg_t Bin = Bout1024 ;
.
.
.
//Block 1055
reg_t Bout1055 = ((~(R1023 ^ (correct31 ^ Q)) & Bin) | (~R990 & (
    correct31 ^ Q)));
reg_t Rem31 = ((~R1023 & (correct31 ^ Q)) | ((correct31 ^ Q) & Bin)
     | (~R1023 & Bin) | (R1023 & ~(correct31 ^ Q) & ~Bin));

Remx= ((Rem31) | (Rem30) | (Rem29) | (Rem28) | (Rem27) | (Rem26) |
    (Rem25) | (Rem24) | (Rem23) | (Rem22) | (Rem21) | (Rem20) | (
    Rem19) | (Rem18) | (Rem17) | (Rem16) | (Rem15) | (Rem14) | (
    Rem13) | (Rem12) | (Rem11) | (Rem10) | (Rem9) | (Rem8) | (Rem7)
     | (Rem6) | (Rem5) | (Rem4) | (Rem3) | (Rem2) | (Rem1) | (Rem0)
    );

WRITE_RD(sext_xlen(Remx));
```

Figure 4.16: Functionality of the approximate **remx** instruction. Source: author.

E24 is a high-performance RISC-V microcontroller design model with a 3 stage pipeline running at 125 MHz and was used to perform power modeling of some instructions. Prof5 allows the user to create detailed profiles of RISC-V programs from the SPIKEX log, generating profiles that include the number of cycles, instructions, and power consumption of each instruction and function. The Prof5 energy model was customized to calculate the power of all approximate instructions. The approximate integer instructions have an average power gain (compared to the non-approximate instruction) of 1.3% [8], single-precision approximate FP instructions have an average power gain of 1.2% [28].

Table 4.5 presents the results from the Prof5 tool on the number of cycles, power

```
//sigZ = 0x01000000 + sigA + sigB;//ORIG
  sigZ = 0x01000000 ^ sigA ^ sigB;//APPROX

//sigA = sigA + expA ? 0x20000000 : sigA;//ORIG
  sigA = sigA ^ expA ? 0x20000000 : sigA;//APPROX

//sigB = sigB + expB ? 0x20000000 : sigB;//ORIG
  sigB = sigB ^ expB ? 0x20000000 : sigB;//APPROX

//sigZ = 0x20000000 + sigA + sigB;//ORIG
  sigZ = 0x20000000 ^ sigA ^ sigB;//APPROX
```

Figure 4.17: Approximate addition operations used in the BIG ALU of the approximate FP addition hardware. Source: author.

```
//sigDiff = sigA - sigB;//ORIG
  sigDiff = ((~sigA & sigB) | (sigB & Cin) |
  (~sigA & Cin) | (sigA & ~sigB & ~Cin));//APPROX

//sigY = sigA + (expA ? 0x40000000 : sigA);//ORIG
  sigY = sigA ^ (expA ? 0x40000000 : sigA);//APPROX

//sigY = sigB + (expB ? 0x40000000 : sigB;//ORIG
  sigY = sigB ^ (expB ? 0x40000000 : sigB;//APPROX
```

Figure 4.18: Approximate subtraction operations used in the BIG ALU of the approximate FP subtraction hardware. Source: author.

(μW) and power difference (%) of non-approximated and approximated instructions. Columns 2-3 present the number of cycles and power of the non-approximate instructions. The number of cycles is the same for non-approximation and approximate instructions. Column 5 shows the power of the approximate instructions designed in this work. Equation (4.2) was used to calculate the power difference in percentage ($\text{Power}_{\text{Diff}}$) between the non-approximated instructions ($\text{Power}_{\text{nap}}$) and approximated ($\text{Power}_{\text{ap}}$) and the results are presented in column 6.

$$Power_{\text{Diff}} = \frac{(Power_{\text{nap}} - Power_{\text{ap}})}{Power_{\text{nap}}} \times 100 \tag{4.2}$$

```
//expZ = expA + expB - 0x7F;//ORIG
  expZ = expA ^ expB - 0x7F);//APPROX

  aux = ((uint_fast64_t) 1<<dist);//ASSIST
//return a>>dist | (a & (aux - 1)) != 0);//ORIG
  approx_sub = ((~aux & 1) | (1 & 0) | (~aux & 0)
  | (aux & ~1 & ~0));//APPROX
  return a>>dist | ((a & approx_sub) != 0);//APPROX


//isTiny = (softfloat_detectTininess ==
    softfloat_tininess_beforeRounding) || (exp < -1)
    || (sig + roundIncrement < 0x80000000);//ORIG
  isTiny = (softfloat_detectTininess ==
    softfloat_tininess_beforeRounding) || (exp < -1)
  || (sig ^ roundIncrement < 0x80000000);//APPROX

//sig = (sig + roundIncrement) >> 7;//ORIG
  sig = (sig ^ roundIncrement) >> 7;//APPROX
```

Figure 4.19: Approximate multiply operations of the approximate FP multiplier hardware. Source: author.

```
//expZ = expA - expB + 0x7F;//ORIG
  expZ = expA - expB ^ 0x7F);//APPROX

//sigZ = sigZ + 2;//ORIG
  sigZ = sigZ ^ 2 ;//APPROX


//sigZ = sigZ - 4;//ORIG
  sigZ = ((~sigZ & 4) | (4 & 0) | (~sigZ & 0)
  | (sigZ & ~4 & ~0));//APPROX
```

Figure 4.20: Approximate division operations of the approximate FP divider hardware. Source: author.

4.4.3 ACCEPT

After inserting the approximate instructions into the RISC-V and SPIKE and extending the mathematical model of the instructions, a tool for testing (compilation) is needed. ACCEPT framework was chosen because it presents the user with code snippets suitable for approximation.

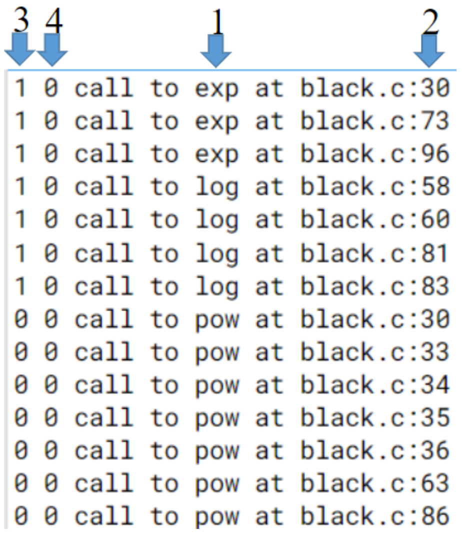
Figure 4.21 shows a code snippet from the log file generated by ACCEPT after

Table 4.5: Cycles and power results of non-approximate and approximate instructions. Source: author.

Instructions	**Cycles**	**Power$_{\textbf{nap}}$ (μW)**	**Instructions**	**Power$_{\textbf{ap}}$ (μW)**	**Power$_{\textbf{Diff}}$(%)**
add	1	2.80	addx	2.76	1.43
sub	1	2.86	subx	2.82	1.40
mul	1	3.09	mulx	3.05	1.29
div	1	3.09	divx	3.05	1.29
fadd	2	3.36	faddx	3.18	5.36
fsub	2	3.43	fsubx	3.34	2.62
fmul	2	3.71	fmulx	3.52	5.12
fdiv	2	3.71	fdivx	3.61	2.70

compiling the application. In this example, the column highlighted by arrow 1 presents mathematical functions (exp, log, and pow) suitable for approximation. The column pointed to by arrow 2 presents the line number where the functions are in the application source code. Arrow 3 is used by the user to choose the approximation level. Level "0", in this example, the mathematical function will be executed using the *math.h* library, in our context, it will be executed without approximation, that is, exactly. If approximation level "1" is chosen, the mathematical function will be executed using the approximate mathematical function of the FastApprox library [41]. The column with the arrow 4 was a contribution of our work, where the user can choose the Loop Perforation (LP) approximation with the approximate instructions, opting for three approximation levels: without LP approximation, LP approximation via ACCEPT, and LP approximation with approximate instructions[9].

[9]The details of the functionality of the FastApprox library and the Loop Perforation with approximate instructions will be presented in chapter 5.



```
1 0 call to exp at black.c:30
1 0 call to exp at black.c:73
1 0 call to exp at black.c:96
1 0 call to log at black.c:58
1 0 call to log at black.c:60
1 0 call to log at black.c:81
1 0 call to log at black.c:83
0 0 call to pow at black.c:30
0 0 call to pow at black.c:33
0 0 call to pow at black.c:34
0 0 call to pow at black.c:35
0 0 call to pow at black.c:36
0 0 call to pow at black.c:63
0 0 call to pow at black.c:86
```

Figure 4.21: ACCEPT. Source: author.

4.5 Final Remarks

This chapter presented the implementation of instructions that use approximate computing techniques in a RISC-V toolset. The RISC-V architecture was chosen for its flexibility and modularity. The SPIKE simulator was used, as it is a robust tool for testing and validating these new instructions. The extension of the Prof5 energy model to include the new instructions demonstrated a commitment to performance analysis and energy efficiency. In addition, the performance evaluation was discussed, emphasizing the relevance of considering the effectiveness and efficiency of these instructions in the context of approximate computing. The contributions of this chapter are significant, as they not only introduce a new set of approximate instructions but also establish a robust workflow for the design, evaluation, and validation of these instructions. Also, the proposal of instructions such as **addx**, **subx**, **mulx**, **divx**, and **remx**, as well as their floating point counterparts, represent an important advance in the practical application of approximate computing. The discussion on tools extension reveals the importance of adapting development systems, including software and hardware, to support this new class of instructions. This emphasizes that the successful implementation of approximate instructions requires not only innovations at the circuit level but also a coordinated effort in adapting tools that allow their integration into real systems. Furthermore, the research carried out in this chapter paves the way for new investigations and developments in the area.

Chapter 5

Approximate Computing Software-Hardware Approach

This chapter exploits the design of code optimization techniques built with the new approximate instructions. From the classical Loop Perforation technique, we developed a new Instruction-Level Loop Perforation (ILLP) approach, using approximate instructions to perform the loop increment. We also redesigned the technique of approximate mathematical functions by implementing Instruction-Level Approximate Functions (ILAF), where original non-approximate instructions were replaced by their approximated counterparts.

5.1 Contextualization

Most AC software techniques aim to solve specific problems or require excessive intervention from the programmer, who needs to identify which application parts are susceptible to approximations [53]. The Loop Perforation (LP) technique is widely used in AC. LP is simple and effective in reducing the amount of computational work by skipping loop iterations and trading precision for other benefits such as performance and power consumption. Research papers [53, 64, 34, 44, 57] have proposed different ways of applying LP, from a simple loop step to adopting heuristic approaches to find the suitable increment value.

A common limitation of LP is that once the perforation degree (*pd*) is established, application metrics (performance, energy) will only improve at the cost of accuracy. One way to overcome this limitation would be to adopt a strategy where *pd* could use approximate hardware resources. A hardware-supported approach could minimize power consumption or improve performance without forcing additional compilation steps or changes to the application code.

We developed the Instruction-Level Loop Perforation (ILLP) approach that relies on approximate hardware instructions. The idea is to use approximate instructions for calculating the next loop iteration. Unlike software-only LP, ILLP calculates the next loop iteration value using approximate hardware. The direct impact of the technique is on application performance when the approximate hardware runs faster than the exact hardware.

The FastApprox library [41] (FAl) is a tool that provides approximated and vectorized versions of mathematical functions that can be used by applications that support approximations. The ACCEPT framework adopts this library to provide approximated versions of well-known mathematical functions in the application source code. Previous experiments with approximate functions on applications achieved speedups ranging from 1.22× to 634×, with a maximum loss of precision of 30% [53]. These approximate results come only from the software technique since the application hardware is unaware of any approximate approach.

From the previous results on AC applied to mathematical functions, we observed an opportunity to improve performance and reduce power consumption by introducing an additional level of approximation by replacing precise (non-approximated) operations with approximate floating point instructions. This new technique offers a hardware-level (instructions) approximation technique over a source code that is already (or not) approximated by a software-level technique. From this insight, we developed Instruction-Level Approximate Functions (ILAF), which incorporate approximate floating point (FP) instructions into FastApprox mathematical functions such as sine, cosine, tangent, exponential, and logarithmic.

To design both approaches (ILLP and ILAF), the ACCEPT compiler, SPIKE simulator, and RISC-V toolchain needed to be extended to support new approximate instructions, generate approximate source code, and simulate approximate applications. To implement these new capabilities, we extended ACCEPT with a new workflow that allows users to annotate the perforation degree, the loop to be perforated, the choice of looping technique (ILLP or software), the choice of which mathematical function(s) to approximate, and the choice of approximation form for the mathematical functions: exact (math.h), fast and faster (FastApprox), and ILAF.

To evaluate ILLP, we conducted experiments on 13 general-purpose applications and subjected each to different *pd* using the original LP software and ILLP. Our approach demonstrated a reduction in the number of instructions for all applications at all drilling levels while maintaining the same level of accuracy.

ILAF was evaluated and tested on six different applications that apply mathematical functions. The results are organized into three versions: the baseline (BL), FastApprox (FAl), and ILAF approach. The experiments evaluated accuracy, execution cycles, and power (μW).

5.2 Instruction-Level Loop Perforation - ILLP

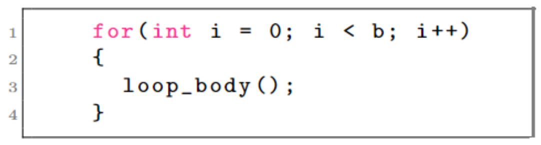
A more detailed exposition of the ILLP technique is covered by presenting the Loop Perforation technique in subsection 5.2.1; The instruction-level LP design is shown in subsection 5.2.2; Subsection 5.2.3 describes the applications and experiments performed to evaluate and validate the approximation techniques; The results and discussion are in subsection 5.2.4.

5.2.1 Loop Perforation Approximation Technique

AC is a design alternative that offers performance with increasingly stringent energy and cost constraints. The of exchanging to exchange reduced accuracy in results for gains in performance and energy consumption has increased interest in this field of study [64]. AC presents approximation techniques for hardware and software designs. In hardware, techniques range from the circuit level to architecture, offering reductions in area and energy consumption in the integrated circuit by employing approximate logic circuits, voltage scaling, and memory density. In software, it achieves better system performance by performing less computational work and reducing memory access [8].

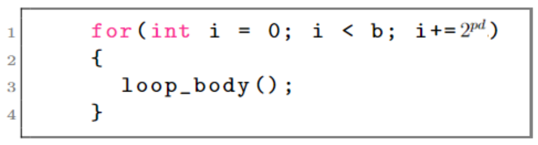
LP is an AC technique that has been gaining appreciation among designers because it is simple, has a general purpose, and is widely applicable to different applications [34]. LP consists of skipping loop iterations to reduce computational workload and gain performance. A simple change in the loop step variable is enough to change its performance, but manually changing one or more loops of an application sometimes becomes more costly than the loop execution itself.

The LP requires a *pd* parameter that indicates how often to skip an iteration at runtime. Figure 5.1 shows an original loop (Fig. 5.1(a)), suitable for perforation, along with a code snippet of the perforated loop (Fig. 5.1(b)) based on a *pd*. The larger the value of *pd*, the fewer iterations the loop will execute, affecting performance and energy consumption.



