APPROXIMATE PROCESSOR DESIGN WITH RISC-V ISA

by

İbrahim Taştan

B.S., Electrical & Electronics Engineering, Boğaziçi University, 2016

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Electrical & Electronics Engineering
Boğaziçi University
2020

dedicated to my lovely daughter Nur Zehra

ACKNOWLEDGEMENTS

My deep gratitude goes first to Prof. Dr. Arda Yurdakul, who expertly guided me through my master thesis. I'm heartily grateful to her, whose encouragement, guidance, and support enabled me to develop this thesis.

I would like to thank Assist. Prof. Dr. İsmail Faik Başkaya, who accepted me as a master student and helped me to finish my master thesis under his supervision. I am also grateful to my thesis committee members for allocating their valuable time and effort to judge my thesis.

I am also thankful to my RISC-V group-mates, namely Mehmet Alp Şarkışla, Ömer Faruk Irmak and Fatih Aşağıdağ, together we developed basics of our RISC-V core. I want to express my special gratitude to Mehmet Alp Şarkışla, who helped me a lot in the process of test code creation. I am also grateful to Mahmut Karaca, who prepared source codes for the software part of this study.

I would like to extend my sincere gratitude to my company, Tübitak Bilgem. With their support, this research was funded by the Turkish Ministry of Science, Industry, and Technology under grant number 58135. Without their support and funding, this project would not have been possible.

I would like to acknowledge with gratitude, the support of my colleagues in TÜTEL. They all kept me going and helped me directly or indirectly to complete this thesis.

I am fully indebted to my family for helping me survive all the stress from this study and not letting me give up. Finally, I owe my finest gratitude to my wife. Without her support, patience, and understanding, this thesis would not have been possible.

ABSTRACT

APPROXIMATE PROCESSOR DESIGN WITH RISC-V ISA

With the rise of the Internet of Things (IoT), low-cost resource-constrained devices have to be more capable than traditional embedded systems, which operate on stringent power budgets. To add new capabilities such as learning, power consumption planning has to be revised. Approximate computing is a promising paradigm for reducing power consumption at the expense of inaccuracy introduced to the computations. In this thesis, we propose a processor with approximate processing functionality for resource-constrained IoT devices. A microprocessor with a dual-datapath mechanism is described in C++ and synthesized with a High-Level Synthesis (HLS) tool. A standard datapath exists for the parts of applications where the calculation should be exact. Additionally, an approximate datapath, which includes approximate computing features that will be more likely to exist in the next generation, low-cost, resource-constrained, and learning IoT devices, is introduced. Coarse-grain control for setting the accuracy of approximate operations is adopted to reduce the number of control signals by grouping the bits so that they can be turned on-off simultaneously. The size of the operands of the approximate operators is dynamically adjusted at the data path without affecting the performance. Based on these features, we propose new approximate adder and multiplier designs and integrate these blocks with a CPU, which benefits from RISC-V ISA. Targeting machine learning applications such as classification and clustering, we have demonstrated that our processor reinforced with approximate operations can save power up to 23% for ASIC implementation while at least 90% top-1 accuracy is achieved on the trained models and test datasets.

ÖZET

RISC-V KOMUT KÜMESİ MİMARİSİYLE YAKLAŞIK İŞLEMCİ TASARIMI

Nesnelerin interneti (IoT)'nin yaygınlaşmasıyla beraber; düşük güç tüketimli, düşük maliyetli ve sınırlı kaynağa sahip IoT cihazlarının, geleneksel gömülü sistemlere göre daha kapsamlı bir yapıda olmasına ihtiyaç duyulmaktadır. Öğrenme gibi yeni kabiliyetlerin bu cihazlara eklenebilmesi için, güç tüketimi yeniden gözden geçirilmelidir. Yaklaşık hesaplama yöntemleri, hesaplamaların tam olarak doğru yapılmasından taviz vererek de olsa, güç tüketimini önemli ölçüde düşürebilmektedir. Bu tez çalışmasında, sınırlı kaynağa sahip IoT cihazları için yaklaşık hesaplama yöntemini kullanan bir işlemci tasarımı sunulmuştur. İki veriyolu olan bu işlemci, C++ programlama diliyle tasarlanmış ve Yüksek Seviye Sentez - High-Level Synthesis (HLS) - araçlarıyla sentezlenmiştir. Normal veriyolunda hesaplamalar tam olarak doğru bir şekilde yapılırken; yeni nesil, düşük maliyetli, kaynakları sınırlı ve öğrenme kabiliyeti olan IoT cihazlarında var olacağını düşündüğümüz yaklaşık veriyolu kısmında ise hesaplamalar yaklaşık olarak yürütülmektedir. Kontrol sinyalleri sayısını düşürmek için bitler gruplandırılmış ve hassasiyet seviyesi düşük bir doğruluk kontrolü mekanizması tasarlanmıştır. Yaklaşık veriyolunda işlenen terimlerin boyutları, performansı etkilemeden, dinamik olarak veriyolunda ayarlanmaktadır. Bu özellikler temelinde, yeni yaklaşık toplayıcı ve çarpıcı blokları literatüre sunulmuş ve RISC-V komut kümesi mimarisiyle tasarlanan bir işlemciye bu bloklar entegre edilmiştir. Sınıflandırma ve kümeleme gibi makine öğrenmesi algoritmalarını hedef alarak yapılan deneylerin sonucunda; önemli ölçüde güç tasarrufu, yüksek doğruluk seviyesiyle beraber elde edilebilmektedir. Tasarlanan işlemcinin uygulamaya özel tümdevre (ASIC) tasarımında, eğitilmiş modeller ve test veri kümeleri üzerinde %23'e varan güç tasarrufu, en az %90 doğrulukla elde edilebildiği gösterilmiştir.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS iv
ABSTRACT v
ÖZET vi
LIST OF FIGURES ix
LIST OF TABLES xi
LIST OF SYMBOLS xiv
LIST OF ACRONYMS/ABBREVIATIONS xv
1. INTRODUCTION 1
2. RELATED WORKS 5
2.1. Approximate Computing 5
2.2. Dynamic Accuracy Control 7
2.3. Approximate Processor Design 9
2.4. IoT Applications 10
2.5. RISC-V ISA 11
3. APPROXIMATE PROCESSOR 13
3.1. Proposed RISC-V Core Structure 13
3.2. Approximate Units 16
3.2.1. Dynamic Sizing 20
3.2.2. Approximate Adder/Subtractor Design 23
3.2.3. Error Probability Analysis of the Proposed Adder 26
3.2.4. Approximate Multiplier Design 32
4. EXPERIMENTS 35
4.1. Experimental Setup 35
4.1.1. Approximate Regions in the Codes 36
4.1.2. Accuracy Evaluation 37
4.1.3. Power Consumption Analysis 38
4.2. Datasets and Algorithms 39
4.2.1. Datasets 39
4.2.2. Algorithms 40

4.3. Experiments 43
4.3.1. Dynamic Sizing 43
4.3.2. Approximate Addition and Subtraction with Exact Multiplication Tests 45
4.3.3. Exact Addition and Subtraction with Approximate Multiplication Tests 47
4.3.4. Approximate Addition, Subtraction and Multiplication Tests . . 49
4.3.5. Approximation Level Modification at Run-time 49
5. DISCUSSION 52
5.1. ASIC Implementation of the Proposed System 53
5.2. Bit-truncation in the Approximate Blocks 54
5.3. Approximate Processor and IoT Use Case Discussion 55
6. CONCLUSION 58
REFERENCES 61
APPENDIX A: Test System for Experiments 69
A.1. start.S File 69
A.2. link.ld File 70
A.3. Linux Commands for Compiling C Codes 75
A.4. Verilog Codes for the Approximate Adder 76
A.5. Verilog Testbench for the Results of the Tests 87
A.6. Constraint File 88
A.7. Writing SAIF File 89
A.8. TCL Script for Vivado 90

LIST OF FIGURES

Figure 2.1. Flow diagram of dynamic accuracy control in our Approximate IoT processor. 8

Figure 3.1. General flow diagram of the implementations in the proposed core. 14

Figure 3.2. C code for XALU block in HLS. op_1 and op_2 refer to two operands and XLEN is the length of a register which is 32. 15

Figure 3.3. *approx_add* C function in HLS and dummy operations inside of it. 18

Figure 3.4. Synthesized verilog code of *approx_add* C function in HLS. This code is modified to call the approximate adder as a submodule. . . 18

Figure 3.5. Flow diagram of dynamic control of the approximation levels and dynamic sizing. 21

Figure 3.6. *fast_clz*32 C function to find the leading zeros for 32-bit operands. 22

Figure 3.7. Basic addition mechanism of the tree adders. Approximation is implemented in the gray cells of the adder tree. 24

Figure 3.8. Sklansky adder tree used for approximate adder design. 25

Figure 3.9. Booth encoder and selector circuits to generate partial products. . 33

Figure 3.10. Partial products structure after booth encoding. s represents the sign extension and black dots are the partial product bits. 34

Figure 3.11. Wallace tree structure for the summation of partial products. All CSAs and final adder are approximate. 34

Figure 4.1. *classifyAPoint* function and MAC loop, highlighted with yellow circle, inside of it where the approximate operations implemented. 41

Figure 4.2. Main error calculation loop for implemented KM code and MAC loop, highlighted with yellow circle, where the approximate operation implemented. 42

Figure 4.3. Main error calculation loop for implemented ANN code and MAC loop, highlighted with yellow circle, where the approximate operation implemented. 44

Figure 4.4. Approximation level modification at run-time (a) accuracy and (b) power saving results. 51

LIST OF TABLES

Table 3.1. R-type RISC-V instruction structure and modification for approximate ADD, SUB and MUL - XADD, XSUB and XMUL - operations. 16

Table 3.2. Truth table for ANDOR operation in the gray cells and its comparison with the approximate results. 27

Table 3.3. The probability tree for the proposed adder when the approximation level is 3. The colored cells show the probability of being false for G values, which are actually the error probabilities at the corresponding bits. 29

Table 3.4. Truth table for ANDOR operation in the gray cells and its comparison with the effect of the approximate results on the other gray cells. 30

Table 3.5. The probability tree for the proposed adder when the approximation level is 1. 31

Table 3.6. The probability tree for the proposed adder when the approximation level is 2. 31

Table 3.7. Truth table for booth selection. 33

Table 4.1. Resource utilization of the core in the target FPGA without approximate blocks. 36

Table 4.2. Implemented datasets and their properties. 40

Table 4.3. Contributions of dynamic sizing to the power saving percentages of the new approximate blocks, given as the average and maximum of the results from different datasets for each algorithm. 45

Table 4.4. Resources for approximate blocks and their area overhead on the CPU. 46

Table 4.5. Accuracy rates of the approximate adders, given as the average of the results from different datasets for each algorithm. 47

Table 4.6. Power saving percentages of the approximate adders, given as the average and maximum of the results from different datasets for each algorithm. 48

Table 4.7. Accuracy rates of the approximate multipliers, given as the average of the results from different datasets for each algorithm. 48

Table 4.8. Power saving percentages of the approximate multipliers, given as the average and the maximum of the results from different datasets for each algorithm. 49

Table 4.9. Accuracy rates of the approximate multipliers, given as the average of the results from different datasets for each algorithm. 50

Table 4.10. Power saving percentages of the approximate multipliers, given as the average and the maximum of the results from different datasets for each algorithm. 50

Table 5.1. Power saving and area overhead of approximate designs in ASIC implementation. 54

Table 5.2. Power saving modes and their corresponding approximation levels for case study proposal. 56

LIST OF SYMBOLS

Π_0	Probability of being false
Π_1	Probability of being true
Π_{E1}	Error probability for all gray cells when the approximate level is 1
Π_{E2}	Error probability for all gray cells when the approximate level is 2
Π_{E3}	Error probability for all gray cells when the approximate level is 3

LIST OF ACRONYMS/ABBREVIATIONS

2D	Two Dimensional
ADD	Addition
ALU	Arithmetic Logic Unit
ANN	Artificial Neural Network
B	Banknote authentication dataset
CPA	Carry Propagate Adder
CPU	Central Processing Unit
CSA	Carry Save Adder
DDoS	Distributed Denial of Service
FPGA	Field Programmable Gate Array
FPU	Floating-Point Unit
FSM	Finite State Machine
G	Generate
GCC	GNU Compiler Collection
GUI	Graphical User Interface
HDL	Hardware Description Language
HLS	High-Level Synthesis
IoT	Internet of Things
ISA	lnstruction Set Architecture
KM	K-means
KNN	K-nearest Neighbor
LR	Wall following robot sensor long dataset
LSB	Least Significant Bit
LW	Wireless indoor localization long dataset
MAC	Multiply and Accumulate
ML	Machine Learning
MSB	Most Significant Bit
MUL	Multiplication

MULDIV	Multiplication Division Unit
P	Propagate
PC	Program Counter
PP	Partial Products
R	Wall following robot sensor dataset
RISC-V	Reduced Instruction Set Computer - V
RMS	Recognition, Mining, and Synthesis
SAIF	Switching Activity Interchange Format
SR	Wall following robot sensor short dataset
SUB	Subtraction
SW	Wireless indoor localization short dataset
W	Wireless indoor localization dataset
XADD	Approximate Addition
XALU	Approximate Arithmetic Logic Unit
XMUL	Approximate Multiplication
XMULDIV	Approximate Multiplication Division Unit
XSUB	Approximate Subtraction

1. INTRODUCTION

As benefits from technology scaling have been diminishing, it has become more important for designers to discover new sources for efficient computing. Besides, the emergence of heavy workloads in some engineering applications, e.g., recognition, mining, inference, data analytics, and vision, desire a considerable amount of power and requires a very long run-time. This point has also drifted people to seek new decent ways in computations to improve power efficiency. Approximate computing has gained the attention of many researchers in recent years because of addressing these needs by proposing energy-efficient computing mechanisms. It is a way to improve the performance or efficiency of computation by sacrificing fully accurate results. Many applications, e.g., image or signal processing, classification, and clustering operations in Machine Learning (ML), where precise results may not be necessary, have benefited from the advantage introduced by this phenomenon. For example, an approximate Multiply and Accumulate (MAC) unit introduced in [1] to perform two dimensional (2D) convolution for image processing. It can improve power savings more than 60 percent, compared to standard, exact MAC, while image quality degradation is in a tolerable range.

Areas of interest for our design are these kinds of applications that show error resiliency, accept incorrect results, or inexact calculation with negligible quality loss. In this study, ML algorithms for classification, clustering, and artificial neural network (ANN) applications are examined. In our RISC-V core design, the main goal is to improve the power efficiency of main calculation loops in these algorithms by taking advantage of the approximate calculations. Desired outputs in these applications need not be unique and fully accurate, finding the true classes for the output is enough. Hence approximate operations are pretty well suited to these ML applications. It is possible with this idea to control the accuracy of the results by adjusting the hardware blocks used in addition, subtraction, and multiplication operations. By means of removing some parts of these blocks used for exact calculation in hardware, we can propose power-efficient and cost-effective designs with the trade-off of precise results,

which may already be unnecessary for the applications. This idea is specialized in this study and carried out for an approximate core design for the resource-constrained Internet of Things (IoT) devices that benefit from ML algorithms for learning.

Learning methods have been recently introduced in IoT applications for improving efficiency in processing massive and complex data [2]. However, feeding sensor data to cloud so as to learn from data has turned out to be inefficient due to network latencies and processing energy [3]. Inefficiency in the processing of all data on the cloud due to network latency and processing energy has pushed researchers to seek efficient ways to implement ML-based operations such as clustering and classification on the resource-constrained IoT edge devices, not only on the cloud. In [4], the classification quality is improved with clustering algorithms to identify anomalies in network traffic on IoT end-devices. Distributed denial of service (DDoS) attacks on critical internet infrastructure are classified in [5] to automatically detect IoT botnets by using a variety of machine learning algorithms on IoT devices.

We utilize our core as an approximate IoT processor for IoT end-devices so that ML algorithms can also be processed at IoT devices. This study aims to design a low-power processor for IoT end-devices that is specialized to implement classification, clustering, and neural network-based ML algorithms in a more power-efficient way. As mentioned previously, error resiliency in these applications make approximate computing more attractive to save a dramatic amount of energy, which helps to extend the battery life of IoT devices. Thus, we use an approximate computing mechanism in our core to provide energy saving in heavy computation workloads of these algorithms when they are processed on IoT devices. As explained in [6], a common issue for many IoT devices is heavy workloads. Besides, their computation methodology is mainly designed to compute precise results even when it is not necessary. Thus, such a design is welcomed for IoT applications that can tolerate inexact operations for the sake of power efficiency.

While designing our Approximate IoT Processor, we follow a guideline that we derived from various studies in the literature: Firstly, approximate low-power computing

at resource-constrained IoT processors needs to be handled at the instruction set level that supports approximate operations [6,7]. Besides, the precision of the computations needs to be adjustable to serve different application requirements [8–10]. Finally, there should be an application framework support to map user-defined regions of software to approximate computing modules [11]. So, in this paper, we propose an embedded processor, which has approximate processing functionality, with the following properties:

- The instruction set extension is minimal: In machine learning, the majority of operations are addition, subtraction, and multiplication. So, we extend base Reduced Instruction Set Computer - V (RISC-V) lnstruction Set Architecture (ISA) only with XADD, XSUB, and XMUL, which stand for the approximate addition, subtraction, and multiplication, respectively. Code pieces that can benefit from approximate instructions can be handled via the plug-in developed in [12] for RISC-V GNU Compiler Collection (GCC).
- We propose a coarse-grain control mechanism for setting the accuracy of approximate operations during run-time. In our proposal, the number of control signals is minimized by setting each control signal to activate a group of bits. To achieve this, we design a parallel-prefix adder and a Wallace tree multiplier. We present three approximation levels to control the accuracy of the computations.
- Exact data types are used so that memory utilization will be maximized, and memory access time will not increase. To reduce power consumption, we adjust the size of the operands of the approximate operators dynamically at the data path. Since approximate operators are faster than the exact ones, dynamic sizing does not deteriorate the performance.
- For monitoring the quality of the decisions resulting from the ML algorithms running on our approximate processor, we rely on the interaction of the IoT device with the IoT user or other constituents of IoT ecosystem.

The rest of the thesis is organized as follows. The second chapter discusses related works. The third chapter introduces the designed processor and approximate units in detail. Conducted experiments are described in the fourth chapter, and discussion of

the results is presented in the fifth chapter. The last chapter summarizes our work and provides outputs of the study.

2. RELATED WORKS

In this chapter, we will set forth related research under five subsections. We summarize approximate computing studies in literature and discuss our approximate adder design with existing methods in the first subsection. In the second section, dynamic accuracy control methods for approximate designs in literature are examined and compared with our method. In the third part, we will introduce several significant points for designing an approximate processor design and make comparisons between existing approximate processors and our proposal. We mention several studies that implement clustering and classification algorithms on IoT devices. We mention studies that implement clustering and classification algorithms on IoT devices in the fourth subsection. In the last section, we discuss the RISC-V ISA that we use in our design.