```
for(int i = 0; i < b; i++)
{
  loop_body();
}
```

(a) Original loop.



```
for(int i = 0; i < b; i+=2^pd)
{
  loop_body();
}
```

(b) Loop perforation.

Figure 5.1: Original loop and after applying loop perforation. Source: author.

Common challenges in adopting LP rely on discovering which loop to perforate and how much to perforate. These challenges are the motivation for a variety of research work that focuses on automatizing the process of finding loops able to be perforated or even adopting different approaches to set the perforation level.

5.2.2 Instruction-Level Loop Perforation Design

Once the compiler selects the loop to be the puncture loop, the iteration value for the variable will be equal to 2^{pd}, Figure 5.1(b). The *pd* is the exponent for the loop iterator to be 2^{pd}, so when $pd = 1$, in the application, the iteration of the "real" loop will be $2^1 = 2$. The values of *pd* adopted in the applications are 1, 2, 4, and 8. Actual perforation values are 2, 4, 16, and 256. It is important to note that code approximation only occurs when *pd* differs from the initial loop iteration step.

Our proposal presents a new LP in which the loop iteration value is not fixed but determined by an approximate instruction. Figure 5.2(a) provides an example of the ILLP technique is inserted into a loop with an incremental loop step. The original operation for the loop step is replaced with a function that invokes an approximate instruction. The approximate function ADDX performs the approximate adder instruction (**addx**).

```
1      for(int i = 0; i < b; i=ADDX(i,2^pd))
2      {
3         loop_body();
4      }
```

(a) Loop perforation with ADDX.

```
1      int ADDX (int i, int 2^pd)
2      {
3            int ADDX;
4            asm volatile {
5               "addx %[z], %[x]; %[y]\n\t"
6               :[z] "=r" (ADDX)
7               :[x] "r" (i), [y] "r" (2^pd) );
8            }
9            // correction_factor
10           if (ADDX <= i)
11              ADDX = i + 2^pd;
12           return (ADDX);
13     }
```

(b) ADDX approximate function.

Figure 5.2: LP with ADDX function call and ADDX function. Source: author.

Figure 5.2(b) presents the assembly code (command `volatile asm`) for the ADDX function. Lines 4-8 indicate that an instruction called **addx** will be executed where parameters i and *pd* will be placed in registers x and y, respectively. The result of the instruction (register z) will be available in the ADDX variable. An offset code (lines 10-11) is used to correct the statement result when it is less than or equal to the current loop step. Note that this offset code is applied to loops with increasing iterator. For a descending loop step, line 10 should evaluate whether the result is greater than or equal to i, and line 11 must be changed to a subtraction operation.

The correction_factor (CF), lines 10-11, could be performed directly in the hardware, but the designer should be aware that CF is only valid to ensure that increment (i) is always progressing. A CF in hardware will be unnecessary for applications that may use the approximate instruction outside the context of LP. Our motivation in designing the **addx** instruction was to keep it very similar to the exact corresponding (**add**) instruction design. It should also be noted that calling the ADDX function in the software was the way to use the approximate instruction.

We extended the ACCEPT to allow users to choose the ILLP in a loop. The first step consists of choosing which loop to perforate. Figure 5.3 illustrates the steps to use ILLP. Given a source code, ACCEPT will analyze the code and identify all loops that can be perforated. A text file is generated presenting the candidate loops to the user (2). In step (3), the user chooses the loop, the *pd*, and the LP: ACCEPT_SPIKE (AS) or Instruction-Level Loop Perforation (ILLP).

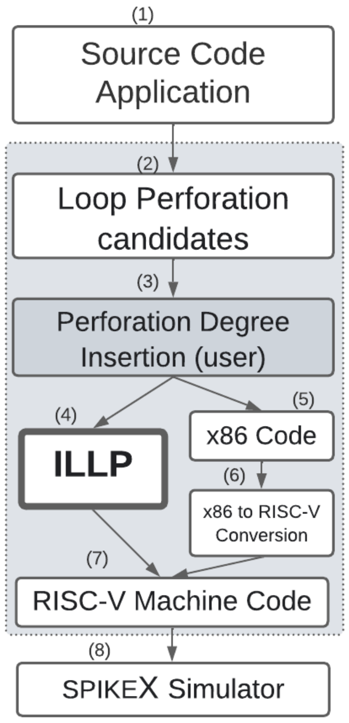


Figure 5.3: Workflow and toolset to apply the ILLP technique. Source: author.

At this point, the compilation flow is divided into two paths: step (4) means that the ILLP now executes the LP, and the output is RISC-V assembly code; step (5) is the standard LP available in ACCEPT. Step (6) is carried out when the AS is chosen. The conversion is required once that ACCEPT is integrated into LLVM version 3.2 which only generates x86 code. Step (7) converts the assembly to machine code. In step (8), the application can be simulated in the SPIKEX simulator (our extension for the SPIKE simulator). SPIKEX supports approximate instructions, such as **addx**, and is fully compatible with the original SPIKE.

The code snippet in Figure 5.4(a) shows a RISC-V assembly code with LP and $pd = 2$ after using the compilation workflow in Figure 5.1(b). The loop control is represented in lines 3-9, while lines 10-13 are part of the loop body. Assuming a loop bound of 100, the *main* function runs 973 instructions. It is worth mentioning that the workflow conversion step (6) does not introduce additional instructions to the AS assembly code. However, we have observed that using the ACCEPT LP can increase the number of instructions to ensure that the loop body is not affected by the perforation.

Figure 5.4(b) presents the RISC-V assembly code of the loop from Figure 5.2(a) after applying the ILLP with $pd = 2$. Lines 8-11 are part of the loop body, line 12 stores the value of pd, and line 14 calls the approximated ($ADDX$) function. Line 5 in the function is the CF, which is carried out only when the result of the approximate instruction (line 3) is less than the current value of the loop control variable stored in register $a5$.

```
...
1   000101ac <main>:
...
2       101e0:    0180006f    j     101f8
3       101e4:    00140513    addi  a0,s0,1
4       101e8:    008535b3    sltu  a1,a0,s0
5       101ec:    00b484b3    add   s1,s1,a1
6       101f0:    00050413    mv    s0,a0
7       101f4:    03350063    beq   a0,s3,10214
8       101f8:    00147513    andi  a0,s0,1
9       101fc:    fe0514e3    bnez  a0,101e4
10      10200:    001a0a13    addi  s4,s4,1
11      10204:    00090513    mv    a0,s2
12      10208:    000a0593    mv    a1,s4
13      1020c:    594080ef    jal   ra,187a0
...
```

(a) Original LP.

```
...
1   0001018c <ADDX>:
2       1018c:    00050793    mv    a5,a0
3       10190:    02b5052b    addx  a0,a0,a1
4       10194:    00a7c463    blt   a5,a0,1019c <ADDX>
5       10198:    00b78533    add   a0,a5,a1
6       1019c:    00008067    ret

7   000101a0 <main>:
...
8       101c8:    00148493    addi  s1,s1,1
9       101cc:    00048593    mv    a1,s1
10      101d0:    c8898513    addi  a0,s3,-888
11      101d4:    1f0000ef    jal   ra,103c4
12      101d8:    00200593    li    a1,2
13      101dc:    00040513    mv    a0,s0
14      101e0:    fadff0ef    jal   ra,1018c <ADDX>
15      101e4:    00050413    mv    s0,a0
16      101e8:    fea950e3    bge   s2,a0,101c8
...
```

(b) LP with the ILLP technique.

Figure 5.4: RISC-V assembly code result of the LP. Source: author.

When analyzing a RISC-V assembly code snippet with LP, with $pd = 2$ and a loop limit of 100, for both techniques, we verify that the conversion step of the AS workflow does not introduce additional instructions to the assembly code. However, we noticed that using ACCEPT can increase the number of instructions to ensure that puncture does not affect the loop body. In this example, ILLP code executes 28.7% fewer instructions (693) than AS (973).

5.2.3 Experimental Setup

We have designed and conducted experiments to evaluate and validate the ILLP by comparing its accuracy, number of instructions, cycles, and energy (μWsec) to the LP in the ACCEPT (AS) and the original code (baseline - BL). The experiments were performed across a set of 13 applications[10]: BINARY_SEARCH (BS) [65], CONV1D [65], CONV2D [65], DIJKSTRA (DIJ) [23], FANNKUCH-REDUX (FAN) [22], FFT [61], FIBONACCI (FIB) [65], FLOYD-WARSHALL (FLOYD) [5], MEDIAN [65], MULT100X100 (MULT) [author], NBODY [22], PI [70], and SPECTRALNORM (SPECTRAL) [22].

We have identified the hotspot function in each application using GProf [25]. For applications such as BS, FAN, FFT, and SPECTRAL, which did not have a loop in their hotspot functions, we added perforation to the loop where the hotspot function is called to enhance the use of LP. In cases where the function had a nested loop, we selected the loop with the least impact on accuracy. Table 5.1 provides a summary of each application, including information on the presence of input and output vectors or matrices, the existence of LP in the hotspots function, whether there is a nested loop, and where the LP is placed (innermost (I) or outermost (O)).

Table 5.1: Applications summary. Input apps, Output apps, LP in Hotspots, Has nested loops, Position LP. Source: author.

APPS	**Input**	**Output**	**LP in Hot-spots**	**Nested loop**	**Position LP**
BINARY_SEARCH	✓	✓			
CONV1D	✓	✓	✓	✓	O
CONV2D	✓	✓	✓	✓	O
DIJKSTRA	✓	✓	✓		
FANNKUCH					
FFT	✓	✓		✓	O
FIBONACCI			✓		
FLOYD	✓	✓	✓	✓	I
MEDIAN	✓	✓	✓		
MULT100X100	✓	✓	✓	✓	O
NBODY			✓	✓	O
PI			✓		
SPECTRALNORM				✓	I

Our experiments used the infrastructure provided by the ACCEPT to insert LP in the code. We simulated all the applications in the SPIKEX RISC-V simulator, following the workflow illustrated in Figure 5.3. We evaluated the perforation on four degrees ($pd = 1$, 2, 4, and 8). Our comparison involved the ILLP results with the ACCEPT LP and the original (BL) code. Importantly, we inserted perforations on the same loops of each application for both the ACCEPT and the ILLP approaches.

We gathered all the necessary information about the application's performance using Prof5. We emphasize here that the simulation of the applications was carried out by SPIKE, which follows the design of a 5-stage RISC-V. Prof5 uses the executable file generated by the SPIKE. Prof5 allowed us to create detailed profiles of RISC-V programs from the SPIKEX log. The profile data generated by Prof5 is the number of cycles, instructions, power, energy, and average power per cycle. We also customize the energy model by entering new instructions and creating custom ones. Approximate instructions like **addx** have also been added to the power model, with an energy savings of 13.92% [8] compared to the default *add* custom instruction by Prof5.

[10]Applications are available for download at: https://github.com/lscad-facom-ufms/ILLP-Loop_Perforation

Table 5.2 presents the results of output, number of instructions, number of cycles, and energy (μWsec) of the original applications (BL). It may be observed that the FAN application has the highest number of instructions among all applications (1,072,887,156) and the highest number of cycles. The FAN, FFT, NBODY, and SPECTRAL applications have the highest energy values.

Table 5.2: Output, number of instructions, number of cycles, and energy. Source: author.

Applications	Output	Instructions	Cycles	Energy (μWsec)
BINARY_SEARCH	500	2,259,486	2,993,337	56.89
CONV1D	1,002	5,144,208	7,180,610	155.20
CONV2D	100	207,223	277,491	5.67
DIJKSTRA	100	1,087,879	1,380,358	27.86
FANNKUCH	38	1,072,887,156	1,274,710,666	25,210.72
FFT	1,024	541,575,553	700,790,974	14,111.47
FIBONACCI	102,334,155	82,678	106,175	2.10
FLOYD	324	253,433	316,767	6.48
MEDIAN	1,000	2,232,393	2,879,240	53.48
MULT100X100	10,000	7,496,393	8,599,491	185.89
NBODY	-0.1690876	318,869,815	422,378,381	8,565.30
PI	3.1480	24,378,464	32,613,086	654.97
SPECTRALNORM	1.2741930	331,087,512	447,130,079	8,932.78

Column Output represents the original (Baseline - BL) output of each application. BS, CONV1D, CONV2D, DIJ, FFT, FLOYD, MEDIAN, and MULT have matrices or arrays elements as outcomes. BS original output has 500 elements, CONV1D has 1,002, CONV2D has 100, DIJ has 100, FFT has 1,024, FLOYD has 324, MEDIAN has 1,000, and MULT has 10,000 elements. FAN, FIB, NBODY, PI, and SPECTRAL have a single number of the output result. When using the LP, we performed the comparison element by element. For example, the MULT application with LP and $pd = 1$ has an output of 5,000 elements, an accuracy loss of 50%.

The precision metric is the first to consider when comparing approximate optimization techniques. We calculated the relative error ($RE = \frac{|AO-BO|}{|BO|}$) of the LP and comparing them to the output results of the baseline (BO) and the Approximate Output (AO). A larger RE indicates greater imprecision.

5.2.4 Results and Discussion

Table 5.3 presents the RE of the applications in each of the techniques. Note that with $pd = 1$, the BS application presented an RE of 0.5000, which is consistent with the drilling that was drilled at 50% of its BL. As the value of pd increases, the RE also increases, causing a significant precision loss, as in CONV1D, which presents an RE of 0.9960 with $pd = 8$.

Figure 5.5 presents the percentage reduction of AS and ILLP instructions compared to BL with $pd = 1$ and $pd = 8$. The reference (zero) represents the instructions of the BL application. Upside-down columns indicate that the number of run instructions of the LP is greater than the BL application. The circled line shows the error percentage of