2.1. Approximate Computing

Approximate computing has been one of the most promising phenomena to address the need for reducing power consumption in computer systems. Current researches in the field of approximate computing have mainly focused on probabilistic or imprecise designs at circuit, architecture, and software level [13]. SALSA (Systematic logic synthesis of approximate circuits) [14] is one of the good examples in the literature for circuit-level approximation, which proposes approximate versions of the logic circuits. Designing an arithmetic datapath using imprecise, approximate adders [13,15] and/or multipliers [16,17] are also very popular examples of circuit-level application of approximate computing. ANN can use approximate calculation methods for complex algorithms to achieve architectural improvement to identify critical neurons in an easier way [18]. Software can also be approximated using techniques like code perforation [13]. Another example at the programming language level, a systematic tool was proposed in [19] to divide the program into the approximate parts and the precise parts. Both parts are mapped to different hardware with different supply voltages, speed-grades, etc., respectively. This technique makes it possible to present a relatively high service quality. In the meantime, due to approximation, energy can be reduced. On hard-

ware, similar ideas can be applied, too. As an example, [20] proposes a ripple carry adder on which a non-uniform voltage scaling technique is applied. Our methodology in this paper covers the first two groups, namely circuit and architecture level approximations. By introducing new approximate adder and multiplier, we will be focusing on circuit-level approximation, while integrating these blocks with a microprocessor to execute some chosen instructions in approximate mode will be a kind of architecture level approximation.

Approximation at circuit-level for adders covers a general implementation flow, which is generating larger approximate blocks by using smaller exact or inexact sub-adders. In [21], 8-bit approximate sub-adders are reconfigured to create 32-bit and 64-bit more efficient approximate designs in terms of critical path delay, area, and power consumption. In [15], fixed sub-adders are used to create a reconfigurable generic adder structure. In [22], the pipeline mechanism is used to generate an approximation methodology by implementing sub-adders. Using sub-adders can provide a good opportunity to reconfigure the structure and diminish critical path delay to a degree. However, critical path delay can still harm the performance of the operation, especially in high-performance applications.

At this point, we believe that proposing an approximation methodology for parallel prefix adders would be useful to implement to reduce critical path length because this type of adders have a better latency than the others [23]. There is a study in the literature that uses parallel-prefix adders in the approximate adder design [24]. Their approximation method includes precise and approximate parts, but parallel-prefix adders are used only in the precise part. Thus, there is no approximation methodology for parallel-prefix adders. As a result, the approximation methodology for this type of adders should be investigated to propose energy-efficient ways for high-performance and error-resilient applications. To address this need, we introduce a simple approximation methodology for parallel-prefix adders, which can be further improved for other applications. In our scenario, we specialized the approximate adder to add a coarse-grain dynamic accuracy control mechanism. We used Sklansky adder as a case study, which has a moderate area and the best delay among parallel-prefix adders. We control

only gray cells in the tree for approximation in order to lower the energy consumption, reduce critical path, and still have acceptable accuracy. This method can be generalized to convert other parallel-prefix adders into approximate ones. For example, Brent-Kung, Kogge-Stone, and other parallel-prefix adders share a similar structure with Sklansky. Hence this idea can also be directly implemented on them.

2.2. Dynamic Accuracy Control

A further point for approximate computation studies in the literature is controlling the accuracy of the approximate modules dynamically [8–10,22]. [8] proposes a new methodology to arrange the degree of approximation in accordance with the changes in the application's noise tolerance within the course of execution. They implement approximate floating-point operations aimed at recognition, mining, and synthesis (RMS) applications, which process massive but noisy input data by probabilistic algorithms, therefore it is suitable to control the accuracy within the course of execution. [9] also proposes a dynamic accuracy configurable mechanism for the calculations in partial product trees of the multipliers and [22] also configure the accuracy of the results during run-time while proposed approximate adder can operate in accurate or approximate mode.

Our approximate processor offers dynamic accuracy control functionality as a programmable feature because each IoT application may require a different accuracy. Fine-grain control, as proposed in [10], offers fine-tuning of the accuracy at the cost of increased latency, size, and power consumption of the controller. However, applying approximate computing in low-cost and resource-constrained IoT processors should come with no or very little overhead in terms of the area while reducing power consumption and lowering the execution time, if possible. Hence, we have preferred to use coarse-grain accuracy control. We create certain power-saving levels by putting three approximation levels for each approximate block. Accuracy levels can be adjusted by users or IoT ecosystems in accordance with their power and accuracy requirements, as shown in Figure 2.1. Approximation Level Control Unit can be configured for IoT end-devices.

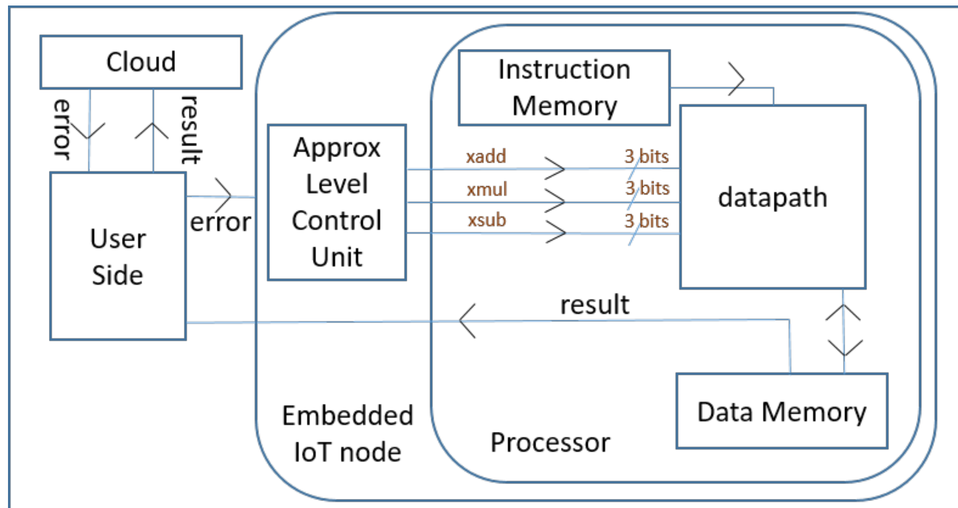

Figure 2.1. Flow diagram of dynamic accuracy control in our Approximate IoT processor.

In the literature, dynamic accuracy control is also achieved via dynamic voltage scaling (DVS) [24,25]. There is no dedicated approximate operation block in DVS; the selected parts of the processor are forced to behave inexactly by lowering the supply voltage. In DVS, accuracy control takes longer than a cycle, especially when an accuracy change is required from approximate mode to exact mode. Besides, it comes with 9% -20% area overhead in the related studies. In our approach, dynamic accuracy control is applied only on the approximate ALU that is augmented in the datapath. Hence the accuracy control circuit need not be big. Its overhead on the entire CPU is much less than 1% in our design. If we calculate the area overhead due to dynamic accuracy control on each approximate block as calculated in [24,25], then it becomes about 4% at most, which is still very small compared to their overhead. Another advantage of our design is that accuracy control takes less than a cycle when switching between different levels of approximation or between one of the approximation levels and exact mode. Since our approximate operators execute faster than the exact ones, dynamic accuracy control circuit does not cause additional latency.

Datatype size and dynamically adjustable operations may also affect consumed power amount, dramatically, as highlighted in [19]. We consider adjusting the size of our approximate blocks in accordance with the size of the operands at run-time just before the execution stage to improve power efficiency a bit more. Adjustment is

implemented at the hardware level, and dynamic configuration is achieved by switching operations.

2.3. Approximate Processor Design

Architectural considerations for approximate calculations in a microprocessor should also be given special attention. According to [7], for the best implementation, designed approximate hard blocks for these operations should be integrated into the architecture at the instruction level. Thus ISA extensions should be made to support these operations in the architecture. We also extend RISC-V ISA for our case to identify approximate operations at the instruction level.

Another architectural issue that is also stressed in [7] is the coverage of the approximate operations. We agree with the idea that approximate calculations must be restricted to only certain operations. For example, we cannot use approximate addition for the immediate value used for branch operations, because any deviation from the real result will collapse the running code. We confine our approximate computations to multiplication, addition, and subtraction by marking instructions that are computed as either approximately or exactly. Approximate result storage may also be taken into consideration separately. While proposed architecture in [7] uses approximation aware cache designs controlled by software or in [19] splits the data memory into two parts as approximate or precise; we did not need to expose our memory parts any modification. Partitioning storage for approximate and exact data causes the under-utilization of both partitions. Besides, memory access time dramatically increases due to the software and hardware incorporated in the data distribution.

The idea of carrying the approximate operations into action by means of a processor is also realized in [11]. In Quora [11], vector operations are approximately executed, but scalar operations are handled with exact operators. The main hardware approximation methodologies in Quora are clock gating, voltage scaling, and truncation. The first method is actually a common low-power technique used in ASIC designs. The second one is also a low-power technique but can also be used in approximation in

the sense that voltage can be dynamically lowered too much to diminish the power consumption at the cost of some error. They truncate least significant bits (LSB)s to control precision dynamically, but as they stressed in [11], there does not exist a hardware module that calculates results directly in an approximate fashion. They have a quality monitor unit that follows the error rate and truncates LSBs accordingly. In our design, we implement scalar operations with approximate and exact operators which can be selected by software. Separate hardware blocks exist for calculating the result approximately and controlling the accuracy dynamically. Hence, two designs have a different approach in terms of approximate CPU design.

A patented work [25] about an approximate processor design can also be found to show the increasing interest in translating approximate design methodologies into processor systems. A proposed processor in [25] includes execution units, such as an integer unit, a single instruction multiple data (SIMD) unit, a multimedia unit, and a floating-point unit. Considering resource-constrained low-power IoT devices, using all of these blocks creates too much area overhead together with a significant increase in energy consumption. We simplify the architecture of the core by putting execution blocks for only logic and integer operations to minimize power consumption and to make it suitable for resource-constrained IoT end-devices. Basic integer addition, subtraction, and multiplication operations can be approximately executed via an additional datapath, namely approximate datapath, added to our CPU with a small area overhead.

2.4. IoT Applications

In our CPU, classification, clustering, and artificial neural network algorithms are implemented. We have conducted experiments on K-nearest neighbor, K-means, and neural network codes to show that an important amount of energy can be saved by our approximate CPU in the ML applications in which these codes are widely used. There are several applications that use K-nearest neighbor [4, 26, 27], K-means [28, 29] and neural networks [30, 31] at IoT devices in such a way that classification, clustering and machine learning processes used in IoT systems can also be carried out on the

resource-constrained IoT devices, not only on the cloud. So, an approximate CPU can help them to decrease power consumption while calculating intensive computation loads of these algorithms. Core architecture, and approximate level control mechanism in this study are designed for IoT end-devices that may have the architecture shown in Figure 2.1.

2.5. RISC-V ISA

RISC-V is an open and free ISA which is more likely to initialize a new era of processor innovation through open standard collaboration. It is born in University of California, Berkeley, (UCB), and its infrastructure is constructed in academia and research. A new level of free, extensible, open for any improvement software and hardware freedom on architecture is delivered by this new architecture [32]. A GNU/GCC software toolchain, an LLVM compiler, a GDB/GNU debugger, a Spike (ISA simulator), QEMU, and a verification suite are the basic tools of RISC-V software.

Providing a long-lived open ISA with broad and significant infrastructure support can be mentioned as the basic intent of RISC-V. It involves documentation, compiler toolchains, operating system ports, reference software simulators, cycle-accurate FPGA emulators, high-performance FPGA computers, efficient ASIC implementations of various target platform designs, configurable processor generators, architecture test suites, and teaching materials [32]. The reason we choose to design new architecture based on RISC-V is the key features of RISC-V. These features can be summarized as:

(i) Introducing a novel software level and free hardware design with the capability of improvement.
(ii) Being an open-source, easier support opportunity from wide-ranging operating systems, tool developers, and software vendors.
(iii) Unlimited potential for future growth due to being open-source, especially for hardware.
(iv) There is no alternative ISA architected like the RISC-V ISA, which allows user extensibility of the architecture without affecting existing extensions or suffering

software fragmentation.

RISC-V has emerged as a new era on the processor design for several years as introducing a new instruction set architecture (ISA) to the literature. Due to being free and an open ISA, there is no need for any microarchitecture licenses in use of RISC-V ISA. This makes all kinds of optimizations, such as low power, performance, security, etc. possible in the architecture level, which attracts a wide variety of researchers from both academic community [33, 34], and private sector [35–37].

This architecture flexibility leads people from very different fields to attempt to design their own microprocessors specialized for the type of their applications. A number of hard/soft cores ranging from simple cores [38, 39] to the most complex superscalar, out-of-order one [35, 40, 41], have been introduced as open-source and hundreds of academic papers related with this ISA have been published so far, and, for sure, a number of projects are also ongoing. Some commercial products from different vendors are also available in the market [36, 37, 42].

Another point behind this interest is that the RISC-V structure enables all users to make suggestions for the architecture at a much earlier point, which was not possible in previous ISAs [32]. Besides, RISC-V has the flexibility to add new custom instructions to the frozen base instructions of the ISA in order to accelerate particular operations or perform special functions [32].

These benefits of RISC-V have also attracted our interest in order to realize the idea of designing a new application-specific approximate microprocessor for resource-constraint IoT end-devices that aims to perform ML algorithms. New instructions (XADD, XSUB, and XMUL) for approximate operations are proposed. RISC-V compiler is specialized for our instructions to compile the new instructions.

3. APPROXIMATE PROCESSOR

This chapter describes the designed core and approximate blocks and explains their structure in detail.

3.1. Proposed RISC-V Core Structure

Our RISC-V core has been developed in C++ and synthesized with 18.1 version of Vivado High-Level Synthesis (HLS) tool. Although [43] indicates that HLS is not the best solution to develop programmable architectures like CPUs, we can benefit from HLS tools for fast prototyping of complex digital hardware in a simpler way by using a high-level or modeling language. Hence, our concern in this paper is to study the effects of approximate arithmetic units in the datapath of a processor. We focus on improving the power efficiency for selected ML applications via simply configurable and adaptable CPU design in which operations can be approximately executed to save power. Our 32-bit core is able to implement all instructions of RV32IM, which is the base integer instructions of RISC-V ISA plus integer multiplication and division instruction sets, except *fence* and *ecall* instructions. This simple core aims specific applications where the operations can be handled by using only integer implementations. We followed a similar approach with Arm Cortex M0 and M3 and did not add a floating-point unit (FPU) to reduce area and lower the energy consumption for low-cost IoT devices.

The generated core implements each instruction like a finite state machine (FSM). It controls the system with *ap_reset*, *ap_start*, *ap_idle*, *ap_ready* and *ap_done* signals. After instructions are loaded into the instruction memory, with *ap_start* becomes high, the first instruction is taken. When the *ap_done* signal becomes high, then the Program Counter (PC) increments and next instruction is taken. This process continues until the implemented code is finished, which means the PC reaches to the value where *ebreak* instruction resides.

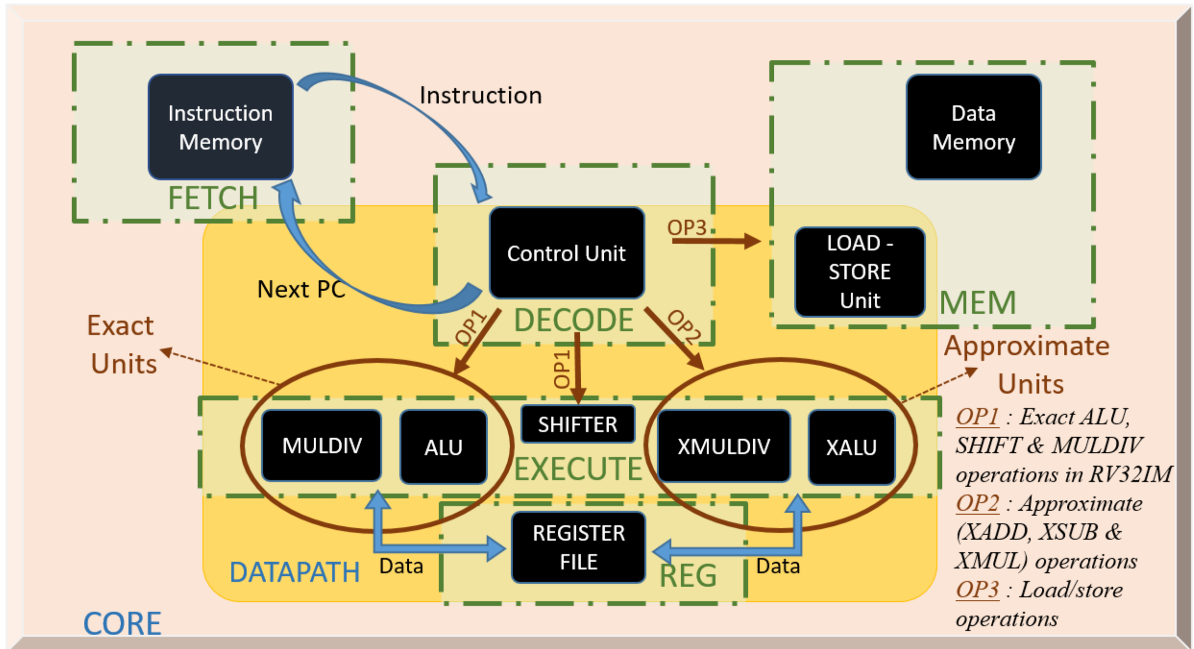

Figure 3.1. General flow diagram of the implementations in the proposed core.

Figure 3.1 shows the general structure of the core. Instruction Memory, where the instruction is fetched from, gives the instruction to the Control Unit, which decodes the signal. Here, conditional operations, namely, branch, and jump, are also decided and executed. It is worth noting that there is no branch prediction block in this design for the sake of simplicity. Immediate values and operands for necessary operations are also determined at the Control Unit. Apart from the conditional operations, process flow varies for different types of instructions after Control Unit. For OP1, and OP2, the execution stage is the next, and the result of the operation is registered after execution. For OP3, which represents load-store operations, data are transferred from the memory to the register or vice versa.

Execute stage consists of two distinct parts, i.e., exact and approximate blocks plus conventional exact shifter block. Exact Part includes ALU and MULDIV blocks, which perform exact calculations for arithmetic and logic instructions. The second main part is Approximate Part, which contains XALU and XMULDIV blocks. Approximate calculations are performed on these blocks. In our current implementation, XALU block contains XADD and XSUB operations, which stand for approximate addition and subtraction, and XMULDIV has only XMUL operation, which is approximate

```
uint32_t xalu(uint32_t op_1,uint32_t op_2,uint8_t ftype, uint8_t func7){
    uint32_t ret = 0;
    uint8_t approx_add_size = 0;
    uint8_t add_sub=0;
    switch (ftype)
    {
        case 0b000:
            switch (func7)
            {
                case 0b1000000: /// XADD
                        add_sub = 0;
                        /// size adjustment ///
                      if (op_1 < op_2)
                          approx_add_size = XLEN - fast_clz32(op_2);
                      else
                          approx_add_size = XLEN - fast_clz32(op_1);
                      /// approximate addition operation ///
                      ret = approx_add((int32_t) op_1,(int32_t) op_2,approx_add_size,add_sub);
                      break;
                case 0b1100000: // XSUB
                      add_sub = 1;
                      ret = approx_add((int32_t) op_1,-(int32_t) op_2, XLEN,add_sub);
//              case 0b1XXXXXXX: // new approximate operation
//                    .
//                    .
//                    .
//                  break;
            }
```

Figure 3.2. C code for XALU block in HLS. op_1 and op_2 refer to two operands and XLEN is the length of a register which is 32.

multiplication. Approximate blocks for other operations, such as division, can be easily added to the Approximate Part of the core. Hence there is no need for further modifications on the architecture. All instruction parsing operations are done for these parts, but specific approximate operations are not defined yet, except for multiplication, addition, and subtraction. Figure 3.2 gives the general structure of the C code for XALU in HLS. Desired approximate counterparts of the exact operations can be added, as shown in Figure 3.2.

New approximate parts of the datapath require new instructions to be added to RISC-V ISA, because there are not any instructions in RISC-V ISA for approximate operations. New instructions are introduced in such a way that they do not overlap with the existing instructions of whole RISC-V ISA. Instead of creating new instructions for approximate operations from scratch which needs more effort for deciding proper and available values for all parts of an instruction, i.e. funct7, funct3, opcode, it is more beneficial to modify exact counterparts of the instructions. For approximate part, R-

Table 3.1. R-type RISC-V instruction structure and modification for approximate ADD, SUB and MUL - XADD, XSUB and XMUL - operations.

Instruction	**[31:25]**	**[24:20]**	**[19:15]**	**[14:12]**	**[11:7]**	**[6:0]**
	funct7	**rs2**	**rs1**	**funct3**	**rd**	**opcode**
ADD	0000000	rs2	rs1	000	rd	0110011
XADD	1000000	rs2	rs1	000	rd	0110011
SUB	0100000	rs2	rs1	000	rd	0110011
XSUB	1100000	rs2	rs1	000	rd	0110011
MUL	0000001	rs2	rs1	000	rd	0110011
XMUL	1000001	rs2	rs1	000	rd	0110011

type instructions are our target. As it can be seen in Table 3.1, both MUL and ADD instructions can be converted to approximate instructions by only changing the most significant bit (MSB) of them which is a part of funct7 codes. As far as we know, there are not any instructions in R-type instructions of RISC-V ISA that uses 1 in MSB, therefore approximate operations can be readily distinguished within the ISA without any clash. Hence, new instructions, shown in Table 3.1, are introduced in such a way that they do not overlap with the existing instructions of whole RISC-V ISA. This small change also facilitates our work in control unit of the core where the MSB of the instruction is just controlled to determine the operation is approximate or not.

3.2. Approximate Units

The architecture of the proposed RISC-V core datapath is customized to process approximate operations together with the exact operation, as described in the previous section. In this section, more detailed information about approximate blocks will be presented. We introduce new approximate designs by focusing on circuit-level approximation while integrating these blocks with a softcore to execute some chosen instructions in approximate mode will be a kind of architecture level approximation. In our design, the precision of the approximate hardware modules is controlled by approximation levels. Our approximation is based on bypassing selected sub-blocks that exist

in the proposed hardware modules. Hence, our approach is different from truncating the least significant bits of the results, as applied in [19, 44]. Approximate operations of our core are addition, subtraction, and multiplication. However, as discussed in the previous section, further approximate modules can also be added to the datapath, as shown in Figure 3.2.

An HLS tool takes the design description in a high-level language as the input and generates the hardware by synthesizing the program constructs to the primitives of the target architecture. HLS provides a fast prototyping and design space exploration for the target hardware. The quality of the design is strongly dependent on the vendor-specific library primitives and user-selected directives that are applied during the code development. Attempting to design an RTL circuit with a high-level language and applying HLS usually produces inefficient hardware. Hence, the approximate adder and multiplier of our core are designed with Verilog Hardware Description Language (HDL) instead of C++. HDL has other several advantages over C++ in such designs because of being a low-level language. It gives us the possibility to control the hardware blocks in the design more efficiently and modify the blocks with more freedom because the hardware blocks used for the operation can be determined directly by the user.

There is another critical point in this configuration, which is worth to mention that describing the core with C++ in HLS and approximate blocks with Verilog creates some difficulties in system integration, although the approximate blocks integration is done at low-level. Approximate operation conditions and instruction parsing should be done in HLS to make the design of the control unit easier for the core.

As can be noticed in the code given in Figure 3.2, a C function, namely *approx_add* which is shown in Figure 3.3, is defined to conduct the approximate operation when it is called. However, this C function does not include the real approximate block. Thus it does not perform the approximate operation; it just represents the approximate operation in HLS. The main task of this pseudo C function, *approx_add*, is to bring the required inputs for the approximate operation and open an output port to return a result to the system.

```
int32_t approx_add(int32_t a, int32_t b, uint8_t approx_add_size, uint8_t add_sub)
{
  int32_t c = 0;
  uint32_t n0 = (a >> 0) & 0x1;
  uint32_t n2 = (a >> 1) & 0x1;
  uint32_t n4 = (a >> 2) & 0x1;
  uint32_t n6 = (a >> 3) & 0x1;
  uint32_t n8 = (a >> 4) & 0x1;
  uint32_t n10 = (a >> 5) & 0x1;
  uint32_t n12 = (a >> 6) & 0x1;
  uint32_t n14 = (a >> 7) & 0x1;
  uint32_t n18 = (b >> 1) & 0x1;
  uint32_t n20 = (b >> 2) & 0x1;
        .
        .
        .
```


Figure 3.3. *approx_add* C function in HLS and dummy operations inside of it.

```
module approx_add (
    input   ap_clk,
    input   ap_rst,
    input   ap_start,
    input   [2:0] approx_level_add,
    input   [2:0] approx_level_sub,
    output  ap_done,
    output  ap_idle,
    output  ap_ready,
    input   [31:0] a,
    input   [31:0] b,
    input   [7:0] approx_add_size,
    input   add_sub,
    output  [31:0] ap_return
);
        .
        .
        .

Sklansky_sizeable u1(
.size_enable(size),
.approx_level(approx_level),
.a(a_p),
.b(b_p),
.ci(1'b0),
.co(),
.s(ap_return));

endmodule //approx_add
```


Figure 3.4. Synthesized verilog code of *approx_add* C function in HLS. This code is modified to call the approximate adder as a submodule.

approx_add function asks from the HLS tool to create necessary port declarations which are used for the operation later, and integrate the approximate operation with the core. However, it is not enough to declare the inputs of the function and leave the body of it empty, because, in synthesizing process, the tool realizes that the block is empty and discard them from the core. We solve this issue by using stub codes for the functions that correspond to the approximate operations. These stub codes include dummy operations as shown in Figure 3.3. Dummy operations are added into the approximate functions so that the tool does not remove the blocks or change the size of their input/outputs. Consequently, the synthesized core contains the synthesized stub codes which we replace by the approximate blocks developed in Verilog. We remove the dummy operations in the *approx_add*, *approx_mul* modules at synthesized Verilog codes, and instantiate our approximate blocks as submodules in these modules, as shown in Figure 3.4 and they perform inexact operations when their service is requested.

Adding our approximate adder to the core is done at HDL level, as stressed in the previous paragraphs. Approximate level control bits are also added to the system at HDL level, not in HLS. Three-bit approximate levels for each operation are added to the top module of the design as inputs. Hence, these bits can be configured independent of the operation of the core. These inputs directly go to *approx_add* or *approx_mul* modules, as shown in Figure 3.4, and controls the gray cells in the adder tree. As it can be noticed in Figure 3.4, our approximate Sklansky adder has a three-bit *approx_level* input. *approx_level_add* or *approx_level_sub* inputs of *approx_add* module is assigned to this input according to the operation type. *add_sub* input of *approx_add* module determines which operation, subtraction or addition, is implemented. If *add_sub* is zero, then *approx_level_add* is appointed to *approx_level*. Otherwise, *approx_level_sub* is appointed to *approx_level*. *approx_level* determines which gray cells are bypassed. If it is "000", it implies no approximation. "001" means the first approximate level, which is bypassing the first seven gray cells. "011" is for the second level, and "111" is used for the third level of approximation. Section 3.2.2 gives more detail about these levels in the adder tree.

We design a dynamically sizeable 32-bit Sklansky parallel-prefix adder and a 16x16 bit Booth Encoded Wallace Tree Multiplier with Verilog HDL as an exact adder and an exact multiplier. Structural coding style is used to facilitate the control of each sub-block more conveniently, although it is more challenging to write the codes in this way. These exact blocks are modified to make approximate calculations. Both approximate designs consist of approximate levels, which can be individually arranged to act as selective-precision approximate operators at run-time. These approximation levels are stored in registers. Each register can be controlled from a circuit, which is out of the core. Thus, the user or the IoT ecosystem can adjust the approximation level while the system is running. We give a direct control for these registers that the approximate levels can be controlled independent of the operation. We only put coarse-grained three levels of approximation for each approximate design to create distinct three levels for power saving modes. Dynamic sizing for unused leading zeros of the operands is controlled directly by datapath. A general diagram for approximation level control and dynamic sizing flow is shown in Figure 3.5.

Subsections describe their dynamically sizeable structure first and then each approximate design separately.

3.2.1. Dynamic Sizing

Data size and dynamically adjustable operations may affect consumed power amount, dramatically, as highlighted in [19]. We considered adjusting the size of our approximate blocks in hardware in accordance with the size of the operands at run-time just before the execution stage to improve power efficiency a bit more. Dynamic sizing produces the minimum size for a block for the exact realization of an operation. Our 32-bit RISC-V core has the capability to implement all 32-bit algebraic operations. However, for the operations that need fewer bits for accurate calculation, unused bits can be fully shut down to avoid consuming power. However, operands change for each operation, so does their size. Therefore, it is difficult to determine a specific size for these blocks. It might be considered to evaluate the dataset before processing it. However, it may cause a big overhead on a processor, especially for big datasets.

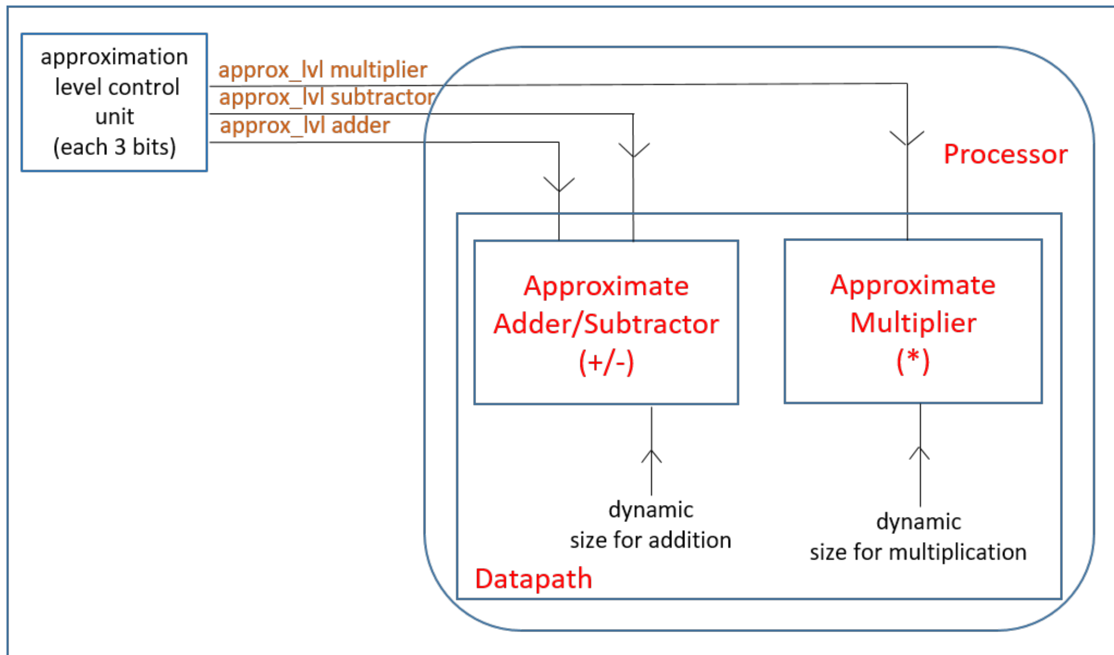

Figure 3.5. Flow diagram of dynamic control of the approximation levels and dynamic sizing.

An alternative and efficient way is adjusting the size of the blocks for each operation. In this study, the size of the adder and multiplier blocks that are used for approximate calculations is dynamically adjustable for each operation. Although this operation is performed on the approximate blocks, it has nothing about approximation, meaning that we still have an accurate adder or multiplier. It is just a way to improve the power efficiency of these blocks a bit more.

Implementation of the dynamic sizing begins inside of the core architecture for this design. The size adjustment part in Figure 3.2 describes the required sizing operations for XADD operation. To determine the minimum size required for a block to handle the operation without any loss, the real size of the operands should be taken into consideration. The real size can be obtained by determining how many bits are actively used to represent the stored value in registers. For an operation, finding all leading zeros of each operand can help us to investigate the real size of the operation. The number of active bits, which is the minimum number of bits that represents the stored value precisely, for an operand can be evaluated by subtracting the register size, XLEN, from the number of the leading zeros as shown in Figure 3.2. Therefore all we

```c
#include <stdint.h>
#include <stdio.h>
#include <assert.h>
#define XLEN 32

uint32_t fast_clz32(uint32_t rs1)
{
if (rs1 == 0)
return XLEN;
assert(sizeof(int) == 4);
return __builtin_clz(rs1);
}
```

Figure 3.6. *fast_clz32* C function to find the leading zeros for 32-bit operands.

need is to find the number of leading zeros with a simple trick.

An instruction set draft for bit manipulation in RISC-V is published on [45], but it has not been officially accepted as a standard by the RISC-V Foundation, yet. In this set, a specific instruction, namely *clz*, which counts leading zeros in a register, is introduced. It may seem useful; however, it requires two more instructions to be added to calculate the real sizes of the operands before the desired operation executed. For example, for single addition or multiplication operation, we need two extra operations for calculating the sizes of the two operands. It will clearly create significant overhead for MAC loops, which is not desired. Instead, *fast_clz32* C function, shown in Figure 3.6, is used to count the leading zeros quickly. *__builtin_clz*() C function, which is provided by GCC, is used to count the leading zeros quickly in this function.

After the approximate operation is received, the leading zeros of the two operands are calculated by this builtin function and subtracted from the full length, 32, to determine the active bits of the operands. The results are given to the approximate blocks, as shown in Figure 3.4 before the operation is executed, and approximate blocks are adjusted accordingly to perform the desired operation more efficiently. For the approximate adder, the active bits are determined for the bigger operands, and the adder size is simply chosen as one bit more of this value due to the possibility of having a carry on MSB. For the multiplier, it is chosen as the sum of the sizes of the two operands.

Dynamic sizing operation for signed values requires a comparison of the two operands. In the case of having negative operands, it is necessary to know which absolute value is greater so as to know whether the result will have leading ones or zeros. Thus, it results in a more complex control mechanism. During our experiments, we found that this complex hardware does not contribute to power reduction. Thus, we use dynamic sizing for operands when they have leading zeroes.

It is observed from several experiments that adjusting the size of the approximate blocks dynamically affects the power significantly. Furthermore, it has no effect on the accuracy of the result, because any operations for the sake of approximate computing are not implemented yet. We can achieve power saving up to 12.8 % compared to the calculations performed on the Exact Part. Experimental results will be explained in Section 4.3.1.

3.2.2. Approximate Adder/Subtractor Design

In this study, we propose a new approximation methodology for parallel-prefix adders. An exact Sklansky adder [46] is designed and modified for the approximate addition operation. We designed a 32-bit signed Sklansky adder [46] and modified it for approximate addition and subtraction operation. The basic addition mechanism of the parallel-prefix adders is given in Figure 3.7. Parallel-prefix adders create a carry chain by taking Propagate (P) and Generate (G) signals as inputs. P0 values represent XOR of two inputs calculated for each bit of two operands, and G0 values AND of each bit of two operands. As it can be seen in Figure 3.8, each row in the tree also includes 32-bit P and G values created by the cells. There are two types of cells in the adder tree, namely black cell, and gray cell. Black cell performs ANDOR operation for the G value of the next row and AND operation for the P value of the next row. The gray cells are used to determine the last values of the G bits. Hence there is no need for further computations of the P values. As a result, they perform only ANDOR operation to find the last G value for each column. After all final carry values are obtained, they

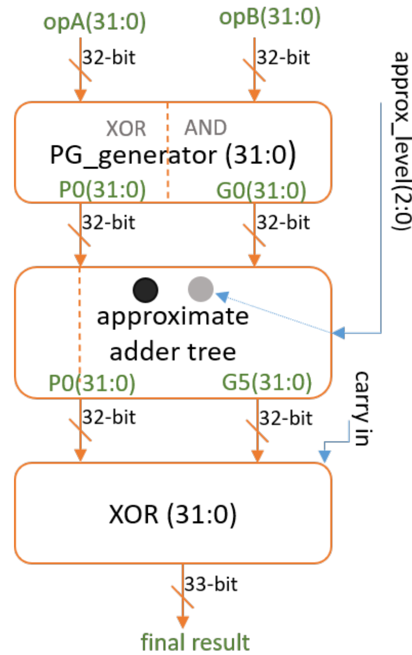

Figure 3.7. Basic addition mechanism of the tree adders. Approximation is implemented in the gray cells of the adder tree.

XORed with the P values calculated from the inputs, and the final result of the 32-bit addition operation is obtained.

The coding of the adder tree is implemented row by row. Black and gray cells in a row of the adder tree are coded separately by using *generate* construct. P and G networks in the tree are coded as two 6x32 arrays, including the P and G inputs. Each next bit for P and G is determined by the corresponding cell. If there is no associated cell for a position in a row, shown as a white cell in Figure 3.8, then P and G values are directly buffered to the next row.

Approximation is performed on gray cells. As shown in Figure 3.8, gray cells are the last cells to determine the final G value for a column in the adder tree. Instead of performing this last ANDOR operation of gray cells, the G value of the previous row can be directly taken as the final G value in order to save the power consumed by these cells. In other words, they can be omitted for the sake of power efficiency, although the final result may be inaccurate. Three approximation levels are defined to control accuracy. The first level omits the gray cells used for the least significant byte.