Table 5.3: Relative Error. Source: author.

Applications	pd = 1	pd = 2	pd = 4	pd = 8
BINARY_SEARCH	0.5000	0.7500	0.9360	0.9960
CONV1D	0.5000	0.7495	0.9371	0.9960
CONV2D	0.5000	0.7000	0.9000	0.9000
DIJKSTRA	0.8500	0.8600	0.8600	0.9100
FANNKUCH	0.5000	0.7368	0.7632	0.7632
FFT	0.5000	0.7500	0.9375	0.9961
FIBONACCI	0.9999	1.0000	1.0000	1.0000
FLOYD	0.4722	0.6420	0.7253	0.7438
MEDIAN	0.5000	0.7500	0.9360	0.9960
MULT100X100	0.5000	0.7500	0.9300	0.9900
NBODY	0.0002	0.0002	0.0002	0.0002
PI	0.2379	0.3784	0.8420	0.9885
SPECTRALNORM	0.6424	0.6557	0.6579	0.6579

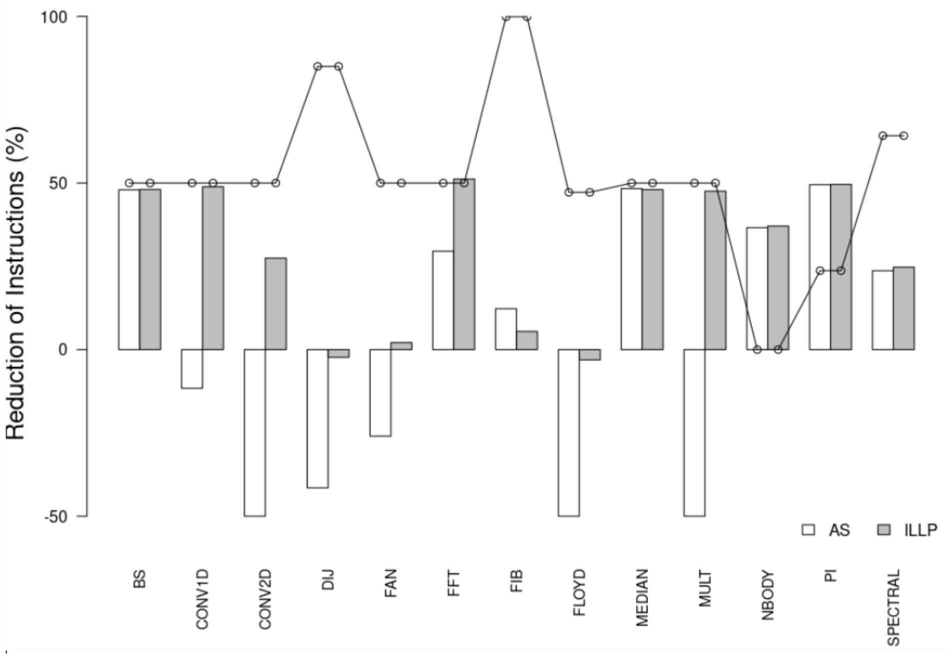
each application and technique, allowing us to evaluate the instruction reduction and the impact on the RE. There is a decrease in the number of instructions in most applications as *pd* increases, just as expected. There is a decrease in the number of instructions in 11 out of 13 applications (highlighted) by the ILLP compared to the BL.

Application DIJ had no significant reduction in instructions over the *pd*. This application has loops with a small number of iterations so that the *pd* level does not impact the number of generated instructions. The ILLP has shown increased instructions compared to BL for DIJ and FLOYD applications. Specifically, the DIJ application had an increase of 2.30% in ILLP_1, while AS_1 had an increase of 41.48%, where _1 is the value of *pd*.

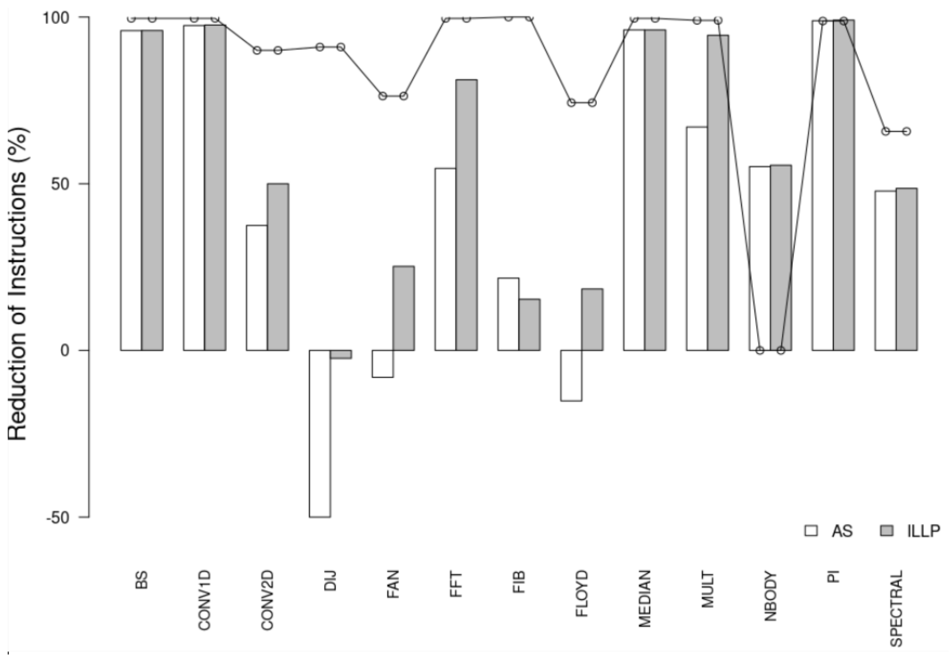
The AS presents an instruction increase in 6 applications compared to BL. The MULT application has an unexpectedly large number of instructions. One explanation for this behavior could be the ACCEPT workflow that makes excessive calls to the internal functions *_mulsi*3 (5016 times) and *_muldi*3 (1,000,280 times), while the ILLP calls the *_mulsi*3 function 16 times and the function *_muldi*3 is not used. Application CONV2D shows an increase of 98.66%. In contrast, CONV2D with the ILLP reduced 27.46%. The FFT application presents a decrease of 51.19% (ILLP_1) compared to the BL code. Meanwhile, the AS technique reduced 29.55% (AS_1) compared to BL.

Figure 5.5(b) shows a significant reduction in instructions with $pd = 8$ in most applications. The FAN application achieves a reduction of 25.22% with ILLP but an increase of 8.08% with AS. On the other hand, the FLOYD application shows an increase in the number of instructions with AS (for all *pd*) but an instruction decrease using ILLP with $pd = 2, 4, 8$.

Figure 5.6 presents the cycle improvement percentage for each application using $pd = 1$ and $pd = 8$. The reference (zero) represents the cycles of the BL application. Figure 5.6(a) shows both AS and ILLP able to achieve a similar cycle reduction for the MEDIAN application, with values of 48.39% and 48.06%, respectively. CONV2D, DIJ, FAN, FLOYD, and MULT applications showed an increase in the number of instructions, leading to an increase in the number of cycles when running on AS. On the other



(a) $pd = 1$.

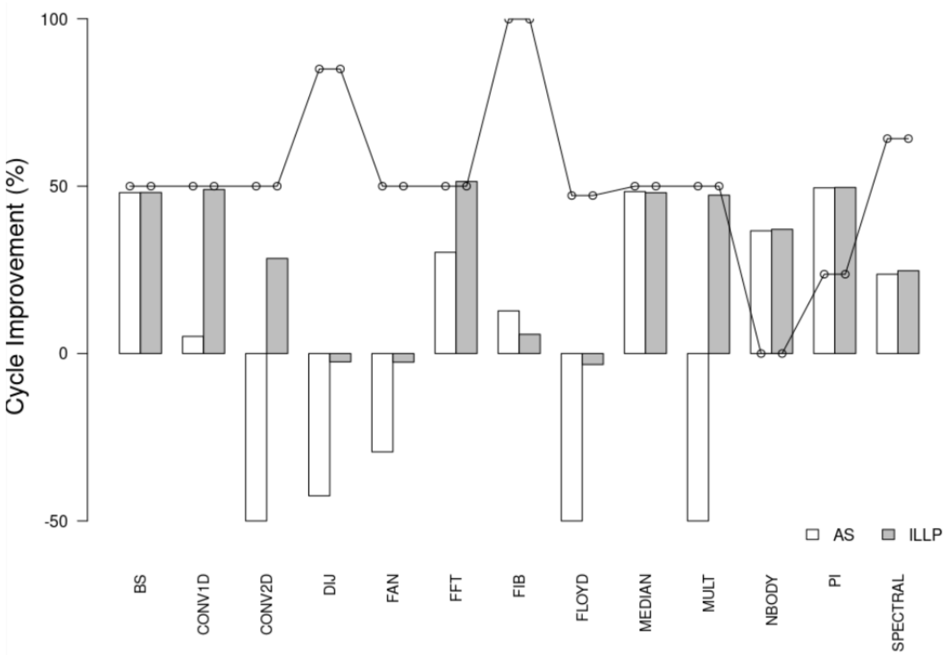


(b) $pd = 8$.

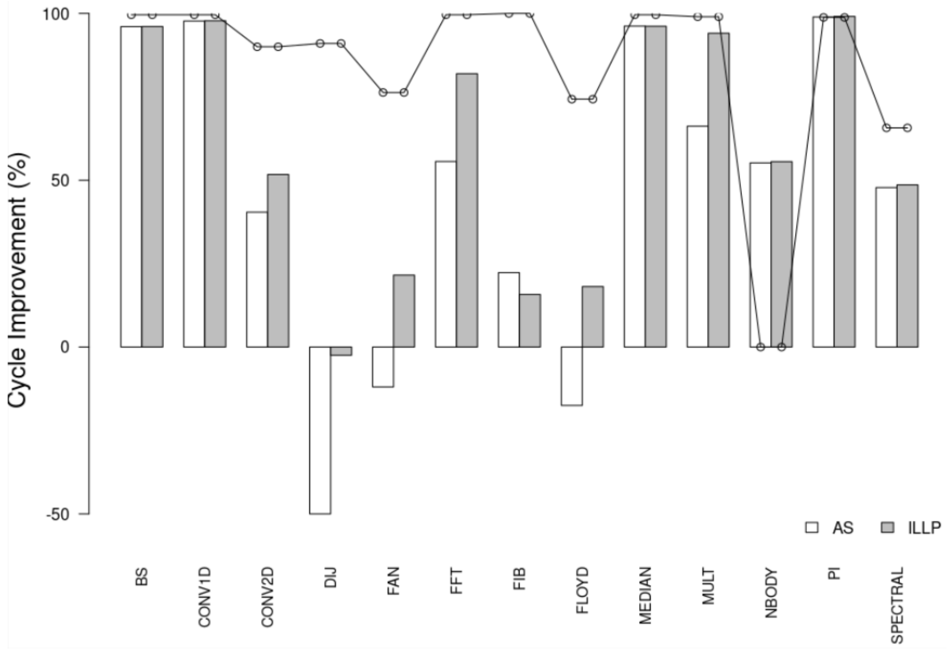
Figure 5.5: AS and ILLP instructions percentage reduction for each application. Source: author.

hand, most of the applications that used ILLP showed cycle improvements. The FFT application, in particular, achieved the most significant cycle reduction (51.40%). The cycle improvement percentage with $pd = 8$ (Figure 5.6(b)) is quite impressive. Notably, cycle reduction gains are greater than 50% and even reach 99.10% for the PI application.

The BS, CONV1D, MEDIAN, NBODY, PI, and SPECTRAL applications have similar values, with a slight advantage for the ILLP.



(a) $pd = 1$.



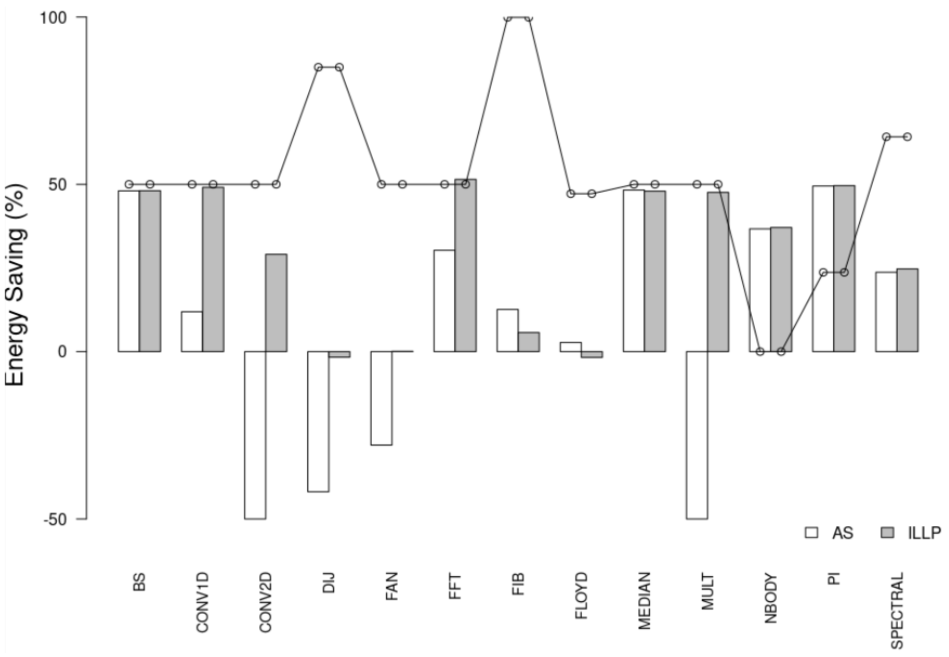
(b) $pd = 8$.

Figure 5.6: AS and ILLP cycle improvement percentage for each application. Source: author.

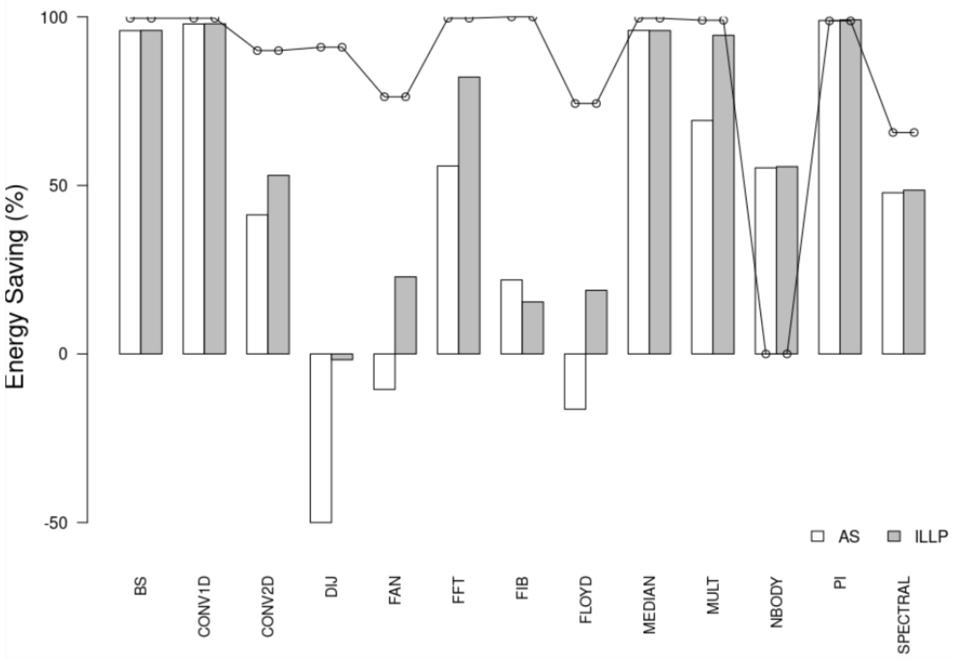
In Figure 5.7, one can see the percentage of energy savings for each application using AS and ILLP with $pd = 1$ and $pd = 8$. Figure 5.7(a) highlights that the ILLP provides better energy reduction values. For instance, the CONV1D application achieved a reduction of 49.11% with ILLP and $pd = 1$ and 11.93% with AS and $pd = 1$. Figure 5.7(b) presents the percentage of energy saving with $pd = 8$, which shows similar behavior to the instructions and cycle metrics. The AS and ILLP results were quite akin in 6 out of 13 applications, whereas ILLP achieved better results in the other 6 applications.

Figure 5.8 presents the boxplot for the reduction in instructions, cycles, and energy metrics, in both techniques, for all pd values $(1, 2, 4, 8)$. Each column presents the results by ordering the lowest to the highest value and organizing the data into quartiles and median. The results indicate that the ILLP outperforms the AS in all metrics and pd. For instance, with a $pd = 1$, ILLP presents an average instruction reduction of 22.55%, whereas the AS only shows a reduction of -1.52%. This trend continues for cycle and energy, where ILLP reaches a reduction of 23% with $pd = 2$ in the number of cycles. However, it is important to note that most results with $pd = 8$ are the same for both techniques, indicating that this pd may be an upper bound for general performance gains (instructions and cycles) and energy savings.

The results brought from the ILLP have proven to be effective in improving several aspects of performance while maintaining accuracy levels equal to the AS. Specifically, the ILLP approach was able to achieve a significant reduction in the number of instructions, cycles, and energy consumption for various applications compared to the BL. For example, the PI application with a $pd = 2$ achieves a 74.61% reduction in the number of instructions, while the FFT application with $pd = 1$ showed a 51.40% reduction in the number of cycles. The PI application using ILLP and $pd = 2$ has 74.49% energy saving compared to the original baseline code.

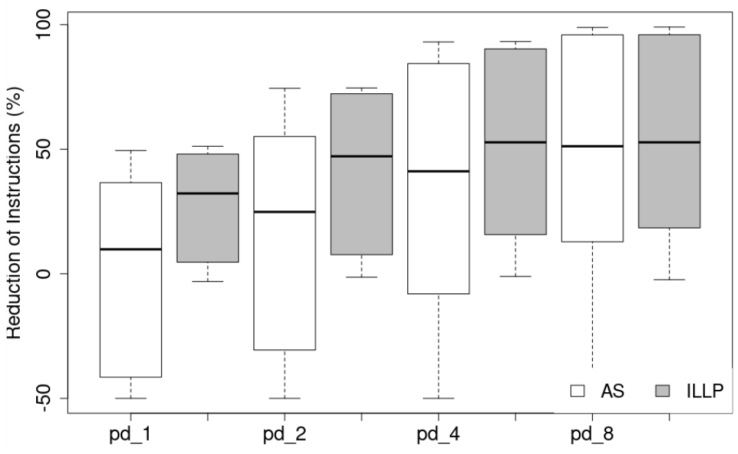


(a) $pd = 1$.

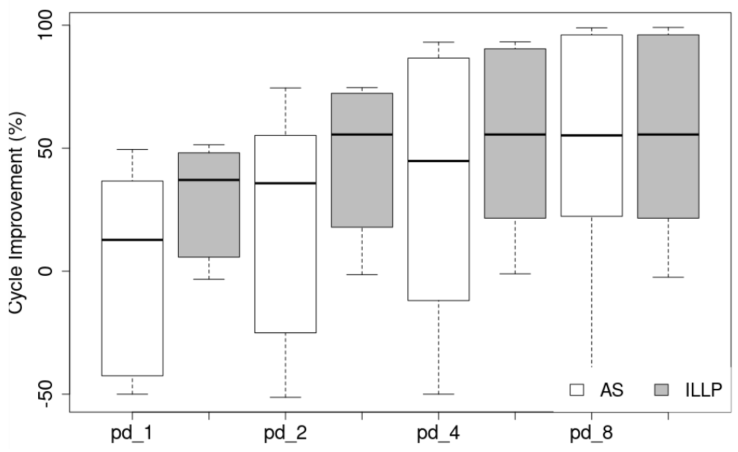


(b) $pd = 8$.

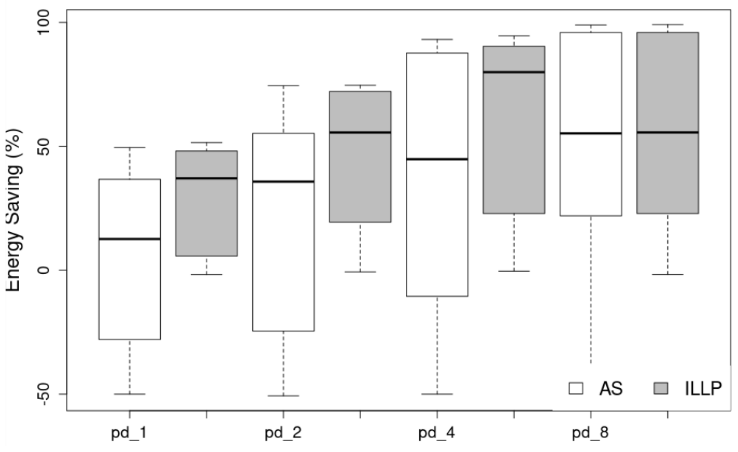
Figure 5.7: AS and ILLP energy saving percentage for each application. Source: author.



(a) Reduction of Instructions.



(b) Cycle Improvement.



(c) Energy Saving.

Figure 5.8: Percentage reduction for instructions, cycles, and energy by technique. Source: author.

5.3 Instruction-Level Approximate Function - ILAF