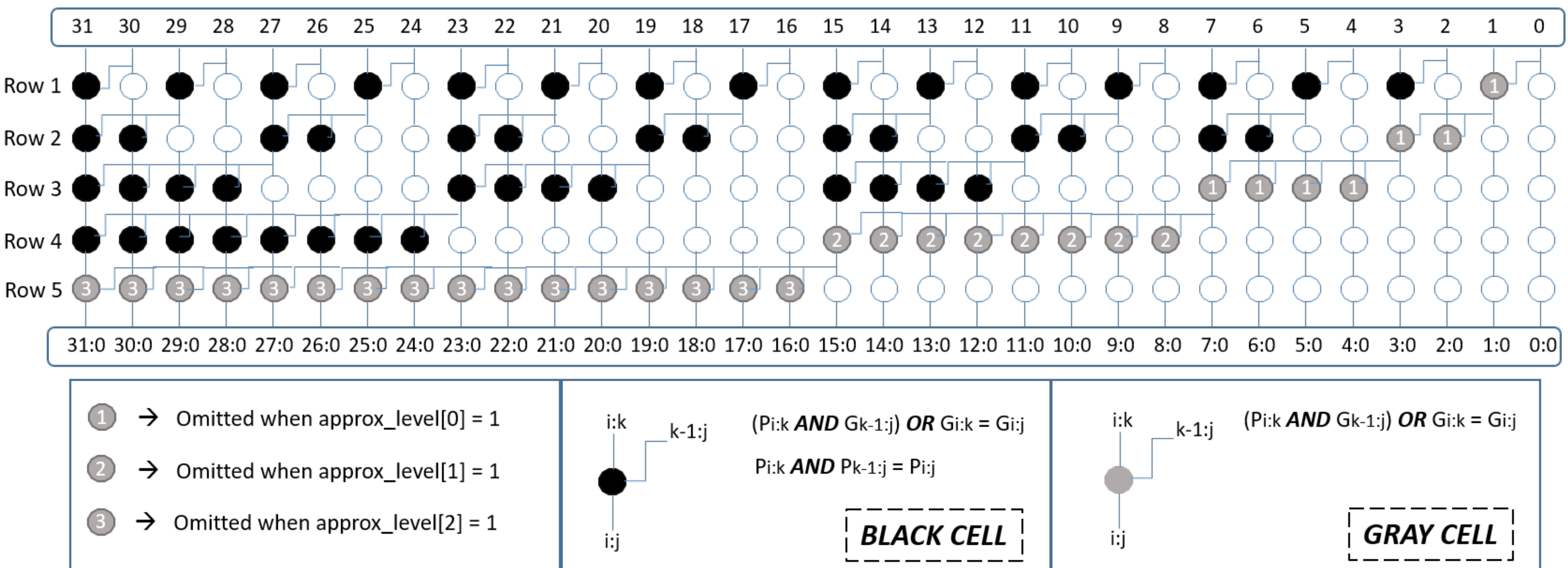

Figure 3.8. Sklansky adder tree used for approximate adder design.

The second level controls the gray cells used for the next eight bits, and the third level gives the decision about the gray cells used the last sixteen bits. We assume that when the approximation level is two, the first level is already approximated, and when it is three, the first two levels are also approximated, although all levels can be controlled independently. Thus, we create test scenarios based on this assumption. When we refer to the approximation level of two, it means that all gray cells in the first four rows are approximated. For the approximation level of three, it omits all gray cells in the tree.

It should be highlighted that this approximate operation is different from truncating LSBs [19, 44] because we do not remove these bits. Instead, we bypass the gray cells and directly use the input value of the gray cells as the tree outputs; hence it still affects the final results.

Subtractor is implemented by negating the second operand and using the same adder block when approximate subtraction (XSUB) operation comes. This provides us an approximate subtractor without consuming any additional resources on-chip.

3.2.3. Error Probability Analysis of the Proposed Adder

The proposed approximate adder can give erroneous results, and the probability of the inaccurate results can be evaluated with different methodologies proposed in literature [15, 47, 48]. [15] tries to model the error probability introduced by their specific adder, GeAr, while [47] calculates the error probability and the mean error in a conventional probability evaluation. [48] provides an efficient error calculation methodology to evaluate arithmetic error rate for deterministic approximate adder architectures by using visibility phenomenon, which shows the restriction of determining each output bit with a subset of the input bits. In this paper, we calculated the error probability of the proposed adder by calculating the error probability for each bit of the output, as calculated in [47]. To do so, we calculated the error probability provided by gray cells that we bypass when the approximate operations are executed.

Table 3.2. Truth table for ANDOR operation in the gray cells and its comparison with the approximate results.

P*i*:*k*	**G***i*:*k*	**G***k*-*1*:*j*	real results	approximate results = (**G***i*:*k*)
0	0	0	0	0
0	0	1	0	0
0	1	0	1	1
0	1	1	1	1
1	0	0	0	0
1	0	1	1	0
1	1	0	1	1
1	1	1	1	1

P*i*:*k* >> propagate value from previous row
G*i*:*k* >> generate value from previous row
G*k*-*1*: *j* >> accumulated generate value from previous columns of previous row

In the approximation process, we omit gray cells in different levels and directly buffer previous G values to the outputs of the adder tree. In a gray cell, ANDOR operation is implemented. Instead of this ANDOR operation, buffering $G_{i:k}$ to the output directly can only be wrong for one possibility out of the eight, as shown with the gray color in Table 3.2. Π_1 is the probability of being true, and Π_0 is the probability of being false. Let Π_{E3} denote the error probability for all gray cells when the approximate level is 3. Then the error probability Π_{E3} becomes the multiplication of the probabilities of $P_{i:k}$ and $G_{k-1:j}$ being true, and the probabilities of $G_{i:k}$ being false as shown in Equation 3.1.

$$\Pi_{E3} = \Pi_1(P_{i:k}) \cdot \Pi_0(G_{i:k}) \cdot \Pi_1(G_{k-1:j}) \tag{3.1}$$

For the first gray cell, at (1:1) - row 1 : column 1, in Figure 3.8 - $P_{0:1}$ is the XOR of the two input, therefore it is true with 1/2 probability. $G_{0:1}$ is the result of AND operation, thus its probability of being false is 3/4. $G_{0:0}$ is also the result of AND operation; thus, its probability of being true is 1/4. Then, the error probability

for the first gray cell becomes (1/2)*(3/4)*(1/4) = 3/32 = 0.09. For the gray cell at (2:2), the error probability depends on the probability of the previous gray cell. It is (1/2)*(3/4)*(29/32) = 87/256 = 0.34. For the gray cell at (2:3), the calculation becomes more difficult because of the additional black cell in the chain. The black cell at (1:3) output should be P = 1 and G = 0 for the wrong case. The possibility for P = 1 is (1/2)*(1/2) = 1/4 and for G = 0 is $(1/2) * (3/4) * (3/4) + (1/2) * (3/4) * (3/4) + (1/2) * (3/4) * (1/4) = 21/32$. Then the error probability for this gray cell (2:3) is (1/4)*(21/32)*(29/32) = 0.15. Calculating the error probabilities becomes much more complicated for the further gray cells because of the difficulties in the evaluation of accumulated probabilities for $P_{i:k}$, $G_{i:k}$, and $G_{k-1:j}$ which are the inputs of these gray cells. For the sake of simplicity, we prepare tree tables which show the probabilities of being true for P and G values after each operation at the associated cell in the adder tree and error probabilities after the last gray cells, as colored cells, in Table 3.3.

In the adder tree, we defined three approximate levels. The first level includes the gray cells in the first three rows of the tree, the second level covers the gray cells in the first four rows, and the third level controls all gray cells. For the approximation of level one, although we do not have any approximation on the gray cells in the fourth and fifth rows, the omission of the gray cells in the first three rows also has an effect on the fourth and fifth rows where the gray cells take the outputs of the last gray cell in the third row. The same effect also exists between the fourth and fifth rows in the approximation level of two. Outputs of the last gray cell in the third row become $G_{k-1:j}$ for the gray cells in the fourth row, and it may affect the result with two possibilities out of eight, as shown in Table 3.4. For the gray cells in the fourth and fifth row, the error probability can be calculated in the same manner. The error probabilities in these cases, Π_{E1} and Π_{E2}, are equal to the sum of the probabilities of the two conditions, which makes the result independent of the probability of $G_{k-1:j}$ being true or false. Therefore, the error probability at the next row where the approximate operations are not used becomes the multiplication of the probabilities of $P_{i:k}$ for being true, and $G_{i:k}$

Table 3.3. The probability tree for the proposed adder when the approximation level is 3. The colored cells show the probability of being false for G values, which are actually the error probabilities at the corresponding bits.

Probability of being true for P values after the operations in each column of each row.																																
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
row 0	0.50	0.50	0.50	0.50	0.50	0.50	0.50	0.50	0.50	0.50	0.50	0.50	0.50	0.50	0.50	0.50	0.50	0.50	0.50	0.50	0.50	0.50	0.50	0.50	0.50	0.50	0.50	0.50	0.50	0.50	0.50	-
row 1	0.25	0.50	0.25	0.50	0.25	0.50	0.25	0.50	0.25	0.50	0.25	0.50	0.25	0.50	0.25	0.50	0.25	0.50	0.25	0.50	0.25	0.50	0.25	0.50	0.25	0.50	0.25	0.50	0.25	0.50	-	-
row 2	0.06	0.13	0.25	0.50	0.06	0.13	0.25	0.50	0.06	0.13	0.25	0.50	0.06	0.13	0.25	0.50	0.06	0.13	0.25	0.50	0.06	0.13	0.25	0.50	0.06	0.13	0.25	0.50	-	-	-	-
row 3	0.00	0.01	0.02	0.03	0.06	0.13	0.25	0.50	0.00	0.01	0.02	0.03	0.06	0.13	0.25	0.50	0.00	0.01	0.02	0.03	0.06	0.13	0.25	0.50	-	-	-	-	-	-	-	-
row 4	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.02	0.03	0.06	0.13	0.25	0.50	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
row 5	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Probability of being true for G values after the operations in each column of each row with approximation level = 3. Note that colored cells show the error probability of the gray cell which is the possibility of the condition shown as gray in ANDOR truth table.																																
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
row 0	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25
row 1	0.34	0.25	0.34	0.25	0.34	0.25	0.34	0.25	0.34	0.25	0.34	0.25	0.34	0.25	0.34	0.25	0.34	0.25	0.34	0.25	0.34	0.25	0.34	0.25	0.34	0.25	0.34	0.25	0.34	0.25	0.09	-
row 2	0.40	0.38	0.34	0.25	0.40	0.38	0.34	0.25	0.40	0.38	0.34	0.25	0.40	0.38	0.34	0.25	0.40	0.38	0.34	0.25	0.40	0.38	0.34	0.25	0.40	0.38	0.34	0.25	0.15	0.34	-	-
row 3	0.42	0.41	0.41	0.40	0.40	0.38	0.34	0.25	0.42	0.41	0.41	0.40	0.40	0.38	0.34	0.25	0.42	0.41	0.41	0.40	0.40	0.38	0.34	0.25	0.03	0.07	0.14	0.32	-	-	-	-
row 4	0.42	0.41	0.41	0.41	0.42	0.41	0.41	0.41	0.42	0.41	0.41	0.40	0.40	0.38	0.34	0.25	0.002	0.004	0.01	0.02	0.04	0.08	0.16	0.36	-	-	-	-	-	-	-	-
row 5	0.00001	0.00002	0.00004	0.00007	0.0001	0.0003	0.0006	0.0012	0.002	0.005	0.009	0.02	0.04	0.08	0.16	0.37	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Table 3.4. Truth table for ANDOR operation in the gray cells and its comparison with the effect of the approximate results on the other gray cells.

Pi:k	**G**i:k	**G**k-1:j	real results	approximate results = (**G**i:k)
0	0	0	0	0
0	0	1	0	0
0	1	0	1	1
0	1	1	1	1
1	0	0	0	1
1	0	1	1	0
1	1	0	1	1
1	1	1	1	1

Pi:k >> propagate value from previous row
Gi:k >> generate value from previous row
Gk-1: j >> accumulated generate value from previous columns of previous row

being false as shown in Equation 3.2.

$$\Pi_{E1} = \Pi_{E2} = \Pi_1(P_{i:k}) \cdot \Pi_0(G_{i:k}) \tag{3.2}$$

The tree table for the probabilities of being true for G values and the error probability for each bit is given in Tables 3.5 and 3.6 for the approximate levels of 1 and 2, respectively. Note that the yellow cells in Tables 3.5 and 3.6 show the error probabilities of the omitted gray cells due to approximation, while the blue cells show the error probabilities on corresponding cells caused by the approximation on the yellow cells.

The final result of the approximate adder is the XOR of the G5 bits, which is the G outputs of the last gray cells, with the P0, as shown in the diagram of Figure 3.7. However, this XOR operation does not affect error probability. Hence, the error

Table 3.5. The probability tree for the proposed adder when the approximation level is 1.

Probability of being true for G values after the operations in each column of each row with approximation level = 1.
Note that colored cells show the error probability of the gray cell which is the possibility of the condition shown as gray in ANDOR truth table.

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
row 0	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25
row 1	0.34	0.25	0.34	0.25	0.34	0.25	0.34	0.25	0.34	0.25	0.34	0.25	0.34	0.25	0.34	0.25	0.34	0.25	0.34	0.25	0.34	0.25	0.34	0.25	0.34	0.25	0.34	0.25	0.34	0.25	0.09	-
row 2	0.40	0.38	0.34	0.25	0.40	0.38	0.34	0.25	0.40	0.38	0.34	0.25	0.40	0.38	0.34	0.25	0.40	0.38	0.34	0.25	0.40	0.38	0.34	0.25	0.40	0.38	0.34	0.25	0.15	0.34	-	-
row 3	0.42	0.41	0.41	0.40	0.40	0.38	0.34	0.25	0.42	0.41	0.41	0.40	0.40	0.38	0.34	0.25	0.42	0.41	0.41	0.40	0.40	0.38	0.34	0.25	0.03	0.07	0.14	0.32	-	-	-	-
row 4	0.42	0.41	0.41	0.41	0.42	0.41	0.41	0.41	0.42	0.41	0.41	0.40	0.40	0.38	0.34	0.25	0.002	0.003	0.01	0.01	0.03	0.05	0.09	0.13	-	-	-	-	-	-	-	-
row 5	0.00001	0.00001	0.00003	0.00005	0.0001	0.0002	0.0004	0.0008	0.002	0.003	0.006	0.01	0.03	0.05	0.09	0.13	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Table 3.6. The probability tree for the proposed adder when the approximation level is 2.

Probability of being true for G values after the operations in each column of each row with approximation level = 2.
Note that colored cells show the error probability of the gray cell which is the possibility of the condition shown as gray in ANDOR truth table.

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
row 0	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25
row 1	0.34	0.25	0.34	0.25	0.34	0.25	0.34	0.25	0.34	0.25	0.34	0.25	0.34	0.25	0.34	0.25	0.34	0.25	0.34	0.25	0.34	0.25	0.34	0.25	0.34	0.25	0.34	0.25	0.34	0.25	0.09	-
row 2	0.40	0.38	0.34	0.25	0.40	0.38	0.34	0.25	0.40	0.38	0.34	0.25	0.40	0.38	0.34	0.25	0.40	0.38	0.34	0.25	0.40	0.38	0.34	0.25	0.40	0.38	0.34	0.25	0.15	0.34	-	-
row 3	0.42	0.41	0.41	0.40	0.40	0.38	0.34	0.25	0.42	0.41	0.41	0.40	0.40	0.38	0.34	0.25	0.42	0.41	0.41	0.40	0.40	0.38	0.34	0.25	0.03	0.07	0.14	0.32	-	-	-	-
row 4	0.42	0.41	0.41	0.41	0.42	0.41	0.41	0.41	0.42	0.41	0.41	0.40	0.40	0.38	0.34	0.25	0.002	0.004	0.01	0.02	0.04	0.08	0.16	0.36	-	-	-	-	-	-	-	-
row 5	0.00001	0.00001	0.00003	0.00005	0.0001	0.0002	0.0004	0.0008	0.002	0.003	0.006	0.01	0.03	0.05	0.09	0.13	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

probability of each bit in the final results will be the same with the one calculated for the adder tree and shown as colored in Table 3.3 for the first approximation level, and in Tables 3.5 and 3.6 for the second and third levels of approximation in this adder design.

Error probability analysis can be beneficial to develop more accurate approximation methodologies from this topology, although we do not make use of it in this study. The first gray cell of each row has a higher error probability than the others, as shown in Tables 3.5 and 3.6. This may cause significant error rates at corresponding bits of the output. For example, when the approximation level is 3, gray cells at (2:2), (3:4), (4:8), and (5:16) have an error probability of about 0.35, which is not small, as shown in Table 3.3. The error probabilities on the other cells are lower than the half of this rate. This means that we are more likely to have erroneous results on these bits. These gray cells can be excluded from the approximation, and more precise results can be achieved.

3.2.4. Approximate Multiplier Design

16x16 bit booth encoded Wallace tree exact multiplier is designed in Verilog HDL. Booth encoding idea is chosen to lower the number of partial products (PP) in multiplication operation, which leads to faster calculation of the final results and reduce the consumed power and covered area, substantially. In this multiplier design, partial products are computed as Radix-4 Booth in the same manner proposed in [49]. A conventional Booth encoder and selector for the Radix-4 Booth proposed in [50] are chosen and implemented, as shown in Figure 3.9. After Radix-4 Booth operation, the number of the PPs decreases from N to $(N+1)/2$ while the length of each PP increased by one bit to accommodate the double of the multiplicand. In our multiplier, the number of the PP decreases from 16 to 9, and the length of the PPs becomes 17-bit due to Radix-4 Booth implementation.

In Radix-4 Booth encoding design; instead of having 0, Y, $2Y$, and $3Y$, we have 0, Y, $2Y$, $-Y$, and $-2Y$ as partial products where Y is the *multiplicand*. According to

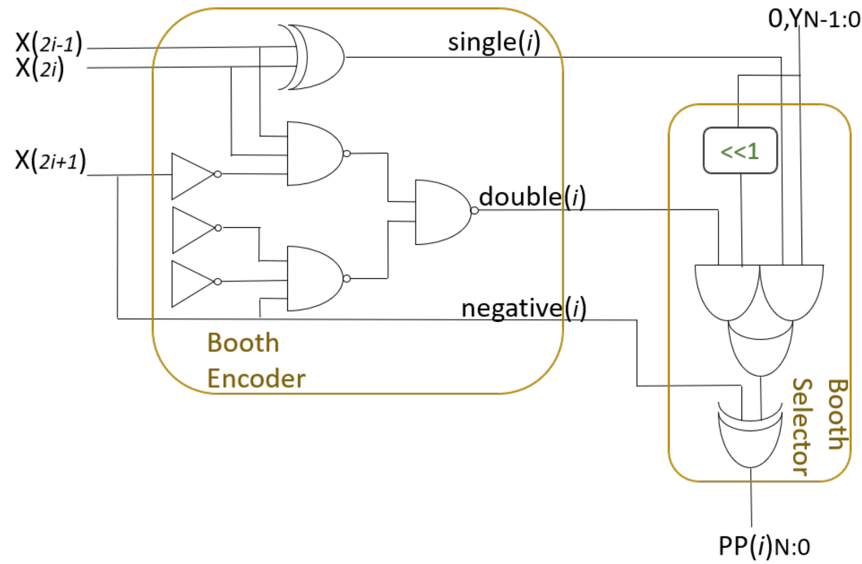

Figure 3.9. Booth encoder and selector circuits to generate partial products.

Table 3.7. Truth table for booth selection.

Inputs			Partial Products	Booth Selects		
X(2i+1)	X(2i)	X(2i+1)	PP(i)	single(i)	double(i)	negative(i)
0	0	0	0	0	0	0
0	0	1	Y	1	0	0
0	1	0	Y	1	0	0
0	1	1	2Y	0	1	0
1	0	0	-Y	0	0	1
1	0	1	-2Y	1	1	1
1	1	0	-Y	1	0	1
1	1	1	-0(=0)	0	0	1

three adjacent X - *multiplier* - value, partial product type is chosen by single, double, or negative signals. The truth table and obtained partial products for these signals are given in Table 3.7. Constructed partial product structure is also given in Figure 3.10. Like the approximate adder, the multiplier block is also designed in such a way that its size is dynamically adjustable during execution in accordance with the operand sizes.