This section introduces the Instruction-Level Approximate Functions (ILAF), which incorporates floating point (FP) approximate instructions into FastApprox's mathematical functions such as sine, cosine, tangent, exponential, and logarithmic.

The Instruction-Level Approximate Function design is presented in subsection 5.3.1. Subsection 5.3.2 describes the applications and the experiments carried out to evaluate and validate the approximation technique. The results and discussion are presented in subsection 5.3.3.

In the era of high-performance computing, the quest for greater efficiency, reduced power consumption, and accuracy in arithmetic operations is a constant challenge, especially in floating point and fixed point computing, where balancing accuracy, performance, and power consumption is crucial. The innovative works [78], [71], and [29] delve into these areas, focusing on pioneering techniques for keeping the accuracy while optimizing the runtime and power consumption of floating point, mixed-precision, and fixed point operations.

The three previous work share a common focus on optimizing arithmetic operations to control accuracy and improve performance and energy efficiency in computing systems. They explore techniques that balance the accuracy with efficient performance, through modeling and mitigating noise in floating point computing, the dynamic flexibility of floating point, or the efficient implementation of fixed point trigonometric functions using the algorithm CORDIC [75]. Each work contributes to enhance computational efficiency, regarding power consumption, hardware resource usage, or arithmetic accuracy, demonstrating the practical implications of these approaches in real-world computing systems.

There are yet other approaches using AC techniques but acting straightly on the mathematical functions accuracy and performance instead of looking only at the operations. The authors in [46], [11], [50], and [66] propose techniques and tools for approximating mathematical functions, all aimed at optimizing computational efficiency while maintaining precision.

When comparing the proposal of this work with those presented previously, one may observe that, unlike other approaches, the proposal seeks to add an instruction level of approximate computing by identifying and replacing accurate (non-approximate) mathematical operations with approximate floating point instructions. Similar to the related research, our technique also works on mathematical functions of an application but is practical to be applied whether as a first approximation step on accurate (non-approximate) functions or even as a second approximation step on software-level approximated mathematical functions.

5.3.1 Instruction-Level Approximate Functions Design

As part of the ILAF design workflow, we adopted a three-level space exploration (DSE) approach (Figure 5.9). At the first level, we focused on how to explore the design of new floating point (FP) approximate instructions (step 1). The focus is to analyze all the set of operations that comprise a complex floating point and replace some of these operations by their approximate versions to improve performance and/or power consumption while keeping controlled accuracy levels. After having new approximate floating point instructions, the second level (step 2) works in the source code of mathematical functions, identifying non-approximate FP instructions that could be replaced with approximate ones. Again, the building of the new approximate mathematical function is guided by the gains in performance and reduced power consumption and looking at the accuracy losses. At the third level (step 3), we aimed to identify and evaluate the functions in the application's source code and determine which ones could be replaced with the approximate mathematical functions. Step 1 was presented in chapter 4. Step 2 will be presented in the following subsection 5.3.1.1. The mapping of approximate functions in the source code of some application (step 3) is presented in Section 5.3.2.

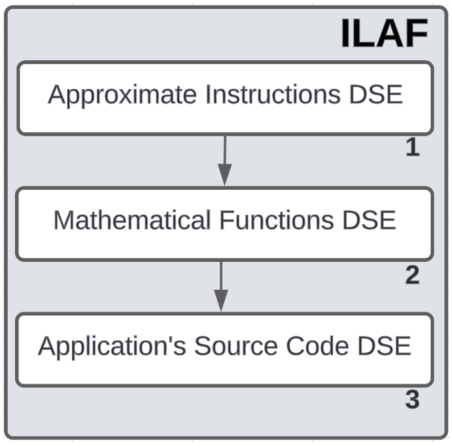

ILAF
Approximate Instructions DSE
1
Mathematical Functions DSE
2
Application's Source Code DSE
3

Figure 5.9: ILAF design workflow. Source: author.

ILAF has adopted the RISC-V instruction-set as the reference ISA to design and evaluate new approximate instructions. The FP instructions in RISC-V use integer operations to manipulate exponents, mantissa, shift, and rounding. Our strategy to design new approximate FP instructions was to exchange some original (accurate) integer operations for approximate integer instructions. The decision-making on which integer operations and instructions should be replaced was done after performing a DSE on those operations and evaluating the impact in accuracy, power, and performance.

The approximate integer operations adopted in this work were first introduced in Catelan et al. [9]. The approximate integer addition follows the RISC-V instruction-set format and is implemented using the InXA1 [1] approximate adder. The approximate integer subtraction was designed and implemented using the APSC4 approximate subtractor [24, 8].

5.3.1.1 Approximate Functions Design

The second step of the design and implementation of ILAF is built on the top of the ACCEPT compiler using the following mathematical functions of the FastApprox library (FAl): SIN, COS, TAN, EXP, LOG, and LOG$_2$. When the user starts the compilation process, ACCEPT presents a list of mathematical functions able to apply approximate optimization. The user may choose 3 options: non-approximate, *fast*, and *faster*. The first option uses the non-approximate mathematical functions from the *math.h* library. *fast* and *faster* options apply versions of the approximated functions from the FastApprox library [53, 41].

We redesign the mathematical functions of FastApprox (*fast* version) by replacing some of the original FP instructions to the new approximate instructions. As an example, Figure 5.10(a) presents a code snippet of the FastLog$_2$ function (log$_2$ approximate mathematical function of FAl). In the FastLog$_2$ function, replacing the original FP division operation/instruction (line 6) to the new **FDIVX** approximate FP instruction (Figure 5.10(b)) could meet the compromise between accuracy, power, and performance. The preliminary experiments improved the number of cycles, instructions, and power, keeping the same accuracy of the original FastLog$_2$ function.