After partial products are obtained, the Wallace tree structure depicted in Figure 3.11 is built to sum all partial products. Carry Save Adders (CSA) with three (3:2), and four (4:2) inputs are used for fast calculation. For the CSAs, we used an approximation methodology described in [47], which proposes approximate (3:2) and

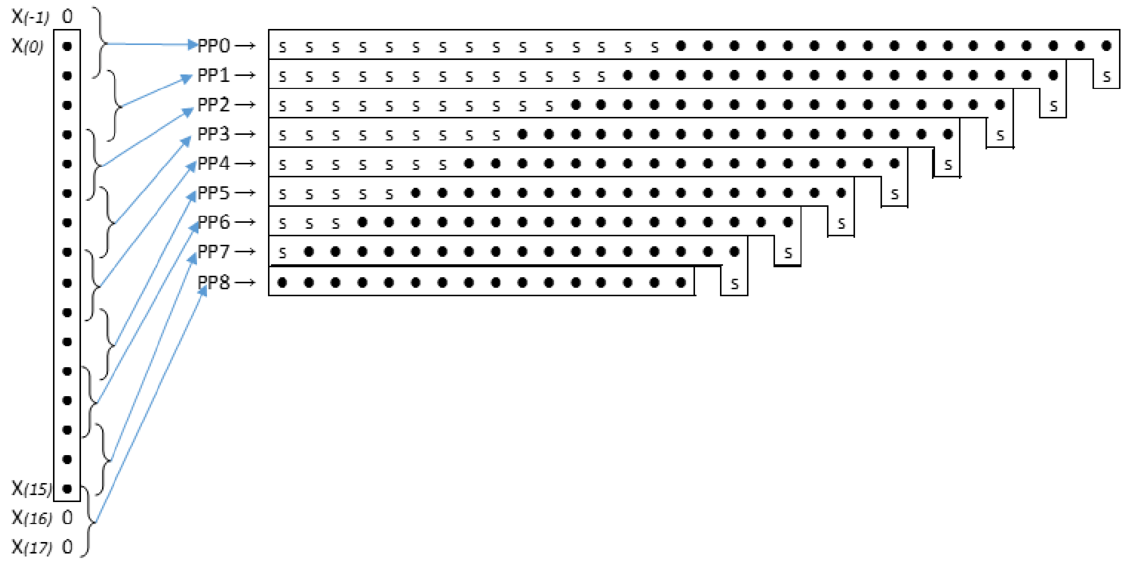

Figure 3.10. Partial products structure after booth encoding. *s* represents the sign extension and black dots are the partial product bits.

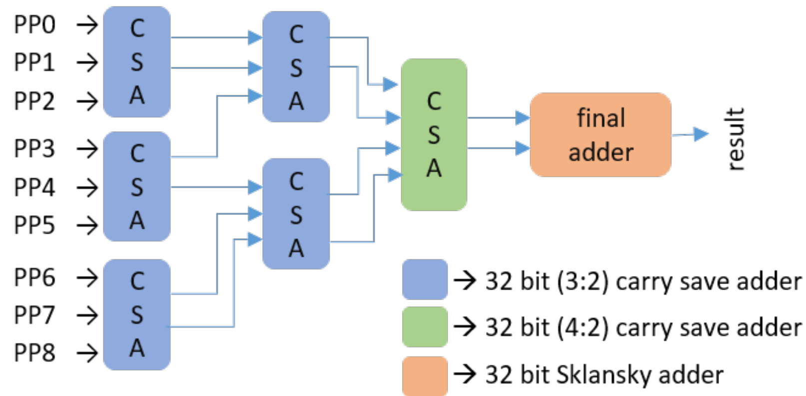

Figure 3.11. Wallace tree structure for the summation of partial products. All CSAs and final adder are approximate.

(4:2) compressors. The other approximate part of this multiplier is the final adder. Instead of using an exact carry propagate adder (CPA), we used the same Sklansky adder designed for the approximate adder. Hence, the same approximation levels and structure are valid for this multiplier, too. Apart from the adders in the Wallace tree, the other parts make exact operations to guarantee the convergence of the outputs to the real results.

4. EXPERIMENTS

We implemented some basic ML algorithms on our microprocessor to evaluate how much power efficiency can be provided by the processor at the expense of accuracy in such applications. Besides, the degree of accuracy loss relative to the amount of saved energy is also essential to measure the effectiveness of the design. K-nearest neighbor (KNN), K-means (KM), and ANN algorithms are chosen and implemented on this CPU for clustering and classifying different types of datasets to make a proof-of-concept study. We mainly compared the results obtained by running the codes in Exact Part with the results in Approximate Part. In Approximate Block, we also used some different approximate adders and multipliers proposed in the literature to make some comparisons between our designs and others and show the design flexibility in terms of modifying subblocks of the core easily.

4.1. Experimental Setup

We synthesized a sample core for Zynq-7000 (xc7z020clg484-1) FPGA by using Xilinx Vivado HLS 18.1. It can run safely with a clock frequency of 100 MHz. This CPU occupies about 3% of on-chip resources in Zynq-7000 FPGA. Resource utilization on the target FPGA can be seen in Table 4.1. For memories, we use 40 KB BRAM for instruction memory and 90 KB BRAM for data memory. Synthesized cores are merged with approximate blocks as described in Section 3.2, and experiments are conducted on the approximate processor.

C codes for KNN, KM, and ANN algorithms are specified, and which parts of the codes will be implemented in an approximate manner is decided by software as described in Section 4.1.1. These C codes for the benchmark algorithms are compiled with 32-bit RISC-V GCC [51] for exact and approximate executions. Compiling the C codes with RISC-V GCC is implemented with Linux commands shown in Appendix A.3. *start.S* and *link.ld* files, which are also given in Appendix A.1 and Appendix A.2, respectively, together with C codes, are used as inputs. *start.S* file describes the

Table 4.1. Resource utilization of the core in the target FPGA without approximate blocks.

Resource	**Utilization**	**Available**	**Utilization (%)**
LUT	1725	53200	3.24
LUTRAM	2	17400	0.01
FF	938	106400	0.88
DSP	6	220	2.72

general flow of the compilation process, while *link.ld* file describes basic initialization steps related to memory operations. Memory sizes are also specified in *link.ld* file. Hence, it should be in accordance with the addresses used in load-store operations. Implemented commands create two output files, which are machine code versions of the instructions, and the data should be saved in the data memory at the beginning of the code. These two files are given to the core as inputs, and their data are read by *readmemh* Verilog command in instruction and data memories, respectively. After instruction memory takes the codes and required values are written into the data memory, codes are begun to run when the start signal, *ap_start*, is given. Machine codes for the approximate operations are created manually from the exact version of the code. Which instructions will be approximate is firstly determined, and their machine codes are changed with the approximate counterparts.

We measure two things in the experiments, namely, power consumption and the accuracy of the result. The last two subsections explain the detail of the setup for these measurements. The first subsection describes the modifications that are done on the software side for marking the operations as approximate or exact.

4.1.1. Approximate Regions in the Codes

In this study, classification, clustering, and artificial neural network algorithms are conducted on our core. Addition, subtraction, and multiplication operations in the main calculation loops are approximately calculated. It is worth to note that we are

not doing any approximate operations for datasets as proposed in [19]. Approximable regions can be created in the code with the help of pragma and attribute operations in C, as described in [12]. The code generator desicribed in [12] adds our custom instructions into the RISC-V GCC with the help of riscv-binutils [52]. After the specialization of the compiler for our approximate instructions, the machine codes created by this compiler can be directly used in our core. Details of this process are described in our published paper [53].

4.1.2. Accuracy Evaluation

For accuracy measurement, exact benchmark codes are executed by using the behavioral simulator of Xilinx Vivado HLS. The files that contain the results of exact executions are regarded as the golden files. Then, the same codes are executed on the synthesized core to verify that its results are the same as the ones in the golden files. Firstly, the core is started to compute the results in Exact parts. These results should be the same with the results coming from the HLS version of the core in order to assure that the machine codes are appropriately generated, and core performs without errors. After this validation, all approximate tests are run on the processor, and results are saved on a text file to calculate their accuracy. Simulations are performed in Xilinx Vivado 18.1 tool, and a Verilog testbench shown in Appendix A.5, which directly reads the memory addresses where the results will be stored after execution is finished, is written to run the tests and write the results into a text file automatically for each case. These results are compared in an excel sheet.

The accuracy metric here is top-1 accuracy, which means that the test results must be exactly the expected answer. Let n denote the number of tests carried out in both exact CPU and approximate CPU, which uses approximate operators only in the annotated regions. Approximate CPU tries to find exactly the same class as the exact CPU model finds. We compare all the results obtained from both parts for n tests, and the percentage of matching cases in all tests gives us the accuracy rate.

4.1.3. Power Consumption Analysis

Making power analysis is the other part of the experiments. Analyses of the consumed power for running codes on the designed core are performed on Power Estimator tool of Xilinx Vivado 18.1. To obtain the best power estimation with high confidence in the tool, these steps should be followed:

(i) A constraint file that describes the inputs and outputs ports, clock specifications, timing exceptions should be added to the design.
(ii) Design should be synthesized and implemented on a target FPGA to obtain placement of the design blocks on the physical device.
(iii) A SAIF (Switching Activity Interchange Format) file, which calculates the toggle rate of the signals should be used to estimate dynamic power accurately. Without a SAIF file, the tool gives a default toggle value, 12.5 %, for all signals except clock for dynamic power estimation. As can be guessed, this does not assure the accuracy of the power estimation. SAIF files can be obtained from the simulation.
(iv) To receive the best estimation from the tool, SAIF file should be written during the post-implementation functional simulation, because this simulation performs the operations based on the placement implemented on the target FPGA. However, it consumes too much time and progresses very slowly compared to the other functional simulation types.
(v) After implementing the design, reading this SAIF file in the report power option gives a power estimation with high confidence.

A constraint file, including clock information, 100 MHz system clock, and primary input-output (I/O) ports on the target FPGA (Zedboard) is written as shown in Appendix A.6. Our core is synthesized and implemented in the tool for the FPGA. After the implementation process, we begin the post-implementation functional simulations and save the toggle counts of all signals in a SAIF file with the commands shown in Appendix A.7. This file is given to the Power Estimator tool to calculate the dynamic power of the system more accurately. All power calculations are done in this

manner, although creating SAIF files with post-implementation simulation requires a considerable amount of time. Power consumption of all cases is recorded and compared with each other.

In experiments, we report dynamic power consumption on the core, because the main difference between the exact and approximate operation can be observed with the change in the dynamic activity of the core. Besides, static power consumption in the conventional FPGAs mainly stemming from the leakages that are independent of the operation. On the other hand, due to the domination of static power in FPGAs, it may not be convenient to design the final product as an FPGA, hence presented power saving here is just to verify the idea. An ASIC implementation where static power is negligible (<1%) may be more beneficial to save significant total power in these applications. ASIC implementation of the core is also done, and obtained results are presented in Section 5.1.

4.2. Datasets and Algorithms

In this section, chosen datasets and algorithms for experiments are explained in detail.

4.2.1. Datasets

KNN, KM, and ANN algorithms are implemented with different datasets to observe the change in the results under the data with different lengths and attributes. Three different datasets are chosen and used in these three algorithms. The first dataset is a wireless indoor localization dataset (W) used in [54, 55]. These data are collected in indoor space by observing signal strengths of seven WiFi signals visible on a smartphone. The decision variable is one of the four rooms. This dataset has a maximum of 8-bit integer numbers. Two different datasets for implementation are extracted from this dataset. One includes 200 data, which refers to short data (SW), and the other contains 2000 data, which is long dataset (LW). Specifying two different lengths in one dataset will help us to measure the effect of the amount of the sample data in the

Table 4.2. Implemented datasets and their properties.

Datasets	**Abbre-viations**	**# of test points**	**# of attributes**	**# of class**	**max. bit-length**
Wifi Localization Data Short Version	SW	200	7	4	8
Wifi Localization Data Long Version	LW	2000	7	4	8
Robot Sensor Data Short Version	SR	200	4	4	16
Robot Sensor Data Long Version	LR	2000	4	4	16
Banknote Authorization Data	B	300	4	2	16

results. The second dataset is the wall following robot sensor data (R) used in [56]. This dataset includes four attributes to decide which direction the robot is following. It contains floating numbers, which is converted to 16-bit integer numbers in order to function in our system. This dataset represents the big value dataset because of including some 16-bit values. Therefore it also has long and short versions (SR and LR). The last dataset is the banknote authentication dataset (B). Like the other datasets, this is also taken from UCI Machine Learning Repository [57]. This dataset has four attributes to decide the two classes. This dataset is chosen to observe the accuracy of the results when we have only two classes. In total, we have five datasets obtained from three different sources, and they are combined with three algorithms to obtain 15 different benchmarks to compare the results. All datasets and their properties are shown in Table 4.2.

4.2.2. Algorithms

KNN is one of the essential classification algorithms in ML and used in some IoT applications [26]. It finds intense application in pattern recognition, data mining,

```
// This function finds classification of point p using
// k nearest neighbour algorithm. It assumes only two
// groups and returns 0 if p belongs to group 0, else
// 1 (belongs to group 1).
int classifyAPoint(Point* arr, int n, int k, Point p)
{
    // Fill distances of all points from p
    for (int i = 0; i < n; i++){
        arr[i].distance =
                    (arr[i].x - p.x) * (arr[i].x - p.x) +
                    (arr[i].z - p.z) * (arr[i].z - p.z) +
                    (arr[i].k - p.k) * (arr[i].k - p.k) +
                    (arr[i].y - p.y) * (arr[i].y - p.y);
    }

    // Sort the Points by distance from p
    selectionsort(arr, n,k);

   // Now consider the first k elements and only 3 groups
```

Figure 4.1. *classifyAPoint* function and MAC loop, highlighted with yellow circle, inside of it where the approximate operations implemented.

and intrusion detection. Because of trying to suggest a low power approximate microprocessor for these kinds of applications, it is beneficial for us to implement this algorithm in our processor to show the improvement. We give some prior data in the mentioned datasets with their class information for training. Then we give some set of test data that are allocated in the best-suited class by analyzing the training set. This analysis is done with the help of distance calculation loops inside a C function called *classifyAPoint* shown in Figure 4.1. In these loops, for a given test point, its distances to the training points are calculated. Then the classes of the shortest three points are found by *selectionsort* function. Finally, the most used class for these three points determines the class of the given point. This loop is implemented for all test points. Distances calculations are MAC loops, highlighted in Figure 4.1, which is quite suitable to implement approximately. It is worth noting that this structure of the KNN algorithm allows us to make a case study for real-time on-chip training. We put some reference data points with known classes into the data memory so that incoming data can be classified correctly and in real-time.

The second ML algorithm for this study is KM, which is also an elementary ML algorithm that makes clustering operation for given datasets. As its name suggests, the KM algorithm identifies k number of centroids, and then allocates every data

```c
/*** main loop */
do {
    /* save error from last step */
    old_error = error, error = 0;

    /* clear old counts and temp centroids */
    for (i = 0; i < k; counts[i++] = 0) {
        for (j = 0; j < m; c1[i][j++] = 0);
    }

    for (h = 0; h < n; h++) {
        /* identify the closest cluster */
        int min_distance = IMAX;
        for (i = 0; i < k; i++){
            int distance = 0;
           // for( j = m; j-- > 0; distance += pow(data[h][j] - c[i][j], 2));
            for (j = m; j-- > 0; j=j){
                distance = distance + (data[h][j] - c[i][j])*(data[h][j] - c[i][j]);}
            if (distance < min_distance) {
                labels[h] = i;
                min_distance = distance;
            }
        }
        /* update size and temp centroid of the destination cluster */
        for (j = m; j-- > 0; c1[labels[h]][j] += data[h][j]);
        counts[labels[h]]++;
        /* update standard error */
        error += min_distance;
    }

    for (i = 0; i < k; i++) { /* update all centroids */
        for (j = 0; j < m; j++) {
            c[i][j] = counts[i] ? c1[i][j] / counts[i] : c1[i][j];
        }
    }

} while (abs(error - old_error) > t);
```

Figure 4.2. Main error calculation loop for implemented KM code and MAC loop, highlighted with yellow circle, where the approximate operation implemented.

point to the nearest cluster, while keeping the centroids as small as possible. Unlike KNN, there is no prior training in KM, which makes it unsupervised learning. A similar distance calculation mechanism can be seen in KM, too. Therefore similar MAC loop approximation is possible for our case study. Main error calculation loop for implemented KM code and where the approximate operations are used shown in Figure 4.2.

Our last algorithm is ANN, which is also very popular in ML. A specific ANN algorithm, genann proposed in [58], is specialized for our case with our datasets. We do training with our datasets and obtain weights for each dataset. Then we give these weights as inputs together with test data to detect their class in our core. Unlike KNN, the training session is implemented outside of the core for the ANN algorithm. In the

ANN algorithm, the input number is the attribute number of the used dataset, 4 and 7, hidden layer is one which has two units for the datasets that have four attributes, and four units for the datasets that have seven attributes. Two units for the output layer are used to determine the classes of the test data. Loop operations in all layers of the neural network are approximately operated, as shown in Figure 4.3, to make a case study for our work.

It should be noted that the calculations in ANN experiments contain only addition and multiplication operations. Thus, we confined our approximation level only to addition and multiplication in this code. That is the reason why we can observe the impact of the approximate subtraction operations on the power consumption in only KNN and KM tests.

A considerable amount of the tests for these benchmark ML algorithms with chosen datasets makes it difficult to conduct all the experiments manually. This situation pushed us to write a script for the automatic generation of the Vivado projects for each specific case. Written scripts are given in Appendix A.8. This script creates Vivado projects with chosen settings, synthesize and implement them. After the implementation, post-implementation functional simulation automatically starts, and saves a SAIF file. It finally reports power including the SAIF file and saves it as a text file. This script generation allows us to run the systems without a GUI, which helps us to save a notable amount of time.

4.3. Experiments

4.3.1. Dynamic Sizing

The first experimental point is to measure the effect of dynamic sizing on power consumption. Here, $AppANew$ refers to the experiments when the approximate operations are confined to addition operations. $AppMNew$ represents the case when the only approximate operations are multiplications. Finally, $AppAMNew$ represents when both of our approximate operators are running. All tests are implemented with

```
int const *genann_run(genann const *ann, int const *inputs) {
    int const *w = ann->weight;
    int *o = ann->output + ann->inputs;
    int const *i = ann->output;
    memcpy(ann->output, inputs, sizeof(int) * ann->inputs);
    int h, j, k;

    if (!ann->hidden_layers) {//no hidden layers
        int *ret = o;
        for (j = 0; j < ann->outputs; ++j) {
            int sum = *w++ * -1;
            sum = sum * 256;
            for (k = 0; k < ann->inputs; ++k) {
                sum += *w++ * i[k];
            }
            *o++ = genann_activation( sum);
        }
        return ret;
    }

    /* Figure input layer */
    for (j = 0; j < ann->hidden; ++j) {
        int sum = *w++ * -1;
        sum = sum * 256;
        for (k = 0; k < ann->inputs; ++k) {
            sum += *w++ * i[k];
        }
        *o++ = genann_activation( sum);
    }
    i += ann->inputs;

    /* Figure hidden layers, if any. */
    for (h = 1; h < ann->hidden_layers; ++h) {
        for (j = 0; j < ann->hidden; ++j) {
            int sum = *w++ * -1;
            sum = sum * 256;
            for (k = 0; k < ann->hidden; ++k) {
                sum += *w++ * i[k];
            }
            *o++ = genann_activation( sum);
        }
        i += ann->hidden;
    }
    int const *ret = o;

    /* Figure output layer. */
    for (j = 0; j < ann->outputs; ++j) {
        int sum = *w++ * -1;
        sum = sum * 256;
        for (k = 0; k < ann->hidden; ++k) {
            sum += *w++ * i[k];
        }
        *o++ = genann_activation( sum);
    }
    return ret;
```

Figure 4.3. Main error calculation loop for implemented ANN code and MAC loop, highlighted with yellow circle, where the approximate operation implemented.

our new designs by choosing the approximation levels as zero, which means the operations are implemented exactly, although they are performed in Approximate Parts. This is the best way to see the effectiveness of the dynamic sizing on power savings. All results of the experiments for dynamically sizing operations are shown in Table 4.3. In this table, power savings are shown in terms of average power and maximum power obtained from different datasets for each algorithm. *AppANew* refers to the new adder, *AppMNew* refers to the new multiplier, and *AppAMNew* refers to the design where both new designs are used. Comparisons are made among different algorithms and different designs. Accuracy of the results is not given because the approximation level is zero, which means that the results are already full-accurate although working in Approximate Parts. Hence, accuracy is not studied in this part. Table 4.3 shows that dynamic sizing can provide power savings up to 12.8%.

Table 4.3. Contributions of dynamic sizing to the power saving percentages of the new approximate blocks, given as the average and maximum of the results from different datasets for each algorithm.

Approximate Design Name	**KNN**		**KM**		**ANN**	
	Avg. Power Saving (%)	**Max. Power Saving (%)**	**Avg. Power Saving (%)**	**Max. Power Saving (%)**	**Avg. Power Saving (%)**	**Max. Power Saving (%)**
ApANew	3.1	5.2	2.1	2.9	1.8	2.5
ApMNew	4.5	7.6	4	5.2	4.9	6.5
ApAMNew	8.2	12.8	5.8	7.9	7.3	10.1

4.3.2. Approximate Addition and Subtraction with Exact Multiplication Tests

In this part, we have confined the approximate operations to the approximate addition and approximate subtraction with XADD and XSUB and executed multiplication on the exact multiplier with MUL instructions. We also tested another approx-

Table 4.4. Resources for approximate blocks and their area overhead on the CPU.

Approximate Module	**LUT amount**	**Area overhead on the core (%)**
ApA1	58	2.2
ApA2	52	2
ApANew	95	3.5
ApASNew	95	3.5
ApM1	227	8.4
ApM2	235	8.7
ApMNew	250	9.3
ApAM1	285	10.6
ApAM2	287	10.7
ApAMNew	345	12.8
ApASMNew	345	12.8

imate adder from the literature. We used a 32-bit AA7 type of adder in DeMAS [59], which is an open-source approximate adders library designed for FPGAs. We used 32-bit AA7 type of adder in DeMAS [59]. This library aims FPGA implementations, and it is described in VHDL, hence very suitable for our implementation. It is possible with this adder to specify how many bits will be approximate. We obtained two approximate adders from AA7: $AppA1$ has 28 exact and 4 approximate bits, $AppA2$ has 24 exact and 8 approximate bits. In our design, $ApANew$ represents only approximate addition, and $ApASNew$ refers to approximate addition and subtraction operations that are implemented on our adder design. The approximation levels of our designs are modified with the bit-length of the datasets. For the 8-bit datasets, it is 2, and for the 16-bit datasets, it is 3.

Approximate blocks cause area overhead on the CPU. Area overheads, shown in Table 4.4, for $AppA1$, $AppA2$ and $AppANew$ are reported as 2.2%, 2% and 3.5%, respectively. Area overhead for $ApASNew$ is the same as $ApANew$ because it uses the same design for subtraction. All three algorithms with five different datasets are

implemented on the core with these four different approximate blocks. The average accuracy of the results and the average and maximum percentage of power gain compared to the operation implemented fully on Exact Part are given in Table 4.5 and Table 4.6, respectively.

Table 4.5. Accuracy rates of the approximate adders, given as the average of the results from different datasets for each algorithm.

Approximate Design Name	**KNN**	**KM**	**ANN**
	Average Accuracy (%)	**Average Accuracy (%)**	**Average Accuracy (%)**
ApA1	99	97.6	91.6
ApA2	90.8	92.4	87.4
ApANew	98	97.9	94.8
ApASNew	92	94.3	-

4.3.3. Exact Addition and Subtraction with Approximate Multiplication Tests

In this part, we have confined the approximate operation to the approximate multiplication with XMUL and executed addition and subtraction on the exact adder with ADD and SUB instructions. Two 16x16 bits approximate multipliers, $AppM1$ and $AppM2$, are chosen in SMApproxLib [60] which is an open-source approximate multiplier library designed for FPGAs by considering the LUT structure and carry chains of modern FPGAs. Our multiplier design, $AppMNew$, together with these two multipliers are added to our core separately. The approximation level of our multiplier is again 2 for the 8-bit datasets, 3 for the 16-bit datasets. $AppM1$, $AppM2$ and $AppMNew$ have 8.4%, 8.7% and 9.3% area overhead, shown in Table 4.4, respectively. Test results are shown in Table 4.7 and Table 4.8.

Table 4.6. Power saving percentages of the approximate adders, given as the average and maximum of the results from different datasets for each algorithm.

Approximate Design Name	**KNN**		**KM**		**ANN**	
	Avg. Power Saving (%)	**Max. Power Saving (%)**	**Avg. Power Saving (%)**	**Max. Power Saving (%)**	**Avg. Power Saving (%)**	**Max. Power Saving (%)**
ApA1	4.3	6.7	3.8	5.2	5.5	7.9
ApA2	8.2	12.1	6.2	8	8.9	10.4
ApANew	9.8	13.7	9.3	12.5	11.7	14.3
ApASNew	19.5	24.1	18.5	23.5	-	-

Table 4.7. Accuracy rates of the approximate multipliers, given as the average of the results from different datasets for each algorithm.

Approximate Design Name	**KNN**	**KM**	**ANN**
	Average Accuracy (%)	**Average Accuracy (%)**	**Average Accuracy (%)**
ApM1	89.2	88.8	86.4
ApM2	88.5	87.4	85
ApMNew	95	95.6	92

Table 4.8. Power saving percentages of the approximate multipliers, given as the average and the maximum of the results from different datasets for each algorithm.

Approximate Design Name	KNN		KM		ANN	
	Avg. Power Saving (%)	Max. Power Saving (%)	Avg. Power Saving (%)	Max. Power Saving (%)	Avg. Power Saving (%)	Max. Power Saving (%)
ApM1	7.8	11.5	8.9	12.5	8.5	10.5
ApM2	8.7	12.1	10.1	13.2	9.2	13.1
ApMNew	13.1	17.9	14.1	17.7	14.7	17.8

4.3.4. Approximate Addition, Subtraction and Multiplication Tests

In the fourth experiment, XADD, XSUB, and XMUL are used together to observe the maximum power gain and the quality of results. $ApA1$ and $ApM1$ are combined as $ApAM1$, while $ApA2$ and $ApM2$ are used together and marked as $ApAM2$. Our approximate modules are also combined as $ApAMNew$ and $ApASMNew$. They are used together with two different combinations, approximate addition, and multiplication, and approximate addition, subtraction, and multiplication, to show the improvement in terms of power. $ApAM1$, $ApAM2$ and $ApAMNew$ have 10.6%, 10.7% and 12.8% area overhead, shown in Table 4.4, respectively. Results can be observed in Table 4.9 and Table 4.10.

4.3.5. Approximation Level Modification at Run-time

The last experiments are about changing the approximation level while codes are running on the CPU. In previous experiments, we fixed the approximation level for the whole codes, but in this part, we split each experiment into three equal parts. In the first part, the codes begin with the first approximation level. In the second, we set the approximation level to two while codes are running. When the last interval comes, the approximation level is set to three. These experiments will show us whether we can

Table 4.9. Accuracy rates of the approximate multipliers, given as the average of the results from different datasets for each algorithm.

Approximate Design Name	**KNN**	**KM**	**ANN**
	Average Accuracy (%)	**Average Accuracy (%)**	**Average Accuracy (%)**
ApAM1	87.6	86.1	84.2
ApAM2	86.5	84.2	82.3
ApAMNew	93	94	90.2
ApASMNew	90	91.8	-

Table 4.10. Power saving percentages of the approximate multipliers, given as the average and the maximum of the results from different datasets for each algorithm.

Approximate Design Name	**KNN**		**KM**		**ANN**	
	Avg. Power Saving (%)	**Max. Power Saving (%)**	**Avg. Power Saving (%)**	**Max. Power Saving (%)**	**Avg. Power Saving (%)**	**Max. Power Saving (%)**
ApAM1	11.3	16.2	11.1	14.5	12.7	15.2
ApAM2	16.1	20.2	14.9	19.8	16.7	20.8
ApAMNew	22.1	27.5	21.5	27.6	23.4	29.8
ApASMNew	31.7	40.3	30.6	35.2	-	-

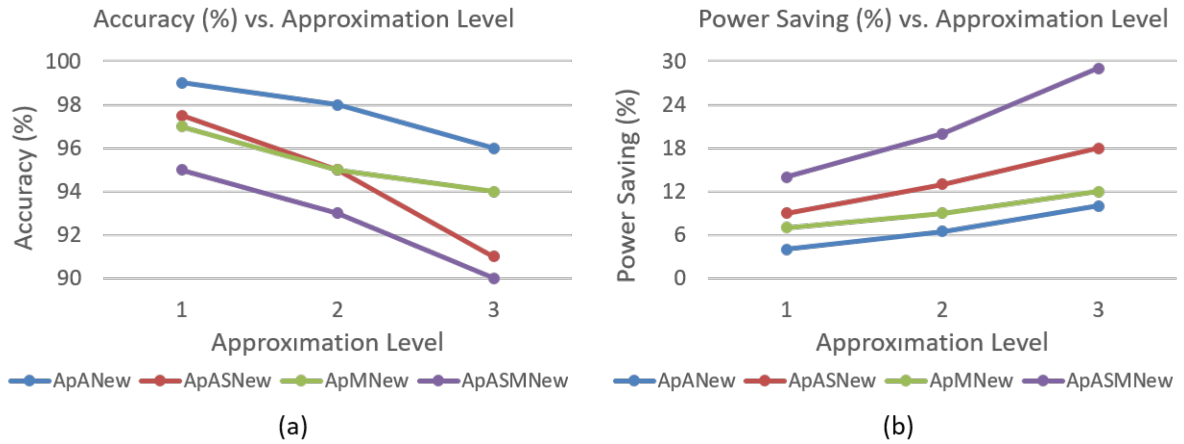

Figure 4.4. Approximation level modification at run-time (a) accuracy and (b) power saving results.

save power or not for dynamically controllable (programmable) approximations. The results shown in Figure 4.4 verify that changing the approximation level dynamically in the course of execution does not give too much overhead on our system, and it can give a similar accuracy and power saving rates with the other experiments that we fixed the approximation level. These experiments are implemented in only our new designs due to the inability to control the other designs in this setup. Figure 4.4 gives the average results for all codes implemented with 16-bit datasets.

5. DISCUSSION

Test results are shown in Tables 4.3 - 4.10 in terms of averages of the results obtained from different datasets for a specific approximate block type. We also add maximum power savings to the tables to indicate the utmost power-saving point reached in these experiments.

In the case of dynamic sizing, Table 4.3 suggests, dynamic sizing can provide up to 12.8% power savings. The overall average of the experimental results shows that the dynamic sizing operations provide about 30% of the power saving in new designs.

As seen in Table 4.5, approximate adders usually give the best accuracy of the result with the least power gain. We have observed that using a longer dataset can provide better accuracy because of having more reference data in the data memory. Accuracy of the ANN is generally the lowest accuracy achieved with the same dataset and the same adder, because the longest datasets which have 2000 points may still not be large enough to yield the best result from the ANN algorithm. Longer data also have a good impact on the improvement in power gain because increasing the amount of the data leads the MAC loops to dominate the codes. Hence most of the running time is covered by these operations, which are performed approximately. Generally, datasets with lower bit-length may be less accurate due to the difficulties of the convergence in smaller intervals for approximate calculations. It may also provide less power saving percentage because used hardware blocks may be much smaller than the longer one. However, it still depends on the length of the dataset and the approximate adder type.

$ApANew$ provides the best power saving rates with almost fully accurate results for almost all cases in the approximate addition and exact multiplication operations. $ApA1$ and $ApA2$ provides a good example of accuracy and power efficiency trade-off. $ApA1$ gives very accurate results with less power saving, but $ApA2$ may improve the power efficiency significantly with a tolerable error rate, which may be more desirable. Making subtraction operations also approximate has yielded a vital power saving.

ApASNew can save power up to 24.3%, which can be regarded as a dramatic power improvement without having an extra subtraction block.

Approximate multipliers reduce power more than the adder because the multiplier hardware blocks are generally much bigger than the adders. However, the accuracy of the results is generally worse than the adder case, since the possibility of the significant deviation from the real result is higher in the multiplication process. It is possible to reach 18% power saving together with the 95% accuracy rate. Bit-length and amount of the data in the datasets are observed to affect the accuracy. The power consumption is similar to the approximate adder cases. *ApMNew* generally provides the best accuracy for all cases, together with the best power saving rates.

The last case where both addition and multiplication are approximately executed provides the most power-efficient design at the expense of accuracy loss, as expected. Table 4.10 shows that the approximate calculations can provide significant power savings up to 40.3% together with a quality loss, which depends on the approximate adder, algorithm, and dataset. For *ApAMNew* and *ApASMNew*, average accuracy is above 90% for all codes, while average power saving is above 20% for all cases.

5.1. ASIC Implementation of the Proposed System

The primary disadvantage of the implementations in FPGA for low power applications is static power. Although we do not take the static power into consideration in the experiments, it may dominate the total power such that the power saving obtained from approximate operations does not improve the battery life of the IoT devices, as suggested. In our case, static power is responsible for about 65% of the total consumption. If we include static power in the calculations, power-saving becomes 14% at most, which can also be considered as an important save, but not enough to say that the final target for this design should be an FPGA. Hence, we also synthesized our designs with fo ASIC implementation.

Table 5.1. Power saving and area overhead of approximate designs in ASIC implementation.

Approximate Module	Average Power Saving	Area overhead on the core (%)
ApANew	7.7	2.3
ApASNew	12.3	2.3
ApMNew	11.6	6.2
ApAMNew	17.3	8.5
ApASMNew	23.1	8.5

We have performed another synthesis run with TSMC 65nm Low Power (LP) library on Cadence tools targeting ASIC. We have used TSMC memories for our data and instruction memories with the same sizes in FPGA. We did not change the main structure of the experiments, and the same tests are implemented on the new design only for power measurements. We calculate the power consumption with the help of a tcf file, which is similar to SAIF, created from the simulation tool to count toggle amounts of all signals. The power consumption of the system for ASIC synthesis is reported in Table 5.1. In ASIC, 23% of the power can be saved while the accuracy rate is above 90%, and static power is less than 1%. Power saving for ASIC design can be further improved by implementing other low power techniques like dynamic voltage scaling and clock gating.

5.2. Bit-truncation in the Approximate Blocks

Bit-truncation is a significant way to save energy, mainly when it is used for floating-point units [19,44]. However, truncation at integer units affects accuracy more dramatically than floating-point operations. To see the effect of bit truncation instead of omitting gray cells, we have performed tests without changing our approximate level control circuit. When first level bits are truncated, accuracy drops to less than 15% with a power saving of more than 30%. When the second level bits are truncated, accuracy drops 5% with a power saving of more than 50%. The best case is achieved

when the approximation level is set to 1. In this case, the accuracy is less than 45%, and power-saving is less than 18%. This error rate is too high, and energy-saving is moderate compared to our design. Hence, it turns out to be inconvenient to implement bit truncation in our system. However, the bit-truncation approach may give better results when bit-by-bit fine-grain control is adopted. A new hybrid approach may be a future work to develop a more efficient design. Coarse-grain approximation levels with fine control bit-truncation inside of these levels can be discussed to implement a new proposal.

5.3. Approximate Processor and IoT Use Case Discussion

Most of the IoT devices are required to run on batteries or other limited energy sources for years, such as home/building automation devices [61]. Thus, energy-efficient hardware solutions that help them to improve their quality of service and extend their battery life are of great importance. In this context, we can provide an efficient framework for IoT end-devices to execute clustering and classification based ML implementations in Approximate Part of our CPU when it is possible.

In our system, we can control the accuracy levels independent of the core. Existing approximate processors [11, 25] correct the precision of arithmetic operations by observing the error at the output of the arithmetic operator. However, this is not the case in our approach. The precision of the arithmetic operations is corrected by receiving the feedback from the IoT device user or other components in the IoT ecosystem. Thus, one of the basic novelty in our approach relies on the method that we handle accuracy adjustment.

We are controlling the accuracy via the approximation levels. However, the user can also view these operations as power control modes. We show an example in Table 5.2, which provides eight different power-saving modes. In the user side, it will be called as power-saving modes, because the higher level will consume more power. However, these modes will control and define our approximate levels for each approximate block, as shown in Table 5.2. In this solution, the user will know that the results may

Table 5.2. Power saving modes and their corresponding approximation levels for case study proposal.

Power Saving Modes	Approximation Level of		
	Adder	Subtractor	Multiplier
0	0	0	0
1	1	1	0
2	1	1	1
3	2	2	1
4	2	2	2
5	3	2	2
6	3	3	2
7	3	3	3

not be fully accurate in power-saving modes, but the accuracy of the system will be controlled by the IoT ecosystem. If the accuracy decreases below a certain level, then the user will receive a warning from the IoT cloud.

As we mentioned throughout this paper, we have targeted the applications where clustering and classifications are heavily used. These are basic ML operations that can also be used in IoT devices mainly to detect attack traffic from IoT botnet [5, 26]. For example, in [26], the authors deal with improving the KNN-based classifiers for online anomaly network traffic identification with the help of clustering. The paper seems to merge classification and clustering algorithms, which are the areas we are also focusing on, to reach better performance for the detection of anomalous network traffic. It can be another use case for this design. Our Approximate IoT processor will be running classification and clustering algorithms in the Approximate Part of our core to improve the performance in terms of power consumption while keeping the accuracy level high by receiving feedback from the IoT server.

Another not-too-distant future use case is also described in our published paper [53], which proposes a smart cooker that uses a neural network model for cooking. The cooker senses the ingredients via the sensors and combines them with user preferences to decide on the cooking profile and duration by making approximate computations. After the cooking time is over, the food is cooked either properly or not. Based on the feedback from the user or the customer services via IoT ecosystem, the processor will adjust its accuracy level.

6. CONCLUSION

In this thesis, the approximate computing phenomenon is studied. A wide variety of studies in the literature show that a significant amount of energy can be saved by approximate computing for fault-tolerant applications. We use a circuit-level approximation technique in a microprocessor to achieve better performance in terms of power-saving. An approximate datapath is introduced and added to a 32-bit integer RISC-V microprocessor. This new datapath includes approximate hardware for addition and multiplication operations. Examining approximate designs in the literature allows us to get familiar with different approximate techniques, including circuit-level, architecture-level, and software-level approximations. Based on circuit-level approximation, we propose our methodology for designing approximate adders. Targeting parallel-prefix adders, we offer an approximation technique based on the adder tree structure of parallel-prefix adders. The proposed method is implemented with a 32-bit Sklansky adder. The same adder is used in a Radix-4 Booth Encoded Wallace tree multiplier to introduce an approximate multiplier.