```
1   static inline float fastlog2 (float x)
2   {
3     ...
4     return
5       y-124.22551499f-(1.498030302f*mx.f)-
6       (1.72587999f / (0.3520887068f+mx.f));
7   }
```

(a) FastLog$_2$ function.

```
1   static inline float fastlog2 (float x)
2   {
3     ...
4     return
5       y-124.22551499f-(1.498030302f*mx.f)-
6       FDIVX(1.72587999f,(0.3520887068f+mx.f));
7   }
```

(b) FastLog$_2$ with **FDIVX** instruction.

Figure 5.10: Code snippet of the FastLog$_2$ function. Source: author.

Each mathematical function implemented in the *fast* version of FAl was submitted to the same design procedure: original FP instructions were replaced to approximate FP instructions. The decision-making on the number of FP instructions to be replaced and where (in the function's source code) to be placed has considered the results of a comprehensive set of DSE experiments. Each experiment is a scenario, a combination of candidate FP instructions, where the approximate instructions were placed in the functions. Each possible scenario has been evaluated looking at the accuracy, performance increase, and power consumption reduction.

From all the evaluated functions, the TAN function had the most significant number of scenarios (10). The chosen scenario was based on the best (lesser) mean absolute percentage error (MAPE) value:

$$MAPE = \frac{1}{n}\sum_{i=1}^{n}|\frac{E_i - A_i}{E_i}| \tag{5.1}$$

where:

- n is the sample size;
- E_i is the original (accurate) value/output;
- A_i is the approximate value/output.

Table 5.4 displays the scenario (the amount of approximate instructions) and its MAPE value for each mathematical function. The best scenario of the SIN function had the most significant number of approximate FP instructions (three **FADDX** and three **FMULX**). The COS and TAN functions had one **FADDX** instruction, the LOG and LOG$_2$ functions had one **FDIVX** instruction and the EXP function had one **FSUBX**.

Table 5.4: Selected approximate instructions and MAPE of each mathematical function. Source: author.

Functions	**FADDX**	**FSUBX**	**FMULX**	**FDIVX**	**MAPE**
SIN	3		3		0.15
COS	1				0.03
TAN	1				0.14
EXP		1			0.00
LOG				1	0.00
LOG$_2$				1	0.00

5.3.2 Experimental Setup

ILAF has been evaluated and tested on six different applications that apply mathematical functions[11]: CUBIC [40], FFBENCH [76], FBENCH [76], BLACKSCHOLE [15], LOG2, and IDENTITY_LOG2.

[11]Applications are available for download at: https://github.com/lscad-facom-ufms/ILAF-ApproxFunction

The type of mathematical functions present in each of the applications, the number of functions, the number of functions replaced by ILAF (number in parenthesis), and the number of executions (bold) are summarized in Table 5.5. For example, the CUBIC application presents three cosine functions, each one run once, but only two were replaced by ILAF.

Table 5.5: Summary of mathematical functions in each application. Source: author.

Applications	**SIN**	**COS**	**TAN**	**EXP**	**LOG**	**LOG$_2$**
CUBIC	-	3 (2) **3**	-	-	-	-
FFBENCH	2 (2) **64**	-	-	-	-	-
FBENCH	6 (0) **12**	2 (2) **2**	1 (0) **4**	-	-	-
BLACKSCHOLE	-	-	-	3 (3) **6**	4 (4) **4**	-
LOG2	-	-	-	-	-	1 (1) **50**
IDENTITY_LOG2	-	-	-	-	-	2 (2) **5000**

Figure 5.11 illustrates the ACCEPT compiler approximate workflow where a user can choose between the FAl or the ILAF approaches. Given a source code (step 1), the ACCEPT compiler will analyze the code and identify all mathematical functions able to approximation (step 2). In step 3, the user will choose the approximate function model. When choosing ILAF (step 4), the mathematical functions of the source code will be replaced by the ones from the FAl with approximate instructions. In step 5, the mathematical functions will be from the FastApprox library. Step 6 translates the assembly to machine code RISC-V. Step 7 is the code simulation using the SPIKEX simulator. SPIKEX is built on top of the SPIKE simulator and it supports all the approximate instructions we have designed in this work.

5.3.3 Results and Discussion

The results and discussion presented in this section are based on a set of applications organized into three different versions: the baseline (BL) where the applications have non-approximate mathematical functions; the FastApprox (FAl), where the applications had the approximate functions from the FastApprox library, and the ILAF approach where the applications used ILAF approximate functions built on the top of the FastApprox library. The experiments evaluated accuracy, running cycles, and power (μW) comparing the original (non-approximate) application to the FAl and ILAF.

The relative error ($RE = \frac{|AO-BO|}{|BO|}$) is the metric adopted to evaluate the application accuracy when using the approximate functions optimization. RE is calculated from the baseline output results (BO) representing the non-approximate applications. The

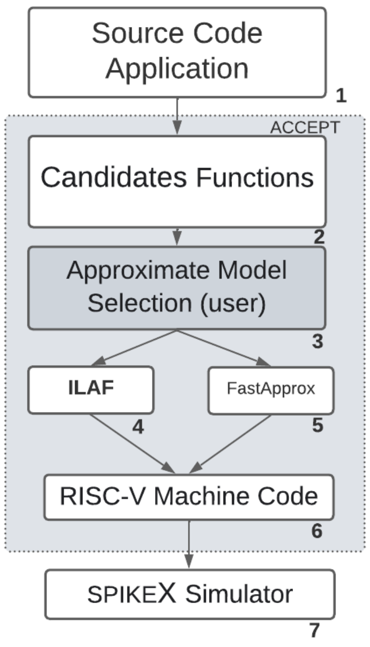


Figure 5.11: ACCEPT approximate functions workflow. Source: author.

Approximate Output (AO) is the output result of the ILAF or the FAl approaches. In the experiments, only two applications (CUBIC and FBENCH) had a maximum $RE = 0.3$ (30%). The remaining applications presented $RE = 0$, meaning no lack of accuracy in using the approximate functions approaches.

The results have shown that the approximate techniques directly impact applications highly dependent on the math function. Other applications may have limited benefits when looking at the performance and power consumption of the whole running code. Table 5.6 shows the percentage improvement of instructions, power, and cycles in each application by using ILAF over BL.

Table 5.6: Percentage improvements of the applications running the ILAF approach compared to the BL. Source: author.

Applications	**Instructions**	**Power**	**Cycles**
CUBIC	94.77	94.18	95.19
FFBENCH	0.02	0.01	0.01
FBENCH	63.10	56.83	76.01
BLACKSCHOLE	88.17	88.71	90.81
LOG2	49.08	23.38	66.62
IDENTITY_LOG2	76.95	77.16	78.39

An improvement of 94.77% in the number of instructions can be seen in the CUBIC application compared to the baseline (BL) version, meaning that 94.77% fewer instructions in the cosine function using our ILAF approach. Figure 5.12 sketches the functions called by the cosine function in each version of the application. One may notice that there is a large difference between instructions (in parenthesis) of the cosine function in Figure 5.12(a) and Figure 5.12(c). This discrepancy was caused by the additional functions brought to the application by the *math.h* library. In Figure 5.12(a), CUBIC runs three calls to cosine that, in turn, calls three other functions (__ieee754_rem_pio2, Kernel_sin, and Kernel_cos). Figure 5.12(b) represents the CUBIC using the approximate cosine function (fast_cos) from the FastApprox library. When using ILAF (Figure 5.12(c)), CUBIC performs one calls to fast_cos and two call to fast_cos1, which is the cosine from ILAF.

An analysis was performed regarding the applications code region where ILAF was applied. The goal was to observe how many instructions (at runtime) of the math functions of FastApprox could be impacted by ILAF. The code coverage metric is presented in equation (5.2), where T_{ILAF} is the number of ILAF approximate instructions applied in a mathematical function. (T_{I}) is the total of instructions of a given FastApprox function. The power improvement ($Power_{\text{imp}}$) is also calculated in equation (5.3). Each instruction of the set of replaced instructions (I_{R}) of a given function is multiplied by its power ($Power_{\text{i}}$), and I_{Approx} is the set of FP approximate instructions of ILAF that replaces the instructions of I_{R}.

$$Coverage = \frac{T_{\text{ILAF}}}{T_{\text{I}}} \tag{5.2}$$

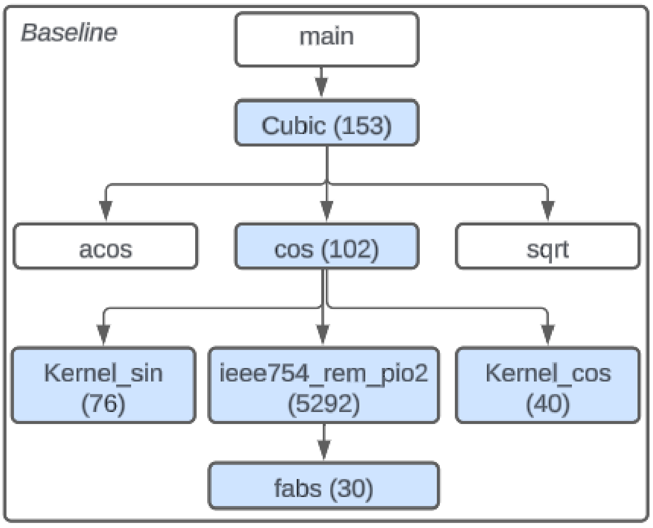
$$Power_{\text{imp}} = \frac{\sum_{i}^{I_{\text{R}}} i \times Power_{\text{i}}}{\sum_{j}^{I_{\text{Approx}}} j \times Power_{\text{j}}} \tag{5.3}$$

Table 5.7 displays the coverage, power improvement, and speedup cycle comparing ILAF to FastApprox. The results are based on the instructions of the mathematical functions that are impacted by our approach.

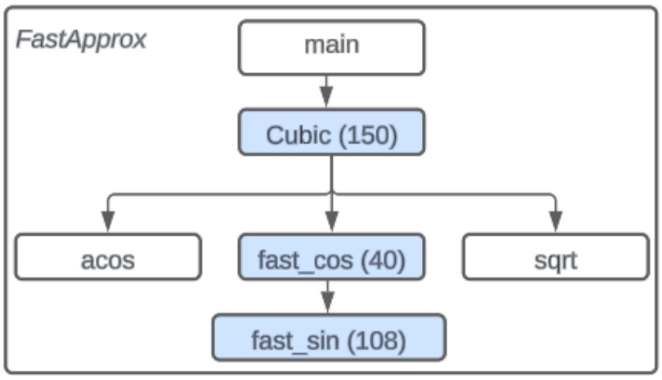
Table 5.7: Coverage and power improvement of ILAF compared to FAl. Source: author.

Applications	**Coverage (%)**	**Power Improvement**
CUBIC	7.50	1.58
FFBENCH	30.60	1.03
FBENCH	1.30	1.04
BLACKSCHOLE	19.00	3.02
LOG2	4.00	1.03
IDENTITY_LOG2	3.80	1.03

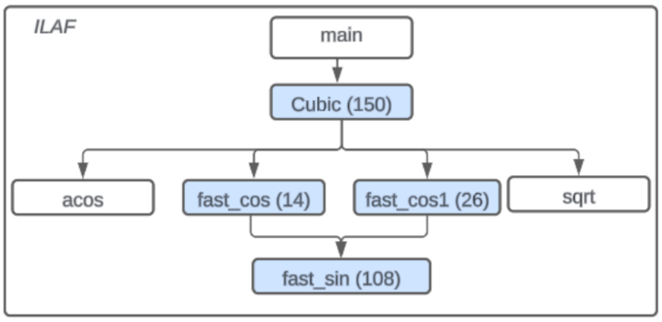
The coverage achieved in the FFBENCH application means that the ILAF approximate instructions covers 30.60% of all instructions (at runtime) of the FastApprox



(a) Cosine function call flow in the *math.h* library (baseline).



(b) Cosine function call flow in the FastApprox library.



(c) Cosine function call flow in the ILAF.

Figure 5.12: Call flow of the cosine function of CUBIC using Baseline, FastApprox, and ILAF. Source: author.

sine function. Specifically, FFBENCH uses two sine functions which are run 64×. Each sine function of ILAF has three **FADDX** and three **FMULX** instructions, so that 384 ($6 \times 64 = 384$) instructions, at runtime, were replaced.

The BLACKSCHOLE application had the largest number of functions impacted by the ILAF technique (three exponentials and four logarithms). The larger use of the ILAF approximate functions is the responsible for the best power improvement (3.02×) among the applications.

Table 5.8 shows a code snippet of the transit_surface function that is part of the cosine function of FastApprox. Table 5.9 depicts the transit_surface function of the ILAF approach. Both Tables highlight some floating point instructions, the number of running instructions, power, and cycles. The FBENCH application source code has 6 sine functions, 2 cosines, and 1 tangent, but the best results are found when ILAF is applied only on the cosine functions. Both functions have the same number of instructions (72) in the code snippet, but ILAF reduces the total power consumption (calculated by multiplying the number of instructions by power). FastApprox had 12 *fadd.s* instructions, while ILAF replaced 8 with *faddx.s*, resulting in 4 remaining *fadd.s* instructions.

Table 5.8: Excerpt from the code of the "transit_surface" function (FBENCH application) using the *FastApprox* library. Source: author.

transit_surface:	**Instructions**	**Power**	**Cycles**
⋮	⋮	⋮	⋮
fadd.s	12	3.36	2
fdiv.s	4	3.71	2
flt.s	8		
c.bnez	20		
ret	16	1.82	2
c.j	12		

Table 5.9: Excerpt from the code of the "transit_surface" function (FBENCH application) using *ILAF*. Source: author.

transit_surface:	**Instructions**	**Power**	**Cycles**
⋮	⋮	⋮	⋮
fadd.s	4	3.36	2
fdiv.s	4	3.71	2
flt.s	8		
c.bnez	20		
faddx.s	8	3.19	2
ret	16	1.82	2
c.j	12		

Experiments were performed on applications with mathematical functions, observing the impacts of the ILAF approach and comparing them to non-approximated applications (baseline) and an approximate approach using the FastApprox library. The

CUBIC application using ILAF achieved a power improvement of 94.18% and a reduction of 95.19% in cycles compared to the baseline. The BLACKSCHOLE application using ILAF achieved a power improvement of 3.02× with only 19% code coverage. These results showed that ILAF significantly improved performance and energy consumption compared to non-approximated and even software-level approximate versions of the applications. Comparison with the FastApprox library revealed that our approximate instruction approach provides power improvement for all applications while maintaining the accuracy levels achieved by the software technique (FastApprox).

The adoption of ILAF poses a challenging task to attain better performance, power improvement, and keeping accepted levels of accuracy. The benefits of ILAF relies on the right decision-making on approximate instruction replacement in the mathematical functions. This challenge comes from the difficulty of performing the design space exploration, given that this task has an exponential complexity. This work applied design space exploration in three levels: the design of the approximate floating point instructions, the design of the approximate functions, and the design of the approximate applications. We had to explore the space of the floating point instructions to choose the proper replacements of the original floating point instructions to the approximate ones. Also, it was necessary to perform the design space exploration in the mathematical functions to identify the instructions that the approximate ones should replace. Lastly, a third exploration step was performed in the applications to evaluate which mathematical functions should be replaced by their approximate counterparts.

5.4 Final Remarks

This chapter presented the design and results of approximate computing approaches integrating software and hardware techniques. The techniques are the Instruction-Level Loop Perforation (ILLP) and Instruction-Level Approximate Functions (ILAF). The results demonstrated that these techniques effectively optimize performance and reduce energy consumption in several applications while maintaining accuracy within acceptable limits. The detailed analysis of the techniques revealed that the strategic combination of software and hardware methods can lead to a better computational resources usage, especially in scenarios where absolute accuracy is not critical. ILLP proved to be particularly advantageous in identifying and optimizing critical loops. At the same time, ILAF stood out in replacing mathematical functions with instructions-approximated versions, offering savings in power consumption. These advances reinforce the potential of approximate computing as a viable solution to modern systems' performance and energy consumption challenges. However, there is room for future improvements and expansions, such as applying ILLP and ILAF techniques in other AC techniques, developing automated tools for identifying approximation opportunities, and exploring other hardware components that can be approximated.