Dynamically controlling the accuracy of the results can be assessed as a further point in the approximate computing research. From our examination, we deduce that while fine-grain accuracy control is encouraged to have better control over the accuracy of the result, it may require more hardware blocks and may create significant area overhead on the design. On the other hand, using coarse-grain accuracy control to create certain accuracy levels may be more efficient. We define three approximation levels and achieve dynamic coarse-grained accuracy control. It provides better area overhead compared to the fine-grain control and still have certain levels to arrange the accuracy in the course of execution.

In the microprocessor design, we keep the architecture very simple and avoid using power-hungry blocks like a floating-point unit to improve power efficiency a bit more. We create custom instructions for our approximate operations and add them to the RISC-V ISA. We control the size of the operands for approximate datapath

and provide dynamic sizing without affecting the accuracy of the results. This idea improves the power saving significantly and does not deteriorate the performance of the core.

Experimental tests on classification and clustering algorithms are implemented to show the effectiveness of the proposed design. MAC loops in these algorithms are calculated approximately, and power savings for different datasets are demonstrated. Subtraction operations in these loops are also approximated to increase power efficiency at the expense of more accuracy loss. Achieved power saving on these ML-based algorithms shows that the proposed processor can be used in IoT end devices that are capable of implementing these types of algorithms for learning. Designed core structure allows configuring the approximation levels for specific IoT applications, as suggested in the use cases in Section 5.3.

Several future works can also be proposed based on this study. Firstly, the error probability analysis of the proposed approximate adder shows that the accuracy of the system can be improved with simple modifications on the same structure for the applications that more sensitive to the erroneous results, while power-saving can still be high. Another future work is to implement the method to the other parallel-prefix adders. The idea of omitting gray cells can be directly applied to them. The technique can also be modified for a specific type of parallel-prefix adders. Truncating operations can also be introduced in the approximate designs together with this method. We analyze that it is inefficient to use truncation in this system directly instead of our approximation method. However, a hybrid format can be proposed to improve power efficiency in the LSBs. ASIC implementation of the design can be further improved by using clock gating and dynamic voltage scaling. Creating power domains for exact and approximate datapaths to shut-down these blocks when they are not used can also be considered to reduce power consumption in ASIC implementation.

In conclusion, we propose and demonstrate the design of an approximate processor which is specialized to perform classification and clustering ML algorithms on IoT end-devices. As it is shown in Fig 3.2, the design is flexible enough to add new ap-

proximate operators though our current processor includes approximate blocks only for addition, subtraction, and multiplication operations. We try to give an approximate methodology for parallel-prefix adders and make a case study on Sklansky adders with a coarse-grain dynamic precision control. We use the same adder for subtraction and for enhancing an approximate multiplier proposed in the literature. Combining with dynamic sizing of the operands during execution time, we achieve significant dynamic power savings for FPGAs, up to 40.3%, in the conducted experiments. Experimental results point out that 30% of overall power savings are obtained by the dynamic sizing idea introduced in this study. ASIC implementation of the core is also demonstrated. In ASIC implementation, power can be saved up to 23%, and static power, which dominates in FPGAs, can be decreased below 1%. Possible application scenarios that implement ML algorithms on IoT end devices are also presented. Experimental results are discussed in detail to explain different factors that affect the results. Several future works that will improve the results obtained from this study are also discussed. This study shows that approximate computing can be efficiently utilized for microprocessor systems to meet the stringent power constraint of IoT end devices that are capable of learning.

REFERENCES

1. Esposito, D., A. G. M. Strollo and M. Alioto, "Low-power approximate MAC unit", *2017 13th Conference on Ph.D. Research in Microelectronics and Electronics (PRIME)*, pp. 81–84, June 2017.

2. Mohammadi, M., A. Al-Fuqaha, S. Sorour and M. Guizani, "Deep Learning for IoT Big Data and Streaming Analytics: A Survey", *IEEE Communications Surveys Tutorials*, Vol. 20, No. 4, pp. 2923–2960, Fourthquarter 2018.

3. La, Q. D., M. V. Ngo, T. Q. Dinh, T. Q. Quek and H. Shin, "Enabling intelligence in fog computing to achieve energy and latency reduction", *Digital Communications and Networks*, Vol. 5, No. 1, pp. 3 – 9, 2019, http://www.sciencedirect.com/science/article/pii/S2352864818301081, artificial Intelligence for Future Wireless Communications and Networking.

4. Su, M.-Y., "Using clustering to improve the KNN-based classifiers for online anomaly network traffic identification", *Journal of Network and Computer Applications*, Vol. 34, No. 2, pp. 722 – 730, 2011, http://www.sciencedirect.com/science/article/pii/S1084804510001785, efficient and Robust Security and Services of Wireless Mesh Networks.

5. Doshi, R., N. Apthorpe and N. Feamster, "Machine Learning DDoS Detection for Consumer Internet of Things Devices", *2018 IEEE Security and Privacy Workshops (SPW)*, pp. 29–35, May 2018.

6. Agarwal, V., R. A. Patil and A. B. Patki, "Architectural Considerations for Next Generation IoT Processors", *IEEE Systems Journal*, Vol. 13, No. 3, pp. 2906–2917, Sep. 2019.

7. Esmaeilzadeh, H., A. Sampson, L. Ceze and D. Burger, "Architecture Support for Disciplined Approximate Programming", *Proceedings of the Seventeenth In-*

ternational Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XVII, pp. 301–312, ACM, New York, NY, USA, 2012, http://doi.acm.org/10.1145/2150976.2151008.

8. Yesil, S., I. Akturk and U. R. Karpuzcu, "Toward Dynamic Precision Scaling", *IEEE Micro*, Vol. 38, No. 4, pp. 30–39, Jul 2018.

9. Leon, V., G. Zervakis, S. Xydis, D. Soudris and K. Pekmestzi, "Walking through the Energy-Error Pareto Frontier of Approximate Multipliers", *IEEE Micro*, Vol. 38, No. 4, pp. 40–49, Jul 2018.

10. Liu, Z., A. Yazdanbakhsh, T. Park, H. Esmaeilzadeh and N. S. Kim, "SiMul: An Algorithm-Driven Approximate Multiplier Design for Machine Learning", *IEEE Micro*, Vol. 38, No. 4, pp. 50–59, Jul 2018.

11. Venkataramani, S., V. K. Chippa, S. T. Chakradhar, K. Roy and A. Raghunathan, "Quality programmable vector processors for approximate computing", *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12, Dec 2013.

12. Karaca, M., *RISCV approximate adder code generator plugin for GCC*, 2019, https://github.com/karacasoft/riscv_approximate_codegen_plugin_for_gcc, accessed at December 2019.

13. Shafique, M., R. Hafiz, S. Rehman, W. El-Harouni and J. Henkel, "Invited: Cross-layer approximate computing: From logic to architectures", *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, June 2016.

14. Venkataramani, S., A. Sabne, V. Kozhikkottu, K. Roy and A. Raghunathan, "SALSA: Systematic logic synthesis of approximate circuits", *DAC Design Automation Conference 2012*, pp. 796–801, June 2012.

15. Shafique, M., W. Ahmad, R. Hafiz and J. Henkel, "A low latency generic accuracy

configurable adder", *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, June 2015.

16. Rehman, S., W. El-Harouni, M. Shafique, A. Kumar, J. Henkel and J. Henkel, "Architectural-space exploration of approximate multipliers", *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–8, Nov 2016.

17. Lin, C. and I. Lin, "High accuracy approximate multiplier with error correction", *2013 IEEE 31st International Conference on Computer Design (ICCD)*, pp. 33–38, Oct 2013.

18. Zhang, Q., T. Wang, Y. Tian, F. Yuan and Q. Xu, "ApproxANN: An approximate computing framework for artificial neural network", *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 701–706, March 2015.

19. Sampson, A., W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze and D. Grossman, "EnerJ: Approximate Data Types for Safe and General Low-power Computation", *SIGPLAN Not.*, Vol. 46, No. 6, pp. 164–174, Jun 2011, http://doi.acm.org/10.1145/1993316.1993518.

20. Kedem, Z. M., V. J. Mooney, K. K. Muntimadugu and K. V. Palem, "An approach to energy-error tradeoffs in approximate ripple carry adders", *IEEE/ACM International Symposium on Low Power Electronics and Design*, pp. 211–216, Aug 2011.

21. Balasubramanian, P. and D. L. Maskell, "Hardware Optimized and Error Reduced Approximate Adder", *Electronics*, Vol. 8, No. 11, 2019, https://www.mdpi.com/2079-9292/8/11/1212.

22. Kahng, A. B. and S. Kang, "Accuracy-configurable adder for approximate arithmetic designs", *DAC Design Automation Conference 2012*, pp. 820–825, June 2012.

23. Yezerla, S. K. and B. Rajendra Naik, "Design and estimation of delay, power

and area for Parallel prefix adders", *2014 Recent Advances in Engineering and Computational Sciences (RAECS)*, pp. 1–6, March 2014.

24. Macedo, M., L. Soares, B. Silveira, C. M. Diniz and E. A. C. da Costa, "Exploring the use of parallel prefix adder topologies into approximate adder circuits", *2017 24th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pp. 298–301, Dec 2017.

25. Henry, G. G., T. Parks and R. E. Hooker, "Processor That Performs Approximate Computing Instructions", , July 2016, https://patents.google.com/patent/US9389863B2/en, uS Patent 9 389 863 B2.

26. Alsouda, Y., S. Pllana and A. Kurti, "A Machine Learning Driven IoT Solution for Noise Classification in Smart Cities", *CoRR*, Vol. abs/1809.00238, 2018, http://arxiv.org/abs/1809.00238.

27. Viegas, E., A. O. Santin, A. França, R. Jasinski, V. A. Pedroni and L. S. Oliveira, "Towards an Energy-Efficient Anomaly-Based Intrusion Detection Engine for Embedded Systems", *IEEE Transactions on Computers*, Vol. 66, No. 1, pp. 163–177, Jan 2017.

28. Yu, Z., "Big Data Clustering Analysis Algorithm for Internet of Things Based on K-Means", *International Journal of Distributed Systems and Technologies*, Vol. 10, pp. 1–12, 01 2019.

29. Suarez, J. and A. Salcedo, "ID3 and k-means Based Methodology for Internet of Things Device Classification", pp. 129–133, 11 2017.

30. Moon, J., S. Kum and S. Lee, "A Heterogeneous IoT Data Analysis Framework with Collaboration of Edge-Cloud Computing: Focusing on Indoor PM10 and PM2.5 Status Prediction", *Sensors*, Vol. 19, No. 14, 2019, https://www.mdpi.com/1424-8220/19/14/3038.

31. Kang, J. and D.-S. Eom, “Offloading and Transmission Strategies for IoT Edge Devices and Networks”, *Sensors*, Vol. 19, No. 4, 2019, https://www.mdpi.com/1424-8220/19/4/835.

32. *RISC-V*, https://riscv.org, accessed on 23 Dec 2019.

33. *Shakti-Open Source Processor Development Ecosystem*, http://shakti.org.in, accessed on 16 Jan 2020.

34. *PULP Platform*, https://pulp-platform.org, accessed on 19 Jan 2020.

35. *Cores-SweRV*, https://github.com/chipsalliance/Cores-SweRV, accessed on 15 Jan 2020.

36. *Open, Lowest Power, Programmable RISC-V Solutions.*, https://www.microsemi.com/product-directory/mi-v-embedded-ecosystem/4406-risc-v-cpus, accessed on 19 Jan 2020.

37. *Rumble Development Corporation*, https://www.rumbledev.com, accessed on 19 Jan 2020.

38. *A 32-bit Microcontroller featuring a RISC-V core*, https://github.com/onchipuis/mriscv, accessed on 18 Jan 2020.

39. *DarkRISCV-opensouce RISC-V implemented from scratch in one night!*, https://github.com/darklife/darkriscv, accessed on 18 Jan 2020.

40. *XuanTie C910 - High-performance 64-bit RISC-V architecture multi-core processor with AI vector acceleration engine*, https://www.t-head.cn/product/c910?spm=a2ouz.12987052.0.0.5c5c6245WIbjoG, accessed on 16 Jan 2020.

41. *BOOM: Berkeley Out-of-Order Machine*, https://github.com/riscv-boom/riscv-boom, accessed on 16 Jan 2020.

42. *Sifive*, https://www.sifive.com, accessed on 19 Jan 2020.

43. Liu, G., J. Primmer and Z. Zhang, "Rapid Generation of High-Quality RISC-V Processors from Functional Instruction Set Specifications", *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6, June 2019.

44. Tolba, M. F., A. H. Madian and A. G. Radwan, "FPGA realization of ALU for mobile GPU", *2016 3rd International Conference on Advances in Computational Tools for Engineering Applications (ACTEA)*, pp. 16–20, July 2016.

45. Wolf, C., *RISC-V Bitmanip (Bit Manipulation) Extension*, 2019, https://github.com/riscv/riscv-bitmanip, accessed at April 2019.

46. Sklansky, J., "Conditional-Sum Addition Logic", *IRE Transactions on Electronic Computers*, Vol. EC-9, No. 2, pp. 226–231, June 1960.

47. Esposito, D., A. G. M. Strollo, E. Napoli, D. D. Caro and N. Petra, "Approximate Multipliers Based on New Approximate Compressors", *IEEE Transactions on Circuits and Systems I: Regular Papers*, Vol. 65, pp. 4169–4182, 2018.

48. Echavarria, J., S. Wildermann, E. Potwigin and J. Teich, "Efficient Arithmetic Error Rate Calculus for Visibility Reduced Approximate Adders", *IEEE Embedded Systems Letters*, Vol. 10, No. 2, pp. 37–40, June 2018.

49. Macsorley, O. L., "High-Speed Arithmetic in Binary Computers", *Proceedings of the IRE*, Vol. 49, No. 1, pp. 67–91, Jan 1961.

50. Goto, G., T. Sato, M. Nakajima and T. Sukemura, "A 54*54-b regularly structured tree multiplier", *IEEE Journal of Solid-State Circuits*, Vol. 27, No. 9, pp. 1229–1236, Sep. 1992.

51. RISC-V, *RISC-V GCC*, 2019, https://github.com/riscv/riscv-gcc, accessed on 18 March 2018.

52. *RISC-V binutils*, https://github.com/riscv/riscv-binutils-gdb, accessed on 25 Nov 2019.

53. Taştan, I., M. Karaca and A. Yurdakul, "Approximate CPU Design for IoT End-Devices with Learning Capabilities", *Electronics*, Vol. 9, No. 1, 2020, https://www.mdpi.com/2079-9292/9/1/125.

54. Bhatt, R., *Fundamental Principles of Optical Lithography*, Independently published, 2017.

55. Rohra, J., P. Thakur, S. J.N., B. Perumal and R. Bhatt, "User Localization in an Indoor Environment using Fuzzy Hybrid of Particle Swarm Optimization & Gravitational Search Algorithm with Neural Networks", , 12 2016.

56. Freire, A. L., G. A. Barreto, M. Veloso and A. T. Varela, "Short-term memory mechanisms in neural network learning of robot navigation tasks: A case study", *2009 6th Latin American Robotics Symposium (LARS 2009)*, pp. 1–6, Oct 2009.

57. Dua, D. and C. Graff, *UCI Machine Learning Repository*, 2019, http://archive.ics.uci.edu/ml, accessed on 23 March 2019.

58. Winkle, L. V., *Genann*, 2019, https://github.com/codeplea/genann, accessed at June 2019.

59. Prabakaran, B. S., S. Rehman, M. A. Hanif, S. Ullah, G. Mazaheri, A. Kumar and M. Shafique, "DeMAS: An efficient design methodology for building approximate adders for FPGA-based systems", *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 917–920, March 2018.

60. Ullah, S., S. S. Murthy and A. Kumar, "SMApproxLib: Library of FPGA-based Approximate Multipliers", *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pp. 1–6, June 2018.

61. Hahm, O., E. Baccelli, H. Petersen and N. Tsiftes, "Operating Systems for Low-End Devices in the Internet of Things: A Survey", *IEEE Internet of Things Journal*, Vol. 3, No. 5, pp. 720–734, Oct 2016.