Chapter 6

Conclusions

Approximate computing presents itself as an innovative and promising approach to address the physical challenges of the end of Dennard scale. It allows optimizations in terms of energy consumption, area usage, and performance through controlled trade-offs in computational precision. This thesis investigated AC techniques, from hardware to software, offering research contributions on both levels. Approximate arithmetic circuits and techniques for applying approximate computing in software were designed and evaluated, such as instruction-level loop perforation (ILLP) and approximate arithmetic functions (ILAF).

As a first general research contribution, this thesis comprehensively reviews approximate arithmetic circuits. The research showed advances in proposing new mixed circuits that accommodate accurate and non-accurate components to control output inaccuracy while improving power consumption. Those circuits worked as building blocks for introducing specific instructions for approximate computing in the RISC-V instruction set. Those instructions, in turn, were used to design new approaches for well-known software techniques: loop-perforation and approximated mathematical functions.

6.1 Specific Conclusions

In the Approximate Computing chapter (chapter 2), a detailed review of the AC state-of-the-art was conducted, addressing its techniques and categorization at software, hardware, and architecture levels. The methodology consisted of a critical analysis of the existing literature, with the organization of the techniques into categories based on different authors. Despite the number of AC taxonomies, there was a gap in the representation of solutions that adopted hardware and software techniques. We have filled this gap up by proposing a new taxonomy encompassing hardware/software techniques regarding the most recent advances in the field.

The work on AC circuits ended up designing and implementing new mixed arithmetic circuits and evaluating their features on FPGA platforms. The results brought evidence that mixed circuits can be an option to balance energy efficiency and accuracy. The experience in designing and evaluating approximate circuits was the motivation to look at the design of new instructions using those circuits. To this end, the RISC-V ISA was extended to support new approximate instructions. The methodology included building new arithmetic instructions (integer and floating point) and adjusting the compiler, toolchain, and simulation tools, thus offering a toolset that automates the approximate code-generation process.

From the new approximate RISC-V instruction-set, we advanced a step further by integrating techniques at software and hardware levels, focusing on instruction-level loop perforation (ILLP) and approximate functions (ILAF). Integrating these techniques in an automated way contributed to evaluating those AC techniques in a larger set of applications.

6.2 Limitations of the Study

During the development of the work, there were several technical challenges:

- Lack of design tools. Sometimes, existing tools were outdated or adopted discontinued resources;
- Constrained applications set. As the tool's outdated features required applications to be highly specific and adaptable to different contexts;
- Extensive data exploration (at up to three levels: instruction, mathematical function, and application) was required to develop approximate mathematical functions.

Some studies were conducted during the work to find better applicability for approximate arithmetic circuits and approximate instructions. One of these studies was the design of an approximate processor using the NEORV32 RISC-V[48] VHDL design. The first results achieved satisfactory accuracy results; however, there was a lack of tools for evaluating other metrics. Another study performed was the adoption of approximate floating point instructions in the Taylor's mathematical functions[84], which presented many possibilities of exchanging accurate instructions for approximate ones, thus requiring a great effort in exploring of the design space.

Another technical challenge in this work was using the ACCEPT framework to generate the RISC-V code. When carrying out our tests with ACCEPT, converting the x86 object file to an RISC-V object file was necessary. Some applications showed a non-compatible increase in the number of instructions; others raised errors in variables that did not convert, thus constraining the applications that could be evaluated.

6.3 Thesis Contributions

This work contributed to an in-depth study of approximate computing, focusing on hardware and software techniques. New methods of designing approximate arithmetic circuits and instructions were exploited, thus combining hardware and software techniques. Among the specific contributions, the following stand out:

- Proposal for an expanded taxonomy, introducing the category of combined software and hardware support for AC techniques;
- Characterization of a wide range of approximate arithmetic circuits, showing impacts on accuracy, area usage, frequency, and power dissipation;
- New approximate mixed designs to improve (minimize) the trade-off between relative error and area usage;
- Physical experiments in FPGA platform, revealing that approximate circuits can take advantage of prototyping technologies to achieve better area usage and energy savings;
- Design of approximate integer and floating point instructions incorporated into the RISC-V instruction set;
- A new approximate technique that uses approximate instructions in Loop Perforation;
- A new approximate technique that uses approximate integer and/or floating point instructions in mathematical functions;
- An infrastructure for approximate experimentation using hardware/software tools: the SPIKE simulator with new approximate instructions, the ACCEPT compiler to use the approximate RISC-V instructions, and the Prof5 profiling tool for power estimation of those instructions.

The following articles were published and highlight the advances and results from the research:

- Santos, R.; Sonahata, R.; Krebs, C.; Catelan, D.; Duenha, L; Segovia, D.; Santos, M.. *Exploração do projeto de sistemas baseados em GPU ciente de dark silicon.* In: Simpósio em Sistemas Computacionais de Alto Desempenho (SSCAD), 20. , 2019, Campo Grande. Porto Alegre: Sociedade Brasileira de Computação, 2019 . p. 358-369. doi: https://doi.org/10.5753/wscad.2019.8682 [62];
- Catelan, D.; Santos, R.. *Exploração do espaço de projeto em arquiteturas heterogêneas cientes de dark silicon e utilizando computação aproximada.* In: Escola

Regional de Alto Desempenho do Centro-Oeste (ERAD-CO), 3. , 2020, Campo Grande. Anais. Porto Alegre: Sociedade Brasileira de Computação, 2020 . p. 5-8. doi: https://doi.org/10.5753/eradco.2020.12644 [6];

- Catelan, D.; Duenha, L.; Santos, R.. *Accuracy and physical characterization of approximate arithmetic circuits.* In Anais do XXI Simpósio em Sistemas Computacionais de Alto Desempenho, pages 143–154, Porto Alegre, RS, Brasil, 2020. SBC. doi: https://doi.org/10.5753/wscad.2020.14065 [7];

- Catelan, D.; Duenha, L.; Santos, R.. *Evaluation and characterization of approximate arithmetic circuits.* In Concurrency and Computation: Practice and Experience, page e6865. Special Issue: WSCAD 2020. PDCAT 2020/PDCAT-PAAP 2020, 2022. https://onlinelibrary.wiley.com/doi/epdf/10.1002/cpe.6865 [8];

- Catelan, D.; Duenha, L.; Wanner, L.; Santos, R.. *Instruction-level Loop Perforation.* In Anais do XXIV Simpósio em Sistemas Computacionais de Alto Desempenho, pages 37–48, Porto Alegre, RS, Brasil, 2023. SBC. https://sol.sbc.org.br/index.php/sscad/article/view/26507 [9];

- Catelan, D.; Sovernigo, F.; Duenha, L.; Santos, R.., *An Instruction-Set Extension to Support Approximate Multicore Processors*, 2024 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW), Hilo, HI, USA, 2024, pp. 23-32, doi: 10.1109/SBAC-PADW64858.2024.00015. https://ieeexplore.ieee.org/document/10764671

6.4 Future Work

Given the advances and challenges observed throughout this work, several opportunities emerge to deepen and expand the contributions. The following suggestions seek to address some gaps and explore new approaches and techniques that can improve the results from this thesis.

- Explore the new Verilog version of the NEORV32 processor, to enable the usage of the analysis resources available in the Qflow [19] tool. Evaluate how Qflow analysis tools (timing, power consumption, and area utilization estimates) can be used to optimize NEORV32 performance in different configurations. Such an approach could not only validate the processor's compatibility with the tool and provide insights on improving future implementations. With the information obtained, it would be possible to create an approximation core and insert it into frameworks such as MultiExplorer [17];

- Design and carry out an automated design space exploration heuristic on the Taylor's series mathematical functions. The goal would be to identify optimal settings for parameters, such as the number of terms in the series, numerical precision, and possibilities for exchanging exact instructions for approximate instructions, in order to optimize the accuracy and efficiency of the calculation. This approach would include the formulation of criteria for performance evaluation, such as accuracy, resource consumption (time and space), number of instructions, cycles, and power;

- Extend the approximate RISC-V ISA by designing new approximate instructions. Likewise, to design new approximate mathematical functions, such as pow and sqrt. This approach would involve defining criteria to balance the accuracy of calculations with system's efficiency, and evaluating the impact of the new instructions and functions on different applications. The goal is to explore the potential of approximate computing for error-tolerant applications, expanding the possibilities of using RISC-V in systems with performance and power constraints.

- Development of a framework based on approximate computing, enabling the adaptation of classical Machine Learning algorithms (K-means, SVM, KNN, and DBSCAN) to scenarios with variable computational constraints, prioritizing the preservation of result quality, reduction of time and resource consumption, and facilitating its use in embedded devices and real-time systems.

Bibliography

[1] H. A. F. Almurib, T. N. Kumar, and F. Lombardi. Inexact designs for approximate low power addition by cell replacement. In *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*, DATE '16, page 660–665, San Jose, CA, USA, 2016. EDA Consortium.

[2] H. Barua and D. Mondal. Approximate computing: A survey of recent trends—bringing greenness to computing and communication. *The Journal of the Institution of Engineers (India): Electronics and Telecommunication Division*, 06 2019.

[3] A. I. Barvinok. A polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed. In *Annual Foundations of Computer Science*, SFCS '93, page 566–572, USA, 1993. IEEE Computer Society. https://doi.org/10.1109/SFCS.1993.366830.

[4] J. Bornholt, E. Torlak, L. Ceze, and D. Grossman. Approximate program synthesis. *Workshop on Approximate Computing Across the Stack (WAX)*, 2015.

[5] K. Boyini. Floyd warshall algorithm, 2022. https://www.tutorialspoint.com/Floyd-Warshall-Algorithm. Accessed: March/2023.

[6] D. Catelan and R. Dos Santos. Exploração do espaço de projeto em arquiteturas heterogêneas cientes de dark silicon e utilizando computação aproximada. In *Anais da III Escola Regional de Alto Desempenho do Centro-Oeste*, pages 5–8, Porto Alegre, RS, Brasil, 2020. SBC.

[7] D. Catelan, L. Duenha, and R. Dos Santos. Accuracy and physical characterization of approximate arithmetic circuits. In *Anais do XXI Simpósio em Sistemas Computacionais de Alto Desempenho*, pages 143–154, Porto Alegre, RS, Brasil, 2020. SBC. https://sol.sbc.org.br/index.php/wscad/article/view/14065.

[8] D. Catelan, L. Duenha, and R. Dos Santos. Evaluation and characterization of approximate arithmetic circuits. In *Concurrency and Computation: Practice and Experience*, page e6865. Special Issue:WSCAD 2020. PDCAT 2020/PDCAT-PAAP 2020, 2022. https://onlinelibrary.wiley.com/doi/epdf/10.1002/cpe.6865.

[9] D. Catelan, L. Duenha, L. Wanner, and R. Dos Santos. Instruction-level loop perforation. In *Anais do XXIV Simpósio em Sistemas Computacionais de Alto Desempenho*, pages 37–48, Porto Alegre, RS, Brasil, 2023. SBC. https://sol.sbc.org.br/index.php/sscad/article/view/26507.

[10] D. Catelan, F. Sovernigo, L. Duenha, and R.. Dos Santos. An instruction-set extension to support approximate multicore processors. In *2024 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*, pages 23–32, 2024.

[11] D. Cattaneo, M. Chiari, G. Magnani, N. Fossati, S. Cherubin, and G. Agosta. Fixm: Code generation of fixed point mathematical functions. *Sustainable Computing: Informatics and Systems*, 29:100478, 2021.

[12] L. Chen, J. Han, W. Liu, and F. Lombardi. Design of approximate unsigned integer non-restoring divider for inexact computing. In *Proceedings of the 25th Edition on Great Lakes Symposium on VLSI*, GLSVLSI '15, page 51–56, New York, NY, USA, 2015. Association for Computing Machinery.

[13] L. Chen, J. Han, W. Liu, and F. Lombardi. On the design of approximate restoring dividers for error-tolerant applications. *IEEE Transactions on Computers*, 65(8):2522–2533, 2016.

[14] V. Chippa, D. Mohapatra, K. Roy, S. Chakradhar, and A. Raghunathan. Scalable effort hardware design. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 22:2004–2016, 09 2014.

[15] T. DiPasqueale (codeslinger). Black-scholes option pricing model in c. https://gist.github.com/codeslinger/472083/. Accessed: September/2023.

[16] R. Dennard, F. Gaensslen, H. Yu, L. Rideout, E. Bassous, and A. Leblanc. Design of ion-implanted mosfets with very small physical dimensions. *IEEE Journal of Solid-Circuits*, pages 256–267, 1974.

[17] R. Devigo, L. Duenha, R. Azevedo, and R. Santos. Multiexplorer: A tool set for multicore system-on-chip design exploration. In *IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 160–161. IEEE, 2015.

[18] K. Du, P. Varman, and K. Mohanram. High performance reliable variable latency carry select addition. In *2012 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1257–1262, March 2012.

[19] R. T. Edwards. Qflow: An open-source digital synthesis flow, 2021. Available online at http://opencircuitdesign.com/qflow/. Accessed: November/2022.

[20] H. Esmaeilzadeh, A. Sampson, M. Ringenburg, L. Ceze, D. Grossman, and D. Burger. Addressing dark silicon challenges with disciplined approximate computing. *Proc. 4th Workshop on Energy-Efficient Design*, 01 2012.

[21] I. Felzmann, J. F. Filho, and L. Wanner. Risk-5: Controlled approximations for risc-v. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11):4052–4063, 2020.

[22] Benchmarks Game, 2022. https://benchmarksgame-team.pages.debian.net/benchmarksgame. Accessed: March/2023.

[23] GeeksforGeeks. Dijkstra's shortest path algorithm, 2022. https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/. Accessed: March/2023.

[24] A. Gorantla and P. Deepa. Design of approximate subtractors and dividers for error tolerant image processing applications. *Journal of Electronic Testing*, pages 1–7, 10 2019.

[25] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A call graph execution profiler. In *Symposium on Compiler Construction*, page 120–126, NY, USA, 1982. Association for Computing Machinery.

[26] V. Gupta, D. Mohapatra, A. Raghunathan, and K. Roy. Low-power digital signal processing using approximate adders. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 32(1):124–137, 2013.

[27] J. Han and M. Orshansky. Approximate computing: An emerging paradigm for energy-efficient design. In *18th IEEE European Test Symposium, ETS 2013, Avignon, France, May 27-30, 2013*, pages 1–6, 2013.

[28] M. Horowitz. 1.1 computing's energy problem (and what we can do about it). In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 10–14, 2014.

[29] N. Iwanaga, T. Abe, and A. Yamawaki. Development of fixed-point trigonometric function library for high-level synthesis. *1st IEEE/IIAE International Conference on Intelligent Systems and Image Processing*, pages 91–94, 2013.

[30] H. Jiang, J. Han, and F. Lombardi. A comparative review and evaluation of approximate adders. In *Proceedings of the 25th Edition on Great Lakes Symposium on VLSI*, GLSVLSI '15, page 343–348, New York, NY, USA, 2015. Association for Computing Machinery.

[31] P. Kulkarni, P. Gupta, and M. Ercegovac. Trading accuracy for power in a multiplier architecture. *J. Low Power Electronics*, 7:490–501, 12 2011.

[32] V. Leon, M. Hanif, G. Armeniakos, X. Jiao, M. Shafique, K. Pekmestzi, and D. Soudris. Approximate computing survey, part i: Terminology and software e hardware approximation techniques, 2023.

[33] L. Li and H. Zhou. On error modeling and analysis of approximate adders. In *2014 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 511–518, Nov 2014.

[34] S. Li, S. Park, and S. Mahlke. Sculptor: Flexible approximation with selective dynamic loop perforation. In *International Conference on Supercomputing*, page 341–351, NY, USA, 2018. Association for Computing Machinery.

[35] J. Liang, J. Han, and F. Lombardi. New metrics for the reliability of approximate and probabilistic adders. *IEEE Transactions on Computers*, 62(9):1760–1771, 2013.

[36] I. Lin, Y. Yang, and C. Lin. High-performance low-power carry speculative addition with variable latency. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 23(9):1591–1603, Sep. 2015.

[37] S. Ma and P. Ampadu. Approximate memory with approximate dct. In *Proceedings of the 2019 on Great Lakes Symposium on VLSI*, GLSVLSI '19, page 355–358, New York, NY, USA, 2019. Association for Computing Machinery.

[38] H.R. Mahdiani, S.M. Fakhraie, and Cameron Lucas. Bio-inspired imprecise computational blocks for efficient vlsi implementation of soft-computing applications. *Circuits and Systems I: Regular Papers, IEEE Transactions on*, 57:850 – 862, 05 2010.

[39] A. Mativi, D. Silveira, and S. Bampi. A survey on approximate memory techniques for error-tolerant applications. In *UFRGS*, 2019. http://www.inf.ufrgs.br/ acmsouza/files/2019-SIM.pdf.

[40] MiBENCH. Solve a cubic polynomial. https://github.com/embecosm/mibench. Accessed: September/2023.

[41] P. Mineiro. Fastapprox library. https://code.google.com/archive/p/fastapprox/. Accessed: August/2024.

[42] S. Mittal. A survey of techniques for approximate computing. *ACM Comput. Surv.*, 48(4):62:1–62:33, March 2016.

[43] D. Mohapatra, V. K. Chippa, A. Raghunathan, and K. Roy. Design of voltage-scalable meta-functions for approximate computing. In *2011 Design, Automation Test in Europe*, pages 1–6, March 2011.

[44] A. Moreno, F. Calle, and C. Pedraza. A low-cost fault tolerance method for arm and risc-v microprocessor-based systems using temporal redundancy and approximate computing through simplified iterations. *Journal of Integrated Circuits and Systems*, 16(3), 2021.

[45] V. Mrázek, R. Hrbáček, Z. Vašíček, and L. Sekanina. Evoapprox8b: Library of approximate adders and multipliers for circuit design and benchmarking of approximation methods. In *Proc. of the 2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 258–261. European Design and Automation Association, 2017.

[46] J. Muller. Elementary functions and approximate computing. *Proceedings of the IEEE*, 108(12):2136–2149, 2020.

[47] S. Muthulakshmi, Chandra Dash, and S. Prabaharan. Memristor augmented approximate adders and subtractors for image processing applications: An approach. *AEU - International Journal of Electronics and Communications*, 91, 05 2018.

[48] S. Nolting. The neorv32 risc-v processor. https://github.com/stnolting/neorv32/tree/v1.10.4. Accessed: June/2022.

[49] K. Palem and A. Lingamneni. Ten years of building broken chips: The physics and engineering of inexact computing. *ACM Trans. Embed. Comput. Syst.*, 12(2s):87:1–87:23, May 2013.

[50] K. Parasyris, J. Diffenderfer, H. Menon, I. Laguna, J. Vanover, R. Vogt, and D. Osei-Kuffuor. Approximate computing through the lens of uncertainty quantification. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2022.

[51] J. Park, H. Esmaeilzadeh, X. Zhang, M. Naik, and W. Harris. Flexjava: language support for safe and modular approximate programming. *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 745–757, 08 2015.

[52] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design RISC-V Edition: The Hardware Software Interface.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2017.

[53] L. Reis and L. Wanner. Functional approximation and approximate parallelization with the accept compiler. In *IEEE 33rd International Symposium on Computer Architecture and High Performance Computing*, pages 188–197, 2021.

[54] L. O. P. Reis. Targeting broad software approximations with the accept compiler. Master's thesis, Campinas State University, Campinas, SP, Brazil, 2021.

[55] RISC-V. Risc-v pk, 2022. Available at: https://github.com/riscv-software-src/riscv-pk. Accessed: September/2023.

[56] RISC-V. Risc-v toolchain, 2022. Available at: https://github.com/riscv-collab/riscv-gnu-toolchain. Accessed: November/2023.

[57] M. Rodriguez-Cancio, B. Combemale, and B. Baudry. Approximate loop unrolling, 2019. Association for Computing Machinery. pages 94-105. 2019. https://doi.org/10.1145/3310273.3323841.

[58] A. Sampson, A. Baixo, B. Ransford, T. Moreau, J. Yip, L. Ceze, and M. Oskin. Accept: A programmer-guided compiler framework for practical approximate computing. *University of Washington Technical Report UW-CSE-15-01*, 1:1–14, 2015.

[59] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. Enerj: Approximate data types for safe and general low-power computation. *SIGPLAN Not.*, 46(6):164–174, June 2011.

[60] A. Sampson, J. Nelson, K. Strauss, and L. Ceze. Approximate storage in solid-state memories. In *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 25–36, 2013.

[61] F. Santos. Cfast fourier transform, 2022. https://github.com/riscv-software-src/riscv-isa-sim. Accessed: March/2023.

[62] R. Santos, R. Sonohata, C. Krebs, D. Catelan, L. Duenha, D. Segovia, and M. Santos. Exploração do projeto de sistemas baseados em gpu ciente de dark silicon. In *31st International Symposium on Computer Architectura and High Performance Computing (WSCAD-SSC)*, 2019.

[63] M. Shafique, O. Hasan, R. Hafiz, S. Mazahir, M. A. Hanif, and S. Rehman. Approximate computing across the hardware and software stacks. *Institution of Engineering and Technology*, pages 497–522, 2019.

[64] S. Sidirogloy-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard. Managing performance vs. accuracy trade-offs with loop perforation. *Association for Computing Machinery*, pages 124–134, 2011.

[65] J. Silveira, L. Castro, V. Araújo, R. Zeli, D. Lazari, M. Guedes, R. Azevedo, and L. Wanner. Prof5: A risc-v profiler tool. In *International Symposium on Computer Architecture and High Performance Computing*, pages 201–210, 2022.

[66] G. Singh, B. Kundu, H. Menon, A. Penev, D. J. Lange, and V. Vassilev. Fast and automatic floating point error analysis with chef-fp. In *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1018–1028, 2023.

[67] J. Singh and J. Kedia. A comparative analysis of different approximate adders used for image compression and image addition. *International Journal of Advanced Research in Electronics and Communication Engineering (IJARECE)*, 7, 2018.

[68] IEEE Computer Society. Ieee standard for floating-point arithmetic, aug 2008.

[69] SPIKE. Spike risc-v isa simulator, 2019. Available at: https://github.com/riscv-software-src/riscv-isa-sim. Accessed: September/2023.

[70] A. Statescu. Program to compute pi using a monte carlo method, 2022. https://gist.github.com/thinkphp/0d56dfd5eb5f91da029a91d4c7676f12. Accessed: March/2023.

[71] G. Tagliavini, A. Marongiu, and L. Benini. Flexfloat: A software library for transprecision computing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(1):145–156, 2020.

[72] S. Venkataramani, S. Chakradhar, K.k Roy, and A. Raghunathan. Approximate computing and the quest for computing efficiency. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2015.

[73] S. Venkataramani, V. Chippa, S. Chakradhar, K. Roy, and A. Raghunathan. Quality programmable vector processors for approximate computing. *MICRO 2013 - Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1–12, 12 2013.

[74] T. G. Viegas. *Técnicas de Computação Aproximada para Implementação de Filtros FIR*. PhD thesis, Técnico Lisboa, 2016.

[75] J. E. Volder. The cordic trigonometric computing technique. *IRE Transactions on Electronic Computers*, EC-8(3):330–334, 1959.

[76] J. Walker. Floating point benchmarks. https://www.fourmilab.ch/fbench/. Accessed: September/2023.

[77] M. Wyse, A. Baixo, T. Moreau, B. Zorn, J. Bornholt, A. Sampson, L. Ceze, and M. Oskin. React: A framework for rapid exploration of approximate computing techniques. *WAX 2015 (colocated with PLDI)*, 2015.

[78] Y. Xiang, L. Li, S. Yuan, W. Zhou, and B. Guo. Metrics, noise propagation models, and design framework for floating-point approximate computing. *IEEE Access*, 9:71039–71052, 2021.

[79] Q. Xu, T. Mytkowicz, and N. Kim. Approximate computing: A survey. *IEEE Design and Test*, 33(1):8–22, 2 2016.

[80] Z. Yang, A. Jain, J. Liang, J. Han, and F. Lombardi. Approximate xor/xnor-based adders for inexact computing. *Proceedings of the IEEE Conference on Nanotechnology*, pages 690–693, 08 2013.

[81] A. Yazdanbakhsh, D. Mahajan, P. Lotfi-Kamran, and H. Esmaeilzadeh. Axbench: A benchmark suite for approximate computing across the system stack. *IEEE Design & Test*, 34:60–68, 2017.

[82] R. Ye, T. Wang, F. Yuan, R. Kumar, and Q. Xu. On reconfiguration-oriented approximate adder design and its application. In *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 48–54, Nov 2013.

[83] N. Zhu, W. L. Goh, G. Wang, and K. S. Yeo. Enhanced low-power high-speed adder for error-tolerant application. In *2010 International SoC Design Conference*, pages 323–327, Nov 2010.

[84] H. Zitane and D.F.M Torres. Generalized taylor's formula for power fractional derivatives. In *Bol. Soc. Mat. Mex. 29, 68*, 2023. https://doi.org/10.1007/s40590-023-00540-0.