APPENDIX A: Test System for Experiments

A.1. start.S File

Please see the footnote.[1]

```
.section .text
.global _start
_start:

reset_handler:
li x1, 0
li x2, 0
li x3, 0
li x4, 0
li x5, 0
li x6, 0
li x7, 0
li x8, 0
li x9, 0
li x10, 0
li x11, 0
li x12, 0
li x13, 0
li x14, 0
li x15, 0

    .option push
    .option norelax
```

[1]This part is originally prepared by Mehmet Alp Şarkışla and Ömer Faruk Irmak.

```
    la gp, __global_pointer$
    .option pop
    la sp, _stack_start

    call __libc_init_array
    call main
    ebreak

.global _init
.global _fini
_init:
_fini:
ret
```

A.2. link.ld File

Please see the footnote.[2]

```
SEARCH_DIR(.)
__DYNAMIC  =  0;

MEMORY
{

/*///////////// instruction memory ////////////////*/
    rom   : ORIGIN = 0, LENGTH = 40K
/*/////////////   data memory      ////////////////*/
    ram   : ORIGIN = 40K, LENGTH = 80K
/*/////////////   stack memory     ////////////////*/
    stack : ORIGIN = 120K, LENGTH = 10K
```

[2]This part is originally prepared by Mehmet Alp Şarkışla and Ömer Faruk Irmak, and modified by İbrahim Taştan for use in this study.

```
}

/* Stack information variables */
_min_stack  = 0x0;   /* (1K - 4) minimum stack space */
_stack_len   = LENGTH(stack);
_stack_end  = ORIGIN(stack);
_stack_start = ORIGIN(stack) + LENGTH(stack);
__global_pointer$ = 0; /* 2K */

/* We have to align each sector to word boundaries as our current
*s19->slm conversion scripts are not able to handle
*non-word aligned sections. */

SECTIONS
{
    .vectors :
    {
        . = ALIGN(4);
        KEEP(*(.vectors))
    } > ram

    .text : {
        . = ALIGN(4);
        _stext = .;
        *(.text)
        _etext  =  .;
        __CTOR_LIST__ = .;
        LONG((__CTOR_END__ - __CTOR_LIST__) / 4 - 2)
        *(.ctors)
        LONG(0)
```

```
    __CTOR_END__ = .;
    __DTOR_LIST__ = .;
    LONG((__DTOR_END__ - __DTOR_LIST__) / 4 - 2)
    *(.dtors)
    LONG(0)
    __DTOR_END__ = .;
    *(.lit)
    *(.shdata)
    _endtext = .;
}  > rom

/*--------------------------------------------------------*/
/* Global constructor/destructor segement                 */
/*--------------------------------------------------------*/

.preinit_array     :
{
  PROVIDE_HIDDEN (__preinit_array_start = .);
  KEEP (*(.preinit_array))
  PROVIDE_HIDDEN (__preinit_array_end = .);
} > ram

.init_array     :
{
  PROVIDE_HIDDEN (__init_array_start = .);
  KEEP (*(SORT(.init_array.*)))
  KEEP (*(.init_array ))
  PROVIDE_HIDDEN (__init_array_end = .);
} > ram

.fini_array     :
```

```
{
  PROVIDE_HIDDEN (__fini_array_start = .);
  KEEP (*(SORT(.fini_array.*)))
  KEEP (*(.fini_array ))
  PROVIDE_HIDDEN (__fini_array_end = .);
} > ram

.rodata : {
    . = ALIGN(4);
    *(.rodata);
    *(.rodata.*)
} > ram

.shbss :
{
    . = ALIGN(4);
    *(.shbss)
} > ram

.data : {
    . = ALIGN(4);
    sdata  =  .;
    _sdata  =  .;
    *(.data);
    *(.data.*)
    edata  =  .;
    _edata  =  .;
} > ram

.bss :
{
```

```
    . = ALIGN(4);
    _bss_start = .;
    *(.bss)
    *(.bss.*)
    *(.sbss)
    *(.sbss.*)
    *(COMMON)
    _bss_end = .;
} > ram

/* ensure there is enough room for stack */
.stack (NOLOAD): {
    . = ALIGN(4);
    . = . + _min_stack ;
    . = ALIGN(4);
    stack = . ;
    _stack = . ;
} > stack

.stab  0 (NOLOAD) :
{
    [ .stab ]
}

.stabstr  0 (NOLOAD) :
{
    [ .stabstr ]
}

.bss :
{
```

```
        . = ALIGN(4);
        _end = .;
    } > ram
}
```

A.3. Linux Commands for Compiling C Codes

Please see the footnote.[3]

```
############ VARIABLES ############
##### set variables
######### Specify the C code which is converted to assembly.
C_CODE = "./knn/knn_sel_robot_1960_40"
######### Specify the instruction set (which extensions of RISC-V ISA)
######### for RISC-V GCC.
I_SET = "rv32im"
######### Specify the optimization level (0, 1, 2 or 3) for RISC-V GCC.
OPT_LEV = "3"

############ COMMANDS ############
##### run commands
######### Remove previous .exe file for any possible clash.
rm _$OPT_LEV.exe
######### Generate new .exe file by using link.ld and start.S files.
riscv32-unknown-elf-gcc -g0 -O$OPT_LEV -march=$I_SET -Wl,--no-relax
-nostartfiles start.S $C_CODE.c -T link.ld -o _$OPT_LEV.exe -lm
######### Generate .hex file for instructions
riscv32-unknown-elf-objcopy -O binary -j .text -j  .text.startup
_$OPT_LEV.exe $C_CODE$OPT_LEV.hex
######### Dump .hex file for readability with pseudo names
```

[3]This part is originally prepared by Mehmet Alp Şarkışla and modified by İbrahim Taştan for use in this study.

```
######### for instructions and registers
riscv32-unknown-elf-objdump -d _$OPT_LEV.exe > $C_CODE$OPT_LEV.idump
######### Dump .hex file for readability  with real names
######### for instructions and registers
riscv32-unknown-elf-objdump --disassembler-options=no-aliases,numeric
-D -g  _$OPT_LEV.exe > $C_CODE$OPT_LEV.all
######### Generate .mem file for the data stored in the memory
riscv32-unknown-elf-objcopy -O binary  --remove-section
.text --remove-section .text.startup
--strip-debug  _$OPT_LEV.exe $C_CODE$OPT_LEV.mem
######### Dump .mem file for readability
riscv32-unknown-elf-objdump  -s  -b binary --adjust-vma=0xa000
$C_CODE$OPT_LEV.mem  > $C_CODE$OPT_LEV.mdump
```

A.4. Verilog Codes for the Approximate Adder

```
//// SKLANSKY APPROXIMATE ADDER TOP MODULE ////

module Sklansky_sizeable(
size_enable,approx_level,a,b,ci,co,s);

 input [31:0] size_enable;
 input [2:0] approx_level;
 input [31:0] a,b;
 output [31:0] s;
 input ci;
 output co;

wire [31:0] P,C;
wire [31:0] G0;
```

```
////  P & G generation ////
pg_gen U1 (
     .size(size_enable),
.a(a),
     .b(b),
     .G(G0),
     .P(P)
    );

//// tree adder instantiation ////
sklansky_tree sklansky_tree_32_bit (
    .approx_level(approx_level),
    .G0(G0),
    .P0(P),
    .C(C)
    );

//// XOR operations to obtain the results ////
xor2 s0 (
        .a(ci),
        .b(P[0]),
        .en(size_enable[0]),
        .o(s[0])
        );

  genvar             i;
  generate
    for (i=1; i<32; i=i+1) begin: sum
xor2 s1_31 (
 .a(C[i-1]),
 .b(P[i]),
```

```
 .en(size_enable[i]),
 .o(s[i])
 );
    end
  endgenerate

assign co   = C[31];

endmodule

//// SKLANSKY TREE CODE ////

module sklansky_tree(
input [31:0] G0,P0,
input [2:0] approx_level,
output [31:0] C
 );

wire [31:0] P[5:0], G[5:0];

assign G[0] = G0;
assign P[0] = P0;

//// creating gray cells row by row //////

gray_cell row_1_g (
    .approx(approx_level[0]),
    .G_ik(G[0][1]),
    .P_ik(P[0][1]),
    .G_k1j(G[0][0]),
    .G_ij(G[1][1])
```

```
    );


  genvar             i;
  generate
    for (i=2; i<4; i=i+1)

      begin: row2_g
gray_cell row_2_g (
                               .approx(approx_level[0]),
 .G_ik(G[1][i]),
 .P_ik(P[1][i]),
 .G_k1j(G[1][1]),
 .G_ij(G[2][i])
 );
      end
  endgenerate

  genvar             j;
  generate
    for (j=4; j<8; j=j+1)

      begin: row3_g
gray_cell row_3_g (
                               .approx(approx_level[0]),
 .G_ik(G[2][j]),
 .P_ik(P[2][j]),
 .G_k1j(G[2][3]),
 .G_ij(G[3][j])
 );
      end
```

```
  endgenerate

  genvar             k;
  generate
    for (k=8; k<16; k=k+1)

      begin: row4_g
gray_cell row_4_g (
                           .approx(approx_level[1]),
 .G_ik(G[3][k]),
 .P_ik(P[3][k]),
 .G_k1j(G[3][7]),
 .G_ij(G[4][k])
 );
      end
  endgenerate



  genvar             l;
  generate
    for (l=16; l<32; l=l+1)

      begin: row5_g
gray_cell row_5_g (
                           .approx(approx_level[2]),
 .G_ik(G[4][l]),
 .P_ik(P[4][l]),
 .G_k1j(G[4][15]),
 .G_ij(G[5][l])
 );
      end
```

```
  endgenerate

//// creating black cells row by row //////

  genvar             m;
  generate
    for (m=3; m<32; m=m+2)

      begin: row1_b
black_cell row_1_b (
 .G_ik(G[0][m]),
 .P_ik(P[0][m]),
 .G_k1j(G[0][m-1]),
 .P_k1j(P[0][m-1]),
 .G_ij(G[1][m]),
 .P_ij(P[1][m])
 );
      end
  endgenerate

  genvar             n;
  genvar             nn;
  generate
    for (n=6; n<32; n=n+4) begin: xx2
for (nn=0; nn<2; nn=nn+1)
      begin: row2_b
black_cell row_2_b (
 .G_ik(G[1][n+nn]),
 .P_ik(P[1][n+nn]),
 .G_k1j(G[1][n-1]),
 .P_k1j(P[1][n-1]),
```

```
 .G_ij(G[2][n+nn]),
 .P_ij(P[2][n+nn])
 );
end
    end
  endgenerate


  genvar             p;
  genvar             pp;
  generate
    for (p=12; p<32; p=p+8) begin: xx3
for (pp=0; pp<4; pp=pp+1)
      begin: row3_b
black_cell row_3_b (
 .G_ik(G[2][p+pp]),
 .P_ik(P[2][p+pp]),
 .G_k1j(G[2][p-1]),
 .P_k1j(P[2][p-1]),
 .G_ij(G[3][p+pp]),
 .P_ij(P[3][p+pp])
 );
end
    end
  endgenerate

  genvar             r;
  generate
    for (r=24; r<32; r=r+1)
      begin: row4_b
black_cell row_4_b (
```

```
 .G_ik(G[3][r]),
 .P_ik(P[3][r]),
 .G_k1j(G[3][23]),
 .P_k1j(P[3][23]),
 .G_ij(G[4][r]),
 .P_ij(P[4][r])
 );
    end
  endgenerate

//// creating buffer cells row by row //////



  genvar             a;
  generate
    for (a=0; a<31; a=a+2)

      begin: row1
buffer_cell row_1_buff (
 .in1(P[0][a]),
 .in2(G[0][a]),
 .out1(P[1][a]),
 .out2(G[1][a])
 );
      end
  endgenerate




  genvar             b;
```

```
  genvar             bb;
  generate
    for (b=0; b<32; b=b+4) begin: yy2
for (bb=0; bb<2; bb=bb+1)
      begin: row2
buffer_cell row_2_buff (
 .in1(P[1][b+bb]),
 .in2(G[1][b+bb]),
 .out1(P[2][b+bb]),
 .out2(G[2][b+bb])
 );
end
    end
  endgenerate

  genvar             c;
  genvar             cc;
  generate
    for (c=0; c<32; c=c+8) begin: yy3
for (cc=0; cc<4; cc=cc+1)
      begin: row3
buffer_cell row_3_buff (
 .in1(P[2][c+cc]),
 .in2(G[2][c+cc]),
 .out1(P[3][c+cc]),
 .out2(G[3][c+cc])
 );
end
    end
  endgenerate
```

```
  genvar             d;
  genvar             dd;
  generate
    for (d=0; d<32; d=d+16) begin: yy4
for (dd=0; dd<8; dd=dd+1)
      begin: row4
buffer_cell row_4_buff (
 .in1(P[3][d+dd]),
 .in2(G[3][d+dd]),
 .out1(P[4][d+dd]),
 .out2(G[4][d+dd])
 );
end
    end
  endgenerate

  genvar             e;

  generate
    for (e=0; e<16; e=e+1) begin: yy5
buffer_cell row_5_buff (
 .in1(P[4][e]),
 .in2(G[4][e]),
 .out1(P[5][e]),
 .out2(G[5][e])
 );
    end
  endgenerate

assign C = G[5];
```

```
endmodule

//// GRAY CELL CODE ////

module gray_cell(
input G_ik,P_ik,G_k1j,
input approx,
output G_ij
    );

reg G_out;

reg G1, G2, P1;

always @(*) begin
     case (approx)
         1'b0: begin
                G_out <= (G_ik)|(P_ik & G_k1j);
               end
         1'b1: begin
                G_out <= (G_ik);
               end
      endcase
end

assign G_ij = G_out;

endmodule
```

A.5. Verilog Testbench for the Results of the Tests

```
module core_tb;

// Inputs
reg ap_clk;
reg ap_rst;
reg ap_start;

// Outputs
wire ap_done;
wire ap_idle;
wire ap_ready;
reg [16:0]Data_Result_Address;
wire [7:0]Data_Result;

// Instantiate the Unit Under Test (UUT)
riscv_core uut (
.ap_clk(ap_clk),
.ap_rst(ap_rst),
.ap_start(ap_start),
.ap_done(ap_done),
.ap_idle(ap_idle),
.ap_ready(ap_ready),
.Data_Result_Address(Data_Result_Address),
.Data_Result(Data_Result)
);

integer f;
  initial begin
forever #5 ap_clk = ~ap_clk;
  end
```

```
  initial begin
f = $fopen("output.txt","w"); // change the name for the core
ap_clk = 0;
ap_rst = 0;
ap_start = 0;
#100;
ap_rst = 1;
ap_start = 1;
#100;
ap_rst = 0;
ap_start = 1;
#155424420; // specify the time when the code finishes.
/// After finishing the code, write the results into a file. ///
        for (Data_Result_Address=40240; Data_Result_Address<40840;
              Data_Result_Address=Data_Result_Address+4 ) begin
              /* Data_Result_Addresses specifies where
                 the results reside in memory. */
                 #10;
                 $fwrite(f,"%d\n",Data_Result);
        end
        $fclose(f);
        $finish;
  end
endmodule
```

A.6. Constraint File

```
### Clock description
set_property PACKAGE_PIN Y9 [get_ports {ap_clk}];  # "GCLK"
create_clock -period 10 -name CLK -waveform {0 5} [get_ports ap_clk]
```

```
set_property IOSTANDARD LVCMOS33
[get_ports -of_objects [get_iobanks 13]]

### Input description
set_property PACKAGE_PIN P16 [get_ports ap_rst]
set_property PACKAGE_PIN R16 [get_ports ap_start]
set_property IOSTANDARD LVCMOS25
[get_ports -of_objects [get_iobanks 34]]

### Output description
set_property PACKAGE_PIN T22 [get_ports ap_done]
set_property PACKAGE_PIN T21 [get_ports ap_idle]
set_property PACKAGE_PIN U22 [get_ports ap_ready]
set_property IOSTANDARD LVCMOS33
[get_ports -of_objects [get_iobanks 33]]
```

A.7. Writing SAIF File

```
### Open post-implementation functional simulation in Vivado
restart
open_saif "core.saif" ## change the name of the SAIF file.
log_saif [get_objects -r *] ## record activity of all signals
add_force {/core_tb/ap_clk} -radix hex {1 0ns} {0 5000ps}
-repeat_every 10000ps
add_force {/core_tb/ap_rst} -radix hex {1 0ns}
add_force {/core_tb/ap_start} -radix hex {1 0ns}
run 100 ns
add_force {/core_tb/ap_rst} -radix hex {0 0ns}
run 43556245 ns ## specify the time when the code finishes.
close_saif
```

A.8. TCL Script for Vivado

```
##################################
###### SET DESIGN VARIABLES ######
##################################

#### COMPUTER LOCATION #### change it for different computers,
#### it specifies where "THESIS" file resides
set COMPUTER_LOCATION "C:/Users/ASUS/Desktop"
#### PROJECT LOCATION ####
#### change it for each algorithm type. Such as KNN_ROBOT_200
set PROJECT_LOCATION
"${COMPUTER_LOCATION}/THESIS/CORES/KNN_ROBOT_200/PROJECTS"
#### PROJECT NAME ####
#### change it for each subcore ExAdd_ExMul, ExAdd_AppMul etc.
set PROJECT_NAME "ExAdd_ExMul"
#### Setting board and FPGA #### do not touch them.
set FPGA "xc7z020clg484-1"
set BOARD "em.avnet.com:zed:part0:1.4"
#### testbench and constraint locations #### do not touch them.
set TESTBENCH_LOCATION "${COMPUTER_LOCATION}/THESIS/TESTBENCHES"
set CONSTRAINT_LOCATION "${COMPUTER_LOCATION}/THESIS/CONSTRAINTS"
#### Instruction types ####
#### defines type of instruction and instruction memory.
#### EI -> exact instructions
#### AI_1 -> exact mul approximate add instructions
#### AI_2 -> exact add approximate mul instructions
#### AI_3 -> approximate add & mul instructions
set Instruction_Memory "EI"
#### approximate code locations ####
#### specify which types of the approximate codes.
set APP_ADD_LOCATION
```

```
"${COMPUTER_LOCATION}/THESIS/APPROXIMATE_CODES/ADD"
set APP_MUL_LOCATION
"${COMPUTER_LOCATION}/THESIS/APPROXIMATE_CODES/MUL/MUL16x16/app_acc"
#### approximate code types ####
#### specify which types of the approximate codes.
set App_Add_Type "AppAddAcc"
set App_Mul_Type "AppMulAcc"
#### power results directory ####
#### it specifies where power_reports & saif files are written.
set POWER_ANALYSIS_DIR
"${PROJECT_LOCATION}/POWER_REPORTS"
#### simulation time ####
#### it specifies the amount of time required for the simulation.
#### change for different algorithms
set SIMULATION_TIME "43556245"

#########################################
###### START DESIGN and READ FILES ######
#########################################

## start_gui ## commented, if you want GUI, uncomment it.
create_project ${PROJECT_NAME}
${PROJECT_LOCATION}/PROJECTS/${PROJECT_NAME} -part ${FPGA}
set_property board_part ${BOARD} [current_project]
add_files -norecurse {${PROJECT_LOCATION}/RAW_CODES/datapath.v
${PROJECT_LOCATION}/RAW_CODES/${Instruction_Memory}.v
${PROJECT_LOCATION}/RAW_CODES/${Instruction_Memory}.dat
${PROJECT_LOCATION}/RAW_CODES/riscv_core_udiv_3cud.v
${PROJECT_LOCATION}/RAW_CODES/riscv_core_sdiv_3bkb.v
${PROJECT_LOCATION}/RAW_CODES/${App_Add_Type}.v
${PROJECT_LOCATION}/RAW_CODES/${App_Mul_Type}.v
```

```
${PROJECT_LOCATION}/RAW_CODES/datapath_rf.v
${PROJECT_LOCATION}/RAW_CODES/datapath_rf_ram.dat
${PROJECT_LOCATION}/RAW_CODES/alu.v
${PROJECT_LOCATION}/RAW_CODES/datapath_mem.v
${PROJECT_LOCATION}/RAW_CODES/datapath_mem_ram.dat
${PROJECT_LOCATION}/RAW_CODES/riscv_core.v}
add_files ${APP_ADD_LOCATION}
add_files ${APP_MUL_LOCATION}
add_files -fileset sim_1 -norecurse ${TESTBENCH_LOCATION}/core_tb.v
import_files -force -norecurse
import_files -fileset constrs_1 -force -norecurse
${CONSTRAINT_LOCATION}/constr_1.xdc
update_compile_order -fileset sources_1
update_compile_order -fileset sources_1

###################################
###### SYNTHESIZE THE DESIGN ######
###################################

launch_runs synth_1 -jobs 8

##################################
###### IMPLEMENT THE DESIGN ######
##################################

launch_runs impl_1 -jobs 8

#######################################
###### REPORT POWER WITHOUT SAIF ######
#######################################
```

```
report_power
-file ${POWER_ANALYSIS_DIR}/${PROJECT_NAME}_pwr_imp_rpt.txt

##########################
###### WRITING SAIF ######
##########################

launch_simulation -mode post-implementation -type functional
source core_tb.tcl
restart
open_saif "${POWER_ANALYSIS_DIR}/${PROJECT_NAME}.saif"
log_saif [get_objects -r *]
add_force {/core_tb/ap_clk} -radix hex {1 0ns} {0 5000ps}
-repeat_every 10000ps
add_force {/core_tb/ap_rst} -radix hex {1 0ns}
add_force {/core_tb/ap_start} -radix hex {1 0ns}
run 100 ns
add_force {/core_tb/ap_rst} -radix hex {0 0ns}
run ${SIMULATION_TIME} ns //
close_saif

####################################
###### REPORT POWER WITH SAIF ######
####################################

read_saif {${POWER_ANALYSIS_DIR}/${PROJECT_NAME}.saif}

report_power -file
${POWER_ANALYSIS_DIR}/${PROJECT_NAME}_pwr_imp_rpt_with_saif.txt
```